Math Library for Unity

Oleg E. Arutyunyan
a.k.a. “devast”

E-mail: oleg.arutyunyan@gmail.com
Forum link: http://forum.unity3d.com/threads/228681-RELEASED-Math-Library-for-Unity
Last updated: 2014.09.15
Library version: 1.3.3

Follow me on Twitter.

The library is created and tested in Unity 4.3.
The documentation is created with Apache Open Office 4.

1 Table of Contents

2 Introduction....8
2.1 General Information....8
2.2 Special Thanks....8
3 Primitive Objects....10
3.1 2D Objects....10
3.1.1 Line2....10
3.1.2 Ray2....13
3.1.3 Segment2....15
3.1.4 AAB2 (Axis-Aligned Box)....17
3.1.5 Box2....20
3.1.6 Circle2....23
3.1.7 Triangle2....26
3.1.8 Polygon2....30
3.2 3D Objects....33
3.2.1 Line3....33
3.2.2 Ray3....35
3.2.3 Segment3....37
3.2.4 Plane3....39
3.2.5 Rectangle3....43
3.2.6 AAB3 (Axis-Aligned Box)....45
3.2.7 Box3....48
3.2.8 Circle3....51
3.2.9 Sphere3....53
3.2.10 Triangle3....55
3.2.11 Polygon3....58
3.2.12 Capsule3....61
4 Intersection....62
4.1 2D Intersection....64
4.1.1 Line2-Line2....64
4.1.2 Line2-Ray2....65
4.1.3 Line2-Segment2....66
4.1.4 Ray2-Ray2....67
4.1.5 Ray2-Segment2....68
4.1.6 Segment2-Segment2....69
4.1.7 Line2-AAB2....70
4.1.8 Ray2-AAB2....71
4.1.9 Segment2-AAB2....72
4.1.10 Line2-Box2....73
4.1.11 Ray2-Box2....74
4.1.12 Segment2-Box2....75
4.1.13 Line2-Circle2....76

4.1.14 Ray2-Circle2....77
4.1.15 Segment2-Circle2....78
4.1.16 Line2-Triangle2....79
4.1.17 Ray2-Triangle2....80
4.1.18 Segment2-Triangle2....81
4.1.19 Line2-ConvexPolygon2....82
4.1.20 Ray2-ConvexPolygon2....83
4.1.21 Segment2-ConvexPolygon2....84
4.1.22 Ray2-Polygon2....85
4.1.23 AAB2-AAB2....87
4.1.24 AAB2-Circle2....88
4.1.25 Box2-Box2....89
4.1.26 Box2-Circle2....90
4.1.27 Circle2-Circle2....91
4.1.28 Triangle2-Triangle2....92
4.1.29 ConvexPolygon2-ConvexPolygon2....94
4.2 3D Intersection....95
4.2.1 Line3-Plane3....95
4.2.2 Ray3-Plane3....96
4.2.3 Segment3-Plane3....97
4.2.4 Line3-Rectangle3....98
4.2.5 Ray3-Rectangle3....99
4.2.6 Segment3-Rectangle3....100
4.2.7 Line3-AAB3....101
4.2.8 Ray3-AAB3....102
4.2.9 Segment3-AAB3....103
4.2.10 Line3-Box3....104
4.2.11 Ray3-Box3....105
4.2.12 Segment3-Box3....106
4.2.13 Line3-Circle3....107
4.2.14 Ray3-Circle3....108
4.2.15 Segment3-Circle3....109
4.2.16 Line3-Sphere3....110
4.2.17 Ray3-Sphere3....111
4.2.18 Segment3-Sphere3....112
4.2.19 Line3-Triangle3....113
4.2.20 Ray3-Triangle3....115
4.2.21 Segment3-Triangle3....117
4.2.22 Line3-Polygon3....119
4.2.23 Ray3-Polygon3....120
4.2.24 Segment3-Polygon3....121
4.2.25 Plane3-AAB3....122
4.2.26 Plane3-Box3....123
4.2.27 Plane3-Sphere3....124
4.2.28 Plane3-Plane3....125

4.2.29 Plane3-Triangle3....126
4.2.30 AAB3-AAB3....127
4.2.31 AAB3-Sphere3....128
4.2.32 Box3-Box3....129
4.2.33 Box3-Sphere3....130
4.2.34 Box3-Capsule3....131
4.2.35 Sphere3-Sphere3....132
4.2.36 Triangle3-Triangle3....134
5 Distance and Projection....136
5.1 2D Distance and Projection....137
5.1.1 Point2-Line2....137
5.1.2 Point2-Ray2....138
5.1.3 Point2-Segment2....139
5.1.4 Line2-Line2....140
5.1.5 Line2-Ray2....141
5.1.6 Line2-Segment2....142
5.1.7 Ray2-Ray2....143
5.1.8 Ray2-Segment2....144
5.1.9 Segment2-Segment2....145
5.1.10 Point2-AAB2....146
5.1.11 Point2-Box2....147
5.1.12 Point2-Circle2....148
5.1.13 Point2-Triangle2....149
5.2 3D Distance and Projection....150
5.2.1 Point3-Line3....150
5.2.2 Point3-Ray3....151
5.2.3 Point3-Segment3....152
5.2.4 Line3-Line3....153
5.2.5 Line3-Ray3....154
5.2.6 Line3-Segment3....155
5.2.7 Line3-Box3....156
5.2.8 Ray3-Ray3....157
5.2.9 Ray3-Segment3....158
5.2.10 Segment3-Segment3....159
5.2.11 Segment3-Box3....160
5.2.12 Point3-Plane3....161
5.2.13 Point3-Rectangle3....162
5.2.14 Point3-AAB3....163
5.2.15 Point3-Box3....164
5.2.16 Point3-Circle3....165
5.2.17 Point3-Sphere3....167
6 Point Containment and Bounding Objects....168
6.1 2D Point Containment....169
6.1.1 Point2 in AAB2....169

6.1.2 Point2 in Box2....170
6.1.3 Point2 in Circle2....170
6.1.4 Point2 in Convex Polygon2....171
6.1.5 Point2 in Polygon2....172
6.1.6 Point2 in Triangle2....173
6.2 Bounding Areas....174
6.2.1 Constructing AAB2....174
6.2.2 Constructing Box2....175
6.2.3 Constructing Circle2....176
6.3 3D Point Containment....178
6.3.1 Point3 in AAB3....178
6.3.2 Point3 in Box3....179
6.3.3 Point3 in Sphere3....179
6.4 Bounding Volumes....180
6.4.1 Constructing AAB3....180
6.4.2 Constructing Box3....181
6.4.3 Constructing Sphere3....182
7 Side Testing....184
7.1 Line2 Tests....184
7.2 Triangle2 Tests....186
7.3 Plane3 Tests....187
8 Geometric Algorithms....189
8.1 Convex hull....189
9 Approximation....191
9.1 2D Approximation....191
9.1.1 LineFit2....191
9.1.2 GaussPointsFit2....191
9.2 3D Approximation....192
9.2.1 LineFit3....192
9.2.2 PlaneFit3....192
9.2.3 GaussPointsFit3....192
10 Numerical Methods....193
10.1 Solving Linear Systems....193
10.2 Eigen Decomposition....194
10.3 Finding Polynomial Roots....195
10.4 Solving ODEs....199
10.5 Integrating....200
11 Curves....201
11.1 Catmull-Rom Spline....201
11.2 Catmull-Rom Spline Editor Extension....206
11.3 Moving Along Spline at Constant Speed....207

11.4 Cubic Spline (natural and closed)....208
12 Misc....209
12.1 Vector2ex....209
12.2 Vector3ex....212
12.3 Quaternionex....216
12.4 Matrix4x4ex....217
12.5 Mathfex....222
12.6 Pseudo Random Numbers Generator....227
12.7 Random Samplers....232
12.7.1 Weighted Sampler....232
12.7.2 Indexed and Non-Indexed Triangle Set Sampler....232
12.7.3 Poisson Disk Sampler....234
12.7.4 Point Set Filtering....235
12.8 ShuffleBag....237
12.9 Util....237
12.10 Logger....238
13 Tutorials....239
13.1 Working with Catmull-Rom Splines....239

2 Introduction

2.1 General Information

Welcome to the “Math Library for Unity” documentation. It will help you to use the library, give examples of usage, describe the types inside the library and so on.

Chapter 2 will describe primitives and their methods. Other types and other methods situated inside of various static classes are described in subsequent chapters and grouped by usage (subsequent chapters also describe some methods inside of the primitives in more details than in Primitives chapter).

All objects are situated inside of Dest.Math namespace, so do not forget to put `using Dest.Math;` in the top of your script files. Everything except spline related types is compiled into dll. Spline class and corresponding editor class are given as source files.

Notice that all primitives (except polygons) are C# structs. By default structs have parameterless constructor which assign default values to the members of a struct. Refer to the primitives documentation to see what can be expected from calling default constructors (it is advisable to use constructors with parameters).

The library contains numerous test prefabs serving two purposes. First is to test the correctness of the methods. Second is to show users of the library examples of the usage. Users are encouraged to use test prefabs and see their code for learning purposes. All of the test prefabs are situated inside Tests folder of the package. Structure of the folder reflects chapter structure of this documentation file. Most of the prefabs are designed to work in editor mode (i.e. has `ExecuteInEditMode` attribute). Users can just drag-and-drop a prefab to the scene and modify it right away (usually it involves changing shape of the primitives). However some of the prefabs require “Play button” to be pushed on. These prefabs often contain “Press Play To Launch” label inside the Inspector panel. In any case opening script source of the test prefab may tell many useful things. For example, almost all test prefabs are inherited from Test_Base class which contains examples of creating all primitives.

Users who believe that they found a bug are encouraged to test the data on the test prefabs first and ensure bug existance. If the bug is found, please submit it to the forum or on the email.

The library uses only `Vector2`, `Vector3`, `Matrix4x4`, `Mathf`, `Debug` types from the Unity API. However spline editor class uses latest Undo functionality from Unity 4.3. If you have previous version of the Unity, just remove spline class, the rest of the library should work fine. Test prefabs also use Gizmos to draw primitives.

As the final note, authors want to mention that the library is optimized whenever possible taking specific of the C# language into consideration. For example, many methods have `ref` or `out` parameters thus minimizing amount of data to be copied. Also garbage generation has been minimized as well. Vast majority of the intersection and distance tests generate no garbage at all working only with data on the stack.

2.2 Special Thanks

Authors want to thank David H. Eberly, his wonderful books “Geometric Tools for

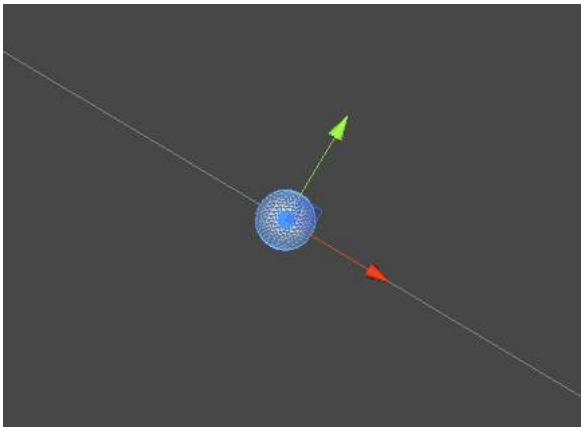
Computer Graphic", "3D Game Engine Design" and his website "Geometric Tools" (http://www.geometrictools.com), containing countless algorithms and papers without which this library could not have been created.

3 Primitive Objects

3.1 2D Objects

This section describes primitive objects in 2D. Library includes line, ray, segment, AAB, box, circle, triangle and polygon primitives in 2D. Each subsection lists available for the user methods.

3.1.1 Line2



A line is represented as P+t*D where P is the line origin, D is a unit-length direction vector, and t is any real number. The user must ensure that D is indeed unit length.

Type
`struct Line2`
Fields
`Center`
`Direction`
Construction
`Line2()`
`Line2(ref Vector2 center, ref Vector2 direction)`
`Line2(Vector2 center, Vector2 direction)`
`CreateFromTwoPoints(ref Vector2 p0, ref Vector2 p1)`
`CreateFromTwoPoints(Vector2 p0, Vector2 p1)`
`CreatePerpToLineTrhoughPoint(Line2 line, Vector2 point)`
`CreateBetweenAndEquidistantToPoints(Vector2 point0, Vector2 point1)`
`CreateParallelToGivenLineAtGivenDistance(Line2 line, float distance)`
Methods
`Eval(float t)`
`SignedDistanceTo(Vector2 point)`
`DistanceTo(Vector2 point)`
`QuerySide(Vector2 point, float epsilon)`
`QuerySideNegative(Vector2 point, float epsilon)`
`QuerySidePositive(Vector2 point, float epsilon)`

`QuerySide(ref Box2 box, float epsilon)`
`QuerySideNegative(ref Box2 box, float epsilon)`
`QuerySidePositive(ref Box2 box, float epsilon)`
`QuerySide(ref AAB2 box, float epsilon)`
`QuerySideNegative(ref AAB2 box, float epsilon)`
`QuerySidePositive(ref AAB2 box, float epsilon)`
`QuerySide(ref Circle2 circle, float epsilon)`
`QuerySideNegative(ref Circle2 circle, float epsilon)`
`QuerySidePositive(ref Circle2 circle, float epsilon)`
`Project(Vector2 point)`
`AngleBetweenTwoLines(Line2 anotherLine, bool acuteAngleDesired)`
`ToString()`

`Vector2 Center`
Line origin

`Vector2 Direction`
Line direction. Must be unit length!

`Line2()`
Creates the line. Users should initialize center and direction manually.

`Line2(ref Vector2 center, ref Vector2 direction)`
Creates the line.
center - Line origin.
direction - Line direction. Must be unit length!

`Line2(Vector2 center, Vector2 direction)`
Creates the line.
center - Line origin.
direction - Line direction. Must be unit length!

`static Line2 CreateFromTwoPoints(ref Vector2 p0, ref Vector2 p1)`
Creates the line. Origin is p0, Direction is Normalized(p1-p0).
p0 - First point.
p1 - Second point.

`static Line2 CreateFromTwoPoints(Vector2 p0, Vector2 p1)`
Creates the line. Origin is p0, Direction is Normalized(p1-p0).
p0 - First point
p1 - Second point

`static Line2 CreatePerpToLineTrhoughPoint(Line2 line, Vector2 point)`
Creates the line which is perpendicular to given line and goes through given point.

`static Line2 CreateBetweenAndEquidistantToPoints(Vector2 point0, Vector2 point1)`
Creates the line which is perpendicular to segment [point0,point1] and line origin goes through middle of the segment.

`static Line2 CreateParallelToGivenLineAtGivenDistance(Line2 line, float distance)`
Creates the line which is parallel to given line on the specified distance from given line.

`Vector2 Eval(float t)`
Evaluates line using P+t*D formula, where P is the line origin, D is a unit-length direction vector, t is parameter.
t - Evaluation parameter

`float SignedDistanceTo(Vector2 point)`
Returns signed distance to a point. Where positive distance is on the right of the line, zero is on the line, negative on the left side of the line.

`float DistanceTo(Vector2 point)`
Returns distance to a point, distance is >= 0f.

`int QuerySide(Vector2 point, float epsilon = MathfEx.ZeroTolerance)`
Determines on which side of the line a point is. Returns +1 if a point is to the right of the line, 0 if it's on the line, -1 if it's on the left.

`bool QuerySideNegative(Vector2 point, float epsilon = MathfEx.ZeroTolerance)`
Returns true if a point is on the negative side of the line, false otherwise.

`bool QuerySidePositive(Vector2 point, float epsilon = MathfEx.ZeroTolerance)`
Returns true if a point is on the positive side of the line, false otherwise.

`int QuerySide(ref Box2 box, float epsilon = MathfEx.ZeroTolerance)`
Determines on which side of the line a box is. Returns +1 if a box is to the right of the line, 0 if it's intersecting the line, -1 if it's on the left.

`bool QuerySideNegative(ref Box2 box, float epsilon = MathfEx.ZeroTolerance)`
Returns true if a box is on the negative side of the line, false otherwise.

`bool QuerySidePositive(ref Box2 box, float epsilon = MathfEx.ZeroTolerance)`
Returns true if a box is on the positive side of the line, false otherwise.

`int QuerySide(ref AAB2 box, float epsilon = MathfEx.ZeroTolerance)`
Determines on which side of the line a box is. Returns +1 if a box is to the right of the line, 0 if it's intersecting the line, -1 if it's on the left.

`bool QuerySideNegative(ref AAB2 box, float epsilon = MathfEx.ZeroTolerance)`
Returns true if a box is on the negative side of the line, false otherwise.

`bool QuerySidePositive(ref AAB2 box, float epsilon = MathfEx.ZeroTolerance)`
Returns true if a box is on the positive side of the line, false otherwise.

`int QuerySide(ref Circle2 circle, float epsilon = MathfEx.ZeroTolerance)`
Determines on which side of the line a circle is. Returns +1 if a circle is to the right of the line, 0 if it's intersecting the line, -1 if it's on the left.

`bool QuerySideNegative(ref Circle2 circle, float epsilon = MathfEx.ZeroTolerance)`
Returns true if a circle is on the negative side of the line, false otherwise.

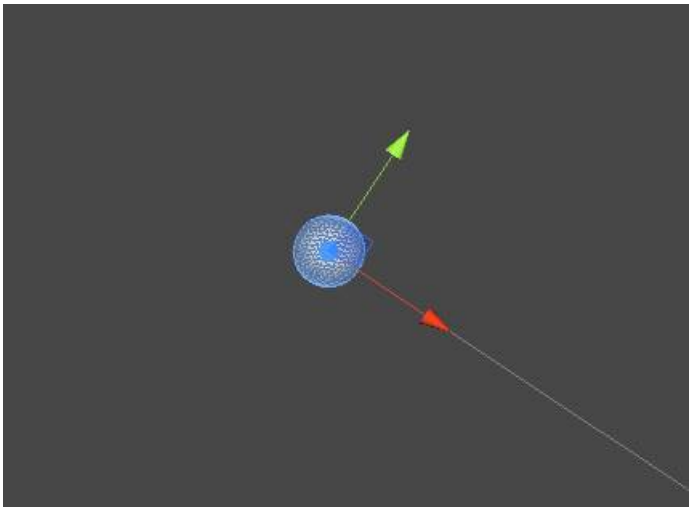
`bool QuerySidePositive(ref Circle2 circle, float epsilon = MathfEx.ZeroTolerance)`
Returns true if a circle is on the positive side of the line, false otherwise.

`Vector2 Project(Vector2 point)`
Returns projected point.

`float AngleBetweenTwoLines(Line2 anotherLine, bool acuteAngleDesired = false)`
Returns angle between this line's direction and another line's direction as: arccos(dot(this.Direction,another.Direction))
If acuteAngleDesired is true, then if resulting angle is > pi/2, then result is transformed to be pi-angle.

`string ToString()`
Returns string representation.

3.1.2 Ray2



A ray is represented as P+t*D, where P is the ray origin, D is a unit-length direction vector, and t >= 0. The user must ensure that D is indeed unit length.

Type
`struct Ray2`
Fields
`Center`
`Direction`
Construction
`Ray2()`
`Ray2(ref Vector2 center, ref Vector2 direction)`
`Ray2(Vector2 center, Vector2 direction)`
Methods
`Eval(float t)`
`DistanceTo(Vector2 point)`
`Project(Vector2 point)`
`string ToString()`

`Vector2 Center`
Ray origin

`Vector2 Direction`
Ray direction. Must be unit length!

`Ray2()`
Creates the ray. Users should initialize center and direction manually.

`Ray2(ref Vector2 center, ref Vector2 direction)`
Creates the ray
center - Ray origin
direction - Ray direction. Must be unit length!

`Ray2(Vector2 center, Vector2 direction)`
Creates the ray
center - Ray origin
direction - Ray direction. Must be unit length!

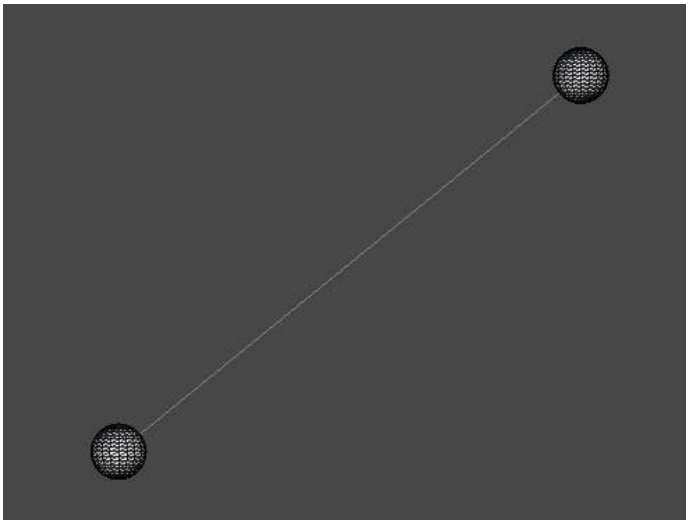
```
Vector2 Eval(float t)
```
Evaluates ray using P+t*D formula, where P is the ray origin, D is a unit-length direction vector, t is parameter.
t - Evaluation parameter

```
float DistanceTo(Vector2 point)
```
Returns distance to a point, distance is >= 0f.

```
Vector2 Project(Vector2 point)
```
Returns projected point

```
string ToString()
```
Returns string representation.

3.1.3 Segment2



A segment is represented as (1-s)*P0+s*P1, where P0 and P1 are the endpoints of the segment and 0 <= s <= 1.

Some algorithms involving segments might prefer a centered representation. This representation is C+t*D, where C = (P0+P1)/2 is the center of the segment, D = (P1-P0)/Length(P1-P0) is a unit-length direction vector for the segment, and |t| <= e. The value e = Length(P1-P0)/2 is the 'extent' (or radius or half-length) of the segment.

Type
`struct Segment2`
Fields
`P0`
`P1`
`Center`
`Direction`
`Extent`
Construction
`Segment2()`
`Segment2(ref Vector2 p0, ref Vector2 p1)`
`Segment2(Vector2 p0, Vector2 p1)`
`Segment2(ref Vector2 center, ref Vector2 direction, float extent)`
`Segment2(Vector2 center, Vector2 direction, float extent)`
Methods
`SetEndpoints(Vector2 p0, Vector2 p1)`
`SetCenterDirectionExtent(Vector2 center, Vector2 direction, float extent)`
`CalcCenterDirectionExtent()`
`CalcEndPoints()`
`Eval(float s)`
`DistanceTo(Vector2 point)`
`Project(Vector2 point)`
`ToString()`

`Vector2 P0`
Start point

`Vector2 P1`
End point

`Vector2 Center`
Segment center

`Vector2 Direction`
Segment direction. Must be unit length!

`float Extent`
Segment half-length

`Segment2()`
Creates the uninitialized segment. To initialize the segment use SetEndpoints() or SetCenterDirectionExtent() methods. It is also possible to set endpoints manually and call CalcCenterDirectionExtent() method or set center/direction/extent manually and call CalcEndPoints() method. Notice that simply setting only endpoints or center/direction/extent will leave segment in wrong state! Always call appropriate methods to keep all fields in the proper state. It is more convenient to use constructors with parameters which initialize the segment properly.

`Segment2(ref Vector2 p0, ref Vector2 p1)`
`Segment2(Vector2 p0, Vector2 p1)`
The constructor computes Center, Direction, and Extent from P0 and P1.
p0 - Segment start point
p1 - Segment end point
See also CalcCenterDirectionExtent() and CalcEndPoints().

`Segment2(ref Vector2 center, ref Vector2 direction, float extent)`
`Segment2(Vector2 center, Vector2 direction, float extent)`
The constructor computes P0 and P1 from Center, Direction, and Extent.
center - Center of the segment
direction - Direction of the segment. Must be unit length!
extent - Half-length of the segment
See also CalcCenterDirectionExtent() and CalcEndPoints().

`void SetEndpoints(Vector2 p0, Vector2 p1)`
Initializes segments from endpoints.

`void SetCenterDirectionExtent(Vector2 center, Vector2 direction, float extent)`
Initializes segment from center, direction and extent.

`void CalcCenterDirectionExtent()`
Call this function when you change P0 or P1. It calculates Center, Direction and Extent from P0 and P1.

`void CalcEndPoints()`
Call this function when you change Center, Direction, Extent. It calculates P0 and P1 from Center, Direction and Extent.

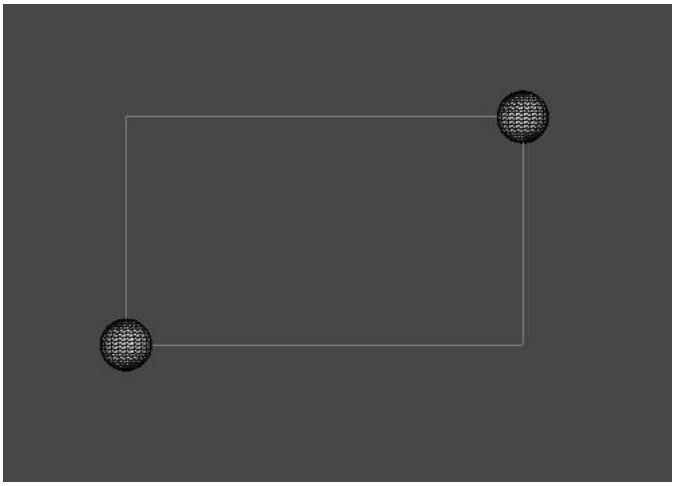
`Vector2 Eval(float s)`
Evaluates segment using (1-s)*P0+s*P1 formula, where P0 and P1 are endpoints, s is parameter.
s - Evaluation parameter

`float DistanceTo(Vector2 point)`
Returns distance to a point, distance is >= 0f.

`Vector2 Project(Vector2 point)`
Returns projected point

`string ToString()`
Returns string representation.

3.1.4 AAB2 (Axis-Aligned Box)



Axis aligned bounding box in 2D.

Type
struct AAB2
Fields
Min
Max
Construction
AAB2()
AAB2(ref Vector2 min, ref Vector2 max)
AAB2(Vector2 min, Vector2 max)
AAB2(float xMin, float xMax, float yMin, float yMax)
CreateFromPoint(ref Vector2 point)
CreateFromPoint(Vector2 point)
CreateFromTwoPoints(ref Vector2 point0, ref Vector2 point1)
CreateFromTwoPoints(Vector2 point0, Vector2 point1)
CreateFromPoints(IEnumerable<Vector2> points)
CreateFromPoints(IList<Vector2> points)
CreateFromPoints(Vector2[] points)
Operators
operator Rect(AAB2 value)
operator AAB2(Rect value)
Methods
CalcCenterExtents(out Vector2 center, out Vector2 extents)
CalcVertices(out Vector2 vertex0, out Vector2 vertex1, out Vector2 vertex2, out Vector2 vertex3)
CalcVertices()
CalcVertices(Vector2[] array)
CalcArea()
DistanceTo(Vector2 point)
Project(Vector2 point)
Contains(ref Vector2 point)

`Contains(Vector2 point)`
`Include(ref Vector2 point)`
`Include(Vector2 point)`
`Include(ref AAB2 box)`
`Include(AAB2 box)`
`ToString()`

`Vector2 Min`
Min point

`Vector2 Max`
Max point

`AAB2()`
Creates AAB. Use it only if you plan to set Min and Max manually. As AAB is a struct, by default C# intializes Min and Max fields to default values (zero vectors). This is inconvenient for the axis-aligned box due to it prevents calling Include() methods straight away. (Imagine creating aab with zero Min/Max and then including axis-aligned box which has entirely negative coordinates of Min/Max. In this case Max coordinate will remain zero vector which is wrong). To fix this issue use constructors with parameters or static contructors. Do not create AAB with Min=PositiveInfinity and Max=NegativeInfinity and then call Include(), due to speed optimizations in Include() methods this will not work properly.

`AAB2(ref Vector2 min, ref Vector2 max)`
`AAB2(Vector2 min, Vector2 max)`
Creates AAB from min and max points.

`AAB2(float xMin, float xMax, float yMin, float yMax)`
Creates AAB. The caller must ensure that xmin <= xmax and ymin <= ymax.

`static AAB2 CreateFromPoint(ref Vector2 point)`
`static AAB2 CreateFromPoint(Vector2 point)`
Creates AAB from single point. Min and Max are set to point. Use Include() method to grow the resulting AAB.

`static AAB2 CreateFromTwoPoints(ref Vector2 point0, ref Vector2 point1)`
`static AAB2 CreateFromTwoPoints(Vector2 point0, Vector2 point1)`
Computes AAB from two points. In case min and max points are known, use constructor instead.

`static AAB2 CreateFromPoints(IEnumerable<Vector2> points)`
`static AAB2 CreateFromPoints(IList<Vector2> points)`
`static AAB2 CreateFromPoints(Vector2[] points)`
Computes AAB from the a of points. Method includes points from the a one by one to create the AAB.
If a set is empty, returns new AAB2().

