

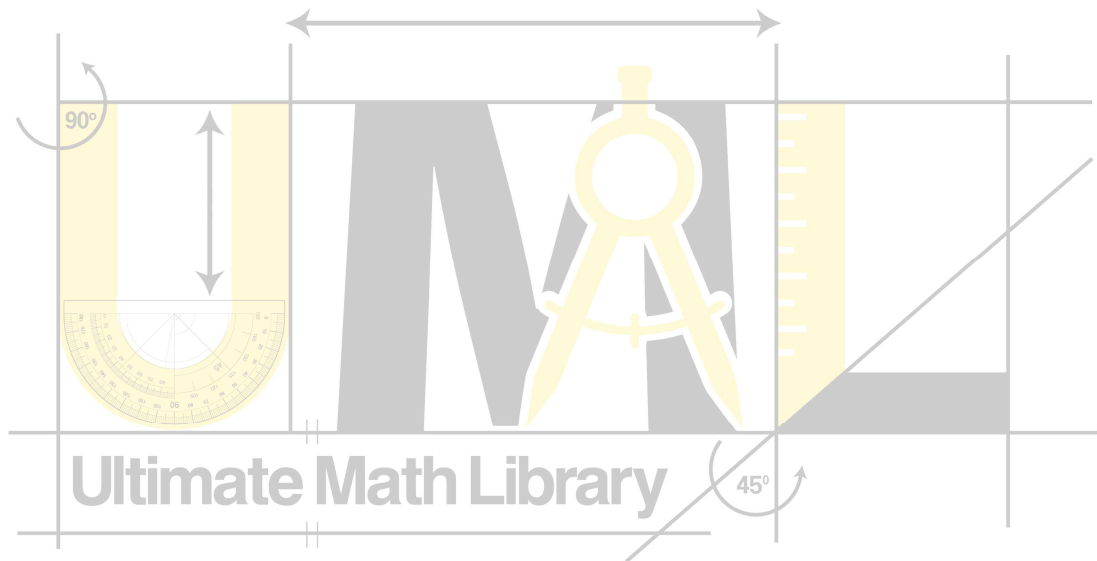
DOCUMENTATION

v1.1

1 TABLE OF CONTENTS

3	Quick Start Guide	3
4	General Math	4
4.1	The <i>UML</i> Class	4
4.2	Numeric Extensions	4
4.3	Polynomials	5
4.3.1	Roots and Discriminants	5
4.3.2	Bernstein Polynomials.....	6
4.4	Complex Numbers.....	7
4.4.1	Roots of Unity	7
5	Curves and Splines	7
5.1	Curve Inheritance Structure.....	8
5.2	Splines.....	8
5.3	Evaluating Curves.....	9
5.4	Types of Curves	9
5.4.1	Bezier Curves.....	9
5.4.2	Catmull-Rom Curves.....	10
5.4.3	Polynomial Curves.....	10
5.5	Derivatives and the Frenet-Serret Apparatus.....	10
5.6	Curve Explorer.....	10
5.6.1	Passing in a Curve.....	11
5.6.2	Curve Explorer Settings.....	11
6	Randomness.....	11
6.1	<i>UMLRandom</i>	12
6.2	Dice Roll	12
6.2.1	Dice Notation	12
6.2.2	Generating Random Numbers Using DiceRoll	12
6.3	Poisson Disk Sampling.....	13
7	Vectors	14
7.1	Swizzling.....	14
7.2	VectorN and VectorNInt	15
7.3	Vector Mapping	15

7.4	Miscellaneous Vector Functionality.....	16
8	Colors	16
8.1	Color Constants.....	16
8.2	Color Extensions.....	16
9	Gizmos.....	17



3 QUICK START GUIDE

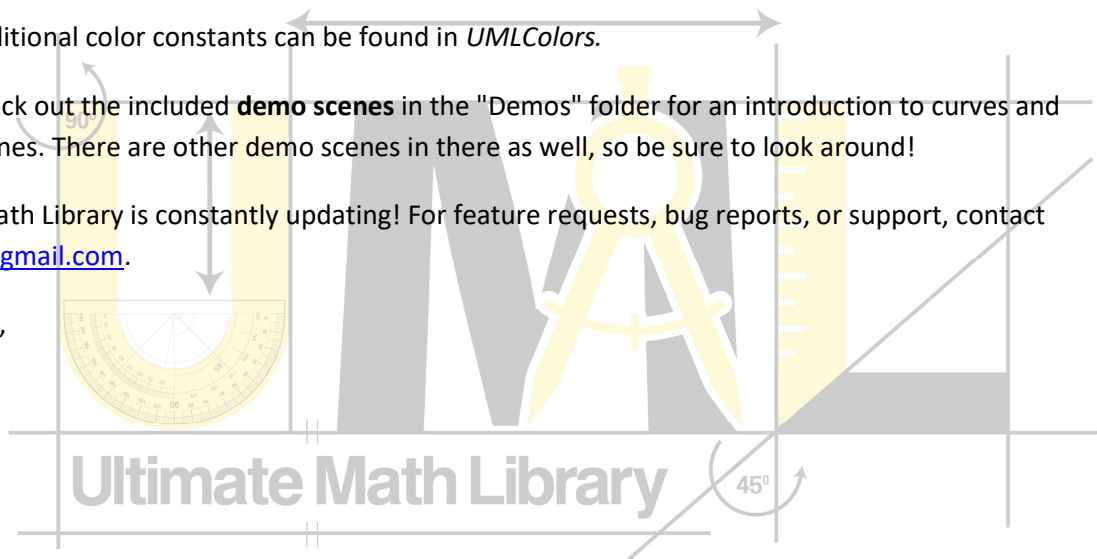
Thank you for purchasing **Ultimate Math Library**, the all-inclusive math library specially designed for Unity!

