Search

photon

Fixed Point

Introduction

In Quantum the `FP` struct (Fixed Point) completely replaces all usages of `floats` and `doubles` to ensure cross-platform determinism. It offers versions of common math data structures like FPVector2, FPVector3, FPMatrix, FPQuaternion, RNGSession, FPBounds2, etc. All systems in Quantum, including physics and navigation, exclusively use `FP` values in their computations.

The fixed-point type implemented in Quantum is `Q48.16`. It has proved to a good balance between precision and performance with a bias towards the latter.

Internally `FP` uses one long to represent the combined fixed-point number (whole number + decimal part); the long value can be accessed and set via `FP.RawValue`.

Quantum's FP Math uses carefully tuned look up tables for fast trigonometric and square root functions (see `QuantumSDK\quantum_unity\Assets\Photon\Quantum\Resources\LUT`).

Back To Top

Parsing FPs

The representable `FP` fraction is limited and never as accurate as a `double`. Parsing is an approximation and will round to the nearest possible `FP` precision. This is reflected in:

- different parsing methods yielding different results on the same machine.

```
FP.FromFloat(1.1f).RawValue != FP.FromString("1.1").RawValue
```

Back To Top

TLDR

- Use from float only during edit time, never inside the simulation or at runtime.
- It is best to convert from raw whenever possible.

Back To Top

FP.FromFloat_UNSAFE()

Converting from `float` is not deterministic due to rounding errors and should **never** be done inside the simulation. Doing such a conversion in the simulation will cause desyncs 100% of the time.

However, it can be used during **edit** or **build time**, when the converted (FP) data is first created and then shared with everyone. **IMPORTANT:** Data generated this way on **different** machines may not be compatible.

```
var v = FP.FromFloat_UNSAFE(1.1f);
```

Back To Top

FP.FromString_UNSAFE()

This will internally parse the `string` as a float and then convert to `FP`. All caveats from `FromFloat_UNSAFE()` apply here as well.

```
var v = FP.FromFloat_UNSAFE("1.1");
```

Back To Top

FP.FromString()

This is deterministic and therefore safe to use anywhere but may not be the most performant option. A typical use case is balancing information (patch) that clients load from a server and then use to update data in Quantum assets.

Be aware of the string locale! It only parses English number formatting for decimals and requires a dot (e.g. 1000.01f).

```
var v = FP.FromFloat("1.1");
```

Back To Top

```
var v = FP.FromRaw(72089);
```

This snippet can be used to create a FP converter window in Unity for convient conversion.

```
using System;
using UnityEditor;
using Photon.Deterministic;

public class FPConverter : EditorWindow {
  private float _f;
  private FP _fp;

  [MenuItem("Quantum/FP Converter")]
  public static void ShowWindow() {
    GetWindow(typeof(FPConverter), false, "FP Converter");
  }

  public virtual void OnGUI() {
    _f = EditorGUILayout.FloatField("Input", _f);
    try {
      _fp = FP.FromFloat_UNSAFE(_f);
      var f = FPPropertyDrawer.GetRawAsFloat(_fp.RawValue);
      var rect = EditorGUILayout.GetControlRect(true);
      EditorGUI.FloatField(rect, "Output FP", f);
      QuantumEditorGUI.Overlay(rect, "(FP)");
      EditorGUILayout.LongField("Output Raw", _fp.RawValue);
    }
    catch (OverflowException e) {
      EditorGUILayout.LabelField("Out of range");
    }
  }
}
```

Back To Top

Const Variables

The `FP._1_10` syntax can not be extended or generated.

`FP` is a struct and can therefore not be used as a constant. It is, however, possible to hard-code and use " `FP` " values in `const` variables:

1. Combine pre-defined `FP._1` `static` getters or `FP.Raw._1` `const` variables.

```
FP foo = FP._1 + FP._0_10;
// or
```

```
const long MagicNumber = FP.Raw._1 + FP.Raw._0_10;

FP foo = default;
foo.RawValue = MagicNumber;
// or
foo = FP.FromRaw(MagicNumber);
```

2. Convert the specific float once to FP and save the raw value as a constant

```
const long MagicNumber = 72089; // 1.1

var foo = FP.FromRaw(MagicNumber);
// or
foo.RawValue = MagicNumber;
```

3. Create the constant inside the Quantum DSL

```
#define FPConst 1.1
```

Then use like this:

```
var foo = default(FP);
foo += Constants.FPConst;
// or
foo.RawValue += Constants.Raw.FPConst;
```

It will generate code to represent the constant in the following way:

```
public static unsafe partial class Constants {
  public const Int32 PLAYER_COUNT = 8;
  /// <summary>1.100006</summary>

  public static FP FPConst {
    [MethodImpl(MethodImplOptions.AggressiveInlining)] get {
      FP result;
      result.RawValue = 72090;
      return result;
    }
  }
  public static unsafe partial class Raw {
    /// <summary>1.100006</summary>
    public const Int64 FPConst = 72090;
  }
}
```

4. Define `readonly` `static` variables in the class.

There is a performance penalty compared to `const` variables and don't forget to mark `readonly` because randomly changing the value during runtime could lead to desyncs.