`static implicit operator Rect(AAB2 value)`
Converts AAB2 to UnityEngine.Rect. Code is Rect.MinMaxRect(value.Min.x, value.Min.y, value.Max.x, value.Max.y)

`static implicit operator AAB2(Rect value)`
Converts UnityEngine.Rect to AAB2. Code is AAB2() { Min = new Vector2(value.xMin, value.yMin), Max = new Vector2(value.xMax, value.yMax) }

`void CalcCenterExtents(out Vector2 center, out Vector2 extents)`
Computes box center and extents (half sizes)

```
void CalcVertices(out Vector2 vertex0, out Vector2 vertex1, out Vector2 vertex2, out Vector2 vertex3)
```
Calculates 4 box corners.
vertex0 - Vector2(Min.x, Min.y)
vertex1 - Vector2(Max.x, Min.y)
vertex2 - Vector2(Max.x, Max.y)
vertex3 - Vector2(Min.x, Max.y)

```
Vector2[] CalcVertices()
```
Calculates 4 box corners and returns them in an allocated array.
See array-less overload for the description.

```
void CalcVertices(Vector2[] array)
```
Calculates 4 box corners and fills the input array with them (array length must be 4).
See array-less overload for the description.

```
float CalcArea()
```
Returns box area

```
float DistanceTo(Vector2 point)
```
Returns distance to a point, distance is >= 0f.

```
Vector2 Project(Vector2 point)
```
Returns projected point

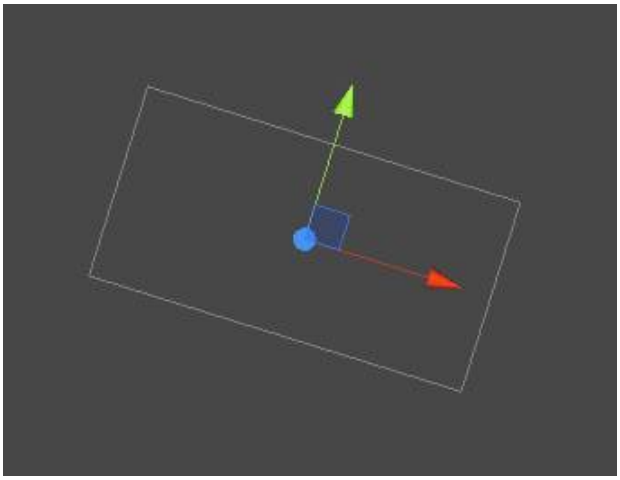
```
bool Contains(ref Vector2 point)
bool Contains(Vector2 point)
```
Tests whether a point is contained by the aab

```
void Include(ref Vector2 point)
void Include(Vector2 point)
```
Enlarges the box to include the point. If the point is inside the AAB does nothing.

```
void Include(ref AAB2 box)
void Include(AAB2 box)
```
Enlarges the aab so it includes another aab.

```
string ToString()
```
Returns string representation.

3.1.5 Box2



A box has center C, axis directions A0 and A1 (perpendicular and unit-length vectors), and extents e0 and e1 (nonnegative numbers). A point X = C + y0*A0 + y1*A1 is inside or on the box whenever |y[i]| <= e[i] for all i.

Type
`struct Box2`
Fields
`Center`
`Axis0`
`Axis1`
`Extents`
Construction
`Box2()`
`Box2(ref Vector2 center, ref Vector2 axis0, ref Vector2 axis1, ref Vector2 extents)`
`Box2(Vector2 center, Vector2 axis0, Vector2 axis1, Vector2 extents)`
`Box2(ref AAB2 box)`
`Box2(AAB2 box)`
`CreateFromPoints(IList<Vector2> points)`
Methods
`GetAxis(int index)`
`CalcVertices(out Vector2 vertex0, out Vector2 vertex1, out Vector2 vertex2, out Vector2 vertex3)`
`CalcVertices()`
`CalcVertices(Vector2[] array)`
`CalcArea()`
`DistanceTo(Vector2 point)`
`Project(Vector2 point)`
`Contains(ref Vector2 point)`
`Contains(Vector2 point)`
`Include(ref Box2 box)`
`Include(Box2 box)`
`ToString()`

`Vector2 Center`
Box center

```
Vector2 Axis0
```
First box axis. Must be unit length!

```
Vector2 Axis1
```
Second box axis. Must be unit length!

```
Vector2 Extents
```
Extents (half sizes) along Axis0 and Axis1. Must be non-negative!

```
Box2()
```
Creates new Box2 instance. Users must fill all the fields manually.

```
Box2(ref Vector2 center, ref Vector2 axis0, ref Vector2 axis1, ref Vector2 extents)
Box2(Vector2 center, Vector2 axis0, Vector2 axis1, Vector2 extents)
```
Creates new Box2 instance.
center - Box center
axis0 - First box axis. Must be unit length!
axis1 - Second box axis. Must be unit length!
extents - Extents (half sizes) along Axis0 and Axis1. Must be non-negative!

```
Box2(ref AAB2 box)
Box2(AAB2 box)
```
Creates Box2 from AxisAlignedBox2

```
static Box2 CreateFromPoints(IList<Vector2> points)
```
Computes oriented bounding box from a set of points. If a set is empty returns new Box2().

```
Vector2 GetAxis(int index)
```
Returns axis by index (0, 1)

```
void CalcVertices(out Vector2 vertex0, out Vector2 vertex1, out Vector2 vertex2, out Vector2 vertex3)
```
Calculates 4 box corners. extAxis[i] is Axis[i]*Extent[i], i=0,1.
vertex0 - Center - extAxis0 - extAxis1
vertex1 - Center + extAxis0 - extAxis1
vertex2 - Center + extAxis0 + extAxis1
vertex3 - Center - extAxis0 + extAxis1

```
Vector2[] CalcVertices()
```
Calculates 4 box corners and returns them in an allocated array. See array-less overload for the description.

```
void CalcVertices(Vector2[] array)
```
Calculates 4 box corners and fills the input array with them (array length must be 4).
See array-less overload for the description.

```
float CalcArea()
```
Returns area of the box as Extents.x * Extents.y * 4

```
float DistanceTo(Vector2 point)
```
Returns distance to a point, distance is >= 0f.

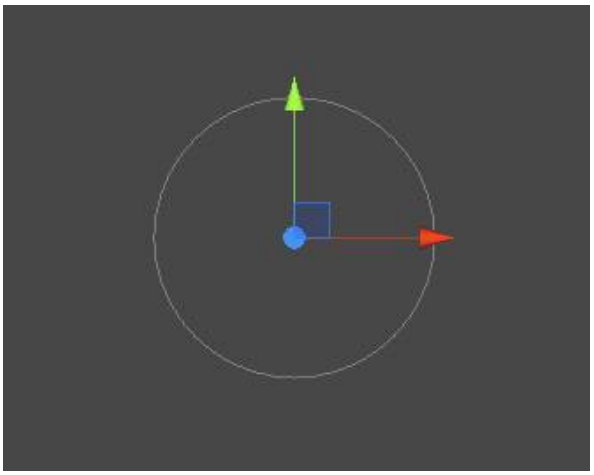
```
Vector2 Project(Vector2 point)
```
Returns projected point

```
bool Contains(ref Vector2 point)
bool Contains(Vector2 point)
```
Tests whether a point is contained by the box

```
void Include(ref Box2 box)
void Include(Box2 box)
```
Enlarges the box so it includes another box.

```
string ToString()
```
Returns string representation.

3.1.6 Circle2



Circle is described by the formula |X - C| ^2=r^2, where C - circle center, r - circle radius.

Type
struct Circle2
Fields
Center
Radius
Construction
Circle2()
Circle2(ref Vector2 center, float radius)
Circle2(Vector2 center, float radius)
CreateFromPointsAAB(IEnumerable<Vector2> points)
CreateFromPointsAAB(IList<Vector2> points)
CreateFromPointsAverage(IEnumerable<Vector2> points)
CreateFromPointsAverage(IList<Vector2> points)
CreateCircumscribed(Vector2 v0, Vector2 v1, Vector2 v2, out Circle2 circle)
CreateInscribed(Vector2 v0, Vector2 v1, Vector2 v2, out Circle2 circle)
Methods
CalcPerimeter()
CalcArea()
Eval(float t)
Eval(float t, float radius)
DistanceTo(Vector2 point)
Project(Vector2 point)
Contains(ref Vector2 point)
Contains(Vector2 point)
Include(ref Circle2 circle)
Include(Circle2 circle)
ToString()

Vector2 Center
Circle center

```
float Radius
```
Circle radius

```
Circle2()
```
Creates circle instance. Users must fill center and radius manually.

```
Circle2(ref Vector2 center, float radius)
Circle2(Vector2 center, float radius)
```
Creates circle from center and radius

```
static Circle2 CreateFromPointsAAB(IEnumerable<Vector2> points)
static Circle2 CreateFromPointsAAB(IList<Vector2> points)
```
Computes bounding circle from a set of points. First compute the axis-aligned bounding box of the points, then compute the circle containing the box. If a set is empty returns new Circle2().

```
static Circle2 CreateFromPointsAverage(IEnumerable<Vector2> points)
static Circle2 CreateFromPointsAverage(IList<Vector2> points)
```
Computes bounding circle from a set of points. Compute the smallest circle whose center is the average of a point set. If a set is empty returns new Circle2().

```
static bool CreateCircumscribed(Vector2 v0, Vector2 v1, Vector2 v2, out Circle2 circle)
```
Creates circle which is circumscribed around triangle. Returns 'true' if circle has been constructed, 'false' otherwise (input points are linearly dependent).

```
static bool CreateInscribed(Vector2 v0, Vector2 v1, Vector2 v2, out Circle2 circle)
```
Creates circle which is insribed into triangle. Returns 'true' if circle has been constructed, 'false' otherwise (input points are linearly dependent).

```
float CalcPerimeter()
```
Returns circle perimeter

```
float CalcArea()
```
Returns circle area

```
Vector2 Eval(float t)
```
Evaluates circle using formula X = C + R*[cos(t), sin(t)] where t is an angle in [0,2*pi).
t - Evaluation parameter

```
Vector2 Eval(float t, float radius)
```
Evaluates disk using formula X = C + radius*[cos(t), sin(t)] where t is an angle in [0,2*pi).
t - Evaluation parameter
radius - Evaluation radius

```
float DistanceTo(Vector2 point)
```
Returns distance to a point, distance is >= 0f.

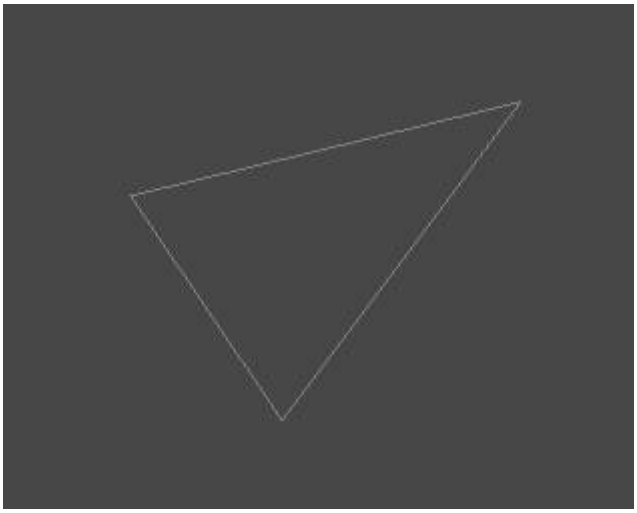
```
Vector2 Project(Vector2 point)
```
Returns projected point

```
bool Contains(ref Vector2 point)
bool Contains(Vector2 point)
```
Tests whether a point is contained by the circle

```
void Include(ref Circle2 circle)
void Include(Circle2 circle)
```
Enlarges the circle so it includes another circle.

```
string ToString()
```
Returns string representation.

3.1.7 Triangle2



Triangle in 2D. Below CW is a shorthand for Clock-Wise and CCW is Counter Clock-Wise.

Test Prefab:
Test_Triangle2

Type
struct Triangle2
Fields
V0
V1
V2
Properties
this[int index] { get; set; }
Construction
Triangle2()
Triangle2(ref Vector2 v0, ref Vector2 v1, ref Vector2 v2)
Triangle2(Vector2 v0, Vector2 v1, Vector2 v2)
Methods
CalcEdge(int edgeIndex)
CalcDeterminant()
CalcOrientation(float threshold)
CalcArea()
CalcArea(ref Vector2 v0, ref Vector2 v1, ref Vector2 v2)
CalcArea(Vector2 v0, Vector2 v1, Vector2 v2)
CalcAnglesDeg()
CalcAnglesDeg(ref Vector2 v0, ref Vector2 v1, ref Vector2 v2)
CalcAnglesDeg(Vector2 v0, Vector2 v1, Vector2 v2)
CalcAnglesRad()
CalcAnglesRad(ref Vector2 v0, ref Vector2 v1, ref Vector2 v2)
CalcAnglesRad(Vector2 v0, Vector2 v1, Vector2 v2)
EvalBarycentric(float c0, float c1)
EvalBarycentric(ref Vector3 baryCoords)
EvalBarycentric(Vector3 baryCoords)

`CalcBarycentricCoords(ref Vector2 point, ref Vector2 v0, ref Vector2 v1, ref Vector2 v2, out Vector3 baryCoords)`
`CalcBarycentricCoords(ref Vector2 point)`
`CalcBarycentricCoords(Vector2 point)`
`DistanceTo(Vector2 point)`
`QuerySideCCW(Vector2 point, float epsilon)`
`QuerySideCW(Vector2 point, float epsilon)`
`Project(Vector2 point)`
`Contains(ref Vector2 point)`
`Contains(Vector2 point)`
`ContainsCCW(ref Vector2 point)`
`ContainsCCW(Vector2 point)`
`ContainsCW(ref Vector2 point)`
`ContainsCW(Vector2 point)`
`ToString()`

`Vector2 V0`
First triangle vertex

`Vector2 V1`
Second triangle vertex

`Vector2 V2`
Third triangle vertex

`Vector2 this[int index]  { get; set; }`
Gets or sets triangle vertex by index: 0, 1 or 2

`Triangle2()`
Creates Triangle2 instance. Users must fill vertices manually.

`Triangle2(ref Vector2 v0, ref Vector2 v1, ref Vector2 v2)`
Creates Triangle2 from 3 vertices

`Triangle2(Vector2 v0, Vector2 v1, Vector2 v2)`
Creates Triangle2 from 3 vertices

`Vector2 CalcEdge(int edgeIndex)`
Returns triangle edge by index 0, 1 or 2. Edge[i] = V[i+1]-V[i].

`float CalcDeterminant()`
Calculates cross product of triangle edges: (V1-V0)x(V2-V0).
If the result is positive then triangle is ordered counter clockwise,
if the result is negative then triangle is ordered clockwise,
if the result is zero then triangle is degenerate.

`Orientations CalcOrientation(float threshold = MathfEx.ZeroTolerance)`
Calculates triangle orientation. See CalcDeterminant() for the description.

`float CalcArea()`
Calculates area of the triangle. It's equal to Abs(Determinant())/2

```
static float CalcArea(ref Vector2 v0, ref Vector2 v1, ref Vector2 v2)
static float CalcArea(Vector2 v0, Vector2 v1, Vector2 v2)
```
Calculates area of the triangle defined by 3 points.

```
Vector3 CalcAnglesDeg()
```
Calculates angles of the triangle in degrees.
Angles are returned in the instance of Vector3 following way: (angle of vertex V0, angle of vertex V1, angle of vertex V2)

```
static Vector3 CalcAnglesDeg(ref Vector2 v0, ref Vector2 v1, ref Vector2 v2)
static Vector3 CalcAnglesDeg(Vector2 v0, Vector2 v1, Vector2 v2)
```
Calculates angles of the triangle defined by 3 points in degrees.
Angles are returned in the instance of Vector3 following way: (angle of vertex V0, angle of vertex V1, angle of vertex V2)

```
Vector3 CalcAnglesRad()
```
Calculates angles of the triangle in radians.
Angles are returned in the instance of Vector3 following way: (angle of vertex V0, angle of vertex V1, angle of vertex V2)

```
static Vector3 CalcAnglesRad(ref Vector2 v0, ref Vector2 v1, ref Vector2 v2)
static Vector3 CalcAnglesRad(Vector2 v0, Vector2 v1, Vector2 v2)
```
Calculates angles of the triangle defined by 3 points in radians.
Angles are returned in the instance of Vector3 following way: (angle of vertex V0, angle of vertex V1, angle of vertex V2)

```
Vector2 EvalBarycentric(float c0, float c1)
```
Gets point on the triangle using barycentric coordinates.
The result is c0*V0 + c1*V1 + c2*V2, 0 <= c0,c1,c2 <= 1, c0+c1+c2=1, c2 is calculated as 1-c0-c1.

```
Vector2 EvalBarycentric(ref Vector3 baryCoords)
Vector2 EvalBarycentric(Vector3 baryCoords)
```
Gets point on the triangle using barycentric coordinates. baryCoords parameter is (c0,c1,c2).
The result is c0*V0 + c1*V1 + c2*V2, 0 <= c0,c1,c2 <= 1, c0+c1+c2=1

```
static void CalcBarycentricCoords(ref Vector2 point, ref Vector2 v0, ref Vector2 v1, ref Vector2 v2, out
Vector3 baryCoords)
```
Calculate barycentric coordinates for the input point with regarding to triangle defined by 3 points.

```
Vector3 CalcBarycentricCoords(ref Vector2 point)
Vector3 CalcBarycentricCoords(Vector2 point)
```
Calculate barycentric coordinates for the input point regarding to the triangle.

```
float DistanceTo(Vector2 point)
```
Returns distance to a point, distance is >= 0f.

```
int QuerySideCCW(Vector2 point, float epsilon = MathfEx.ZeroTolerance)
```
Determines on which side of the triangle a point is. Returns +1 if a point is outside of the triangle, 0 if it's on the triangle border, -1 if it's inside the triangle. Method must be called for CCW ordered triangles.

```
int QuerySideCW(Vector2 point, float epsilon = MathfEx.ZeroTolerance)
```
Determines on which side of the triangle a point is. Returns +1 if a point is outside of the triangle, 0 if it's on the triangle border, -1 if it's inside the triangle. Method must be called for CW ordered triangles.

```
Vector2 Project(Vector2 point)
```
Returns projected point

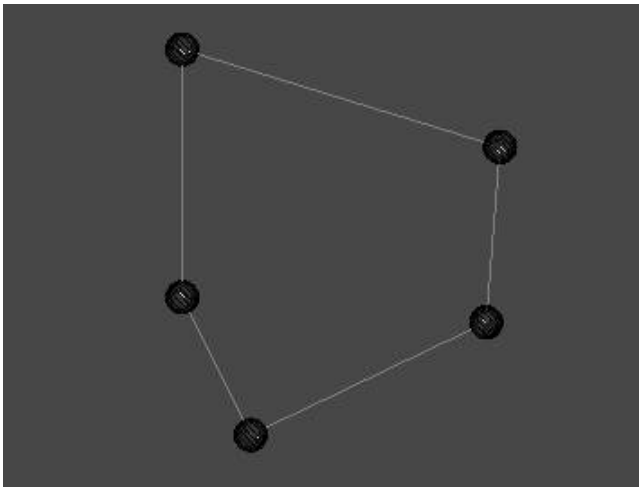
```
bool Contains(ref Vector2 point)
bool Contains(Vector2 point)
```
Tests whether a point is contained by the triangle (CW or CCW ordered). Note however that if the triangle is CCW then points which are on triangle border considered inside, but if the triangle is CW then points which are on triangle border considered outside.
For consistent (and faster) test use appropriate overloads for CW and CCW triangles.

```
bool ContainsCCW(ref Vector2 point)
bool ContainsCCW(Vector2 point)
```
Tests whether a point is contained by the CCW triangle

```
bool ContainsCW(ref Vector2 point)
bool ContainsCW(Vector2 point)
```
Tests whether a point is contained by the CW triangle

```
string ToString()
```
Returns string representation.

3.1.8 Polygon2



Represents 2D polygon (vertex count must be >= 3). Polygon has single border. Many algorithms also require polygon to be convex, it's responsibility of a user to provide such polygons. Polygon contains vertices and edges. Vertex is a simple vector, edge is described below. Notice that Polygon2 is a class not struct.

Test Prefab:
Test_Polygon2

Type
`struct Edge2`
Fields
`Point0` Edge start vertex
`Point1` Edge end vertex
`Direction` Unit length direction vector
`Normal` Unit length normal vector
`Length` Edge length

Type
`class Polygon2`
Properties
`Vertices { get; }`
`Edges { get; }`
`VertexCount { get; }`
`this[int vertexIndex] { get; set; }`
Construction
`Polygon2()`
`Polygon2(Vector2[] vertices)`
`Polygon2(int vertexCount)`
`CreateProjected(Polygon3 polygon, ProjectionPlanes projectionPlane)`
Methods
`GetEdge(int edgeIndex)`
`UpdateEdges()`
`UpdateEdge(int edgeIndex)`

`CalcCenter()`
`CalcPerimeter()`
`CalcArea()`
`IsConvex(out Orientations orientation, float threshold)`
`IsConvex(float threshold)`
`HasZeroCorners(float threshold)`
`ReverseVertices()`
`ContainsConvexQuadCCW(ref Vector2 point)`
`ContainsConvexQuadCCW(Vector2 point)`
`ContainsConvexQuadCW(ref Vector2 point)`
`ContainsConvexQuadCW(Vector2 point)`
`ContainsConvexCCW(ref Vector2 point)`
`ContainsConvexCCW(Vector2 point)`
`ContainsConvexCW(ref Vector2 point)`
`ContainsConvexCW(Vector2 point)`
`ContainsSimple(ref Vector2 point)`
`ContainsSimple(Vector2 point)`
`ToSegmentArray()`
`ToString()`

`Vector2[] Vertices { get; }`
Gets vertices array (do not change the data, use only for traversal)

`Edge2[] Edges { get; }`
Gets edges array (do not change the data, use only for traversal)

`int VertexCount { get; }`
Polygon vertex count

`Vector2 this[int vertexIndex] { get; set; }`
Gets or sets polygon vertex

`Polygon2()`
Default constructor is unavailable to the users. Use constructors with parameters.

`Polygon2(Vector2[] vertices)`
Creates polygon from an array of vertices (array is copied)

`Polygon2(int vertexCount)`
Creates polygon setting number of vertices. Vertices then can be filled using indexer.

`static Polygon2 CreateProjected(Polygon3 polygon, ProjectionPlanes projectionPlane)`
Creates Polygon2 instance from Polygon3 instance by projecting Polygon3 vertices onto one of three base planes (on practice just dropping one of the coordinates).

`Edge2 GetEdge(int edgeIndex)`
Returns polygon edge

`void UpdateEdges()`
Updates all polygon edges. Use after vertex change.

```
void UpdateEdge(int edgeIndex)
```
Updates certain polygon edge. Use after vertex change.

```
Vector2 CalcCenter()
```
Returns polygon center

```
float CalcPerimeter()
```
Returns polygon perimeter length

```
float CalcArea()
```
Returns polygon area (polygon must be simple, i.e. without self-intersections).

```
bool IsConvex(out Orientations orientation, float threshold = MathfEx.ZeroTolerance)
```
Tests if the polygon is convex and returns orientation

```
bool IsConvex(float threshold = MathfEx.ZeroTolerance)
```
Tests if the polygon is convex

```
bool HasZeroCorners(float threshold = MathfEx.ZeroTolerance)
```
Returns true if polygon contains some edges which have zero angle between them.

```
void ReverseVertices()
```
Reverses polygon vertex order

```
bool ContainsConvexQuadCCW(ref Vector2 point)
bool ContainsConvexQuadCCW(Vector2 point)
```
Tests whether a point is contained by the convex CCW 4-sided polygon (the caller must ensure that polygon is indeed CCW ordered)

```
bool ContainsConvexQuadCW(ref Vector2 point)
bool ContainsConvexQuadCW(Vector2 point)
```
Tests whether a point is contained by the convex CW 4-sided polygon (the caller must ensure that polygon is indeed CW ordered)

```
bool ContainsConvexCCW(ref Vector2 point)
bool ContainsConvexCCW(Vector2 point)
```
Tests whether a point is contained by the convex CCW polygon (the caller must ensure that polygon is indeed CCW ordered)

```
bool ContainsConvexCW(ref Vector2 point)
bool ContainsConvexCW(Vector2 point)
```
Tests whether a point is contained by the convex CW polygon (the caller must ensure that polygon is indeed CW ordered)

```
bool ContainsSimple(ref Vector2 point)
bool ContainsSimple(Vector2 point)
```
Tests whether a point is contained by the simple polygon (i.e. without self intersection). Non-convex polygons are allowed, orientation is irrelevant.
Note that points which are on border may be classified differently depending on the point position.

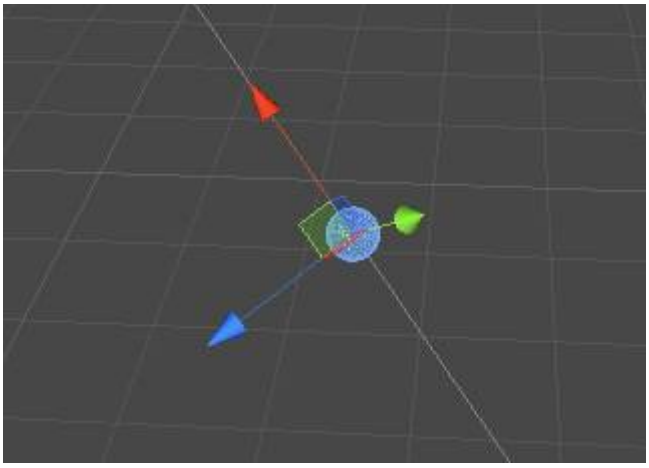
```
Segment2[] ToSegmentArray()
```
Converts the polygon to segment array

```
string ToString()
```
Returns string representation.

3.2 3D Objects

This section describes primitive objects in 3D. Library includes line, ray, segment, plane, rectangle, AAB, box, circle, sphere, triangle and polygon primitives in 3D. Each subsection lists available for the user methods.

3.2.1 Line3



The line is represented as P+t*D where P is the line origin, D is a unit-length direction vector, and t is any real number. The user must ensure that D is indeed unit length.

Type
`struct Line3`
Fields
`Center`
`Direction`
Construction
`Line3()`
`Line3(ref Vector3 center, ref Vector3 direction)`
`Line3(Vector3 center, Vector3 direction)`
`static Line3 CreateFromTwoPoints(ref Vector3 p0, ref Vector3 p1)`
`static Line3 CreateFromTwoPoints(Vector3 p0, Vector3 p1)`
Methods
`Eval(float t)`
`DistanceTo(Vector3 point)`
`Project(Vector3 point)`
`AngleBetweenTwoLines(Line3 anotherLine, bool acuteAngleDesired)`
`ToString()`

`Vector3 Center`
Line origin

`Vector3 Direction`
Line direction. Must be unit length!

`Line3()`
Creates the line. Users should initialize center and direction manually.

```
Line3(ref Vector3 center, ref Vector3 direction)
Line3(Vector3 center, Vector3 direction)
```
Creates the line
center - Line origin
direction - Line direction. Must be unit length!

```
static Line3 CreateFromTwoPoints(ref Vector3 p0, ref Vector3 p1)
static Line3 CreateFromTwoPoints(Vector3 p0, Vector3 p1)
```
Creates the line. Origin is p0, Direction is Normalized(p1-p0).
p0 - First point
p1 - Second point

```
Vector3 Eval(float t)
```
Evaluates line using P+t*D formula, where P is the line origin, D is a unit-length direction vector, t is parameter.
t - Evaluation parameter

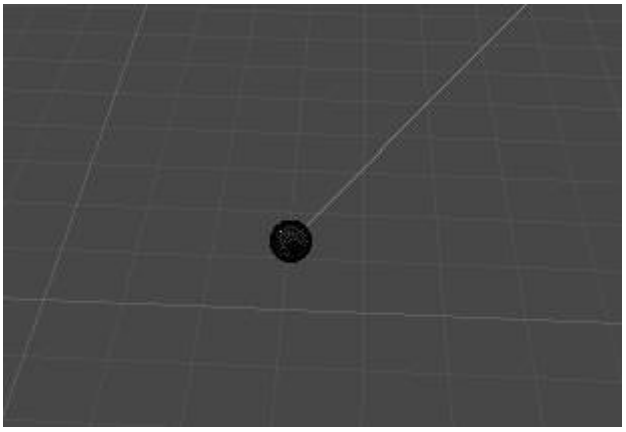
```
float DistanceTo(Vector3 point)
```
Returns distance to a point, distance is >= 0f.

```
Vector3 Project(Vector3 point)
```
Returns projected point

```
float AngleBetweenTwoLines(Line3 anotherLine, bool acuteAngleDesired = false)
```
Returns angle between this line's direction and another line's direction as: arccos(dot(this.Direction,another.Direction)). If acuteAngleDesired is true, then in resulting angle is > pi/2, then result is transformed to be pi-angle.

```
string ToString()
```
Returns string representation.