Here are some quick tips to get you started:

- All library scripts are contained directly within the namespace *Nickmiste.UltimateMathLibrary*.
- The class *UML* serves as a direct replacement for Unity's *Mathf*. All functionality is replicated exactly, and a ton of new functionality has been added!
- Also check out *UMLRandom* for a similar style replacement to Unity's *Random* class, and *UMLGizmos* for additional gizmos.
- Additional color constants can be found in *UMLColors*.
- Check out the included **demo scenes** in the "Demos" folder for an introduction to curves and splines. There are other demo scenes in there as well, so be sure to look around!

Ultimate Math Library is constantly updating! For feature requests, bug reports, or support, contact nickmiste@gmail.com.

Best wishes,
Nickmiste



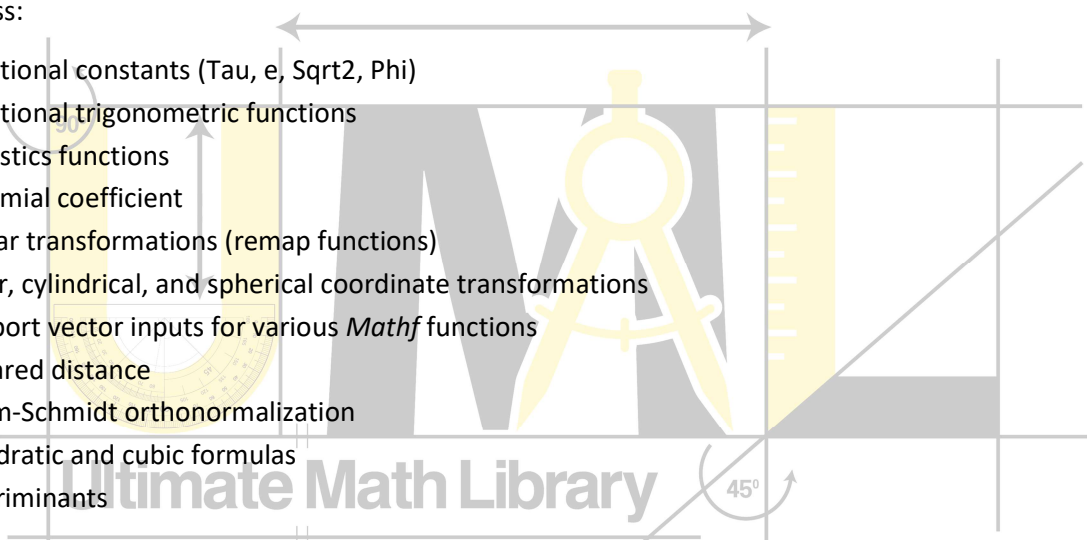
4 GENERAL MATH

This section contains an overview of the general math functionality included in UltimateMathLibrary.

4.1 THE *UML* CLASS

The *UML* class is at the core of UltimateMathLibrary. It serves as a direct replacement for Unity's *Mathf*. All functionality is replicated exactly, and a ton of new functionality has been added. Though there is nothing wrong with using *Mathf*, it is recommended to only use *UML* once installing UltimateMathLibrary to avoid having to search between two different classes for the function you need. If installing UltimateMathLibrary midway through development, there is no need to replace existing *Mathf* calls with *UML* calls.

In addition to all base *Mathf* functions, here are some of the other types of functions that you can find in the *UML* class:

- 
- Additional constants (Tau, e, Sqrt2, Phi)
 - Additional trigonometric functions
 - Statistics functions
 - Binomial coefficient
 - Linear transformations (remap functions)
 - Polar, cylindrical, and spherical coordinate transformations
 - Support vector inputs for various *Mathf* functions
 - Squared distance
 - Gram-Schmidt orthonormalization
 - Quadratic and cubic formulas
 - Discriminants

Note that this list is non-exhaustive. For a full list of functions, refer to your IDE's code completion.