Back To Top

Casting

Implicit casting to `FP` from `int` , `uint` , `short` , `ushort` , `byte` , `sbyte` is allowed and safe.

```
FP v = (FP)1;
FP v = 1;
```

Explicit casting from `FP` to `float` or `double` is possible, but obviously should not be used inside the simulation.

```
var v = (double)FP._1;
var v = (float)FP._1;
```

Casting to integer and back is safe though.

```
FP v = (FP)(int)FP._1;
```

Unsafe casts are marked as `[obsolete]` and will cause a `InvalidOperationException` .

```
FP v = 1.1f;  // ERROR
FP v = 1.1d;  // ERROR
```

Back To Top

Inlining

All low-level Quantum systems use manual inlined `FP` arithmetic to extract every ounce of performance possible. Fixed point math uses integer division and multiplication. To achieve this, the result or dividend are bit-shifted by the FP-Precision (16) before or after the calculation.

```
var v = parameter * FP._0_01;

// inlined integer math
FP v = default;
v.RawValue = (parameter.RawValue * FP._0_01.RawValue) >> FPLut.PRECISION;
```

```
var v = parameter / FP._0_01;

// inlined integer math
```

photon

Back To Top

Overflow

`FP.UseableMax` represents the highest `FP` number that can be multiplied with itself and not cause an overflow (exceeding `long` range).

```
FP.UseableMax
    Decimal: 32767.9999847412
    Raw: 2147483647
    Binary: 1111111111111111111111111111111 = 31 bit


FP.UseableMin
    Decimal: -32768
    Raw: -2147483648
    Binary: 10000000000000000000000000000000 = 32 bit
```

Back To Top

Precision

The general `FP` precision is decent when the numbers are kept within a certain range `(0.01..1000)`. FP-math related precision problems usually produce inaccurate results and tend to make systems (based on math) unstable. A very common case is to multiply very high or small numbers and than returning to the original range by division for example. The resulting numbers lose precision.

Another example is this method excerpt from `ClosestDistanceToTriangle`. Where `t0` is calculated from multiplying two dot products with each other, where a dot-product is already also a result of multiplications. This is a problem when very accurate results are expected. A way to mitigate this issue is shifting the values artificially before the calculation then shift the result back. This will work when the ranges of the input are somewhat known.

```
var diff = p - v0;
var edge0 = v1 - v0;
var edge1 = v2 - v0;
var a00 = Dot(edge0, edge0);
var a01 = Dot(edge0, edge1);
var a11 = Dot(edge1, edge1);
var b0 = -Dot(diff, edge0);
var b1 = -Dot(diff, edge1);
var t0 = a01 * b1 - a11 * b0;
var t1 = a01 * b0 - a00 * b1;
// ...
closestPoint = v0 + t0 * edge0 + t1 * edge1;
```

FPAnimationCurves

Unity comes with a set of tools which are useful to express some values in the form of curves. It comes with a custom editor for such curves, which are then serialised and can be used in runtime in order to evaluate the value of the curve in some specific point.

There are many situations where curves can be used, such as expressing steering information when implementing vehicles, the utility value during the decision making for an AI agent (as done in Bot SDK's Utility Theory), getting the multiplier value for some attack's damage, and so on.

The Quantum SDK already comes with its own implementation of an animation curve, named `FPAnimationCurve` , evaluated as FPs. This custom type, when inspected directly on Data Assets and Components in Unity, are drawn with Unity's default Animation Curve editors, whose data are then internally baked into the deterministic type.

Back To Top

Polling Data From An FPAnimationCurve

The code needed to poll some data from a curve, with Quantum code, is very similar to the Unity's version:

```
// This returns the pre-baked value, interpolated accordingly to the curve's configura
FP myValue = myCurve.Evaluate(FP._0_50);
```

Back To Top

Creating FPAnimationCurves Directly On The Simulation

Here are the snippets to create a deterministic animation curve from scratch, directly on the simulation:

```
// Creating a simple, linear curve with five key points
// Change the parameter as prefered
public static class FPAnimationCurveUtils
{
    public static FPAnimationCurve CreateLinearCurve(FPAnimationCurve.WrapMode preWrapM
    {
        return new FPAnimationCurve
        {
            Samples = new FP[5] { FP._0, FP._0_25, FP._0_50, FP._0_75, FP._1 },
            PostWrapMode = (int)postWrapMode,
            PreWrapMode = (int)preWrapMode,
            StartTime = 0,
            EndTime = 1,
            Resolution = 32
        };
    }
}
```

```
// It can also be used directly to pre-initialise a curve in an asset
public unsafe partial class CollectibleData
{
    public FPAnimationCurve myCurve = FPAnimationCurveUtils.CreateLinearCurve(FPAnimat
```

Back To Top

Baking Unity's AnimationCurve Into A FPAnimationCurve

The snippets needed for converting a regular Unity `AnimationCurve` into an `FPAnimationCurve` can be found HERE.

To Document Top

photon

Connect

Public Discord
YouTube
Facebook
Twitter
Blog
Contact Us

Languages

English
日本語
한국어
简体中文
繁体中文