3.2.2 Ray3



The ray is represented as P+t*D, where P is the ray origin, D is a unit-length direction vector, and t >= 0. The user must ensure that D is indeed unit length.

Type
`struct Ray3`
Fields
`Center`
`Direction`
Construction
`Ray3()`
`Ray3(ref Vector3 center, ref Vector3 direction)`
`Ray3(Vector3 center, Vector3 direction)`
Operators
`operator Ray(Ray3 value)`
`operator Ray3(Ray value)`
Methods
`Eval(float t)`
`DistanceTo(Vector3 point)`
`Project(Vector3 point)`
`ToString()`

`Vector3 Center`
Ray origin.

`Vector3 Direction`
Ray direction. Must be unit length!

`Ray3()`
Creates the ray. Users should initialize center and direction manually.

`Ray3(ref Vector3 center, ref Vector3 direction)`
`Ray3(Vector3 center, Vector3 direction)`
Creates the ray
center - Ray origin
direction - Ray direction. Must be unit length!

```
static implicit operator Ray(Ray3 value)
```
Converts Ray3 to UnityEngine.Ray

```
static implicit operator Ray3(Ray value)
```
Converts UnityEngine.Ray to Ray3

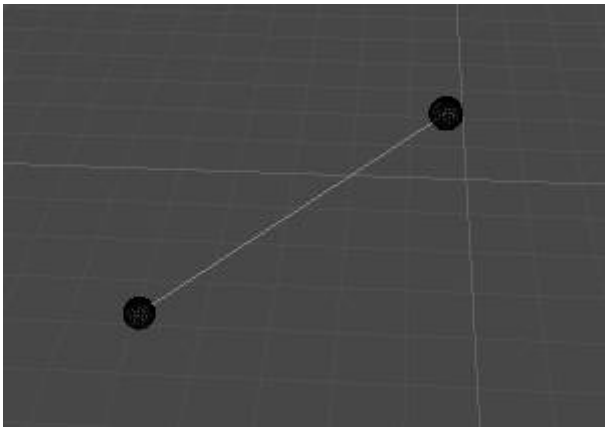
```
Vector3 Eval(float t)
```
Evaluates ray using P+t*D formula, where P is the ray origin, D is a unit-length direction vector, t is parameter.
t - Evaluation parameter

```
float DistanceTo(Vector3 point)
```
Returns distance to a point, distance is >= 0f.

```
Vector3 Project(Vector3 point)
```
Returns projected point

```
string ToString()
```
Returns string representation.

3.2.3 Segment3



The segment is represented as (1-s)*P0+s*P1, where P0 and P1 are the endpoints of the segment and 0 <= s <= 1. Some algorithms involving segments might prefer a centered representation similar to how oriented bounding boxes are defined. This representation is C+t*D, where C = (P0+P1)/2 is the center of the segment, D = (P1-P0)/Length(P1-P0) is a unit-length direction vector for the segment, and |t| <= e. The value e = Length(P1-P0)/2 is the 'extent' (or radius or half-length) of the segment.

Type
`struct Segment3`
Fields
`P0`
`P1`
`Center`
`Direction`
`Extent`
Construction
`Segment3()`
`Segment3(ref Vector3 p0, ref Vector3 p1)`
`Segment3(Vector3 p0, Vector3 p1)`
`Segment3(ref Vector3 center, ref Vector3 direction, float extent)`
`Segment3(Vector3 center, Vector3 direction, float extent)`
Methods
`SetEndpoints(Vector3 p0, Vector3 p1)`
`SetCenterDirectionExtent(Vector3 center, Vector3 direction, float extent)`
`CalcCenterDirectionExtent()`
`CalcEndPoints()`
`Eval(float s)`
`DistanceTo(Vector3 point)`
`Project(Vector3 point)`
`ToString()`

`Vector3 P0`
Start point

`Vector3 P1`
End point

```
Vector3 Center
```
Segment center

```
Vector3 Direction
```
Segment direction. Must be unit length!

```
float Extent
```
Segment half-length

```
Segment3()
```
Creates the uninitialized segment. To initialize the segment use SetEndpoints() or SetCenterDirectionExtent() methods. It is also possible to set endpoints manually and call CalcCenterDirectionExtent() method or set center/direction/extent manually and call CalcEndPoints() method. Notice that simply setting only endpoints or center/direction/extent will leave segment in wrong state! Always call appropriate methods to keep all fields in the proper state. It is more convenient to use constructors with parameters which initialize the segment properly.

```
Segment3(ref Vector3 p0, ref Vector3 p1)
Segment3(Vector3 p0, Vector3 p1)
```
The constructor computes Center, Direction, and Extent from P0 and P1.
p0 - Segment start point
p1 - Segment end point

```
Segment3(ref Vector3 center, ref Vector3 direction, float extent)
Segment3(Vector3 center, Vector3 direction, float extent)
```
The constructor computes P0 and P1 from Center, Direction, and Extent.
center - Center of the segment
direction - Direction of the segment. Must be unit length!
extent - Half-length of the segment

```
void SetEndpoints(Vector3 p0, Vector3 p1)
```
Initializes segments from endpoints.

```
void SetCenterDirectionExtent(Vector3 center, Vector3 direction, float extent)
```
Initializes segment from center, direction and extent.

```
void CalcCenterDirectionExtent()
```
Call this function when you change P0 or P1.

```
void CalcEndPoints()
```
Call this function when you change Center, Direction, or Extent.

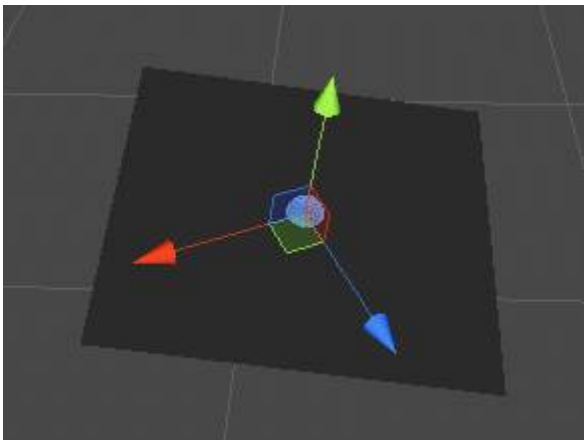
```
Vector3 Eval(float s)
```
Evaluates segment using (1-s)*P0+s*P1 formula, where P0 and P1 are endpoints, s is parameter.
s - Evaluation parameter

```
float DistanceTo(Vector3 point)
```
Returns distance to a point, distance is >= 0f.

```
Vector3 Project(Vector3 point)
```
Returns projected point

```
string ToString()
```
Returns string representation.

3.2.4 Plane3



The plane is represented as Dot(N,X)=c where N is a unit-length normal vector, c is the plane constant, and X is any point on the plane. The user must ensure that the normal vector is unit length.

Type
struct Plane3
Fields
Normal
Constant
Construction
Plane3()
Plane3(ref Vector3 normal, float constant)
Plane3(Vector3 normal, float constant)
Plane3(ref Vector3 normal, ref Vector3 point)
Plane3(Vector3 normal, Vector3 point)
Plane3(ref Vector3 p0, ref Vector3 p1, ref Vector3 p2)
Plane3(Vector3 p0, Vector3 p1, Vector3 p2)
Operators
operator Plane(Plane3 value)
operator Plane3(Plane value)
Methods
CalcOrigin()
CreateOrthonormalBasis(out Vector3 u, out Vector3 v, out Vector3 n)
SignedDistanceTo(Vector3 point)
DistanceTo(Vector3 point)
QuerySide(Vector3 point, float epsilon)
QuerySideNegative(Vector3 point, float epsilon)
QuerySidePositive(Vector3 point, float epsilon)
QuerySide(ref Box3 box, float epsilon)
QuerySideNegative(ref Box3 box, float epsilon)
QuerySidePositive(ref Box3 box, float epsilon)
QuerySide(ref AAB3 box, float epsilon)
QuerySideNegative(ref AAB3 box, float epsilon)
QuerySidePositive(ref AAB3 box, float epsilon)
QuerySide(ref Sphere3 sphere, float epsilon)
QuerySideNegative(ref Sphere3 sphere, float epsilon)

QuerySidePositive(ref Sphere3 sphere, float epsilon)
Project(Vector3 point)
ProjectVector(Vector3 vector)
AngleBetweenPlaneNormalAndLine(Line3 line)
AngleBetweenPlaneNormalAndLine(Vector3 direction)
AngleBetweenPlaneAndLine(Line3 line)
AngleBetweenPlaneAndLine(Vector3 direction)
AngleBetweenTwoPlanes(Plane3 anotherPlane)
ToString()

`Vector3 Normal`
Plane normal. Must be unit length!

`float Constant`
Plane constant c from the equation Dot(N,X) = c

`Plane3()`
Creates plane instance. Users must fill normal and constant manually.

`Plane3(ref Vector3 normal, float constant)`
`Plane3(Vector3 normal, float constant)`
Creates the plane by specifying N and c directly.
normal - Must be unit length!

`Plane3(ref Vector3 normal, ref Vector3 point)`
`Plane3(Vector3 normal, Vector3 point)`
N is specified, c = Dot(N,P) where P is a point on the plane.
normal - Must be unit length!

`Plane3(ref Vector3 p0, ref Vector3 p1, ref Vector3 p2)`
`Plane3(Vector3 p0, Vector3 p1, Vector3 p2)`
Creates the plane from 3 points. N = Cross(P1-P0,P2-P0)/Length(Cross(P1-P0,P2-P0)), c = Dot(N,P0) where P0, P1, P2 are points on the plane.

`static implicit operator Plane(Plane3 value)`
Converts Plane3 to UnityEngine.Plane

`static implicit operator Plane3(Plane value)`
Converts UnityEngine.Plane to Plane3

`Vector3 CalcOrigin()`
Returns N*c

`void CreateOrthonormalBasis(out Vector3 u, out Vector3 v, out Vector3 n)`
Creates orthonormal basis from plane. In the output n - is the plane normal.

`float SignedDistanceTo(Vector3 point)`
Compute d = Dot(N,P)-c where N is the plane normal and c is the plane constant. This is a signed distance. The sign of the return value is positive if the point is on the positive side of the plane, negative if the point is on the negative side, and zero if the point is on the plane.

`float DistanceTo(Vector3 point)`
Returns distance to a point, distance is >= 0f.

```
int QuerySide(Vector3 point, float epsilon = MathfEx.ZeroTolerance)
```
Determines on which side of the plane a point is. Returns +1 if a point is on the positive side of the plane, 0 if it's on the plane, -1 if it's on the negative side. The positive side of the plane is the half-space to which the plane normal points.

```
bool QuerySideNegative(Vector3 point, float epsilon = MathfEx.ZeroTolerance)
```
Returns true if a point is on the negative side of the plane, false otherwise.

```
bool QuerySidePositive(Vector3 point, float epsilon = MathfEx.ZeroTolerance)
```
Returns true if a point is on the positive side of the plane, false otherwise.

```
int QuerySide(ref Box3 box, float epsilon = MathfEx.ZeroTolerance)
```
Determines on which side of the plane a box is. Returns +1 if a box is on the positive side of the plane, 0 if it's intersecting the plane, -1 if it's on the negative side. The positive side of the plane is the half-space to which the plane normal points.

```
bool QuerySideNegative(ref Box3 box, float epsilon = MathfEx.ZeroTolerance)
```
Returns true if a box is on the negative side of the plane, false otherwise.

```
bool QuerySidePositive(ref Box3 box, float epsilon = MathfEx.ZeroTolerance)
```
Returns true if a box is on the positive side of the plane, false otherwise.

```
int QuerySide(ref AAB3 box, float epsilon = MathfEx.ZeroTolerance)
```
Determines on which side of the plane a box is. Returns +1 if a box is on the positive side of the plane, 0 if it's intersecting the plane, -1 if it's on the negative side. The positive side of the plane is the half-space to which the plane normal points.

```
bool QuerySideNegative(ref AAB3 box, float epsilon = MathfEx.ZeroTolerance)
```
Returns true if a box is on the negative side of the plane, false otherwise.

```
bool QuerySidePositive(ref AAB3 box, float epsilon = MathfEx.ZeroTolerance)
```
Returns true if a box is on the positive side of the plane, false otherwise.

```
int QuerySide(ref Sphere3 sphere, float epsilon = MathfEx.ZeroTolerance)
```
Determines on which side of the plane a sphere is. Returns +1 if a sphere is on the positive side of the plane, 0 if it's intersecting the plane, -1 if it's on the negative side. The positive side of the plane is the half-space to which the plane normal points.

```
bool QuerySideNegative(ref Sphere3 sphere, float epsilon = MathfEx.ZeroTolerance)
```
Returns true if a sphere is on the negative side of the plane, false otherwise.

```
bool QuerySidePositive(ref Sphere3 sphere, float epsilon = MathfEx.ZeroTolerance)
```
Returns true if a sphere is on the positive side of the plane, false otherwise.

```
Vector3 Project(Vector3 point)
```
Returns projected point

```
Vector3 ProjectVector(Vector3 vector)
```
Returns projected vector

```
float AngleBetweenPlaneNormalAndLine(Line3 line)
```
Returns angle in radians between plane normal and line direction which is: arccos(dot(normal,direction))

`float AngleBetweenPlaneNormalAndLine(Vector3 direction)`
Returns angle in radians between plane normal and line direction which is: arccos(dot(normal,direction)). Direction will be normalized.

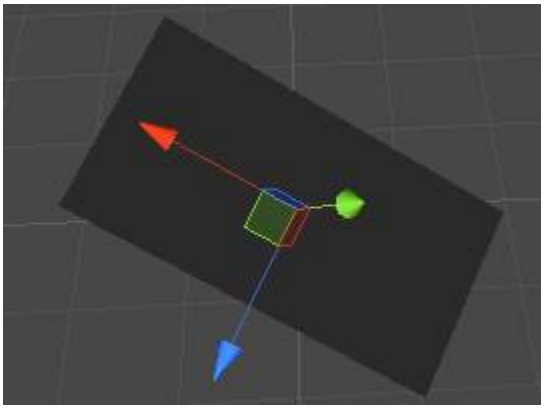
`float AngleBetweenPlaneAndLine(Line3 line)`
Returns angle in radians between plane itself and line direction which is: pi/2 — arccos(dot(normal,direction))

`float AngleBetweenPlaneAndLine(Vector3 direction)`
Returns angle in radians between plane itself and direction which is: pi/2 - arccos(dot(normal,direction)). Direction will be normalized.

`float AngleBetweenTwoPlanes(Plane3 anotherPlane)`
Returns angle in radians between this plane's normal and another plane's normal as: arccos(dot(this.Normal,another.Normal))

`string ToString()`
Returns string representation.

3.2.5 Rectangle3



Test Prefab:
Test_Rectangle3

Represents 2D box embedded in 3D space. Points are R(s,t)=C+s0*U0+s1*U1, where C is the center of the rectangle, U0 and U1 are unit-length and perpendicular axes. The parameters s0 and s1 are constrained by |s0|<=e0 and |s1|<=e1, where e0>0 and e1>0 are called the extents of the rectangle.

Test prefab shows construction from four points using static method.

Type
`struct Rectangle3`
Fields
`Center`
`Axis0`
`Axis1`
`Normal`
`Extents`
Construction
`Rectangle3()`
`Rectangle3(ref Vector3 center, ref Vector3 axis0, ref Vector3 axis1, Vector2 extents)`
`Rectangle3(Vector3 center, Vector3 axis0, Vector3 axis1, Vector2 extents)`
`CreateFromCCWPoints(Vector3 p0, Vector3 p1, Vector3 p2, Vector3 p3)`
`CreateFromCWPoints(Vector3 p0, Vector3 p1, Vector3 p2, Vector3 p3)`
Methods
`CalcVertices(out Vector3 vertex0, out Vector3 vertex1, out Vector3 vertex2, out Vector3 vertex3)`
`CalcVertices()`
`CalcVertices(Vector3[] array)`
`CalcArea()`
`DistanceTo(Vector3 point)`
`Project(Vector3 point)`
`ToString()`

`Vector3 Center`
Rectangle center

`Vector3 Axis0`
First rectangle axis. Must be unit length!

`Vector3 Axis1`
Second rectangle axis. Must be unit length!

```
Vector3 Normal
```
Rectangle normal which is Cross(Axis0, Axis1). Must be unit length!

```
Vector2 Extents
```
Extents (half sizes) along Axis0 and Axis1. Must be non-negative!

```
Rectangle3()
```
Creates Rectangle3 instance. Users must fill all the fields manually.

```
Rectangle3(ref Vector3 center, ref Vector3 axis0, ref Vector3 axis1, ref Vector2 extents)
Rectangle3(Vector3 center, Vector3 axis0, Vector3 axis1, Vector2 extents)
```
Creates new Rectangle3 instance.
center - Rectangle center.
axis0   - First box axis. Must be unit length!
axis1   - Second box axis. Must be unit length!
extents - Extents (half sizes) along Axis0 and Axis1. Must be non-negative!

```
static Rectangle3 CreateFromCCWPoints(Vector3 p0, Vector3 p1, Vector3 p2, Vector3 p3)
```
Creates rectangle from 4 counter clockwise ordered points.
Center=(p0+p2)/2,
Axis0=Normalized(p1-p0),
Axis1=Normalized(p2-p1).
The user therefore must ensure that the points are indeed represent rectangle to obtain meaningful result.

```
static Rectangle3 CreateFromCWPoints(Vector3 p0, Vector3 p1, Vector3 p2, Vector3 p3)
```
Creates rectangle from 4 clockwise ordered points.
Center=(p0+p2)/2,
Axis0=Normalized(p2-p1),
Axis1=Normalized(p1-p0).
The user therefore must ensure that the points are indeed represent rectangle to obtain meaningful result.

```
void CalcVertices(out Vector3 vertex0, out Vector3 vertex1, out Vector3 vertex2, out Vector3 vertex3)
```
Calculates 4 box corners. extAxis[i] is Axis[i]*Extent[i], i=0,1.
vertex0 - Center - extAxis0 - extAxis1
vertex1 - Center + extAxis0 - extAxis1
vertex2 - Center + extAxis0 + extAxis1
vertex3 - Center - extAxis0 + extAxis1

```
Vector3[] CalcVertices()
```
Calculates 4 box corners and returns them in an allocated array. Look array-less method for the description.

```
void CalcVertices(Vector3[] array)
```
Calculates 4 box corners and fills the input array with them (array length must be 4).
Look array-less method for the description.

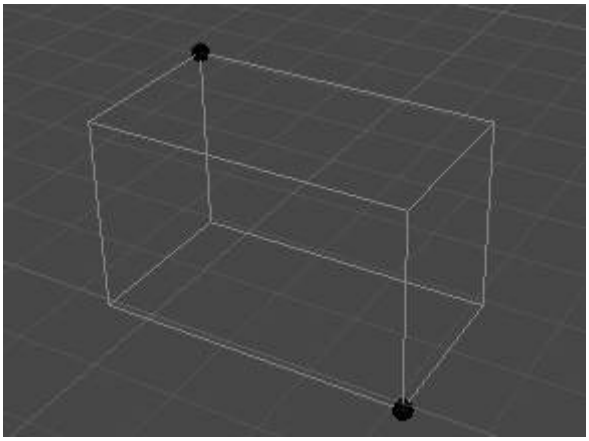
```
float CalcArea()
```
Returns area of the box as Extent.x*Extent.y*4

```
float DistanceTo(Vector3 point)
```
Returns distance to a point, distance is >= 0f.

```
Vector3 Project(Vector3 point)
```
Returns projected point

```
string ToString()
```
Returns string representation.

3.2.6 AAB3 (Axis-Aligned Box)



Axis aligned bounding box in 3D

Type
struct AAB3
Fields
Min
Max
Construction
AAB3()
AAB3(ref Vector3 min, ref Vector3 max)
AAB3(Vector3 min, Vector3 max)
AAB3(float xMin, float xMax, float yMin, float yMax, float zMin, float zMax)
static AAB3 CreateFromPoint(Vector3 point)
static AAB3 CreateFromTwoPoints(ref Vector3 point0, ref Vector3 point1)
static AAB3 CreateFromTwoPoints(Vector3 point0, Vector3 point1)
static AAB3 CreateFromPoints(IEnumerable<Vector3> points)
static AAB3 CreateFromPoints(IList<Vector3> points)
static AAB3 CreateFromPoints(Vector3[] points)
Operators
operator Bounds(AAB3 value)
operator AAB3(Bounds value)
Methods
CalcCenterExtents(out Vector3 center, out Vector3 extents)
CalcVertices(out Vector3 vertex0, out Vector3 vertex1, out Vector3 vertex2, out Vector3 vertex3, out Vector3 vertex4, out Vector3 vertex5, out Vector3 vertex6, out Vector3 vertex7)
CalcVertices()
CalcVertices(Vector3[] array)
CalcVolume()
DistanceTo(Vector3 point)
Project(Vector3 point)
Contains(ref Vector3 point)
Contains(Vector3 point)

`Include(ref Vector3 point)`
`Include(Vector3 point)`
`Include(ref AAB3 box)`
`Include(AAB3 box)`
`ToString()`

`Vector3 Min`
Min point

`Vector3 Max`
Max point

`AAB3()`
Creates AAB. Use it only if you plan to set Min and Max manually. As AAB is a struct, by default C# intializes Min and Max fields to default values (zero vectors). This is inconvenient for the axis-aligned box due to it prevents calling Include() methods straight away. (Imagine creating aab with zero Min/Max and then including axis-aligned box which has entirely negative coordinates of Min/Max. In this case Max coordinate will remain zero vector which is wrong). To fix this issue use constructors with parameters or static contructors. Do not create AAB with Min=PositiveInfinity and Max=NegativeInfinity and then call Include(), due to speed optimizations in Include() methods this will not work properly.

`AAB3(ref Vector3 min, ref Vector3 max)`
`AAB3(Vector3 min, Vector3 max)`
Creates AAB from min and max points.

`AAB3(float xMin, float xMax, float yMin, float yMax, float zMin, float zMax)`
Creates AAB. The caller must ensure that xmin <= xmax, ymin <= ymax and zmin <= zmax.

`static AAB3 CreateFromPoint(ref Vector3 point)`
`static AAB3 CreateFromPoint(Vector3 point)`
Creates AAB from single point. Min and Max are set to point. Use Include() method to grow the resulting AAB.

`static AAB3 CreateFromTwoPoints(ref Vector3 point0, ref Vector3 point1)`
`static AAB3 CreateFromTwoPoints(Vector3 point0, Vector3 point1)`
Computes AAB from two points. In case min and max points are known, use constructor instead.

`static AAB3 CreateFromPoints(IEnumerable<Vector3> points)`
`static AAB3 CreateFromPoints(IList<Vector3> points)`
`static AAB3 CreateFromPoints(Vector3[] points)`
Computes AAB from a set of points. Method includes points from a set one by one to create the AAB.
If a set is empty, returns new AAB3().

`static implicit operator Bounds(AAB3 value)`
Converts AAB3 to UnityEngine.Bounds

`static implicit operator AAB3(Bounds value)`
Converts UnityEngine.Bounds to AAB3

`void CalcCenterExtents(out Vector3 center, out Vector3 extents)`
Computes box center and extents (half sizes)

```
void CalcVertices(Vector3 vertex0, out Vector3 vertex1, out Vector3 vertex2, out Vector3 vertex3,
Vector3 vertex4, out Vector3 vertex5, out Vector3 vertex6, out Vector3 vertex7)
```
Calculates 8 box corners.
vertex0 - Vector3(Min.x, Min.y, Min.z)
vertex1 - Vector3(Max.x, Min.y, Min.z)
vertex2 - Vector3(Max.x, Max.y, Min.z)
vertex3 - Vector3(Min.x, Max.y, Min.z)
vertex4 - Vector3(Min.x, Min.y, Max.z)
vertex5 - Vector3(Max.x, Min.y, Max.z)
vertex6 - Vector3(Max.x, Max.y, Max.z)
vertex7 - Vector3(Min.x, Max.y, Max.z)

```
Vector3[] CalcVertices()
```
Calculates 8 box corners and returns them in an allocated array.
See array-less overload for the description.

```
void CalcVertices(Vector3[] array)
```
Calculates 8 box corners and fills the input array with them (array length must be 8).
See array-less overload for the description.

```
float CalcVolume()
```
Returns box volume

```
float DistanceTo(Vector3 point)
```
Returns distance to a point, distance is >= 0f.

```
Vector3 Project(Vector3 point)
```
Returns projected point

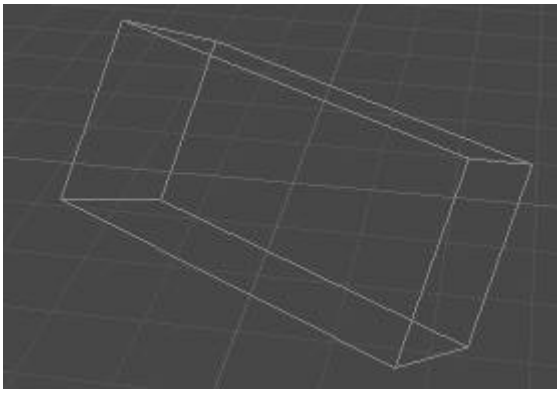
```
bool Contains(ref Vector3 point)
bool Contains(Vector3 point)
```
Tests whether a point is contained by the aab

```
void Include(ref Vector3 point)
void Include(Vector3 point)
```
Enlarging the box to include the point. If the point is inside the AAB does nothing.

```
void Include(ref AAB3 box)
void Include(AAB3 box)
```
Enlarges the aab so it includes another aab.

```
string ToString()
```
Returns string representation.

3.2.7 Box3



A box has center C, axis directions U[0], U[1], and U[2] (mutually perpendicular unit-length vectors), and extents e[0], e[1], and e[2] (all nonnegative numbers). A point X=C+y[0]*U[0]+y[1]*U[1]+y[2]*U[2] is inside or on the box whenever |y[i]|<=e[i] for all i.

Type
`struct Box3`
Fields
`Center`
`Axis0`
`Axis1`
`Axis2`
`Extents`
Construction
`Box3()`
`Box3(ref Vector3 center, ref Vector3 axis0, ref Vector3 axis1, ref Vector3 axis2, ref Vector3 extents)`
`Box3(Vector3 center, Vector3 axis0, Vector3 axis1, Vector3 axis2, Vector3 extents)`
`Box3(ref AAB3 box)`
`Box3(AAB3 box)`
`CreateFromPoints(IList<Vector3> points)`
Methods
`GetAxis(int index)`
`void CalcVertices(out Vector3 vertex0, out Vector3 vertex1, out Vector3 vertex2, out Vector3 vertex3, out Vector3 vertex4, out Vector3 vertex5, out Vector3 vertex6, out Vector3 vertex7)`
`CalcVertices()`
`CalcVertices(Vector3[] array)`
`CalcVolume()`
`DistanceTo(Vector3 point)`
`Project(Vector3 point)`
`Contains(ref Vector3 point)`
`Contains(Vector3 point)`
`Include(ref Box3 box)`
`Include(Box3 box)`
`ToString()`

`Vector3 Center`
Box center

```
Vector3 Axis0
```
First box axis. Must be unit length!

```
Vector3 Axis1
```
Second box axis. Must be unit length!

```
Vector3 Axis2
```
Third box axis. Must be unit length!

```
Vector3 Extents
```
Extents (half sizes) along Axis0, Axis1 and Axis2. Must be non-negative!

```
Box3()
```
Creates new Box3 instance. Users must fill all the fields manually.

```
Box3(ref Vector3 center, ref Vector3 axis0, ref Vector3 axis1, ref Vector3 axis2, ref Vector3 extents)
Box3(Vector3 center, Vector3 axis0, Vector3 axis1, Vector3 axis2, Vector3 extents)
```
Creates new Box3 instance.
center - Box center
axis0 - First box axis. Must be unit length!
axis1 - Second box axis. Must be unit length!
axis2 - Third box axis. Must be unit length!
extents - Extents (half sizes) along Axis0, Axis1 and Axis2. Must be non-negative!

```
Box3(ref AAB3 box)
Box3(AAB3 box)
```
Create Box3 from AxisAlignedBox3

```
static Box3 CreateFromPoints(IList<Vector3> points)
```
Computes oriented bounding box from a set of points. If a set is empty returns new Box3().

```
Vector3 GetAxis(int index)
```
Returns axis by index (0, 1, 2)

```
void CalcVertices(out Vector3 vertex0, out Vector3 vertex1, out Vector3 vertex2, out Vector3 vertex3, out
Vector3 vertex4, out Vector3 vertex5, out Vector3 vertex6, out Vector3 vertex7)
```
Calculates 8 box corners. extAxis[i] is Axis[i]*Extent[i], i=0,1,2
vertex0 - Center - extAxis0 - extAxis1 - extAxis2
vertex1 - Center + extAxis0 - extAxis1 - extAxis2
vertex2 - Center + extAxis0 + extAxis1 - extAxis2
vertex3 - Center - extAxis0 + extAxis1 - extAxis2
vertex4 - Center - extAxis0 - extAxis1 + extAxis2
vertex5 - Center + extAxis0 - extAxis1 + extAxis2
vertex6 - Center + extAxis0 + extAxis1 + extAxis2
vertex7 - Center - extAxis0 + extAxis1 + extAxis2

```
Vector3[] CalcVertices()
```
Calculates 8 box corners and returns them in an allocated array. See array-less overload for the description.

```
void CalcVertices(Vector3[] array)
```
Calculates 8 box corners and fills the input array with them (array length must be 8).
See array-less overload for the description.

```
float CalcVolume()
```
Returns volume of the box as Extents.x * Extents.y * Extents.z * 8

```
float DistanceTo(Vector3 point)
```
Returns distance to a point, distance is >= 0f.