4.2 NUMERIC EXTENSIONS

Extension methods to square and cube a number have been added to *float* and *int* types. Note that these extension methods are equivalent to *UML.Square()* and *UML.Cube()*.



```
float x = 5f;
Debug.Log(x.Squared()); // Output: 25
Debug.Log(x.Cubed()); // Output: 125
```

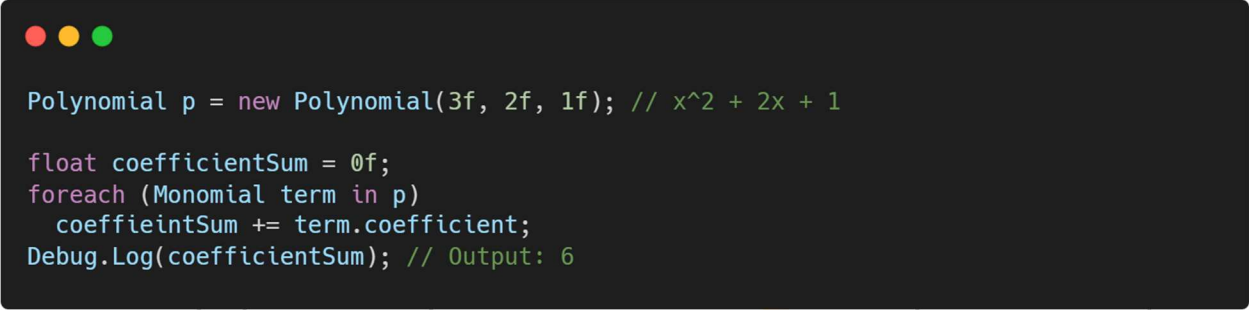
Figure 4.1 Example of squaring and cubing a floating-point type

There are a few other extension methods as well, most of which have equivalents in *UML*.

4.3 POLYNOMIALS

New in version 1.1, the *Polynomial* type can be used to perform all sorts of useful operations on polynomials.

To create a polynomial, call the constructor and supply the coefficients for each term in order of increasing degree. For example, to create the polynomial $x^2 + 2x + 3$, you would call `new Polynomial(3, 2, 1)`. Once you have a polynomial, you can iterate through each of its terms, which are of the type *Monomial*. A *Monomial* is a simple struct that stores a term's degree and coefficient.

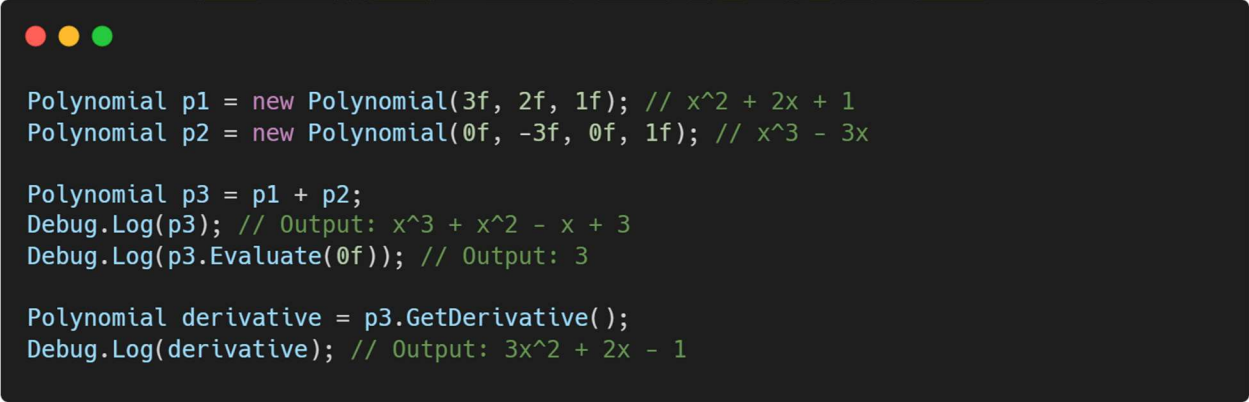


```
Polynomial p = new Polynomial(3f, 2f, 1f); // x^2 + 2x + 1

float coefficientSum = 0f;
foreach (Monomial term in p)
    coefficientSum += term.coefficient;
Debug.Log(coefficientSum); // Output: 6
```

Figure 4.2 Iterating through a polynomial's terms to find the sum of its coefficients

You can easily evaluate polynomials at an input, perform arithmetic, and take derivatives and integrals. These operations are also defined on monomials where they make sense.



```
Polynomial p1 = new Polynomial(3f, 2f, 1f); // x^2 + 2x + 1
Polynomial p2 = new Polynomial(0f, -3f, 0f, 1f); // x^3 - 3x

Polynomial p3 = p1 + p2;
Debug.Log(p3); // Output: x^3 + x^2 - x + 3
Debug.Log(p3.Evaluate(0f)); // Output: 3

Polynomial derivative = p3.GetDerivative();
Debug.Log(derivative); // Output: 3x^2 + 2x - 1
```

Figure 4.3 Performing basic polynomial operations

For help visualizing polynomials, see the section below on polynomial curves.