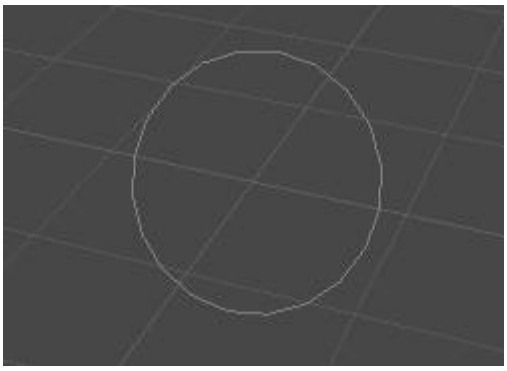
```
Vector3 Project(Vector3 point)
```
Returns projected point

```
bool Contains(ref Vector3 point)
bool Contains(Vector3 point)
```
Tests whether a point is contained by the box

```
void Include(ref Box3 box)
void Include(Box3 box)
```
Enlarges the box so it includes another box.

```
string ToString()
```
Returns string representation.

3.2.8 Circle3



Represents 2D circle embedded in 3D space. The plane containing the circle is Dot(N,X-C)=0, where X is any point in the plane. Vectors U, V, and N form an orthonormal set (matrix [U V N] is orthonormal and has determinant 1). The circle within the plane is parameterized by X=C+R*(cos(t)*U + sin(t)*V), where t is an angle in [0,2*pi).

Type
`struct Circle3`
Fields
`Center`
`Axis0`
`Axis1`
`Normal`
`Radius`
Construction
`Circle3()`
`Circle3(ref Vector3 center, ref Vector3 axis0, ref Vector3 axis1, float radius)`
`Circle3(Vector3 center, Vector3 axis0, Vector3 axis1, float radius)`
`Circle3(ref Vector3 center, ref Vector3 normal, float radius)`
`Circle3(Vector3 center, Vector3 normal, float radius)`
`static bool CreateCircumscribed(Vector3 v0, Vector3 v1, Vector3 v2, out Circle3 circle)`
`static bool CreateInscribed(Vector3 v0, Vector3 v1, Vector3 v2, out Circle3 circle)`
Methods
`float CalcPerimeter()`
`float CalcArea()`
`Vector3 Eval(float t)`
`Eval(float t, float radius)`
`DistanceTo(Vector3 point, bool solid)`
`Project(Vector3 point, bool solid)`
`ToString()`

`Vector3 Center`
Circle center.

`Vector3 Axis0`
First circle axis. Must be unit length!

`Vector3 Axis1`
Second circle axis. Must be unit length!

```
Vector3 Normal
```
Circle normal which is Cross(Axis0, Axis1). Must be unit length!

```
float Radius
```
Circle radius.

```
Circle3()
```
Creates Circle3 instance. Users must fill all the fields manually.

```
Circle3(ref Vector3 center, ref Vector3 axis0, ref Vector3 axis1, float radius)
Circle3(Vector3 center, Vector3 axis0, Vector3 axis1, float radius)
```
Creates new circle instance from center, axes and radius. Normal is calculated as cross product of the axes.
axis0 - Must be unit length!
axis1 - Must be unit length!

```
Circle3(ref Vector3 center, ref Vector3 normal, float radius)
Circle3(Vector3 center, Vector3 normal, float radius)
```
Creates new circle instance. Computes axes from specified normal.
normal - Must be unit length!

```
static bool CreateCircumscribed(Vector3 v0, Vector3 v1, Vector3 v2, out Circle3 circle)
```
Creates circle which is circumscribed around triangle. Returns 'true' if circle has been constructed, 'false' otherwise (input points are linearly dependent).

```
static bool CreateInscribed(Vector3 v0, Vector3 v1, Vector3 v2, out Circle3 circle)
```
Creates circle which is insribed into triangle. Returns 'true' if circle has been constructed, 'false' otherwise (input points are linearly dependent).

```
float CalcPerimeter()
```
Returns circle perimeter

```
float CalcArea()
```
Returns circle area

```
Vector3 Eval(float t)
```
Evaluates circle using formula X = C + R*cos(t)*U + R*sin(t)*V where t is an angle in [0,2*pi).
t - Evaluation parameter

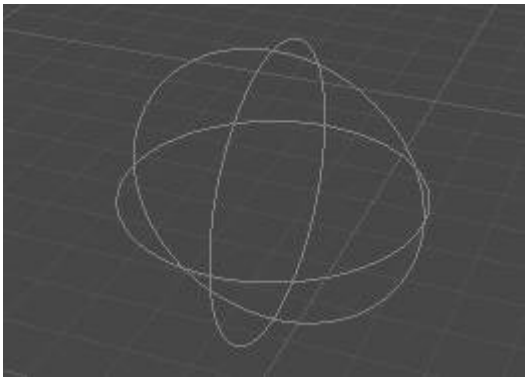
```
Vector3 Eval(float t, float radius)
```
Evaluates disk using formula X = C + radius*cos(t)*U + radius*sin(t)*V where t is an angle in [0,2*pi).
t - Evaluation parameter
radius - Evaluation radius

```
float DistanceTo(Vector3 point, bool solid = true)
```
Returns distance to a point, distance is >= 0f.

```
Vector3 Project(Vector3 point, bool solid = true)
```
Returns projected point

```
string ToString()
```
Returns string representation.

3.2.9 Sphere3



The sphere is represented as |X-C| = R where C is the center and R is the radius.

Type
struct Sphere3
Fields
Center
Radius
Construction
Sphere3()
Sphere3(ref Vector3 center, float radius)
Sphere3(Vector3 center, float radius)
CreateFromPointsAAB(IEnumerable<Vector3> points)
CreateFromPointsAAB(IList<Vector3> points)
CreateFromPointsAverage(IEnumerable<Vector3> points)
CreateFromPointsAverage(IList<Vector3> points)
CreateCircumscribed(Vector3 v0, Vector3 v1, Vector3 v2, Vector3 v3, out Sphere3 sphere)
CreateInscribed(Vector3 v0, Vector3 v1, Vector3 v2, Vector3 v3, out Sphere3 sphere)
Methods
CalcArea()
CalcVolume()
Eval(float theta, float phi)
DistanceTo(Vector3 point)
Project(Vector3 point)
Contains(ref Vector3 point)
Contains(Vector3 point)
Include(ref Sphere3 sphere)
Include(Sphere3 sphere)
ToString()

`Vector3 Center`
Sphere center

`float Radius`
Sphere radius

```
Sphere3()
```
Creates Sphere3 instance. Users must fill center and radius manually.

```
Sphere3(ref Vector3 center, float radius)
Sphere3(Vector3 center, float radius)
```
Creates Sphere3 from center and radius

```
static Sphere3 CreateFromPointsAAB(IEnumerable<Vector3> points)
static Sphere3 CreateFromPointsAAB(IList<Vector3> points)
```
Computes bounding sphere from a set of points. First compute the axis-aligned bounding box of the points, then compute the sphere containing the box. If a set is empty returns new Sphere3().

```
static Sphere3 CreateFromPointsAverage(IEnumerable<Vector3> points)
static Sphere3 CreateFromPointsAverage(IList<Vector3> points)
```
Computes bounding sphere from a set of points. Compute the smallest sphere whose center is the average of a point set. If a set is empty returns new Sphere3().

```
static bool CreateCircumscribed(Vector3 v0, Vector3 v1, Vector3 v2, Vector3 v3, out Sphere3 sphere)
```
Creates sphere which is circumscribed around tetrahedron. Returns 'true' if sphere has been constructed, 'false' otherwise (input points are linearly dependent).

```
static bool CreateInscribed(Vector3 v0, Vector3 v1, Vector3 v2, Vector3 v3, out Sphere3 sphere)
```
Creates sphere which is insribed into tetrahedron. Returns 'true' if sphere has been constructed, 'false' otherwise (input points are linearly dependent).

```
float CalcArea()
```
Returns sphere area

```
float CalcVolume()
```
Returns sphere volume

```
Vector3 Eval(float theta, float phi)
```
Evaluates sphere using formula X = C + R*[cos(theta)*sin(phi) , sin(theta)*sin(phi) , cos(phi)], where 0 <= theta,phi < 2*pi.

```
float DistanceTo(Vector3 point)
```
Returns distance to a point, distance is >= 0f.

```
Vector3 Project(Vector3 point)
```
Returns projected point

```
bool Contains(ref Vector3 point)
bool Contains(Vector3 point)
```
Tests whether a point is contained by the sphere

```
void Include(ref Sphere3 sphere)
void Include(Sphere3 sphere)
```
Enlarges the sphere so it includes another sphere.

```
string ToString()
```
Returns string representation.

3.2.10 Triangle3

Triangle in 3D.

Type
struct Triangle3
Fields
V0
V1
V2
Properties
this[int index] { get; set; }
Construction
Triangle3()
Triangle3(ref Vector3 v0, ref Vector3 v1, ref Vector3 v2)
Triangle3(Vector3 v0, Vector3 v1, Vector3 v2)
Methods
CalcEdge(int edgeIndex)
CalcNormal()
CalcArea()
CalcArea(ref Vector3 v0, ref Vector3 v1, ref Vector3 v2)
CalcArea(Vector3 v0, Vector3 v1, Vector3 v2)
CalcAnglesDeg()
CalcAnglesDeg(ref Vector3 v0, ref Vector3 v1, ref Vector3 v2)
CalcAnglesDeg(Vector3 v0, Vector3 v1, Vector3 v2)
CalcAnglesRad()
CalcAnglesRad(ref Vector3 v0, ref Vector3 v1, ref Vector3 v2)
CalcAnglesRad(Vector3 v0, Vector3 v1, Vector3 v2)
EvalBarycentric(float c0, float c1)
EvalBarycentric(ref Vector3 baryCoords)
EvalBarycentric(Vector3 baryCoords)
CalcBarycentricCoords(ref Vector3 point, ref Vector3 v0, ref Vector3 v1, ref Vector3 v2, out Vector3 baryCoords)
CalcBarycentricCoords(ref Vector3 point)
CalcBarycentricCoords(Vector3 point)
ToString()

```
Vector3 V0
```
First triangle vertex

```
Vector3 V1
```
Second triangle vertex

```
Vector3 V2
```
Third triangle vertex

```
Vector3 this[int index] { get; set; }
```
Gets or sets triangle vertex by index: 0, 1 or 2

```
Triangle3()
```
Creates Triangle3 instance. Users must fill vertices manually.

```
Triangle3(ref Vector3 v0, ref Vector3 v1, ref Vector3 v2)
```
Creates Triangle3 from 3 vertices

```
Triangle3(Vector3 v0, Vector3 v1, Vector3 v2)
```
Creates Triangle3 from 3 vertices

```
Vector3 CalcEdge(int edgeIndex)
```
Returns triangle edge by index 0, 1 or 2
Edge[i] = V[i+1]-V[i]

```
Vector3 CalcNormal()
```
Returns triangle normal as (V1-V0)x(V2-V0)

```
float CalcArea()
```
Returns triangle area as 0.5*Abs(Length((V1-V0)x(V2-V0)))

```
static float CalcArea(ref Vector3 v0, ref Vector3 v1, ref Vector3 v2)
static float CalcArea(Vector3 v0, Vector3 v1, Vector3 v2)
```
Returns triangle area defined by 3 points.

```
Vector3 CalcAnglesDeg()
```
Calculates angles of the triangle in degrees.
Angles are returned in the instance of Vector3 following way: (angle of vertex V0, angle of vertex V1, angle of vertex V2)

```
static Vector3 CalcAnglesDeg(ref Vector3 v0, ref Vector3 v1, ref Vector3 v2)
static Vector3 CalcAnglesDeg(Vector3 v0, Vector3 v1, Vector3 v2)
```
Calculates angles of the triangle defined by 3 points in degrees.
Angles are returned in the instance of Vector3 following way: (angle of vertex V0, angle of vertex V1, angle of vertex V2)

```
Vector3 CalcAnglesRad()
```
Calculates angles of the triangle in radians.
Angles are returned in the instance of Vector3 following way: (angle of vertex V0, angle of vertex V1, angle of vertex V2)

```
static Vector3 CalcAnglesRad(ref Vector3 v0, ref Vector3 v1, ref Vector3 v2)
static Vector3 CalcAnglesRad(Vector3 v0, Vector3 v1, Vector3 v2)
```
Calculates angles of the triangle defined by 3 points in radians.
Angles are returned in the instance of Vector3 following way: (angle of vertex V0, angle of vertex V1, angle of vertex V2)

```
Vector3 EvalBarycentric(float c0, float c1)
```
Gets point on the triangle using barycentric coordinates.
The result is c0*V0 + c1*V1 + c2*V2, 0 <= c0,c1,c2 <= 1, c0+c1+c2=1, c2 is calculated as 1-c0-c1.

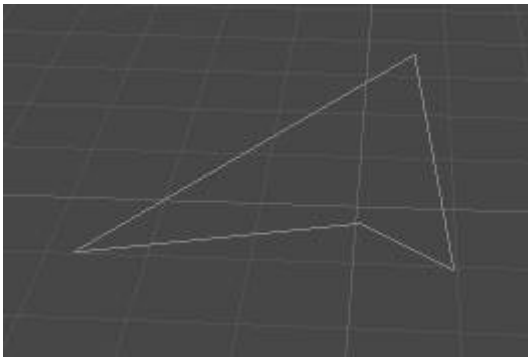
```
Vector3 EvalBarycentric(ref Vector3 baryCoords)
Vector3 EvalBarycentric(Vector3 baryCoords)
```
Gets point on the triangle using barycentric coordinates. baryCoords parameter is (c0,c1,c2).
The result is c0*V0 + c1*V1 + c2*V2, 0 <= c0,c1,c2 <= 1, c0+c1+c2=1

```
static void CalcBarycentricCoords(ref Vector3 point, ref Vector3 v0, ref Vector3 v1, ref Vector3 v2, out
Vector3 baryCoords)
```
Calculate barycentric coordinates for the input point with regarding to triangle defined by 3 points.

```
Vector3 CalcBarycentricCoords(ref Vector3 point)
Vector3 CalcBarycentricCoords(Vector3 point)
```
Calculate barycentric coordinates for the input point regarding to the triangle.

```
string ToString()
```
Returns string representation.

3.2.11 Polygon3



Represents 3d planar polygon (vertex count must be >= 3). Polygon has single border. Many algorithms also require polygon to be convex, it's responsibility of a user to provide such polygons. Polygon contains vertices and edges. Vertex is a simple vector, edge is described below. Notice that Polygon3 is a class not struct.

Test Prefab:
Test_Polygon3

Type
`struct Edge3`
Fields
`Vector3 Point0` Edge start vertex
`Vector3 Point1` Edge end vertex
`Vector3 Direction` Unit length direction vector
`Vector3 Normal` Unit length normal vector
`float Length` Edge length

Type
`class Polygon3`
Properties
`Vertices { get; }`
`Edges { get; }`
`VertexCount { get; }`
`this[int vertexIndex] { get; set; }`
`Plane { get; set; }`
Construction
`Polygon3()`
`Polygon3(Vector3[] vertices, Plane3 plane)`
`Polygon3(int vertexCount, Plane3 plane)`
Methods
`SetVertexProjected(int vertexIndex, Vector3 vertex)`
`ProjectVertices()`
`GetEdge(int edgeIndex)`
`UpdateEdges()`
`UpdateEdge(int edgeIndex)`

`CalcCenter()`
`CalcPerimeter()`
`HasZeroCorners(float threshold)`
`ReverseVertices()`
`ToSegmentArray()`
`ToString()`

`Vector3[] Vertices { get; }`
Gets vertices array (do not change the data, use only for traversal)

`Edge3[] Edges { get; }`
Gets edges array (do not change the data, use only for traversal)

`int VertexCount { get; }`
Polygon vertex count

`Vector3 this[int vertexIndex] { get; set; }`
Gets or sets polygon vertex. The caller is responsible for supplying the points which lie in the polygon's plane.

`Plane3 Plane { get; set; }`
Gets or sets polygon plane. After plane change reset all vertices manually or call ProjectVertices() to project all vertices automatically.

`Polygon3()`
Default constructor is unavailable to the users. Use constructors with parameters.

`Polygon3(Vector3[] vertices, Plane3 plane)`
Creates polygon from an array of vertices (array is copied). The caller is responsible for supplying the points which lie in the polygon's plane.

`Polygon3(int vertexCount, Plane3 plane)`
Creates polygon setting number of vertices. Vertices then can be filled using indexer.

`void SetVertexProjected(int vertexIndex, Vector3 vertex)`
Sets polygon vertex and ensures that it will lie in the plane by projecting it.

`void ProjectVertices()`
Projects polygon vertices onto polygon plane.

`Edge3 GetEdge(int edgeIndex)`
Returns polygon edge

`void UpdateEdges()`
Updates all polygon edges. Use after vertex change.

`void UpdateEdge(int edgeIndex)`
Updates certain polygon edge. Use after vertex change.

`Vector3 CalcCenter()`
Returns polygon center

```
float CalcPerimeter()
```
Returns polygon perimeter length

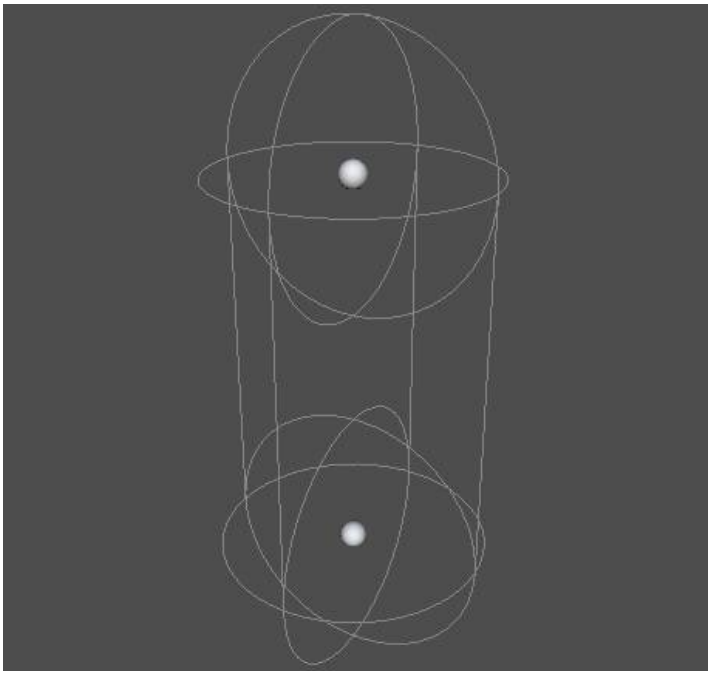
```
bool HasZeroCorners(float threshold = MathfEx.ZeroTolerance)
```
Returns true if polygon contains some edges which have zero angle between them.

```
void ReverseVertices()
```
Reverses polygon vertex order

```
Segment3[] ToSegmentArray()
```
Converts the polygon to segment array

```
string ToString()
```
Returns string representation.

3.2.12 Capsule3



Capsule is defined by the volume around the segment with certain radius.

Type
`struct Capsule3`
Fields
`Segment`
`Radius`
Construction
`Capsule3()`
`Capsule3(ref Segment3 segment, float radius)`
`Capsule3(Segment3 segment, float radius)`

`Segment3 Segment`
Capsule base segment

`float Radius`
Capsule radius

`Capsule3()`
Creates Capsule3 instance. Users must fill segment and radius manually.

`Capsule3(ref Segment3 segment, float radius)`
`Capsule3(Segment3 segment, float radius)`
Creates Capsule3 from segment and radius

4 Intersection

This chapter describes various intersection methods. Intersection goes as follow: on the input there are two primitive objects, on the output – flag indiciating whether the intersection has occurred and optionally information about the intersection case. There are two types of intersection methods in the library – “test” and “find”. “Test” methods only tell whether there is an intersection, while “find” method also give information (e.g. points of intersection). Test methods are usually faster (except the cases when “test” method is the same as “find” method). As “find” methods are generally harder to implement, there are cases where only “test” routine is available for the certain primitives. Every intersection method is contained inside static class `Intersection`. Most of the “find” methods return intersection data as a struct. Also all intersection methods are accompanied with test prefabs which are situated inside *Tests\Prefabs\Intersection* folder. They show examples of using intersection methods and output data from those methods.

Large amount of methods return intersection type between the primitives among other data. These types are collected inside `IntersectionTypes` enumeration. All methods could be divided into 2D or 3D group which are described in the subsequent sections, except for the intersection of 1D intervals which is described in this section. Also all 2D or 3D intersection methods return `bool` value as the result. It's equal to `true` when there is an intersection between primitives `false` otherwise.

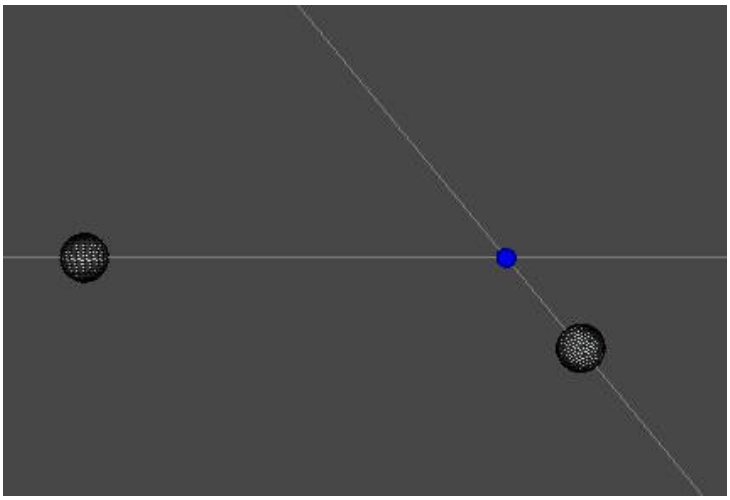
Type
`class Intersection`
Methods
`static int FindSegment1Segment1(float seg0Start, float seg0End, float seg1Start, float seg1End, out float w0, out float w1)` Finds intersection of 1d intervals. Endpoints of the intervals must be sorted, i.e. seg0Start must be <= seg0End, seg1Start must be <= seg1End. Returns 0 if intersection is empty, 1 - if intervals intersect in one point, 2 - if intervals intersect in segment. w0 and w1 will contain intersection point in case intersection occurs.

Type
`enum IntersectionTypes`
Members
`Empty` Entities do not intersect
`Point` Entities intersect in a point
`Segment` Entities intersect in a segment
`Ray` Entities intersect in a ray
`Line` Entities intersect in a line
`Polygon` Entities intersect in a polygon

`Plane` Entities intersect in a plane
`Polyhedron` Entities intersect in a polyhedron
`Other` Entities intersect somehow

4.1 2D Intersection

4.1.1 Line2-Line2



Test Prefab:
Test_IntrLine2Line2

Type
`class Intersection`
Methods

`static bool TestLine2Line2(ref Line2 line0, ref Line2 line1, out IntersectionTypes intersectionType)`
`static bool TestLine2Line2(ref Line2 line0, ref Line2 line1)`
Tests whether two lines intersect. Returns true if intersection occurs (IntersectionTypes.Point, IntersectionTypes.Line), or false if lines do not intersect (IntersectionTypes.Empty).

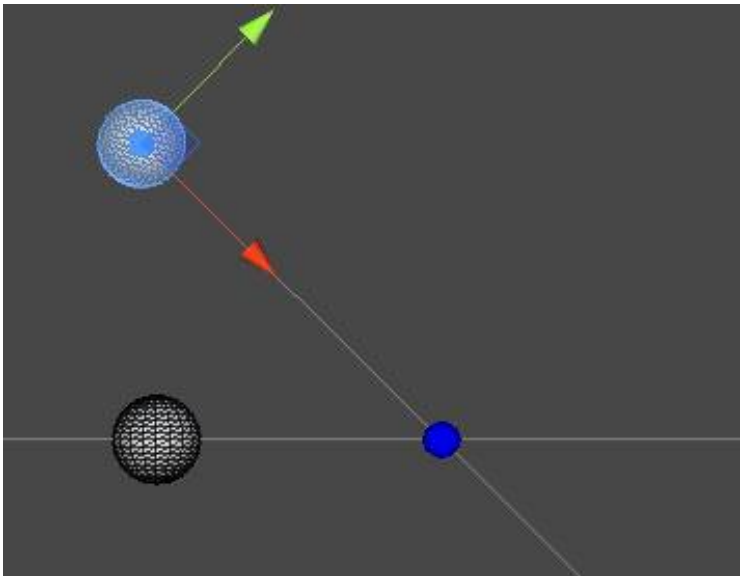
`static bool FindLine2Line2(ref Line2 line0, ref Line2 line1, out Line2Line2Intr info)` Tests whether two lines intersect and finds actual intersection parameters. Returns true if intersection occurs (IntersectionTypes.Point, IntersectionTypes.Line), or false if lines do not intersect (IntersectionTypes.Empty).

Type
`struct Line2Line2Intr` Contains information about intersection of two Line2
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point or IntersectionTypes.Line (lines are the same) if intersection occured otherwise IntersectionTypes.Empty
`Vector2 Point` In case of IntersectionTypes.Point constains single point of intersection. Otherwise Vector2.zero.
`float Parameter` In case of IntersectionTypes.Point contains evaluation parameter of single intersection point according to first line. Otherwise 0.

Example

```
IntersectionTypes intersectionType;
bool test = Intersection.TestLine2Line2(ref line0, ref line1, out intersectionType);
Line2Line2Intr info;
bool find = Intersection.FindLine2Line2(ref line0, ref line1, out info);
```

4.1.2 Line2-Ray2



Test Prefab:
Test_IntrLine2Ray2

Type
`class Intersection`
Methods

`static bool TestLine2Ray2(ref Line2 line, ref Ray2 ray, out IntersectionTypes intersectionType)`
`static bool TestLine2Ray2(ref Line2 line, ref Ray2 ray)`
Tests whether line and ray intersect. Returns true if intersection occurs (IntersectionTypes.Point, IntersectionTypes.Ray), or false if line and ray do not intersect (IntersectionTypes.Empty).

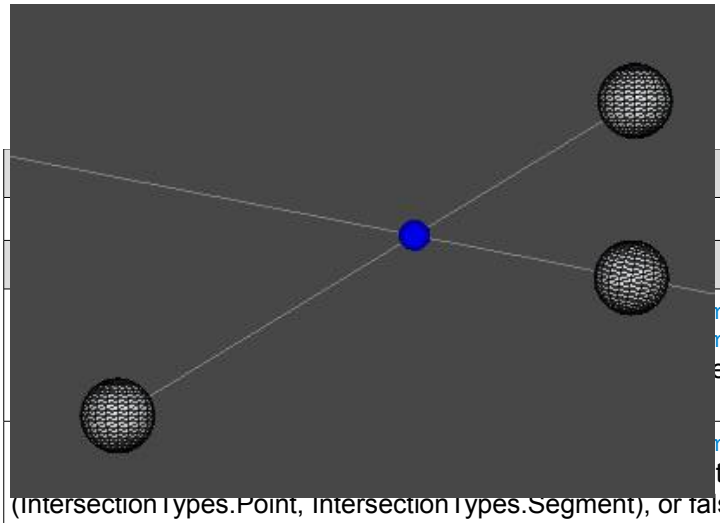
`static bool FindLine2Ray2(ref Line2 line, ref Ray2 ray, out Line2Ray2Intr info)` Tests whether line and ray intersect and finds actual intersection parameters. Returns true if intersection occurs (IntersectionTypes.Point, IntersectionTypes.Ray), or false if line and ray do not intersect (IntersectionTypes.Empty).

Type
`struct Line2Ray2Intr` Contains information about intersection of Line2 and Ray2
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point or IntersectionTypes.Segment (line and ray are collinear) if intersection occured otherwise IntersectionTypes.Empty
`Vector2 Point` In case of IntersectionTypes.Point constains single point of intersection. Otherwise Vector2.zero.
`float Parameter` In case of IntersectionTypes.Point contains evaluation parameter of single intersection point according to line. Otherwise 0.