4.3.1 Roots and Discriminants

In many applications, it is useful to find the roots of a polynomial, or the input values where the polynomial evaluates to 0. UltimateMathLibrary supports finding the roots of polynomials up to degree 3.

```

Polynomial p = new Polynomial(-10f, 3f, 6f, 1f);
float[] roots = p.GetRoots();

foreach (float root in roots)
    Debug.Log(root);

// Output:
// 1
// -2
// -5

```

Figure 4.4 Computing the roots of a cubic polynomial

Sometimes you don't need to know the exact roots of a polynomial, but you still care about the *nature* of the roots. In these cases, it is useful to look at the polynomial's discriminant, which is cheaper than computing the roots. UltimateMathLibrary supports computing discriminants for polynomials up to degree 4.

```

Polynomial p = new Polynomial(-5f, 6f, 9f, -8f, 9f);
float discriminant = p.GetDiscriminant();

Debug.Log(discriminant); // Output: -49005648
// Using the discriminant, we can conclude the polynomial has 2 distinct real roots

```

Figure 4.5 Computing the discriminant of a quartic polynomial

The following chart shows the number of distinct real roots of a polynomial based on if the discriminant is positive, negative or 0:

	Positive	Negative	Zero
Degree 2	2	0	1
Degree 3	3	1	1 or 2
Degree 4	0 or 4	2	1, 2, or 3

Figure 4.6 A guide on interpreting discriminants

Finding the roots of higher degree polynomials can be tricky. For polynomials of degree 5 or higher, it is impossible to compute the exact roots using a formula. However, they can be approximated using a technique called Newton's method. Newton's method works by starting with an initial guess for where the root is and using the function's derivative to iteratively find closer and closer approximations. There are several shortcomings of this technique, but it is still useful in many situations. To use Newton's method, call the method *NewtonsMethod()* on a polynomial with the relevant arguments.

4.3.2 Bernstein Polynomials

Bernstein basis polynomials are used extensively in computer graphics, especially in the computation of Bezier curves. You can compute them using the static method *Bernstein()* in the *Polynomial* class. You

can also bundle all $n+1$ Bernstein basis polynomials of degree n in an array using the static method `BernsteinBasis()`.

```
//Compute the 0th Bernsitein basis polynomial of degree 2.
Polynomial b02 = Polynomial.Bernstein(0, 2); // Result: x^2 - 2x + 1

//Compute all Bernstein basis polynomials of degree 2.
Polynomial[] basis = Polynomial.BernsteinBasis(2);
// Result:
// 0: x^2 - 2x + 1
// 1: -2x^2 + 2x
// 2: x^2
```

Figure 4.7 Computing Bernstein basis polynomials

4.4 COMPLEX NUMBERS

A complex number is a number of the form $a + bi$, where i is called the *imaginary unity*. It has the special property that $i^2 = -1$. UltimateMathLibrary adds the *Complex* type, which allows creating and interacting with complex numbers. Complex numbers have many applications and are also used internally throughout UltimateMathLibrary.

4.4.1 Roots of Unity

An *n*th root of unity is a complex number z such that $z^n = 1$ for a positive integer n . There are exactly n nth roots of unity for all n , and all roots of unity have modulus (magnitude) 1. Roots of unity have a lot of applications when dealing with complex numbers, and UltimateMathLibrary makes it easy to compute them with the static method `Complex.RootOfUnity()`.

5 CURVES AND SPLINES

Curves and splines are a major feature of UltimateMathLibrary. Currently, the main types of curves that are supported are Bezier curves and Catmull-Rom curves, though more types of curves are planned for the future.

5.1 CURVE INHERITANCE STRUCTURE

To get the most out of curves in UltimateMathLibrary, it is important to understand their basic inheritance structure.

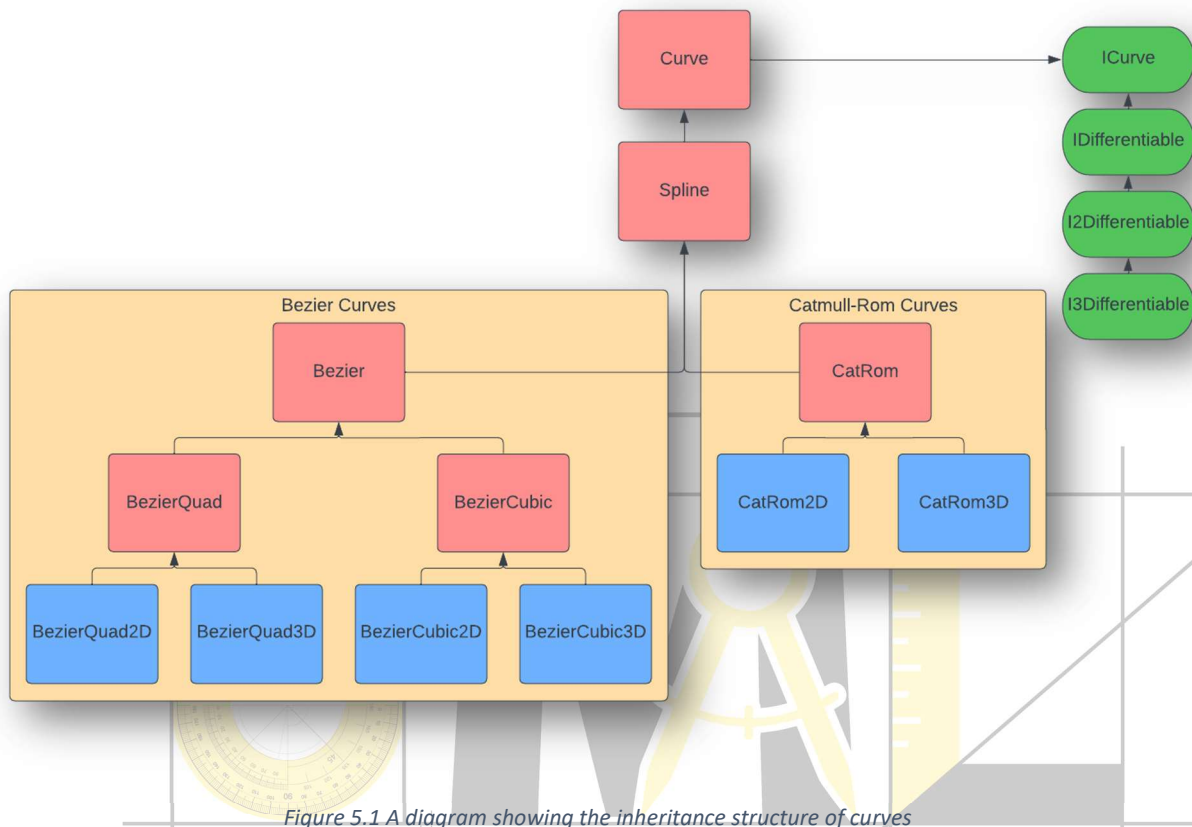


Figure 5.1 A diagram showing the inheritance structure of curves

In the diagram above, red nodes represent abstract classes and green nodes represent interfaces. The actual classes that you will be directly instantiating and interacting with are marked in blue. Not all curve types are shown, but the basic structure remains the same.

Note that many curves also inherit from the *IDifferentiable*, *I2Differentiable*, and *I3Differentiable* interfaces, but these lines have been omitted in the above diagram to avoid clutter. For more information on differentiable curves, see the section on derivatives below.

5.2 SPLINES

Splines are a type of curve that use control points to define their behavior. Most curves that are used in game development (such as Bezier curves and Catmull-Rom curves) are types of splines. In UltimateMathLibrary, all splines inherit from *Spline*, which itself inherits from *Curve*.

A spline's control points can be accessed via indexer, or iterated through by using the spline's enumerator. There are also methods to add and remove control points.

```

Spline spline = new CatRom3D(controlPoints);

//Access a control point via indexer
spline[0] = new Vector3(2f, 3f, 1f);

//Iterate through each control point
foreach (Vector3 controlPoint in spline)
    DoSomething(controlPoint);

```

Figure 5.2 Working with a spline's control points

5.3 EVALUATING CURVES

Curves are evaluated using a *t*-value that represents the progress along the curve. To get the point along a curve at a given *t*-value *t*, use the method *Curve.Evaluate(t)*. *T*-values generally range from 0 to 1, though many curves use the integer part of the *t*-value to denote which segment of the curve to evaluate at. For example, a *t*-value of 2.6 evaluates to the point 60% of the way along the third segment of the curve.

5.4 TYPES OF CURVES

This section details the different types of curves that are included in UltimateMathLibrary. Each type of curve has a 2D and a 3D variant. For example, for a *CatRom* is the abstract class from which you would use either *CatRom2D* or *CatRom3D*.

This documentation is not meant to be an introduction to the behavior of these curves, but rather to show how to use them within UltimateMathLibrary.

5.4.1 Bezier Curves

A Bezier curve is a type of spline that gives you a lot of control based on the position of its control points. The curve doesn't go through all its control points, but every point has influence over the curve.

There are two main types of Bezier curves – quadratic and cubic. If you are unsure which type of Bezier curve you want, you are probably looking for cubic Bezier curves, as those are the more commonly used variant. The base classes for each variant are *BezierQuad* and *BezierCubic*, respectively.

Starting with version 1.1, UML also supports generalized *n*-degree Bezier curves. The base class for general Bezier curves is *BezierGeneral*. The constructor for general Bezier curves takes a list of control points, as well as the curve's degree (which can be changed later with *SetDegree()*). Each segment of an *n*-degree Bezier curve consists of *n*+1 control points. The curve goes through the first and last control points in each segment, and the other points are used to influence the behavior of the curve. Note that if the degree is set to either 2 or 3, you should use *BezierQuad* or *BezierCubic* instead as those are equivalent with some extra performance optimizations.

5.4.2 Catmull-Rom Curves

A Catmull-Rom curve is a type of spline with the useful property that it goes through all its control points.

In addition to the control points, you can also change a Catmull-Rom curve's tension and alpha value. The typical range for both of these values is 0 to 1, but any value can be passed in. Its alpha value can also be passed in via the *CatRomType* enum. *CatRomType.Uniform* corresponds to an alpha value of 0, *CatRomType.Centripital* corresponds to an alpha value of 0.5, and *CatRomType.Chordal* corresponds to an alpha value of 1.

5.4.3 Polynomial Curves

Polynomial curves allow you to treat a polynomial as a curve. Unlike most other curves in UML, a polynomial curve is not a spline. To use it, specify a polynomial and optionally an interval to define the curve on. Polynomial curves are especially useful when paired with the curve explorer to visualize polynomials, their derivatives, and other properties.