Example

```
IntersectionTypes intersectionType;
bool test = Intersection.TestLine2Ray2(ref line, ref ray, out intersectionType);
Line2Ray2Intr info;
bool find = Intersection.FindLine2Ray2(ref line, ref ray, out info);
```

4.1.3 Line2-Segment2



Test Prefab:
Test_IntrLine2Segment2

`ment2 segment, out IntersectionTypes intersectionType)`
`ment2 segment)`
`ersection occurs (IntersectionTypes.Point,`
`not intersect (IntersectionTypes.Empty).`

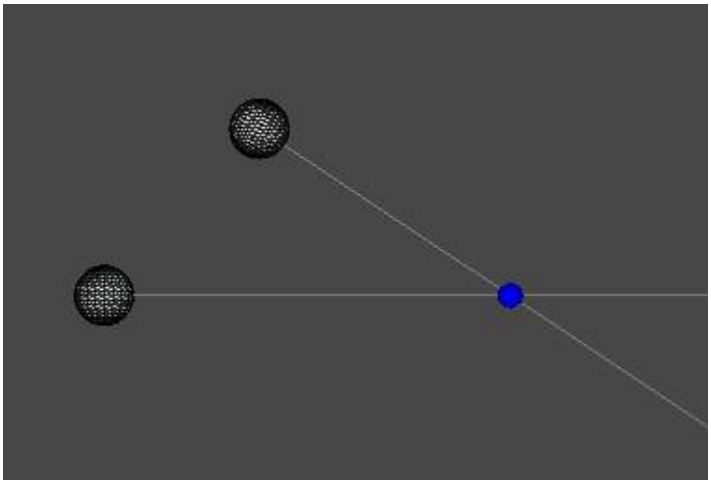
`ment2 segment, out Line2Segment2Intr info)` `tersection parameters. Returns true if intersection occurs (IntersectionTypes.Point, IntersectionTypes.Segment), or false if line and segment do not intersect (IntersectionTypes.Empty).

Type
`struct Line2Segment2Intr` Contains information about intersection of Line2 and Segment2
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point or IntersectionTypes.Segment (line and segment are collinear) if intersection occured otherwise IntersectionTypes.Empty
`Vector2 Point` In case of IntersectionTypes.Point constains single point of intersection. Otherwise Vector2.zero.
`float Parameter` In case of IntersectionTypes.Point contains evaluation parameter of single intersection point according to line. Otherwise 0.

Example

```
IntersectionTypes intersectionType;
bool test = Intersection.TestLine2Segment2(ref line, ref segment, out intersectionType);
Line2Segment2Intr info;
bool find = Intersection.FindLine2Segment2(ref line, ref segment, out info);
```

4.1.4 Ray2-Ray2



Test Prefab:
Test_IntrRay2Ray2

Type
`class Intersection`
Methods

`static bool TestRay2Ray2(ref Ray2 ray0, ref Ray2 ray1, out IntersectionTypes intersectionType)`
`static bool TestRay2Ray2(ref Ray2 ray0, ref Ray2 ray1)`
Tests whether two rays intersect. Returns true if intersection occurs (IntersectionTypes.Point, IntersectionTypes.Ray), or false if rays do not intersect (IntersectionTypes.Empty).

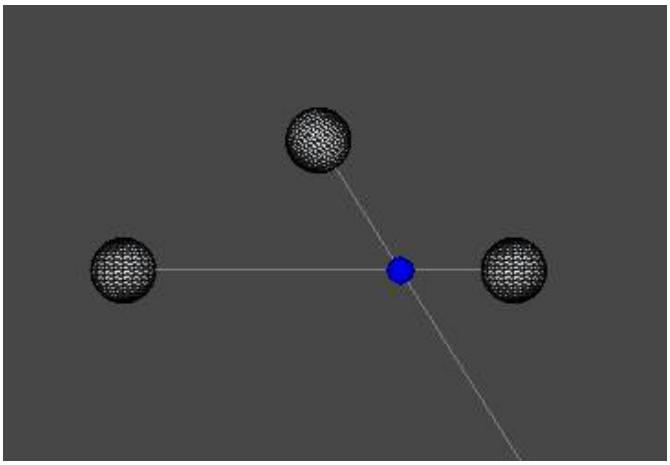
`static bool FindRay2Ray2(ref Ray2 ray0, ref Ray2 ray1, out Ray2Ray2Intr info)` Tests whether two rays intersect and finds actual intersection parameters. Returns true if intersection occurs (IntersectionTypes.Point, IntersectionTypes.Ray), or false if rays do not intersect (IntersectionTypes.Empty).

Type
`struct Ray2Ray2Intr` Contains information about intersection of two Ray2
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point or IntersectionTypes.Ray (rays are collinear and overlap in more than one point) if intersection occured otherwise IntersectionTypes.Empty
`Vector2 Point` In case of IntersectionTypes.Point constains single point of intersection. In case of IntersectionTypes.Ray contains second ray's origin. Otherwise Vector2.zero.
`float Parameter` In case of IntersectionTypes.Point contains evaluation parameter of single intersection point according to first ray. In case of IntersectionTypes.Ray contains evaluation parameter of the second ray's origin according to first ray. Otherwise 0.

Example

```
IntersectionTypes intersectionType;
bool test = Intersection.TestRay2Ray2(ref ray0, ref ray1, out intersectionType);
Ray2Ray2Intr info;
bool find = Intersection.FindRay2Ray2(ref ray0, ref ray1, out info);
```

4.1.5 Ray2-Segment2



Test Prefab:
Test_IntrRay2Segment2

Type
`class Intersection`
Methods

`static bool TestRay2Segment2(ref Ray2 ray, ref Segment2 segment, out IntersectionTypes intersectionType)`
`static bool TestRay2Segment2(ref Ray2 ray, ref Segment2 segment)`
Tests whether ray and segment intersect. Returns true if intersection occurs (IntersectionTypes.Point, IntersectionTypes.Segment), or false if ray and segment do not intersect (IntersectionTypes.Empty).

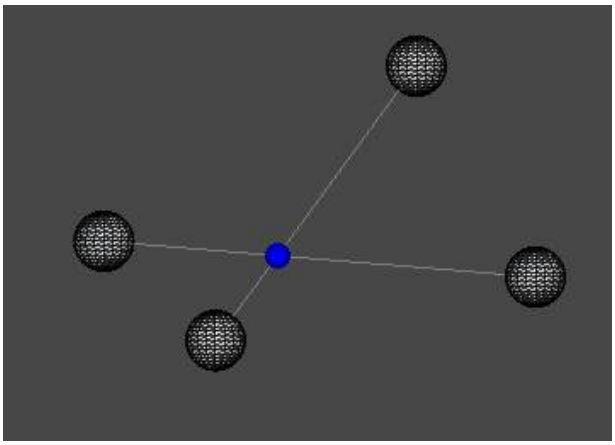
`static bool FindRay2Segment2(ref Ray2 ray, ref Segment2 segment, out Ray2Segment2Intr info)` Tests whether ray and segment intersect and finds actual intersection parameters. Returns true if intersection occurs (IntersectionTypes.Point, IntersectionTypes.Segment), or false if ray and segment do not intersect (IntersectionTypes.Empty).

Type
`struct Ray2Segment2Intr` Contains information about intersection of Ray2 and Segment2
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point or IntersectionTypes.Segment (ray and segment are collinear and overlap in more than one point) if intersection occured otherwise IntersectionTypes.Empty
`Vector2 Point0` In case of IntersectionTypes.Point constains single point of intersection. In case of IntersectionTypes.Segment contains first point of intersection. Otherwise Vector2.zero.
`Vector2 Point1` In case of IntersectionTypes.Segment contains second point of intersection. Otherwise Vector2.zero.
`float Parameter0` In case of IntersectionTypes.Point contains evaluation parameter of single intersection point according to ray. In case of IntersectionTypes.Segment contains evaluation parameter of the first intersection point according to ray. Otherwise 0.
`float Parameter1` In case of IntersectionTypes.Segment contains evaluation parameter of the second intersection point according to ray. Otherwise 0.

Example

```
IntersectionTypes intersectionType;
bool test = Intersection.TestRay2Segment2(ref ray, ref segment, out intersectionType);
Ray2Segment2Intr info;
bool find = Intersection.FindRay2Segment2(ref ray, ref segment, out info);
```

4.1.6 Segment2-Segment2



Test Prefab:
Test_IntrSegment2Segment2

Type
`class Intersection`
Methods

`static bool TestSegment2Segment2(ref Segment2 segment0, ref Segment2 segment1, out IntersectionTypes intersectionType)`
`static bool TestSegment2Segment2(ref Segment2 segment0, ref Segment2 segment1)`
Tests whether two segments intersect. Returns true if intersection occurs (IntersectionTypes.Point, IntersectionTypes.Segment), or false if segments do not intersect (IntersectionTypes.Empty).

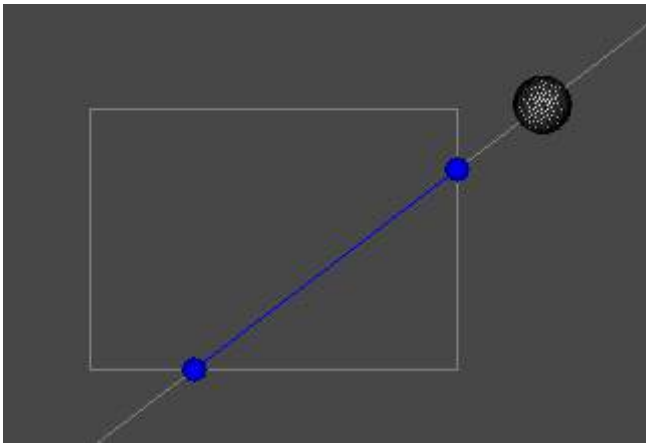
`static bool FindSegment2Segment2(ref Segment2 segment0, ref Segment2 segment1, out Segment2Segment2Intr info)` Tests whether two segments intersect and finds actual intersection parameters. Returns true if intersection occurs (IntersectionTypes.Point, IntersectionTypes.Segment), or false if segments do not intersect (IntersectionTypes.Empty).

Type
`struct Segment2Segment2Intr` Contains information about intersection of two Segment2.
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point or IntersectionTypes.Segment (segments are collinear and overlap in more than one point) if intersection occured otherwise IntersectionTypes.Empty.
`Vector2 Point0` In case of IntersectionTypes.Point contains single point of intersection. In case of IntersectionTypes.Segment contains first point of intersection. Otherwise Vector2.zero.
`Vector2 Point1` In case of IntersectionTypes.Segment contains second point of intersection. Otherwise Vector2.zero.
`float Parameter0` In case of IntersectionTypes.Point contains evaluation parameter of single intersection point according to first segment. In case of IntersectionTypes.Segment contains evaluation parameter of the first intersection point according to first segment. Otherwise 0.
`float Parameter1` In case of IntersectionTypes.Segment contains evaluation parameter of the second intersection point according to first segment. Otherwise 0.

Example

```
IntersectionTypes intersectionType;
bool test = Intersection.TestSegment2Segment2(ref segment0, ref segment1, out intersectionType);
Segment2Segment2Intr info;
bool find = Intersection.FindSegment2Segment2(ref segment0, ref segment1, out info);
```

4.1.7 Line2-AAB2



Test Prefab:
Test_IntrLine2AAB2

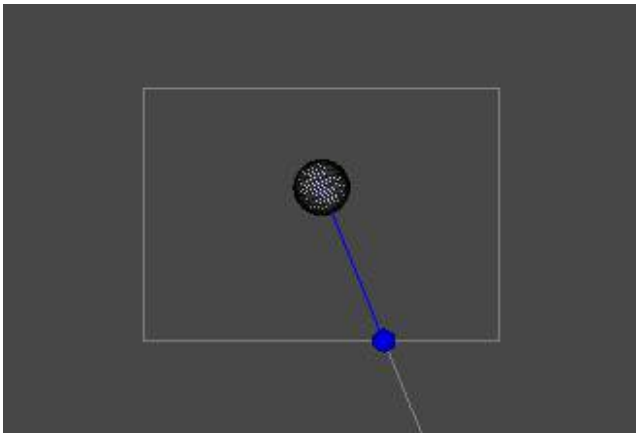
Type
`class Intersection`
Methods
`static bool TestLine2AAB2(ref Line2 line, ref AAB2 box)` Tests if a line intersects an axis aligned box. Returns true if intersection occurs false otherwise.
`static bool FindLine2AAB2(ref Line2 line, ref AAB2 box, out Line2AAB2Intr info)` Tests if a line intersects an axis aligned box and finds intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Line2AAB2Intr` Contains information about intersection of Line2 and AxisAlignedBox2
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point or IntersectionTypes.Segment if intersection occured otherwise IntersectionTypes.Empty
`int Quantity` Number of intersection points. IntersectionTypes.Empty: 0; IntersectionTypes.Point: 1; IntersectionTypes.Segment: 2.
`Vector2 Point0` First intersection point
`Vector2 Point1` Second intersection point

Example

```
bool test = Intersection.TestLine2AAB2(ref line, ref box);
Line2AAB2Intr info;
bool find = Intersection.FindLine2AAB2(ref line, ref box, out info);
```

4.1.8 Ray2-AAB2



Test Prefab:
Test_IntrRay2AAB2

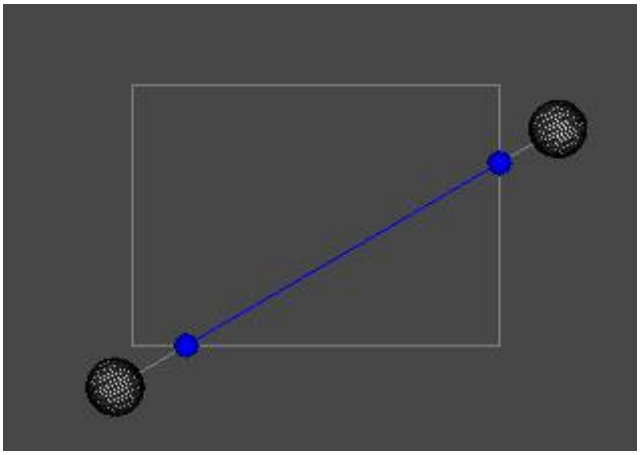
Type
`class Intersection`
Methods
`static bool TestRay2AAB2(ref Ray2 ray, ref AAB2 box)` Tests if a ray intersects an axis aligned box. Returns true if intersection occurs false otherwise.
`static bool FindRay2AAB2(ref Ray2 ray, ref AAB2 box, out Ray2AAB2Intr info)` Tests if a ray intersects an axis aligned box and finds intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Ray2AAB2Intr` Contains information about intersection of Line2 and AxisAlignedBox2
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point or IntersectionTypes.Segment if intersection occured otherwise IntersectionTypes.Empty
`int Quantity` Number of intersection points. IntersectionTypes.Empty: 0; IntersectionTypes.Point: 1; IntersectionTypes.Segment: 2.
`Vector2 Point0` First intersection point
`Vector2 Point1` Second intersection point

Example

```
bool test = Intersection.TestRay2AAB2(ref ray, ref box);
Ray2AAB2Intr info;
bool find = Intersection.FindRay2AAB2(ref ray, ref box, out info);
```

4.1.9 Segment2-AAB2



Test Prefab:
Test_IntrSegment2AAB2

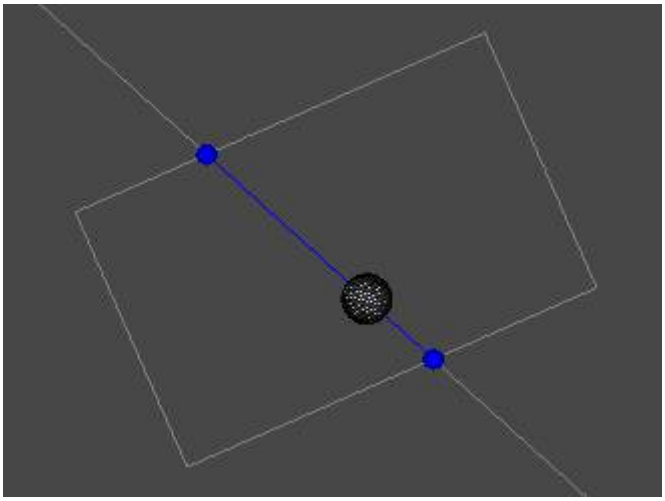
Type
`class Intersection`
Methods
`static bool TestSegment2AAB2(ref Segment2 segment, ref AAB2 box)` Tests if a segment intersects an axis aligned box. Returns true if intersection occurs false otherwise.
`static bool FindSegment2AAB2(ref Segment2 segment, ref AAB2 box, out Segment2AAB2Intr info)` Tests if a segment intersects an axis aligned box and finds intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Segment2AAB2Intr` Contains information about intersection of Line2 and AxisAlignedBox2
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point or IntersectionTypes.Segment if intersection occured otherwise IntersectionTypes.Empty
`int Quantity` Number of intersection points. IntersectionTypes.Empty: 0; IntersectionTypes.Point: 1; IntersectionTypes.Segment: 2.
`Vector2 Point0` First intersection point
`Vector2 Point1` Second intersection point

Example

```
bool test = Intersection.TestSegment2AAB2(ref segment, ref box);
Segment2AAB2Intr info;
bool find = Intersection.FindSegment2AAB2(ref segment, ref box, out info);
```

4.1.10 Line2-Box2



Test Prefab:
Test_IntrLine2Box2

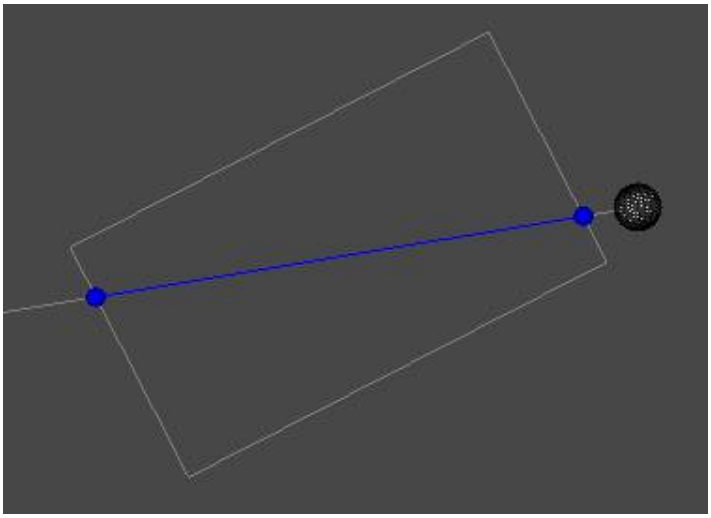
Type
`class Intersection`
Methods
`static bool TestLine2Box2(ref Line2 line, ref Box2 box)` Tests whether line and box intersect. Returns true if intersection occurs false otherwise.
`static bool FindLine2Box2(ref Line2 line, ref Box2 box, out Line2Box2Intr info)` Tests whether line and box intersect and finds actual intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Line2Box2Intr` Contains information about intersection of Line2 and Box2
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point or IntersectionTypes.Segment if intersection occured otherwise IntersectionTypes.Empty
`int Quantity` Number of intersection points. IntersectionTypes.Empty: 0; IntersectionTypes.Point: 1; IntersectionTypes.Segment: 2.
`Vector2 Point0` First intersection point
`Vector2 Point1` Second intersection point

Example

```
bool test = Intersection.TestLine2Box2(ref line, ref box);
Line2Box2Intr info;
bool find = Intersection.FindLine2Box2(ref line, ref box, out info);
```

4.1.11 Ray2-Box2



Test Prefab:
Test_IntrRay2Box2

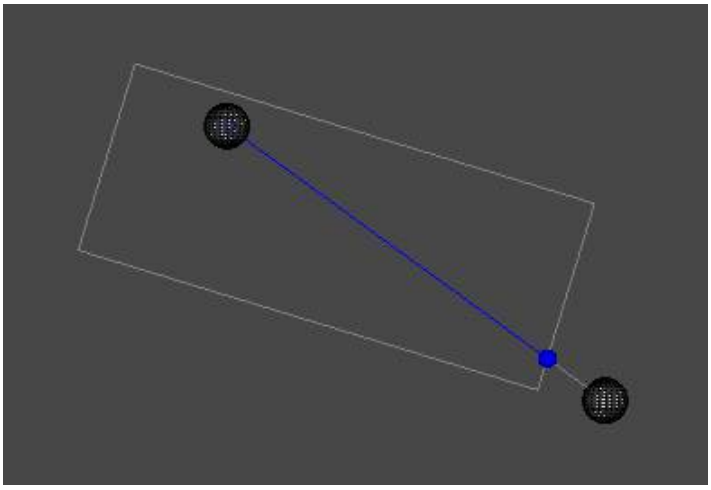
Type
`class Intersection`
Methods
`static bool TestRay2Box2(ref Ray2 ray, ref Box2 box)` Tests whether ray and box intersect. Returns true if intersection occurs false otherwise.
`static bool FindRay2Box2(ref Ray2 ray, ref Box2 box, out Ray2Box2Intr info)` Tests whether ray and box intersect and finds actual intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Ray2Box2Intr` Contains information about intersection of Ray2 and Box2
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point or IntersectionTypes.Segment if intersection occured otherwise IntersectionTypes.Empty
`int Quantity` Number of intersection points. IntersectionTypes.Empty: 0; IntersectionTypes.Point: 1; IntersectionTypes.Segment: 2.
`Vector2 Point0` First intersection point
`Vector2 Point1` Second intersection point

Example

```
bool test = Intersection.TestRay2Box2(ref ray, ref box);
Ray2Box2Intr info;
bool find = Intersection.FindRay2Box2(ref ray, ref box, out info);
```

4.1.12 Segment2-Box2



Test Prefab:
Test_IntrSegment2Box2

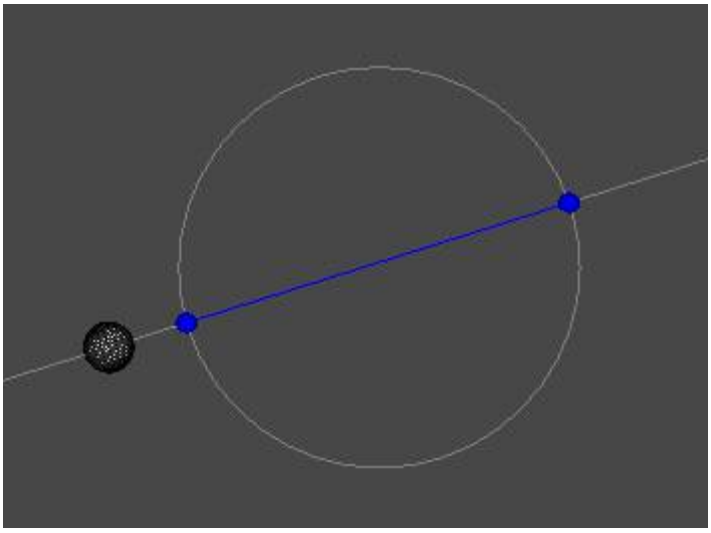
Type
`class Intersection`
Methods
`static bool TestSegment2Box2(ref Segment2 segment, ref Box2 box)` Tests whether segment and box intersect. Returns true if intersection occurs false otherwise.
`static bool FindSegment2Box2(ref Segment2 segment, ref Box2 box, out Segment2Box2Intr info)` Tests whether segment and box intersect and finds actual intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Segment2Box2Intr` Contains information about intersection of Segment2 and Box2
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point or IntersectionTypes.Segment if intersection occured otherwise IntersectionTypes.Empty
`int Quantity` Number of intersection points. IntersectionTypes.Empty: 0; IntersectionTypes.Point: 1; IntersectionTypes.Segment: 2.
`Vector2 Point0` First intersection point
`Vector2 Point1` Second intersection point

Example

```
bool test = Intersection.TestSegment2Box2(ref segment, ref box);
Segment2Box2Intr info;
bool find = Intersection.FindSegment2Box2(ref segment, ref box, out info);
```

4.1.13 Line2-Circle2



Test Prefab:
Test_IntrLine2Circle2

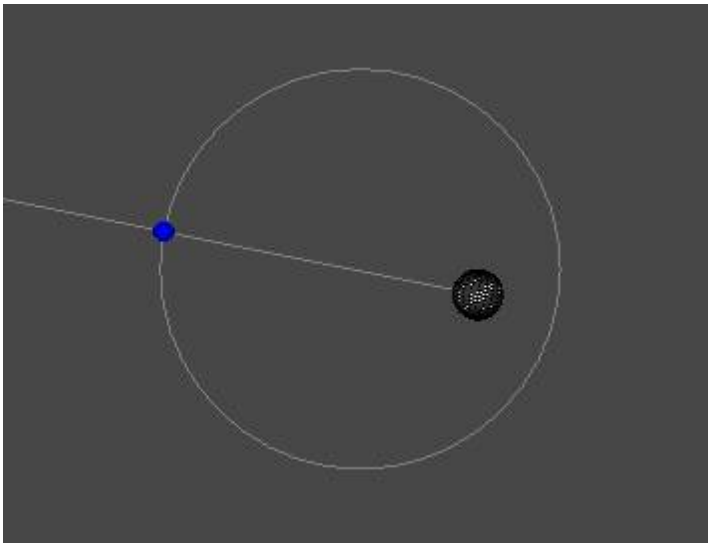
Type
`class Intersection`
Methods
`static bool TestLine2Circle2(ref Line2 line, ref Circle2 circle)` Tests whether line and circle intersect. Returns true if intersection occurs false otherwise.
`static bool FindLine2Circle2(ref Line2 line, ref Circle2 circle, out Line2Circle2Intr info)` Tests whether line and circle intersect and finds actual intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Line2Circle2Intr` Contains information about intersection of Line2 and Circle2
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point or IntersectionTypes.Segment if intersection occured otherwise IntersectionTypes.Empty
`Vector2 Point0` First point of intersection (in case of IntersectionTypes.Point or IntersectionTypes.Segment)
`Vector2 Point1` Second point of intersection (in case of IntersectionTypes.Segment)

Example

```
bool test = Intersection.TestLine2Circle2(ref line, ref circle);
Line2Circle2Intr info;
bool find = Intersection.FindLine2Circle2(ref line, ref circle, out info);
```

4.1.14 Ray2-Circle2



Test Prefab:
Test_IntrRay2Circle2

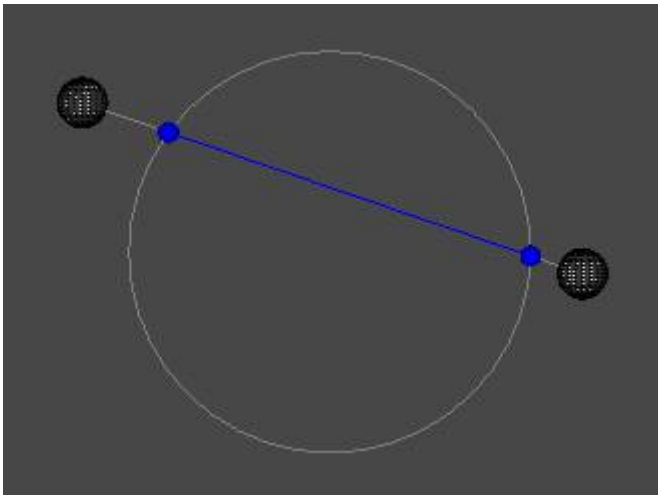
Type
`class Intersection`
Methods
`static bool TestRay2Circle2(ref Ray2 ray, ref Circle2 circle)` Tests whether ray and circle intersect. Returns true if intersection occurs false otherwise.
`static bool FindRay2Circle2(ref Ray2 ray, ref Circle2 circle, out Ray2Circle2Intr info)` Tests whether ray and circle intersect and finds actual intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Ray2Circle2Intr` Contains information about intersection of Ray2 and Circle2
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point or IntersectionTypes.Segment if intersection occured otherwise IntersectionTypes.Empty
`Vector2 Point0` First point of intersection (in case of IntersectionTypes.Point or IntersectionTypes.Segment)
`Vector2 Point1` Second point of intersection (in case of IntersectionTypes.Segment)

Example

```
bool test = Intersection.TestRay2Circle2(ref ray, ref circle);
Ray2Circle2Intr info;
bool find = Intersection.FindRay2Circle2(ref ray, ref circle, out info);
```

4.1.15 Segment2-Circle2



Test Prefab:
Test_IntrSegment2Circle2

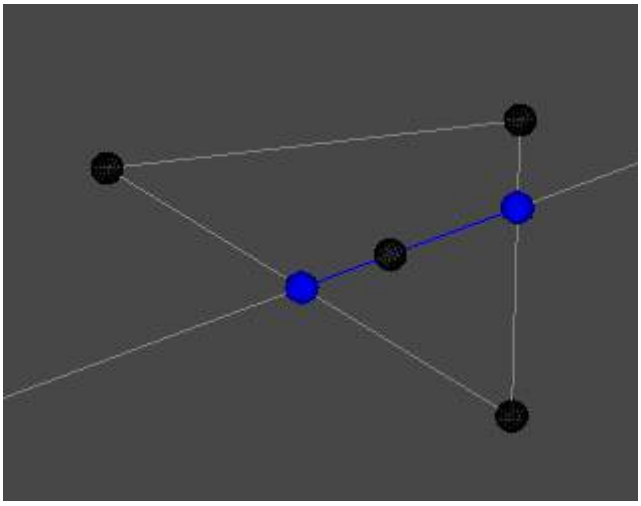
Type
`class Intersection`
Methods
`static bool TestSegment2Circle2(ref Segment2 segment, ref Circle2 circle)` Tests whether segment and circle intersect. Returns true if intersection occurs false otherwise.
`static bool FindSegment2Circle2(ref Segment2 segment, ref Circle2 circle, out Segment2Circle2Intr info)` Tests whether segment and circle intersect and finds actual intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Segment2Circle2Intr` Contains information about intersection of Segment2 and Circle2
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point or IntersectionTypes.Segment if intersection occured otherwise IntersectionTypes.Empty
`Vector2 Point0` First point of intersection (in case of IntersectionTypes.Point or IntersectionTypes.Segment)
`Vector2 Point1` Second point of intersection (in case of IntersectionTypes.Segment)

Example

```
bool test = Intersection.TestSegment2Circle2(ref segment, ref circle);
Segment2Circle2Intr info;
bool find = Intersection.FindSegment2Circle2(ref segment, ref circle, out info);
```

4.1.16 Line2-Triangle2



Test Prefab:
Test_IntrLine2Triangle2

Type
`class Intersection`
Methods

`static bool TestLine2Triangle2(ref Line2 line, ref Triangle2 triangle, out IntersectionTypes intersectionType)`
`static bool TestLine2Triangle2(ref Line2 line, ref Triangle2 triangle)`
Tests whether line and triangle intersect. Returns true if intersection occurs false otherwise.