5.5 DERIVATIVES AND THE FRENET-SERRET APPARATUS

Differentiable curves inherit from *IDifferentiable*. If a curve can be differentiated more than once, it will also inherit from *I2Differentiable* or *I3Differentiable*. This enables you to call *GetDerivative()*, *GetSecondDerivative()*, and/or *GetThirdDerivative()* on the curve.

Differentiable curves also contain support for the Frenet-Serret apparatus. You can call the extension method *GetTangent()* on all differentiable curves, *GetNormal()* on all 2-differentiable curves, and *GetBinormal()* on all 2-differentiable curves in 3D. If a curve is 2-differentiable, you can call *GetFrenetFrame()* on it to bundle the point of application, tangent, normal, (and binormal for 3D curves) vectors into one object. The curve explorer (see below) contains an easy way to visualize Frenet frames on all compatible curves.

To complete the Frenet-Serret apparatus, you can also call *GetCurvature()* on all 2-differentiable curves and *GetTorsion()* on all 3-differentiable curves in 3D to get the curvature and torsion.

5.6 CURVE EXPLORER

The curve explorer provides a way to intuitively visualize a curve and explore its properties. To use it, add a *CurveExplorer2D* or *CurveExplorer3D* component to an empty game object. You can also check out the curve explorer **demo scene** included in the "Demos" folder.

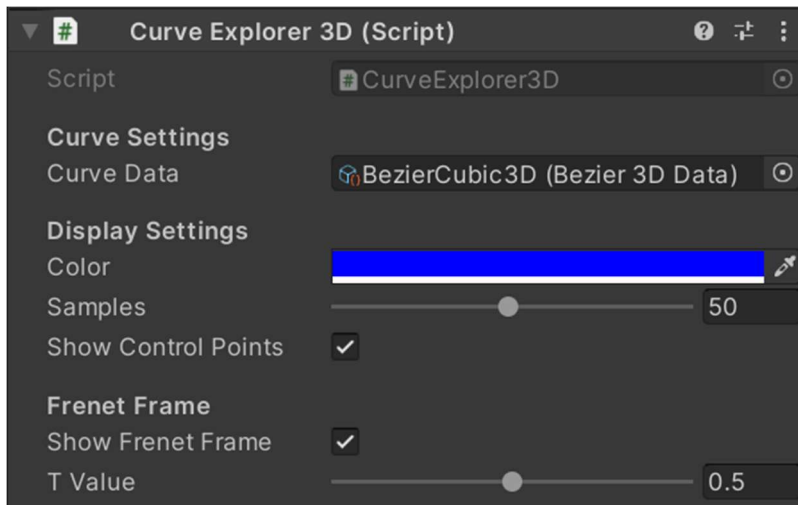


Figure 5.3 The curve explorer interface

5.6.1 Passing in a Curve

To pass a curve into the curve explorer, you must use the *CurveData* scriptable object. To do this, go to *Assets > Create > UltimateMathLibrary > CurveData* and select the type of curve that you want. Note that some *CurveData* types correspond to multiple *Curve* types. For example, *Bezier3DData* is used to visualize both *BezierQuad3D* and *BezierCubic3D* curves.

Once you've passed a curve in, you should see the curve appear in your scene. If you don't see anything, make sure that Gizmos are enabled and that the curve you're passing in is valid. Certain curves, such as Bezier curves and Catmull-Rom curves, require a minimum number of points to be valid, so make sure that you've passed in enough points.

If you aren't sure where to start or which types of curves to create, check out the **example curves** in the "Demos/ExampleCurveData" folder. These are a handful of built-in curve data examples to help you get started.

5.6.2 Curve Explorer Settings

Try playing with the various settings to learn more about the behavior of your curve. The curve explorer updates in real time, so you can also adjust the curve settings to see how they affect the curve.

Showing the Frenet frame can be enlightening when working with differentiable curves. With this setting enabled, you can use the t-value slider to show the Frenet frame at specific points along the curve. Note that this t-value is normalized – a value of 0 represents the point at the start of the curve and a value of 1 represents the point at the end of the curve.

You can make also the curve bigger by increasing the scale of the game object that the *CurveExplorer* component is attached to. The position and rotation of the game object will also affect the transform of the drawn curve.

6 RANDOMNESS

This section contains an overview of the randomness functionality included in UltimateMathLibrary.

6.1 UMLRANDOM

Similar to how *UML* is a direct replacement to Unity's *Mathf*, the *UMLRandom* class is a direct replacement to Unity's *Random* class. All functionality is replicated exactly, and a ton of new functionality has been added. Though there is nothing wrong with using *Random*, it is recommended to only use *UMLRandom* once installing *UltimateMathLibrary* to avoid having to search between two different classes for the function you need. If installing *UltimateMathLibrary* midway through development, there is no need to replace existing *Random* calls with *UMLRandom* calls.

6.2 DICE ROLL

The *DiceRoll* class is a convenient way to generate random numbers that should be familiar to players of tabletop roleplaying games.

6.2.1 Dice Notation

You can instantiate a new *DiceRoll* using dice notation. Dice notation generally takes the form "*NdF*," where *N* and *F* are variables separated by the letter *d*, which stands for *dice*.

- *N* represents the number of dice to be rolled. If *N* is 1, it can optionally be omitted.
- *F* represents the number of faces on each die.