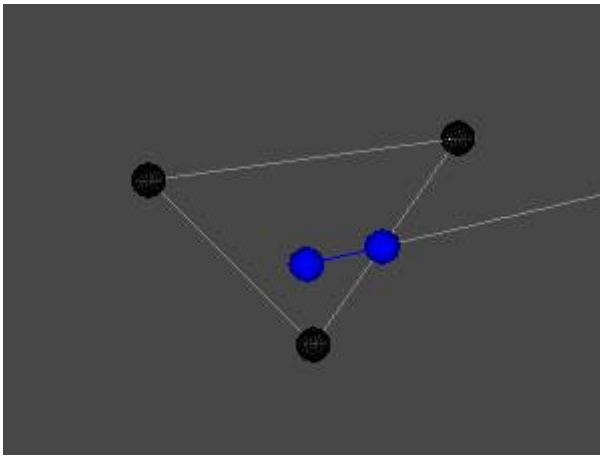
`static bool FindLine2Triangle2(ref Line2 line, ref Triangle2 triangle, out Line2Triangle2Intr info)` Tests whether line and triangle intersect and finds actual intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Line2Triangle2Intr` Contains information about intersection of Line2 and Triangle2
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point or IntersectionTypes.Segment if intersection occured otherwise IntersectionTypes.Empty
`int Quantity` Number of intersection points
`Vector2 Point0` First intersection point
`Vector2 Point1` Second intersection point

Example

```
IntersectionTypes intersectionType;
bool test = Intersection.TestLine2Triangle2(ref line, ref triangle, out intersectionType);
Line2Triangle2Intr info;
bool find = Intersection.FindLine2Triangle2(ref line, ref triangle, out info);
```

4.1.17 Ray2-Triangle2



Test Prefab:
Test_IntrRay2Triangle2

Type
`class Intersection`
Methods

`static bool TestRay2Triangle2(ref Ray2 ray, ref Triangle2 triangle, out IntersectionTypes intersectionType)`
`static bool TestRay2Triangle2(ref Ray2 ray, ref Triangle2 triangle)`
Tests whether ray and triangle intersect. Returns true if intersection occurs false otherwise.

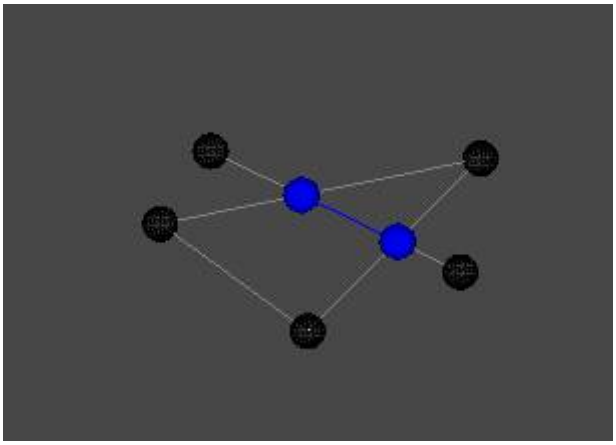
`static bool FindRay2Triangle2(ref Ray2 ray, ref Triangle2 triangle, out Ray2Triangle2Intr info)` Tests whether ray and triangle intersect and finds actual intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Ray2Triangle2Intr` Contains information about intersection of Ray2 and Triangle2
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point or IntersectionTypes.Segment if intersection occured otherwise IntersectionTypes.Empty
`int Quantity` Number of intersection points
`Vector2 Point0` First intersection point
`Vector2 Point1` Second intersection point

Example

```
IntersectionTypes intersectionType;
bool test = Intersection.TestRay2Triangle2(ref ray, ref triangle, out intersectionType);
Ray2Triangle2Intr info;
bool find = Intersection.FindRay2Triangle2(ref ray, ref triangle, out info);
```

4.1.18 Segment2-Triangle2



Test Prefab:
Test_IntrSegment2Triangle2

Type
`class Intersection`
Methods

`static bool TestSegment2Triangle2(ref Segment2 segment, ref Triangle2 triangle, out IntersectionTypes intersectionType)`
`static bool TestSegment2Triangle2(ref Segment2 segment, ref Triangle2 triangle)`
Tests whether segment and triangle intersect. Returns true if intersection occurs false otherwise.

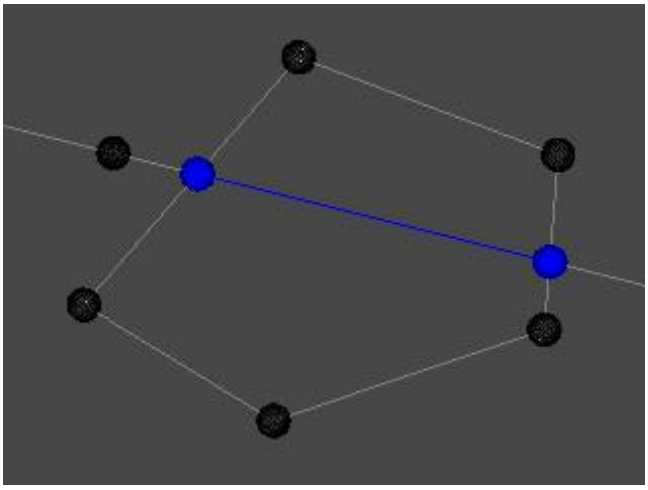
`static bool FindSegment2Triangle2(ref Segment2 segment, ref Triangle2 triangle, out Segment2Triangle2Intr info)` Tests whether segment and triangle intersect and finds actual intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Segment2Triangle2Intr` Contains information about intersection of Segment2 and Triangle2
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point or IntersectionTypes.Segment if intersection occured otherwise IntersectionTypes.Empty
`int Quantity` Number of intersection points
`Vector2 Point0` First intersection point
`Vector2 Point1` Second intersection point

Example

```
IntersectionTypes intersectionType;
bool test = Intersection.TestSegment2Triangle2(ref segment, ref triangle, out intersectionType);
Segment2Triangle2Intr info;
bool find = Intersection.FindSegment2Triangle2(ref segment, ref triangle, out info);
```

4.1.19 Line2-ConvexPolygon2



Test Prefab:
Test_IntrLine2ConvexPolygon2

Note that polygon must be convex and CCW ordered for Find method.

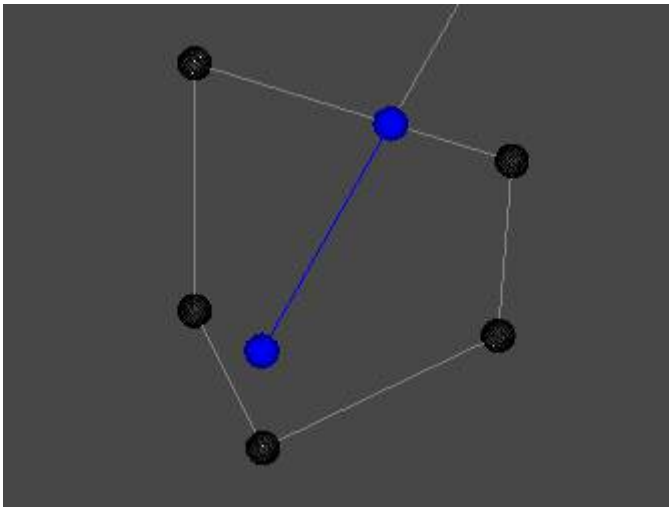
Type
`class Intersection`
Methods
`static bool TestLine2ConvexPolygon2(ref Line2 line, Polygon2 convexPolygon)` Tests if a line intersects a convex polygon. Returns true if intersection occurs false otherwise.
`static bool FindLine2ConvexPolygon2(ref Line2 line, Polygon2 convexPolygon, out Line2ConvexPolygon2Intr info)` Tests if a line intersects a convex ccw ordered polygon and finds intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Line2ConvexPolygon2Intr` Contains information about intersection of Line2 and convex ccw ordered Polygon2
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point or IntersectionTypes.Segment if intersection occured otherwise IntersectionTypes.Empty
`int Quantity` Number of intersection points. IntersectionTypes.Empty: 0; IntersectionTypes.Point: 1; IntersectionTypes.Segment: 2.
`Vector2 Point0` First intersection point
`Vector2 Point1` Second intersection point
`float Parameter0` Line evaluation parameter of the first intersection point
`float Parameter1` Line evaluation parameter of the second intersection point

Example

```
bool test = Intersection.TestLine2ConvexPolygon2(ref line, convexPolygon);
Line2ConvexPolygon2Intr info;
bool find = Intersection.FindLine2ConvexPolygon2(ref line, convexPolygon, out info);
```

4.1.20 Ray2-ConvexPolygon2



Test Prefab:
Test_IntrRay2ConvexPolygon2

Note that polygon must be convex and CCW ordered for Find method.

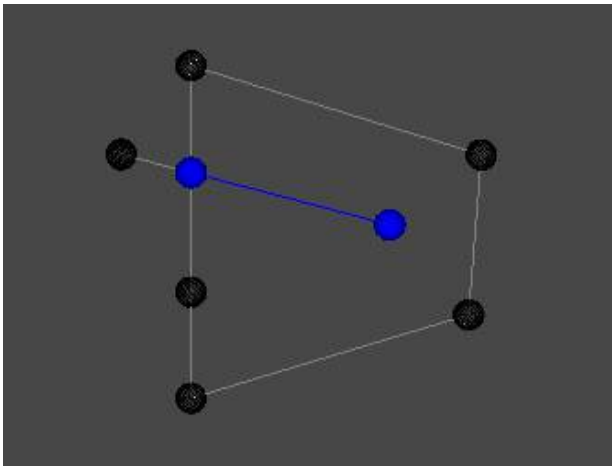
Type
`class Intersection`
Methods
`static bool TestRay2ConvexPolygon2(ref Ray2 ray, Polygon2 convexPolygon)` Tests if a ray intersects a convex polygon. Returns true if intersection occurs false otherwise.
`static bool FindRay2ConvexPolygon2(ref Ray2 ray, Polygon2 convexPolygon, out Ray2ConvexPolygon2Intr info)` Tests if a ray intersects a convex ccw ordered polygon and finds intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Ray2ConvexPolygon2Intr` Contains information about intersection of Ray2 and convex ccw ordered Polygon2
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point or IntersectionTypes.Segment if intersection occured otherwise IntersectionTypes.Empty
`int Quantity` Number of intersection points. IntersectionTypes.Empty: 0; IntersectionTypes.Point: 1; IntersectionTypes.Segment: 2.
`Vector2 Point0` First intersection point
`Vector2 Point1` Second intersection point
`float Parameter0` Ray evaluation parameter of the first intersection point
`float Parameter1` Ray evaluation parameter of the second intersection point

Example

```
bool test = Intersection.TestRay2ConvexPolygon2(ref ray, convexPolygon);
Ray2ConvexPolygon2Intr info;
bool find = Intersection.FindRay2ConvexPolygon2(ref ray, convexPolygon, out info);
```

4.1.21 Segment2-ConvexPolygon2



Test Prefab:
Test_IntrSegment2ConvexPolygon2

Note that polygon must be convex and CCW ordered.

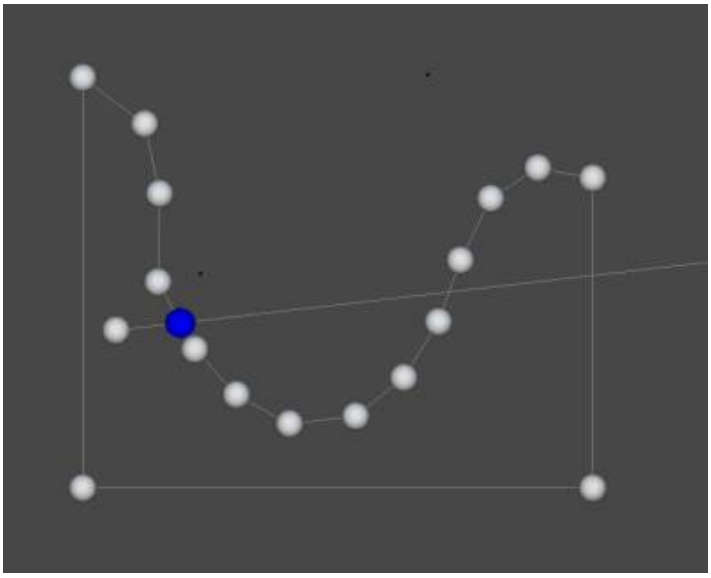
Type
`class Intersection`
Methods
`static bool TestSegment2ConvexPolygon2(ref Segment2 segment, Polygon2 convexPolygon)` Tests if a ray intersects a convex ccw ordered polygon. Returns true if intersection occurs false otherwise.
`static bool FindSegment2ConvexPolygon2(ref Segment2 segment, Polygon2 convexPolygon, out Segment2ConvexPolygon2Intr info)` Tests if a ray intersects a convex ccw ordered polygon and finds intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Segment2ConvexPolygon2Intr` Contains information about intersection of Segment2 and convex ccw ordered Polygon2
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point or IntersectionTypes.Segment if intersection occured otherwise IntersectionTypes.Empty
`int Quantity` Number of intersection points. IntersectionTypes.Empty: 0; IntersectionTypes.Point: 1; IntersectionTypes.Segment: 2.
`Vector2 Point0` First intersection point
`Vector2 Point1` Second intersection point
`float Parameter0` Segment evaluation parameter of the first intersection point
`float Parameter1` Segment evaluation parameter of the second intersection point

Example

```
bool test = Intersection.TestSegment2ConvexPolygon2(ref segment, convexPolygon);
Segment2ConvexPolygon2Intr info;
bool find = Intersection.FindSegment2ConvexPolygon2(ref segment, convexPolygon, out info);
```

4.1.22 Ray2-Polygon2



Test Prefab:
Test_IntrRay2Polygon2

Any polygon is allowed. Also there are overloads which accept segment array rather than polygon. Read comments below.

Type
`class Intersection`
Methods
`static bool TestRay2Polygon2(ref Ray2 ray, Polygon2 polygon)` Tests if a ray intersects a polygon. Returns true if intersection occurs false otherwise.
`static bool TestRay2Polygon2(ref Ray2 ray, Segment2[] segments)` Tests if a ray intersects a segment array. Returns true if intersection occurs false otherwise. Using this method allows to pass non-closed polyline instead of a polygon. Also if you have static polygon which is queried often, it is better to convert polygon to Segment2 array once and then call this method. Overload which accepts a polygon will convert edges to Segment2 every time, while this overload simply accepts Segment2 array and avoids this overhead.
`static bool FindRay2Polygon2(ref Ray2 ray, Polygon2 polygon, out Ray2Polygon2Intr info)` Tests if a ray intersects a polygon and finds intersection parameters. Returns true if intersection occurs false otherwise.
`static bool FindRay2Polygon2(ref Ray2 ray, Segment2[] segments, out Ray2Polygon2Intr info)` Tests if a ray intersects a polygon and finds intersection parameters. Returns true if intersection occurs false otherwise. Using this method allows to pass non-closed polyline instead of a polygon. Also if you have static polygon which is queried often, it is better to convert polygon to Segment2 array once and then call this method. Overload which accepts a polygon will convert edges to Segment2 every time, while this overload simply accepts Segment2 array and avoids this overhead.

Basically these methods do Ray2-Segment2 intersection with every edge of the polygon/polyline. Test methods simply return as soon as any intersection is found, while find methods will look for closest intersection iterating through all the edges. Thus in case when polygon contains very large amount of edges it maybe profitable to split a polygon into smaller parts and do coarse check beforehand (e.g. Ray2-AAB2 test on the smaller polygon AAB).

Type
`struct Ray2Polygon2Intr` Contains information about intersection of Ray2 and general Polygon2

Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point or IntersectionTypes.Segment (ray and some polygon segment are collinear and overlap in more than one point) if intersection occured otherwise IntersectionTypes.Empty
`Vector2 Point0` In case of IntersectionTypes.Point constains single point of intersection. In case of IntersectionTypes.Segment contains first point of intersection. Otherwise Vector2.zero.
`Vector2 Point1` In case of IntersectionTypes.Segment contains second point of intersection. Otherwise Vector2.zero.
`float Parameter0` In case of IntersectionTypes.Point contains evaluation parameter of single intersection point according to ray. In case of IntersectionTypes.Segment contains evaluation parameter of the first intersection point according to ray. Otherwise 0.
`float Parameter1` In case of IntersectionTypes.Segment contains evaluation parameter of the second intersection point according to ray. Otherwise 0.

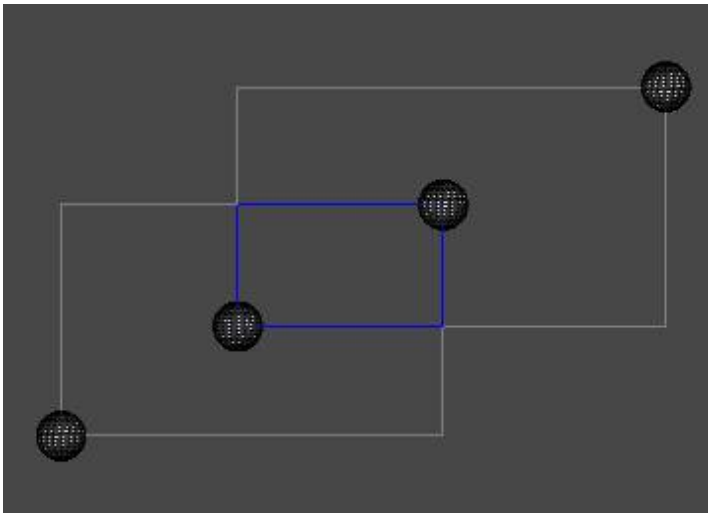
Example

```
// Polyon2 polygon is defined somewhere...
// Use this overload if polygon changes often
Ray2Polygon2Intr info;
bool find = Intersection.FindRay2Polygon2(ref ray, polygon, out info);
```

Example

```
// Polyon2 polygon is defined somewhere...
// Use this overload if polygon is not changing. Thus you can convert polygon once and then use the array
// in tests every time.
// Also this overload allows to pass non-closed polyline (note that polyons are always closed).
Segment2[] segments = polygon.ToSegmentArray(); // Convert polygon to segment array first
Ray2Polygon2Intr info;
bool find = Intersection.FindRay2Polygon2(ref ray, segments, out info);
```

4.1.23 AAB2-AAB2

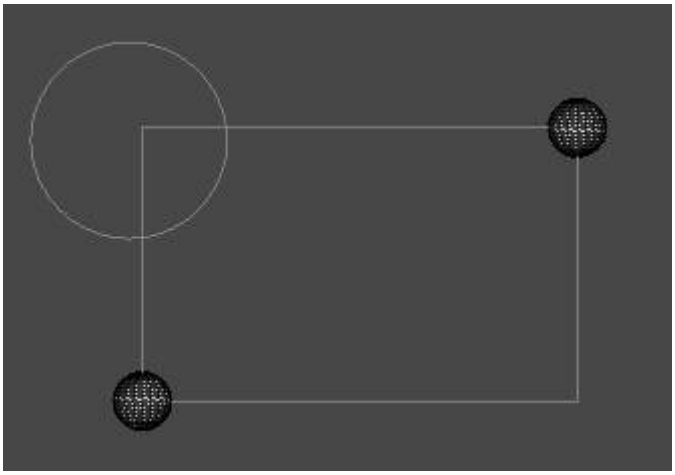


Test Prefab:
Test_IntrAAB2AAB2

Type
`class Intersection`
Methods
`static bool TestAAB2AAB2(ref AAB2 box0, ref AAB2 box1)` Tests whether two AAB intersect. Returns true if intersection occurs false otherwise.
`static bool TestAAB2AAB2OverlapX(ref AAB2 box0, ref AAB2 box1)` Checks whether two aab has x overlap
`static bool TestAAB2AAB2OverlapY(ref AAB2 box0, ref AAB2 box1)` Checks whether two aab has y overlap
`static bool FindAAB2AAB2(ref AAB2 box0, ref AAB2 box1, out AAB2 intersection)` Tests whether two AAB intersect and finds intersection which is AAB itself. Returns true if intersection occurs false otherwise.

Example
`AAB2 intr;` `bool find = Intersection.FindAAB2AAB2(ref box0, ref box1, out intr);`

4.1.24 AAB2-Circle2

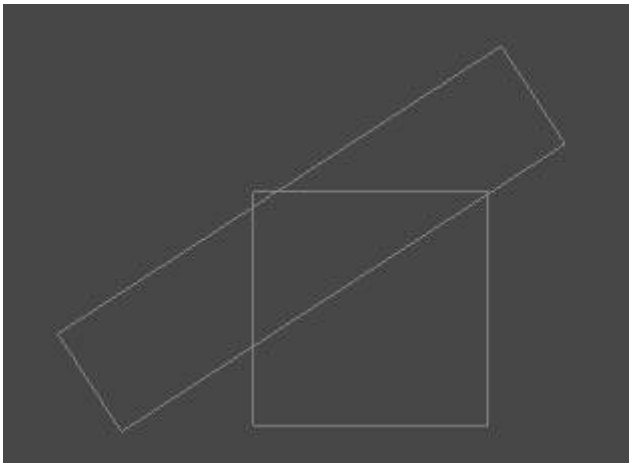


Test Prefab:
Test_IntrAAB2Circle2

Type
`class Intersection`
Methods
`static bool TestAAB2Circle2(ref AAB2 box, ref Circle2 circle)` Tests if an axis aligned box intersects a circle. Returns true if intersection occurs false otherwise.

Example
`bool test = Intersection.TestAAB2Circle2(ref box, ref circle);`

4.1.25 Box2-Box2

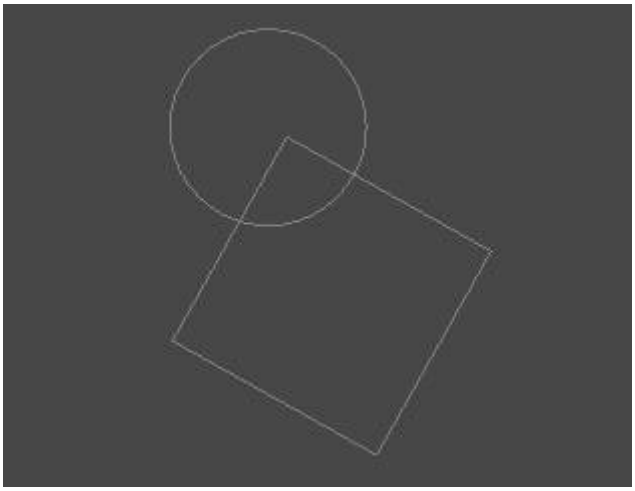


Test Prefab:
Test_IntrBox2Box2

Type
`class Intersection`
Methods
`static bool TestBox2Box2(ref Box2 box0, ref Box2 box1)` Tests if a box intersects another box. Returns true if intersection occurs false otherwise.

Example
`bool test = Intersection.TestBox2Box2(ref box0, ref box1);`

4.1.26 Box2-Circle2

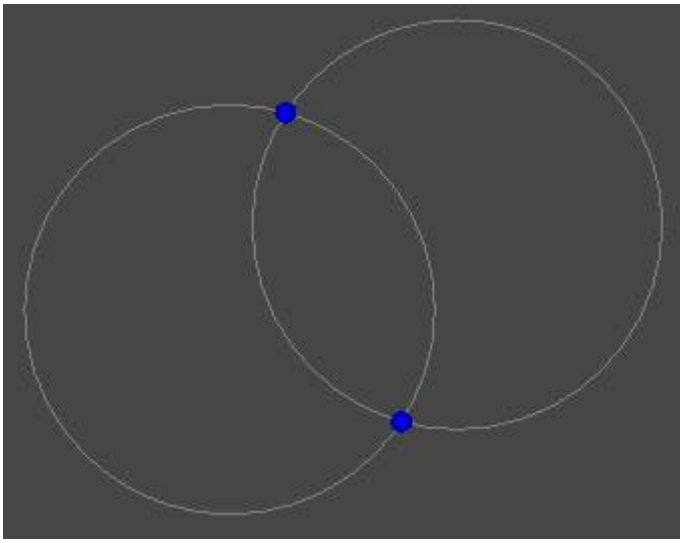


Test Prefab:
Test_IntrBox2Circle2

Type
`class Intersection`
Methods
`static bool TestBox2Circle2(ref Box2 box, ref Circle2 circle)` Tests if a box intersects a circle. Returns true if intersection occurs false otherwise.

Example
`bool test = Intersection.TestBox2Circle2(ref box, ref circle);`

4.1.27 Circle2-Circle2



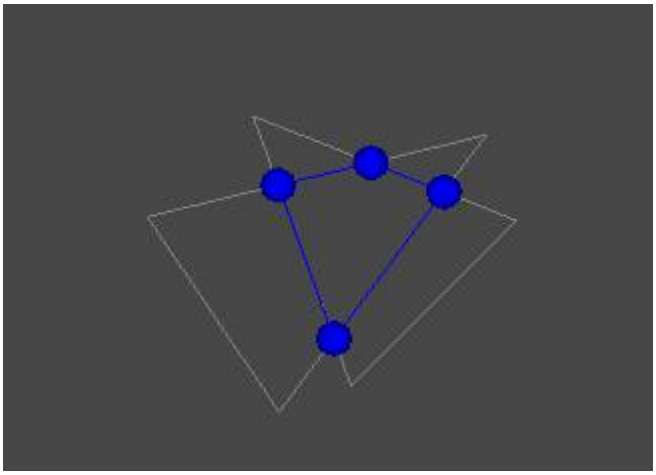
Test Prefab:
Test_IntrCircle2Circle2

Type
`class Intersection`
Methods
`static bool TestCircle2Circle2(ref Circle2 circle0, ref Circle2 circle1)` Tests if a circle intersects another circle. Returns true if intersection occurs false otherwise.
`static bool FindCircle2Circle2(ref Circle2 circle0, ref Circle2 circle1, out Circle2Circle2Intr info)` Tests if a circle intersects another circle and finds intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Circle2Circle2Intr` Contains information about intersection of two Circle2. The quantity Q is 0, 1, or 2. When Q > 0, the interpretation depends on the intersection type. IntersectionTypes.Point: Q distinct points of intersection IntersectionTypes.Other: The circles are the same
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point if there is intersection, IntersectionTypes.Other if circles are the same and IntersectionTypes.Empty if circles do not intersect
`int Quantity` Number of intersection points
`Vector2 Point0` First intersection point
`Vector2 Point1` Second intersection point

Example
`Circle2Circle2Intr info;` `bool find = Intersection.FindCircle2Circle2(ref circle0, ref circle1, out info);`

4.1.28 Triangle2-Triangle2



Test Prefab:
Test_IntrTriangle2Triangle2

Note that both triangles must be CCW ordered.