An optional additive modifier can be appended using the form "*NdF+A*," where *A* is the amount to add to the total each time the dice are rolled.

Dice notation is also used by the *DiceRoll.ToString()* method.

You can also instantiate a *DiceRoll* with *N*, *F*, and *A* directly using the constructor *DiceRoll(N, F A)*.

6.2.2 Generating Random Numbers Using DiceRoll

Once you've instantiated a dice roll, using it to generate random numbers is simple. You can call the method *DiceRoll.Roll()* to simulate rolling the dice. This method will return the sum of each die plus the additive modifier, if present. If you require the result of each die, you can call the overload *DiceRoll.Roll(out int[] results)*. The return value will be the same, but the value of each die will be stored in the results array.

```

DiceRoll diceRoll = new DiceRoll("2d6+11"); //equivalent to new DiceRoll(2, 6, 11);
int total = diceRoll.Roll(out int[] results);
Debug.Log($"Rolled a {results[0]} and a {results[1]}. Final result is {total}.");

// Example output:
// Rolled a 2 and a 6. Final result is 19.
```

Figure 6.1 *DiceRoll* Usage Example

6.3 POISSON DISK SAMPLING

Generating random points in a region uniformly can create undesired results. Poisson disk sampling is an algorithm to efficiently generate random points in a region while ensuring a minimum distance between each point. The below image shows the difference between uniform sampling and Poisson disk sampling.

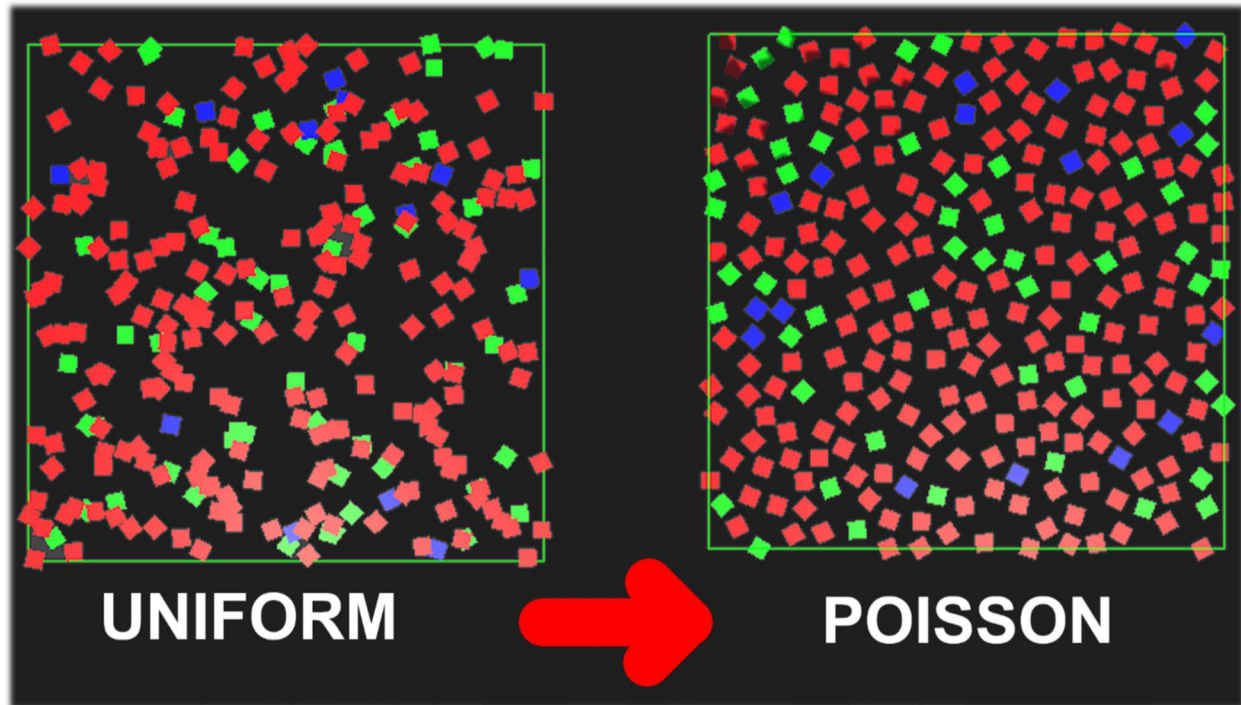


Figure 6.2 An example showing the difference between uniform sampling and Poisson disk sampling

This can be used to procedurally generate trees or other foliage on terrain, place hair on a character model, generate stars in the sky, among other things.