Type
`class Intersection`
Methods
`static bool TestTriangle2Triangle2(ref Triangle2 triangle0, ref Triangle2 triangle1)` Tests if a triangle intersects another triangle (both triangles must be ordered counter clockwise). Returns true if intersection occurs false otherwise.
`static bool FindTriangle2Triangle2(ref Triangle2 triangle0, ref Triangle2 triangle1, out Triangle2Triangle2Intr info)` Tests if a triangle intersects another triangle and finds intersection parameters (both triangles must be ordered counter clockwise). Returns true if intersection occurs false otherwise.

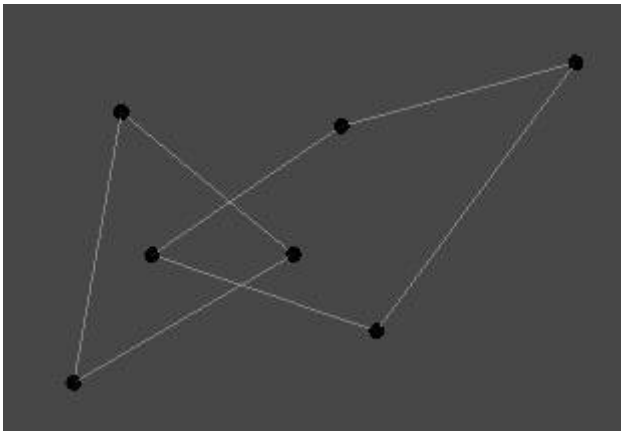
Type
`struct Triangle2Triangle2Intr` Contains information about intersection of Triangle2 and Triangle2
Fields
`IntersectionTypes IntersectionType` Equals to: IntersectionTypes.Empty if no intersection occurs; IntersectionTypes.Point if triangles are touching in a point; IntersectionTypes.Segment if triangles are touching in a segment; IntersectionTypes.Polygon if triangles intersect.
`int Quantity` Number of intersection points. IntersectionTypes.Empty: 0; IntersectionTypes.Point: 1; IntersectionTypes.Segment: 2; IntersectionTypes.Polygon: 3 to 6.
`Vector2 Point0` Intersection point 0
`Vector2 Point1` Intersection point 1
`Vector2 Point2` Intersection point 2
`Vector2 Point3` Intersection point 3

`Vector2 Point4` Intersection point 4
`Vector2 Point5` Intersection point 5
Properties
`Vector2 this[int i] { get; }` Gets intersection point by index (0 to 5). Points could be also accessed individually using Point0,...,Point5 fields.

Example

```
bool test = Intersection.TestTriangle2Triangle2(ref triangle0, ref triangle1);
Triangle2Triangle2Intr info;
bool find = Intersection.FindTriangle2Triangle2(ref triangle0, ref triangle1, out info);
```

4.1.29 ConvexPolygon2-ConvexPolygon2



Test Prefab:
Test_IntrConvexPolygon2ConvexPolygon2

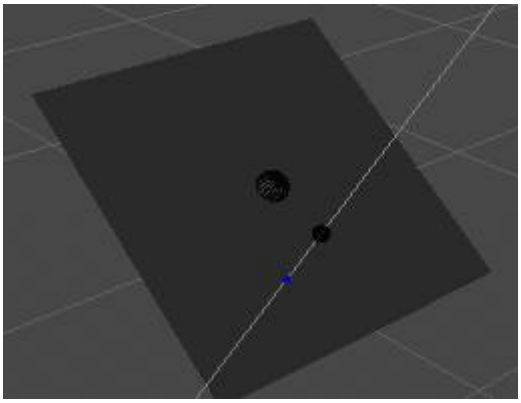
Note that both polygons must be convex and CCW ordered.

Type
`class Intersection`
Methods
`static bool TestConvexPolygon2ConvexPolygon2(Polygon2 convexPolygon0, Polygon2 convexPolygon1)` Tests whether two convex CCW ordered polygons intersect. Returns true if intersection occurs false otherwise. Note that caller is responsibile for supplying proper polygons (convex and CCW ordered).

Example
`bool test = Intersection.TestConvexPolygon2ConvexPolygon2(pol0, pol1);`

4.2 3D Intersection

4.2.1 Line3-Plane3



Test Prefab:
Test_IntrLine3Plane3

Type
`class Intersection`
Methods

`static bool TestLine3Plane3(ref Line3 line, ref Plane3 plane, out IntersectionTypes intersectionType)`
`static bool TestLine3Plane3(ref Line3 line, ref Plane3 plane)`
Tests if a line intersects a plane. Returns true if intersection occurs false otherwise.

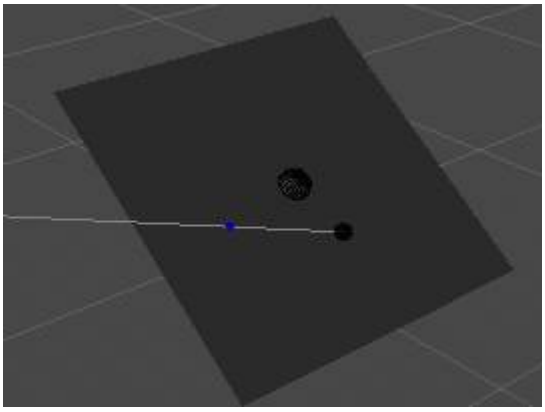
`static bool FindLine3Plane3(ref Line3 line, ref Plane3 plane, out Line3Plane3Intr info)` Tests if a line intersects a plane and finds intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Line3Plane3Intr` Contains information about intersection of Line3 and Plane3
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point or IntersectionTypes.Line (a line lies in a plane) if intersection occured otherwise IntersectionTypes.Empty
`Vector3 Point` Intersection point (in case of IntersectionTypes.Point)
`float LineParameter` Line evaluation parameter of the intersection point (in case of IntersectionTypes.Point)

Example

```
IntersectionTypes intersectionType;
bool test = Intersection.TestLine3Plane3(ref line, ref plane, out intersectionType);
Line3Plane3Intr info;
bool find = Intersection.FindLine3Plane3(ref line, ref plane, out info);
```

4.2.2 Ray3-Plane3



Test Prefab:
Test_IntrRay3Plane3

Type
`class Intersection`
Methods

`static bool TestRay3Plane3(ref Ray3 ray, ref Plane3 plane, out IntersectionTypes intersectionType)`
`static bool TestRay3Plane3(ref Ray3 ray, ref Plane3 plane)`
Tests if a ray intersects a plane. Returns true if intersection occurs false otherwise.

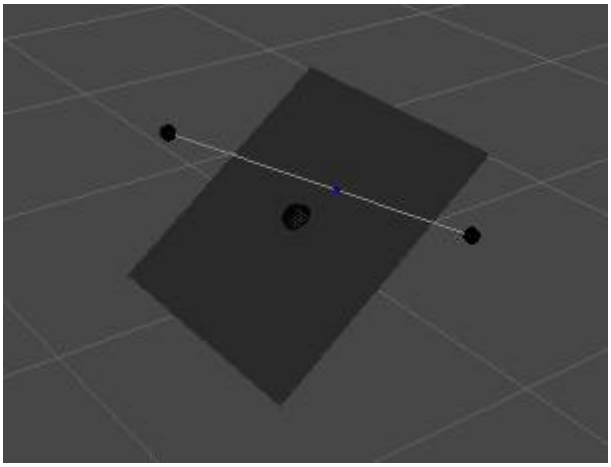
`static bool FindRay3Plane3(ref Ray3 ray, ref Plane3 plane, out Ray3Plane3Intr info)` Tests if a ray intersects a plane and finds intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Ray3Plane3Intr` Contains information about intersection of Ray3 and Plane3
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point or IntersectionTypes.Ray (a ray lies in a plane) if intersection occured otherwise IntersectionTypes.Empty
`Vector3 Point` Intersection point (in case of IntersectionTypes.Point)
`float RayParameter` Ray evaluation parameter of the intersection point (in case of IntersectionTypes.Point)

Example

```
IntersectionTypes intersectionType;
bool test = Intersection.TestRay3Plane3(ref ray, ref plane, out intersectionType);
Ray3Plane3Intr info;
bool find = Intersection.FindRay3Plane3(ref ray, ref plane, out info);
```

4.2.3 Segment3-Plane3



Test Prefab:
Test_IntrSegment3Plane3

Type
`class Intersection`
Methods

`static bool TestSegment3Plane3(ref Segment3 segment, ref Plane3 plane, out IntersectionTypes intersectionType)`
`static bool TestSegment3Plane3(ref Segment3 segment, ref Plane3 plane)`
Tests if a segment intersects a plane. Returns true if intersection occurs false otherwise.

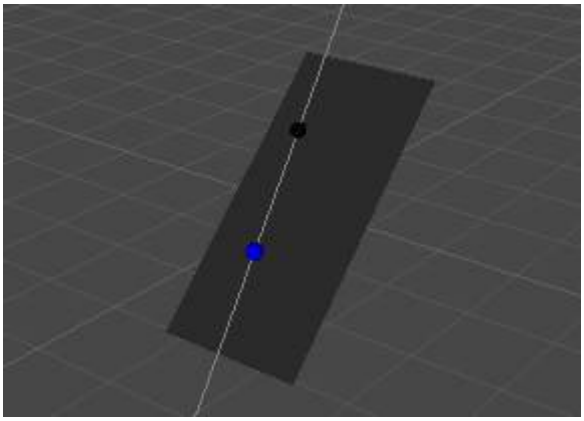
`static bool FindSegment3Plane3(ref Segment3 segment, ref Plane3 plane, out Segment3Plane3Intr info)` Tests if a segment intersects a plane and finds intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Segment3Plane3Intr` Contains information about intersection of Segment3 and Plane3
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point or IntersectionTypes.Segment (a segment lies in a plane) if intersection occured otherwise IntersectionTypes.Empty
`Vector3 Point` Intersection point (in case of IntersectionTypes.Point)
`float SegmentParameter` Segment evaluation parameter of the intersection point (in case of IntersectionTypes.Point)

Example

```
IntersectionTypes intersectionType;
bool test = Intersection.TestSegment3Plane3(ref segment, ref plane, out intersectionType);
Segment3Plane3Intr info;
bool find = Intersection.FindSegment3Plane3(ref segment, ref plane, out info);
```

4.2.4 Line3-Rectangle3



Test Prefab:
Test_IntrLine3Rectangle3

Note that rectangle considered to be solid (that is closed area, not just border).

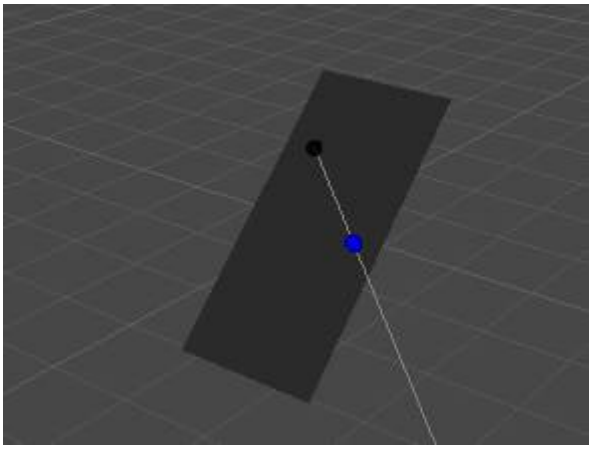
Type
`class Intersection`
Methods
`static bool TestLine3Rectangle3(ref Line3 line, ref Rectangle3 rectangle)` Tests if a line intersects a solid rectangle. Returns true if intersection occurs false otherwise.
`static bool FindLine3Rectangle3(ref Line3 line, ref Rectangle3 rectangle, out Line3Rectangle3Intr info)` Tests if a line intersects a solid rectangle and finds intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Line3Rectangle3Intr` Contains information about intersection of Line3 and Rectangle3 (rectangle considered to be solid)
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point if intersection occured otherwise IntersectionTypes.Empty (including the case when a line lies in the plane of a rectangle)
`Vector3 Point` Intersection point

Example

```
Line3Rectangle3Intr info;
bool find = Intersection.FindLine3Rectangle3(ref line, ref rectangle, out info);
```

4.2.5 Ray3-Rectangle3



Test Prefab:
Test_IntrRay3Rectangle3

Note that rectangle considered to be solid (that is closed area, not just border).

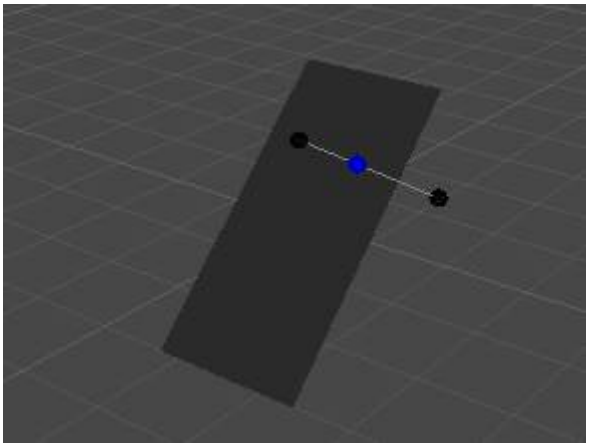
Type
`class Intersection`
Methods
`static bool TestRay3Rectangle3(ref Ray3 ray, ref Rectangle3 rectangle)` Tests if a ray intersects a solid rectangle. Returns true if intersection occurs false otherwise.
`static bool FindRay3Rectangle3(ref Ray3 ray, ref Rectangle3 rectangle, out Ray3Rectangle3Intr info)` Tests if a ray intersects a solid rectangle and finds intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Ray3Rectangle3Intr` Contains information about intersection of Ray3 and Rectangle3 (rectangle considered to be solid)
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point if intersection occured otherwise IntersectionTypes.Empty (including the case when a ray lies in the plane of a rectangle)
`Vector3 Point` Intersection point

Example

```
Ray3Rectangle3Intr info;
bool find = Intersection.FindRay3Rectangle3(ref ray, ref rectangle, out info);
```

4.2.6 Segment3-Rectangle3



Test Prefab:
Test_IntrSegment3Rectangle3

Note that rectangle considered to be solid (that is closed area, not just border).

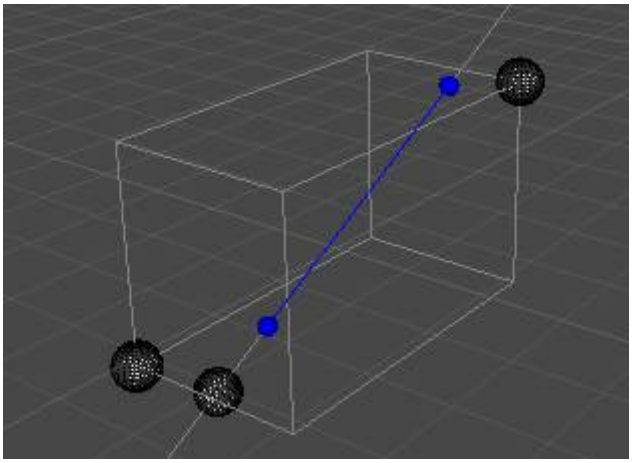
Type
`class Intersection`
Methods
`static bool TestSegment3Rectangle3(ref Segment3 segment, ref Rectangle3 rectangle)` Tests if a segment intersects a solid rectangle. Returns true if intersection occurs false otherwise.
`static bool FindSegment3Rectangle3(ref Segment3 segment, ref Rectangle3 rectangle, out Segment3Rectangle3Intr info)` Tests if a segment intersects a solid rectangle and finds intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Segment3Rectangle3Intr` Contains information about intersection of Segment3 and Rectangle3 (rectangle considered to be solid)
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point if intersection occured otherwise IntersectionTypes.Empty (including the case when a segment lies in the plane of a rectangle)
`Vector3 Point` Intersection point

Example

```
Segment3Rectangle3Intr info;
bool find = Intersection.FindSegment3Rectangle3(ref segment, ref rectangle, out info);
```

4.2.7 Line3-AAB3



Test Prefab:
Test_IntrLine3AAB3

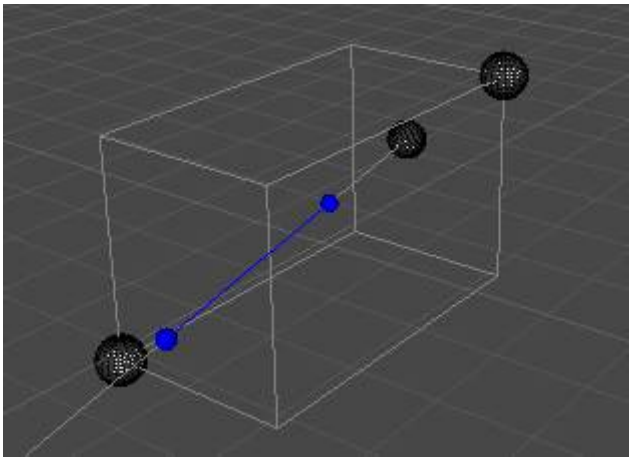
Type
`class Intersection`
Methods
`static bool TestLine3AAB3(ref Line3 line, ref AAB3 box)` Tests if a line intersects an axis aligned box. Returns true if intersection occurs false otherwise.
`static bool FindLine3AAB3(ref Line3 line, ref AAB3 box, out Line3AAB3Intr info)` Tests if a line intersects an axis aligned box and finds intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Line3AAB3Intr` Contains information about intersection of Line3 and AxisAlignedBox3
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point or IntersectionTypes.Segment if intersection occured otherwise IntersectionTypes.Empty
`int Quantity` Number of intersection points. IntersectionTypes.Empty: 0; IntersectionTypes.Point: 1; IntersectionTypes.Segment: 2.
`Vector3 Point0` First intersection point
`Vector3 Point1` Second intersection point

Example

```
bool test = Intersection.TestLine3AAB3(ref line, ref box);
Line3AAB3Intr info;
bool find = Intersection.FindLine3AAB3(ref line, ref box, out info);
```

4.2.8 Ray3-AAB3



Test Prefab:
Test_IntrRay3AAB3

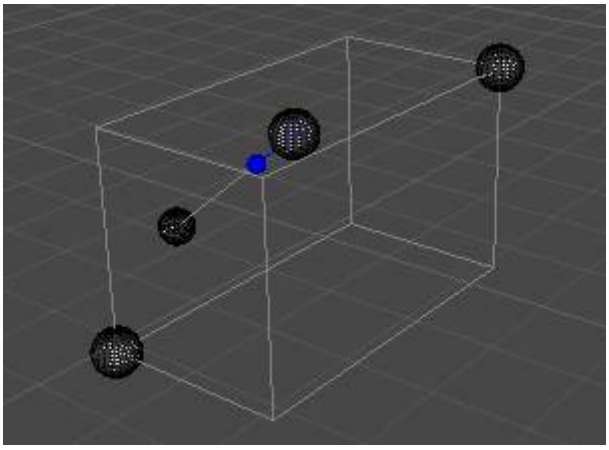
Type
`class Intersection`
Methods
`static bool TestRay3AAB3(ref Ray3 ray, ref AAB3 box)` Tests if a ray intersects an axis aligned box. Returns true if intersection occurs false otherwise.
`static bool FindRay3AAB3(ref Ray3 ray, ref AAB3 box, out Ray3AAB3Intr info)` Tests if a ray intersects an axis aligned box and finds intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Ray3AAB3Intr` Contains information about intersection of Ray3 and AxisAlignedBox3
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point or IntersectionTypes.Segment if intersection occured otherwise IntersectionTypes.Empty
`int Quantity` Number of intersection points. IntersectionTypes.Empty: 0; IntersectionTypes.Point: 1; IntersectionTypes.Segment: 2.
`Vector3 Point0` First intersection point
`Vector3 Point1` Second intersection point

Example

```
bool test = Intersection.TestRay3AAB3(ref ray, ref box);
Ray3AAB3Intr info;
bool find = Intersection.FindRay3AAB3(ref ray, ref box, out info);
```

4.2.9 Segment3-AAB3



Test Prefab:
Test_IntrSegment3AAB3

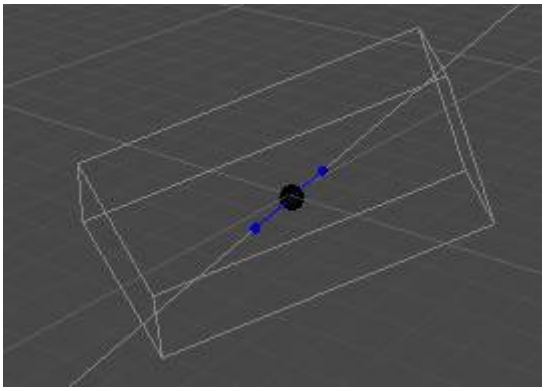
Type
`class Intersection`
Methods
`static bool TestSegment3AAB3(ref Segment3 segment, ref AAB3 box)` Tests if a segment intersects an axis aligned box. Returns true if intersection occurs false otherwise.
`static bool FindSegment3AAB3(ref Segment3 segment, ref AAB3 box, out Segment3AAB3Intr info)` Tests if a segment intersects an axis aligned box and finds intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Segment3AAB3Intr` Contains information about intersection of Segment3 and AxisAlignedBox3
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point or IntersectionTypes.Segment if intersection occured otherwise IntersectionTypes.Empty
`int Quantity` Number of intersection points. IntersectionTypes.Empty: 0; IntersectionTypes.Point: 1; IntersectionTypes.Segment: 2.
`Vector3 Point0` First intersection point
`Vector3 Point1` Second intersection point

Example

```
bool test = Intersection.TestSegment3AAB3(ref segment, ref box);
Segment3AAB3Intr info;
bool find = Intersection.FindSegment3AAB3(ref segment, ref box, out info);
```

4.2.10 Line3-Box3



Test Prefab:
Test_IntrLine3Box3

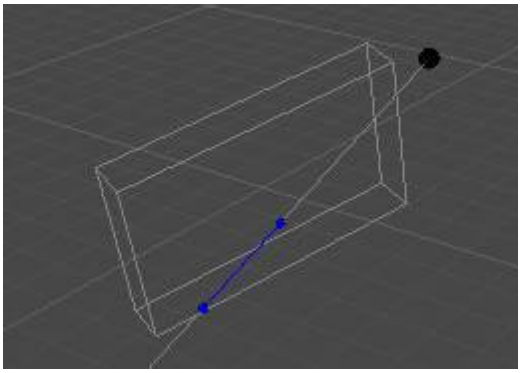
Type
`class Intersection`
Methods
`static bool TestLine3Box3(ref Line3 line, ref Box3 box)` Tests if a line intersects a box. Returns true if intersection occurs false otherwise.
`static bool FindLine3Box3(ref Line3 line, ref Box3 box, out Line3Box3Intr info)` Tests if a line intersects a box and finds intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Line3Box3Intr` Contains information about intersection of Line3 and Box3
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point or IntersectionTypes.Segment if intersection occured otherwise IntersectionTypes.Empty
`int Quantity` Number of intersection points. IntersectionTypes.Empty: 0; IntersectionTypes.Point: 1; IntersectionTypes.Segment: 2.
`Vector3 Point0` First intersection point
`Vector3 Point1` Second intersection point

Example

```
bool test = Intersection.TestLine3Box3(ref line, ref box);
Line3Box3Intr info;
bool find = Intersection.FindLine3Box3(ref line, ref box, out info);
```

4.2.11 Ray3-Box3



Test Prefab:
Test_IntrRay3Box3

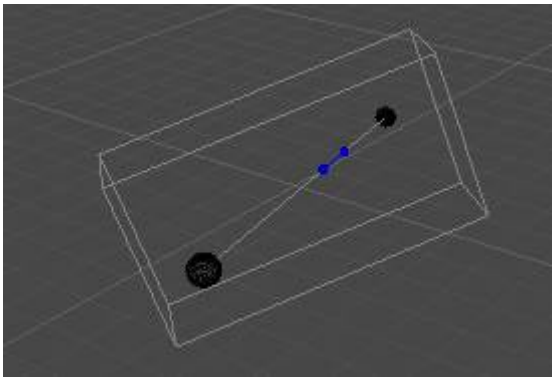
Type
`class Intersection`
Methods
`static bool TestRay3Box3(ref Ray3 ray, ref Box3 box)` Tests if a ray intersects a box. Returns true if intersection occurs false otherwise.
`static bool FindRay3Box3(ref Ray3 ray, ref Box3 box, out Ray3Box3Intr info)` Tests if a ray intersects a box and finds intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Ray3Box3Intr` Contains information about intersection of Ray3 and Box3
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point or IntersectionTypes.Segment if intersection occured otherwise IntersectionTypes.Empty
`int Quantity` Number of intersection points. IntersectionTypes.Empty: 0; IntersectionTypes.Point: 1; IntersectionTypes.Segment: 2.
`Vector3 Point0` First intersection point
`Vector3 Point1` Second intersection point

Example

```
bool test = Intersection.TestRay3Box3(ref ray, ref box);
Ray3Box3Intr info;
bool find = Intersection.FindRay3Box3(ref ray, ref box, out info);
```

4.2.12 Segment3-Box3



Test Prefab:
Test_IntrSegment3Box3

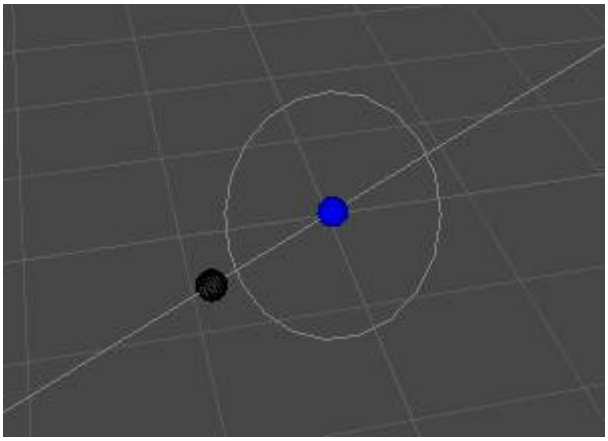
Type
`class Intersection`
Methods
`static bool TestSegment3Box3(ref Segment3 segment, ref Box3 box)` Tests if a segment intersects a box. Returns true if intersection occurs false otherwise.
`static bool FindSegment3Box3(ref Segment3 segment, ref Box3 box, out Segment3Box3Intr info)` Tests if a segment intersects a box and finds intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Segment3Box3Intr` Contains information about intersection of Segment3 and Box3
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point or IntersectionTypes.Segment if intersection occured otherwise IntersectionTypes.Empty
`int Quantity` Number of intersection points. IntersectionTypes.Empty: 0; IntersectionTypes.Point: 1; IntersectionTypes.Segment: 2.
`Vector3 Point0` First intersection point
`Vector3 Point1` Second intersection point

Example

```
bool test = Intersection.TestSegment3Box3(ref segment, ref box);
Segment3Box3Intr info;
bool find = Intersection.FindSegment3Box3(ref segment, ref box, out info);
```

4.2.13 Line3-Circle3



Test Prefab:
Test_IntrLine3Circle3

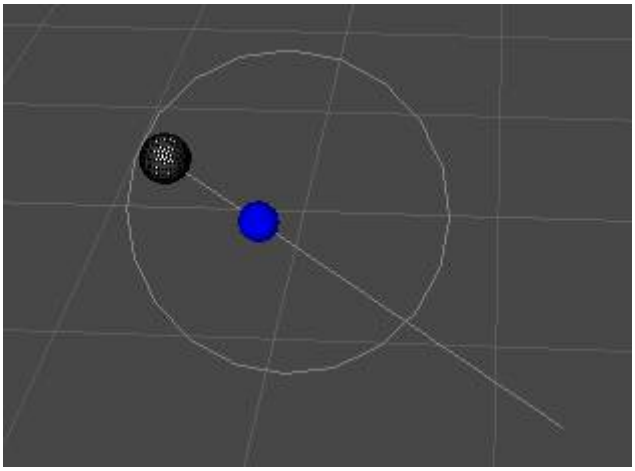
Note that circle considered to be solid (that is closed area, not just border).

Type
`class Intersection`
Methods
`static bool TestLine3Circle3(ref Line3 line, ref Circle3 circle)` Tests if a line intersects a solid circle. Returns true if intersection occurs false otherwise.
`static bool FindLine3Circle3(ref Line3 line, ref Circle3 circle, out Line3Circle3Intr info)` Tests if a line intersects a solid circle and finds intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Line3Circle3Intr` Contains information about intersection of Line3 and Circle3 (circle considered to be solid)
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point if intersection occured otherwise IntersectionTypes.Empty (including the case when a line lies in the plane of a circle)
`Vector3 Point` Intersection point

Example
`Line3Circle3Intr info;` `bool find = Intersection.FindLine3Circle3(ref line, ref circle, out info);`

4.2.14 Ray3-Circle3



Test Prefab:
Test_IntrRay3Circle3

Note that circle considered to be solid (that is closed area, not just border).