To use Poisson disk sampling over a circular region, specify the center of the sampling region, the radius of the circle, and the minimum distance to guarantee between points.

```
PoissonDiskSampler.Sample(Vector2.zero, 10f, 1f);
```

Figure 6.3 Sample points over a circle of radius 10, guaranteeing a minimum distance of 1 between points

To use Poisson disk sampling over a rectangular region, pass in a *Vector2* representing the rectangular bounds instead of the radius.



```
PoissonDiskSampler.Sample(Vector2.zero, new Vector2(10f, 10f), 1f);
```

Figure 6.4 Sample points over a square of radius 10, guaranteeing a minimum distance of 1 between points

You can also use Poisson disk sampling over a custom region by supplying a method that returns true if a given point is contained within the sample region (and false otherwise).



```
PoissonDiskSampler.Sample(Vector2.zero,
    v => Vector2.Distance(v, new Vector2( 10f, 0f) <= 20f) ||
    Vector2.Distance(v, new Vector2(-10f, 0f) <= 20f,
    1f);
```

Figure 6.5 Example of using Poisson disk sampling over a custom region - the union of two circles

For an example of Poisson disk sampling in action, including a detailed explanation of how to sample over a custom region, check out the Poisson disk sampling **demo scene** included in the “Demos” folder.

Note that Poisson disk sampling is currently only supported in 2D. Support for sampling in 3D is planned for a later update.

7 VECTORS

This section contains an overview of the vector functionality included in UltimateMathLibrary.

7.1 SWIZZLING

Swizzling is a form of syntactic sugar that should be familiar to those who have worked with shader languages such as HLSL. UltimateMathLibrary introduces a swizzle-like syntax within C#. Though C# does not support swizzling, a similar effect is achieved using extension methods. The supported types are *Vector2*, *Vector3*, *Vector2Int*, and *Vector3Int*.

```

Vector2 v = new Vector2(0f, 1f);
Vector3 w = new Vector3(1f, 2f, 3f);

//Swizzle Vector2
Debug.Log(v.XX()); // (0, 0)
Debug.Log(v.YX()); // (1, 0)

//Swizzle Vector3
Debug.Log(w.ZYY()); // (3, 2, 2)
Debug.Log(w.XZY()); // (1, 3, 2)

//Swizzle between Vector2 and Vector3
Debug.Log(v.XXX()); // (0, 0, 0)
Debug.Log(w.XZ()); // (1, 3);

```

Figure 7.1 Swizzling Usage Example

7.2 VECTORN AND VECTORNINT

UltimateMathLibrary introduces the types *VectorN* and *VectorNInt* to represent generalized n-dimensional vectors. Components can be accessed via property indexer. Note that the dimension cannot be changed once instantiated.

While support is currently limited, all basic vector operations such as addition, multiplication by constant, and dot product are supported. Support for more operations is planned for future updates.

Many vector operations (such as the dot product) are only defined on vectors of the same dimension. When attempting to perform these operations on vectors of different dimensions, a *DimensionMismatchException* is thrown.

Implicit type conversions to and from *Vector2*, *Vector3*, and *Vector4* exist (likewise for their integer vector equivalents). Note that when converting a *VectorN* of a lower dimension to a higher dimension, the remaining components are set to 0.

7.3 VECTOR MAPPING

Though primarily intended for internal use, the *VectorMapping* class can be used as a convenient way to perform component-wise operations on vectors. All vector types are supported (including *VectorN* and

VectorNInt), and each has two associated mapping functions – one for a unary operation and another for a binary operation. See the usage example below.

```

Vector3 v = new Vector3(5f, -2f, 0.5f);
Vector3 w = new Vector3(3f, -6f, -12f);

//Unary example
Vector3 clamped = VectorMapping.Map(v, x => UML.ClampNeg1To1(x));

//Binary example
Vector3 min = VectorMapping.Map(v, w, (a, b) => UML.Min(a, b));

Debug.Log(clamped); // Output: (1, -1, 0.5)
Debug.Log(min);    // Output: (3, -6, -12)

```

Figure 7.2 Vector Mapping Usage Example

7.4 MISCELLANEOUS VECTOR FUNCTIONALITY

In addition to the above features, the following vector functionality is included in UltimateMathLibrary:

- Conversion between Cartesian, polar, spherical, and cylindrical coordinates (included in the *UML* class and as *Transform* extension methods).
- Implicit conversion between *Vector2* and *Complex*.
- Various vector equivalents to methods only defined on numeric types in *Mathf* (such as *Round()*) have been added to the *UML* class.

8 COLORS

Though not directly related to math, UltimateMathLibrary includes some functionality relating to colors with the main goal being to increase convenience and quality of life.

8.1 COLOR CONSTANTS

The *UMLColors* class contains nearly 150 color constants beyond what Unity includes by default. Note that all colors have an alpha value of 1 except for *UMLColors.transparent*, which is white with an alpha value of 0. This is different from Unity's *Color.clear*, which is black with an alpha value of 0.

8.2 COLOR EXTENSIONS

UltimateMathLibrary includes a few basic color extension methods for getting and setting a color's hue, saturation, and value, multiplying colors, and returning a new color with a modifier alpha value.

9 GIZMOS

The class *UMLGizmos* contains various additional gizmos not included in Unity. It functions similarly to Unity's *Gizmos* class and can be used anywhere that gizmos are supported. Unlike the *UML* and *UMLRandom* classes, *UMLGizmos* does not fully replace Unity's *Gizmos* class and you will need to use both classes to find the gizmos that you want to use. Here are a few examples of the types of gizmos included in *UMLGizmos*:

- Dotted lines
- Circles
- Triangles
- Curves and splines
- Frenet frames
- Polynomials