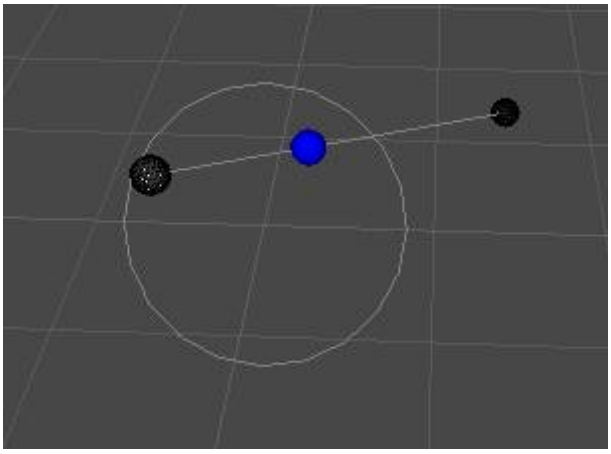
Type
`class Intersection`
Methods
`static bool TestRay3Circle3(ref Ray3 ray, ref Circle3 circle)` Tests if a ray intersects a solid circle. Returns true if intersection occurs false otherwise.
`static bool FindRay3Circle3(ref Ray3 ray, ref Circle3 circle, out Ray3Circle3Intr info)` Tests if a ray intersects a solid circle and finds intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Ray3Circle3Intr` Contains information about intersection of Ray3 and Circle3 (circle considered to be solid)
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point if intersection occured otherwise IntersectionTypes.Empty (including the case when a ray lies in the plane of a circle)
`Vector3 Point` Intersection point

Example

```
Ray3Circle3Intr info;
bool find = Intersection.FindRay3Circle3(ref ray, ref circle, out info);
```

4.2.15 Segment3-Circle3



Test Prefab:
Test_IntrSegment3Circle3

Note that circle considered to be solid (that is closed area, not just border).

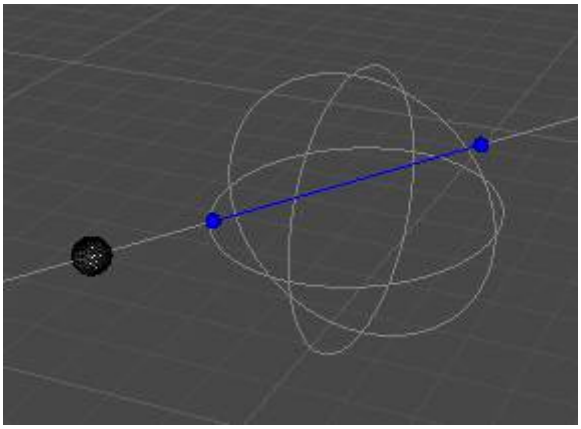
Type
`class Intersection`
Methods
`static bool TestSegment3Circle3(ref Segment3 segment, ref Circle3 circle)` Tests if a segment intersects a solid circle. Returns true if intersection occurs false otherwise.
`static bool FindSegment3Circle3(ref Segment3 segment, ref Circle3 circle, out Segment3Circle3Intr info)` Tests if a segment intersects a solid circle and finds intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Segment3Circle3Intr` Contains information about intersection of Segment3 and Circle3 (circle considered to be solid)
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point if intersection occured otherwise IntersectionTypes.Empty (including the case when a segment lies in the plane of a circle)
`Vector3 Point` Intersection point

Example

```
Segment3Circle3Intr info;
bool find = Intersection.FindSegment3Circle3(ref segment, ref circle, out info);
```

4.2.16 Line3-Sphere3



Test Prefab:
Test_IntrLine3Sphere3

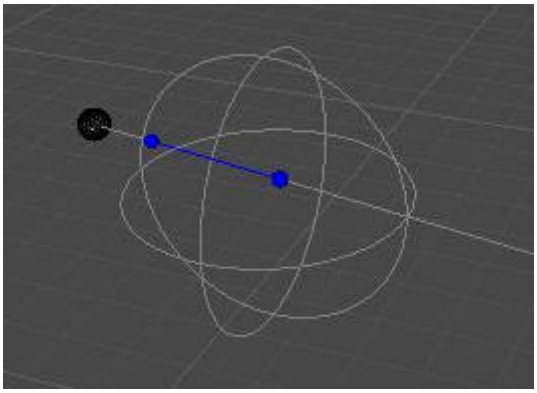
Type
`class Intersection`
Methods
`static bool TestLine3Sphere3(ref Line3 line, ref Sphere3 sphere)` Tests if a line intersects a sphere. Returns true if intersection occurs false otherwise.
`static bool FindLine3Sphere3(ref Line3 line, ref Sphere3 sphere, out Line3Sphere3Intr info)` Tests if a line intersects a sphere and finds intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Line3Sphere3Intr` Contains information about intersection of Line3 and Sphere3
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point or IntersectionTypes.Segment if intersection occured otherwise IntersectionTypes.Empty
`int Quantity` Number of intersection points. IntersectionTypes.Empty: 0; IntersectionTypes.Point: 1; IntersectionTypes.Segment: 2.
`Vector3 Point0` First intersection point
`Vector3 Point1` Second intersection point
`float LineParameter0` Line evaluation parameter of the first intersection point
`float LineParameter1` Line evaluation parameter of the second intersection point

Example

```
bool test = Intersection.TestLine3Sphere3(ref line, ref sphere);
Line3Sphere3Intr info;
bool find = Intersection.FindLine3Sphere3(ref line, ref sphere, out info);
```

4.2.17 Ray3-Sphere3



Test Prefab:
Test_IntrRay3Sphere3

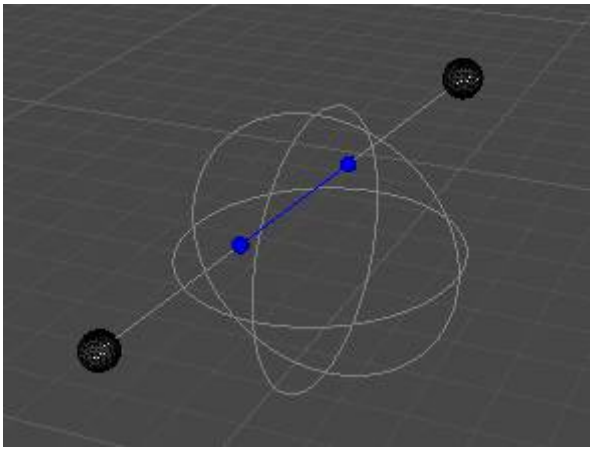
Type
`class Intersection`
Methods
`static bool TestRay3Sphere3(ref Ray3 ray, ref Sphere3 sphere)` Tests if a ray intersects a sphere. Returns true if intersection occurs false otherwise.
`static bool FindRay3Sphere3(ref Ray3 ray, ref Sphere3 sphere, out Ray3Sphere3Intr info)` Tests if a ray intersects a sphere and finds intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Ray3Sphere3Intr` Contains information about intersection of Ray3 and Sphere3
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point or IntersectionTypes.Segment if intersection occured otherwise IntersectionTypes.Empty
`int Quantity` Number of intersection points. IntersectionTypes.Empty: 0; IntersectionTypes.Point: 1; IntersectionTypes.Segment: 2.
`Vector3 Point0` First intersection point
`Vector3 Point1` Second intersection point
`float RayParameter0` Ray evaluation parameter of the first intersection point
`float RayParameter1` Ray evaluation parameter of the second intersection point

Example

```
bool test = Intersection.TestRay3Sphere3(ref ray, ref sphere);
Ray3Sphere3Intr info;
bool find = Intersection.FindRay3Sphere3(ref ray, ref sphere, out info);
```

4.2.18 Segment3-Sphere3



Test Prefab:
Test_IntrSegment3Sphere3

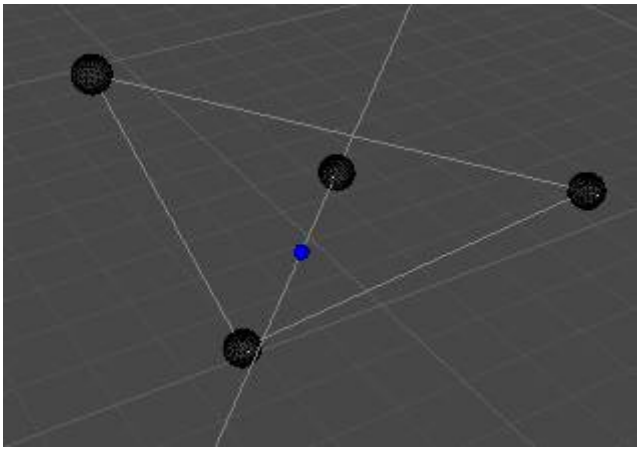
Type
`class Intersection`
Methods
`static bool TestSegment3Sphere3(ref Segment3 segment, ref Sphere3 sphere)` Tests if a segment intersects a sphere. Returns true if intersection occurs false otherwise.
`static bool FindSegment3Sphere3(ref Segment3 segment, ref Sphere3 sphere, out Segment3Sphere3Intr info)` Tests if a segment intersects a sphere and finds intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Segment3Sphere3Intr` Contains information about intersection of Segment3 and Sphere3
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point or IntersectionTypes.Segment if intersection occured otherwise IntersectionTypes.Empty
`int Quantity` Number of intersection points. IntersectionTypes.Empty: 0; IntersectionTypes.Point: 1; IntersectionTypes.Segment: 2.
`Vector3 Point0` First intersection point
`Vector3 Point1` Second intersection point
`float SegmentParameter0` Segment evaluation parameter of the first intersection point
`float SegmentParameter1` Segment evaluation parameter of the second intersection point

Example

```
bool test = Intersection.TestSegment3Sphere3(ref segment, ref sphere);
Segment3Sphere3Intr info;
bool find = Intersection.FindSegment3Sphere3(ref segment, ref sphere, out info);
```

4.2.19 Line3-Triangle3



Test Prefab:
Test_IntrLine3Triangle3

Note that due to intersection with triangles is very common operation library provides additonal overloads for user convenience.

Also note that tests return `false` when a line lies in the plane of a triangle.

Type
`class Intersection`
Methods

`static bool TestLine3Triangle3(ref Line3 line, ref Triangle3 triangle, out IntersectionTypes intersectionType)`
`static bool TestLine3Triangle3(ref Line3 line, ref Vector3 v0, ref Vector3 v1, ref Vector3 v2, out IntersectionTypes intersectionType)`
`static bool TestLine3Triangle3(ref Line3 line, Vector3 v0, Vector3 v1, Vector3 v2, out IntersectionTypes intersectionType)`
`static bool TestLine3Triangle3(ref Line3 line, ref Triangle3 triangle)`
`static bool TestLine3Triangle3(ref Line3 line, ref Vector3 v0, ref Vector3 v1, ref Vector3 v2)`
`static bool TestLine3Triangle3(ref Line3 line, Vector3 v0, Vector3 v1, Vector3 v2)`
Tests if a line intersects a triangle. Returns true if intersection occurs false otherwise.

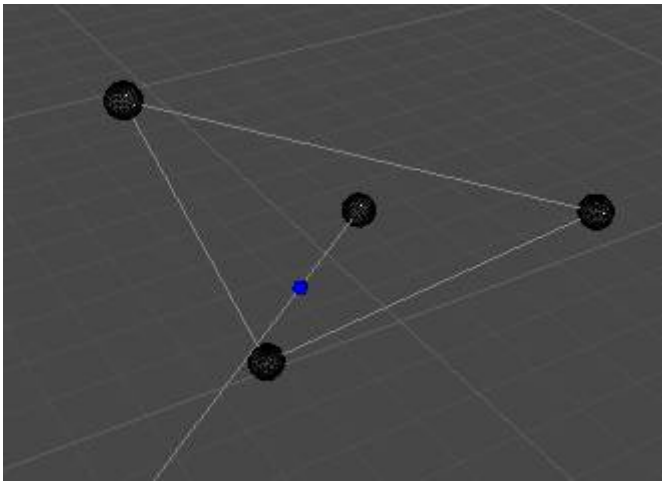
`static bool FindLine3Triangle3(ref Line3 line, ref Triangle3 triangle, out Line3Triangle3Intr info)`
`static bool FindLine3Triangle3(ref Line3 line, ref Vector3 v0, ref Vector3 v1, ref Vector3 v2, out Line3Triangle3Intr info)`
`static bool FindLine3Triangle3(ref Line3 line, Vector3 v0, Vector3 v1, Vector3 v2, out Line3Triangle3Intr info)`
Tests if a line intersects a triangle and finds intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Line3Triangle3Intr` Contains information about intersection of Line3 and Triangle3
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point if intersection occured otherwise IntersectionTypes.Empty (even when a line lies in a triangle plane)
`Vector3 Point` Intersection point (in case of IntersectionTypes.Point)
`float LineParameter` Line evaluation parameter of the intersection point (in case of IntersectionTypes.Point)
`float TriBary0` First barycentric coordinate of the intersection point
`float TriBary1` Second barycentric coordinate of the intersection point
`float TriBary2` Third barycentric coordinate of the intersection point

Example

```
IntersectionTypes intersectionType;
bool test = Intersection.TestLine3Triangle3(ref line, ref triangle, out intersectionType);
Line3Triangle3Intr info;
bool find = Intersection.FindLine3Triangle3(ref line, ref triangle, out info);
```

4.2.20 Ray3-Triangle3



Test Prefab:
Test_IntrRay3Triangle3

Note that due to intersection with triangles is very common operation library provides additonal overloads for user convenience.

Also note that tests return `false` when a ray lies in the plane of a triangle.

Type
`class Intersection`
Methods

`static bool TestRay3Triangle3(ref Ray3 ray, ref Triangle3 triangle, out IntersectionTypes intersectionType)`
`static bool TestRay3Triangle3(ref Ray3 ray, ref Vector3 v0, ref Vector3 v1, ref Vector3 v2, out IntersectionTypes intersectionType)`
`static bool TestRay3Triangle3(ref Ray3 ray, Vector3 v0, Vector3 v1, Vector3 v2, out IntersectionTypes intersectionType)`
`static bool TestRay3Triangle3(ref Ray3 ray, ref Triangle3 triangle)`
`static bool TestRay3Triangle3(ref Ray3 ray, ref Vector3 v0, ref Vector3 v1, ref Vector3 v2)`
`static bool TestRay3Triangle3(ref Ray3 ray, Vector3 v0, Vector3 v1, Vector3 v2)`
Tests if a ray intersects a triangle. Returns true if intersection occurs false otherwise.

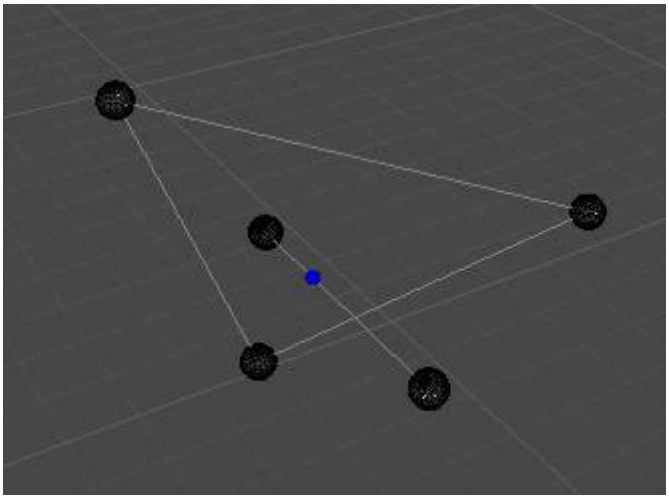
`static bool FindRay3Triangle3(ref Ray3 ray, ref Triangle3 triangle, out Ray3Triangle3Intr info)`
`static bool FindRay3Triangle3(ref Ray3 ray, ref Vector3 v0, ref Vector3 v1, ref Vector3 v2, out Ray3Triangle3Intr info)`
`static bool FindRay3Triangle3(ref Ray3 ray, Vector3 v0, Vector3 v1, Vector3 v2, out Ray3Triangle3Intr info)`
Tests if a ray intersects a triangle and finds intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Ray3Triangle3Intr` Contains information about intersection of Ray and Triangle3
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point if intersection occured otherwise IntersectionTypes.Empty (even when a ray lies in a triangle plane)
`Vector3 Point` Intersection point (in case of IntersectionTypes.Point)
`float RayParameter` Ray evaluation parameter of the intersection point (in case of IntersectionTypes.Point)
`float TriBary0` First barycentric coordinate of the intersection point
`float TriBary1` Second barycentric coordinate of the intersection point
`float TriBary2` Third barycentric coordinate of the intersection point

Example

```
IntersectionTypes intersectionType;
bool test = Intersection.TestRay3Triangle3(ref ray, ref triangle, out intersectionType);
Ray3Triangle3Intr info;
bool find = Intersection.FindRay3Triangle3(ref ray, ref triangle, out info);
```

4.2.21 Segment3-Triangle3



Test Prefab:
Test_IntrSegment3Triangle3

Note that due to intersection with triangles is very common operation library provides additonal overloads for user convenience.

Also note that tests return `false` when a segment lies in the plane of a triangle.

Type
`class Intersection`
Methods

`static bool TestSegment3Triangle3(ref Segment3 segment, ref Triangle3 triangle, out IntersectionTypes intersectionType)`
`static bool TestSegment3Triangle3(ref Segment3 segment, ref Vector3 v0, ref Vector3 v1, ref Vector3 v2, out IntersectionTypes intersectionType)`
`static bool TestSegment3Triangle3(ref Segment3 segment, Vector3 v0, Vector3 v1, Vector3 v2, out IntersectionTypes intersectionType)`
`static bool TestSegment3Triangle3(ref Segment3 segment, ref Triangle3 triangle)`
`static bool TestSegment3Triangle3(ref Segment3 segment, ref Vector3 v0, ref Vector3 v1, ref Vector3 v2)`
`static bool TestSegment3Triangle3(ref Segment3 segment, Vector3 v0, Vector3 v1, Vector3 v2)`
Tests if a segment intersects a triangle. Returns true if intersection occurs false otherwise.

`static bool FindSegment3Triangle3(ref Segment3 segment, ref Triangle3 triangle, out Segment3Triangle3Intr info)`
`static bool FindSegment3Triangle3(ref Segment3 segment, ref Vector3 v0, ref Vector3 v1, ref Vector3 v2, out Segment3Triangle3Intr info)`
`static bool FindSegment3Triangle3(ref Segment3 segment, Vector3 v0, Vector3 v1, Vector3 v2, out Segment3Triangle3Intr info)`
Tests if a segment intersects a triangle and finds intersection parameters. Returns true if intersection occurs false otherwise.

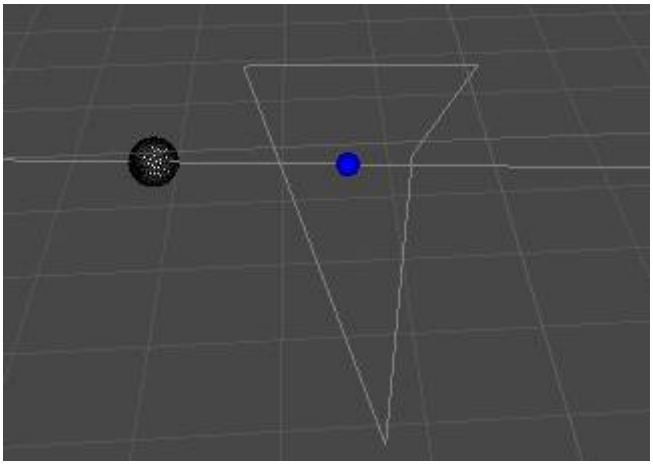
Type
`struct Segment3Triangle3Intr` Contains information about intersection of Segment3 and Triangle3
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point if intersection occured otherwise IntersectionTypes.Empty (even when a segment lies in a triangle plane)
`Vector3 Point` Intersection point (in case of IntersectionTypes.Point)
`float SegmentParameter` Segment evaluation parameter of the intersection point (in case of IntersectionTypes.Point)
`float TriBary0` First barycentric coordinate of the intersection point

`float TriBary1` Second barycentric coordinate of the intersection point
`float TriBary2` Third barycentric coordinate of the intersection point

Example

```
IntersectionTypes intersectionType;
bool test = Intersection.TestSegment3Triangle3(ref segment, ref triangle, out intersectionType);
Segment3Triangle3Intr info;
bool find = Intersection.FindSegment3Triangle3(ref segment, ref triangle, out info);
```

4.2.22 Line3-Polygon3



Test Prefab:
Test_IntrLine3Polygon3

Note that polygon considered to be solid (that is closed area, not just border). Polygon can be non-convex with any orientation, but must be planar.

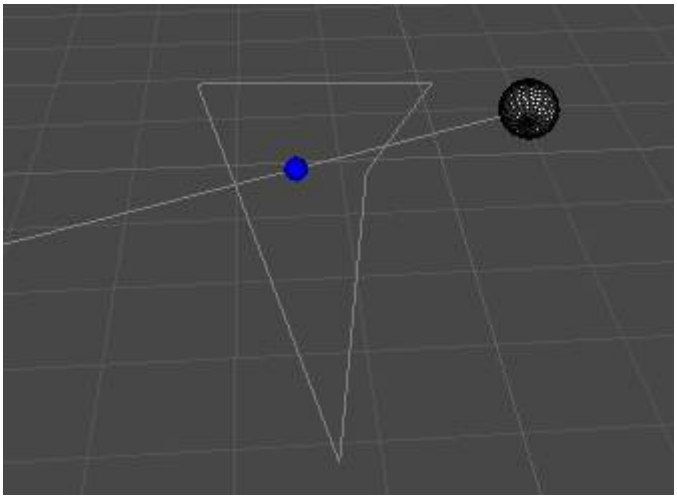
Type
`class Intersection`
Methods
`static bool TestLine3Polygon3(ref Line3 line, Polygon3 polygon)` Tests if a line intersects a solid polygon. Returns true if intersection occurs false otherwise.
`static bool FindLine3Polygon3(ref Line3 line, Polygon3 polygon, out Line3Polygon3Intr info)` Tests if a line intersects a solid polygon and finds intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Line3Polygon3Intr` Contains information about intersection of Line3 and Polygon3 (polygon considered to be solid)
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point if intersection occured otherwise IntersectionTypes.Empty (including the case when a line lies in the plane of a polygon)
`Vector3 Point` Intersection point

Example

```
Line3Polygon3Intr info;
bool find = Intersection.FindLine3Polygon3(ref line, polygon, out info);
```

4.2.23 Ray3-Polygon3



Test Prefab:
Test_IntrRay3Polygon3

Note that polygon considered to be solid (that is closed area, not just border). Polygon can be non-convex with any orientation, but must be planar.

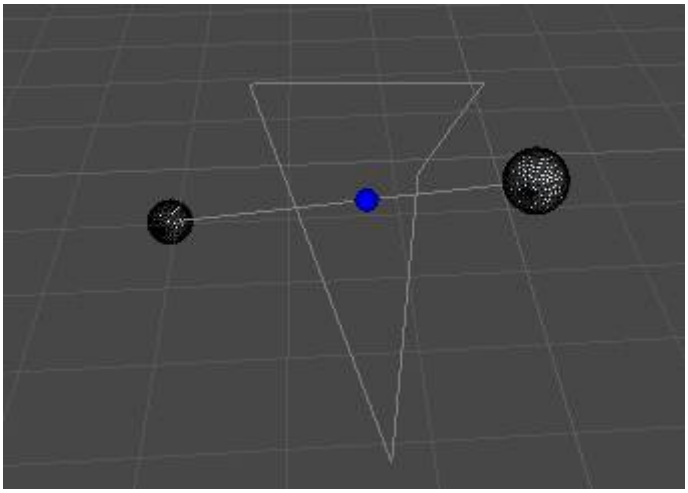
Type
`class Intersection`
Methods
`static bool TestRay3Polygon3(ref Ray3 ray, Polygon3 polygon)` Tests if a line intersects a solid polygon. Returns true if intersection occurs false otherwise.
`static bool FindRay3Polygon3(ref Ray3 ray, Polygon3 polygon, out Ray3Polygon3Intr info)` Tests if a ray intersects a solid polygon and finds intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Ray3Polygon3Intr` Contains information about intersection of Ray3 and Polygon3 (polygon considered to be solid)
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point if intersection occured otherwise IntersectionTypes.Empty (including the case when a ray lies in the plane of a polygon)
`Vector3 Point` Intersection point

Example

```
Ray3Polygon3Intr info;
bool find = Intersection.FindRay3Polygon3(ref ray, polygon, out info);
```

4.2.24 Segment3-Polygon3



Test Prefab:
Test_IntrSegment3Polygon3

Note that polygon considered to be solid (that is closed area, not just border). Polygon can be non-convex with any orientation, but must be planar.

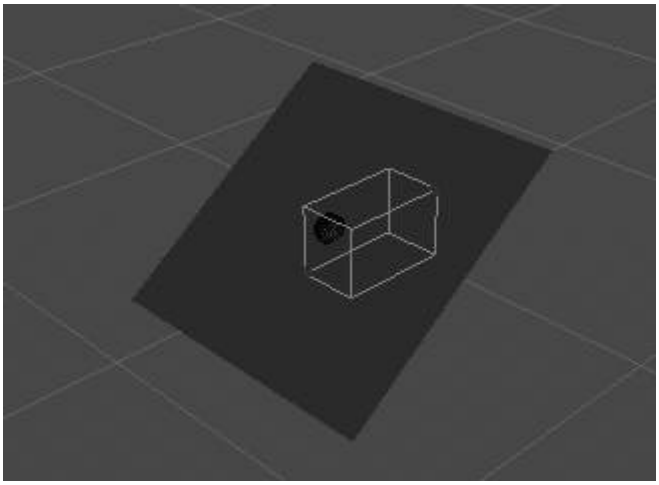
Type
`class Intersection`
Methods
`static bool TestSegment3Polygon3(ref Segment3 segment, Polygon3 polygon)` Tests if a segment intersects a solid polygon. Returns true if intersection occurs false otherwise.
`static bool FindSegment3Polygon3(ref Segment3 segment, Polygon3 polygon, out Segment3Polygon3Intr info)` Tests if a segment intersects a solid polygon and finds intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Segment3Polygon3Intr` Contains information about intersection of Segment3 and Polygon3 (polygon considered to be solid)
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point if intersection occured otherwise IntersectionTypes.Empty (including the case when a segment lies in the plane of a polygon)
`Vector3 Point` Intersection point

Example

```
Segment3Polygon3Intr info;
bool find = Intersection.FindSegment3Polygon3(ref segment, polygon, out info);
```

4.2.25 Plane3-AAB3

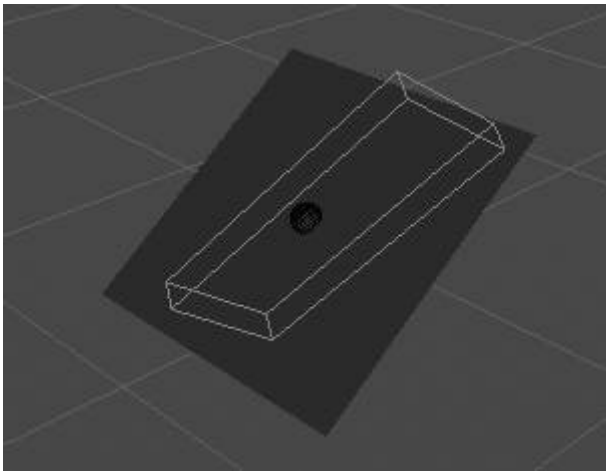


Test Prefab:
Test_IntrPlane3AAB3

Type
`class Intersection`
Methods
`static bool TestPlane3AAB3(ref Plane3 plane, ref AAB3 box)` Tests if a plane intersects a box. Returns true if intersection occurs false otherwise.

Example
`bool test = Intersection.TestPlane3AAB3(ref plane, ref box);`

4.2.26 Plane3-Box3

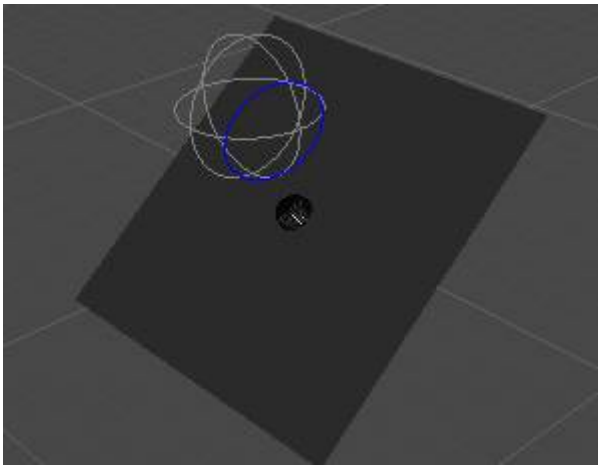


Test Prefab:
Test_IntrPlane3Box3

Type
`class Intersection`
Methods
`static bool TestPlane3Box3(ref Plane3 plane, ref Box3 box)` Tests if a plane intersects a box. Returns true if intersection occurs false otherwise.

Example
`bool test = Intersection.TestPlane3Box3(ref plane, ref box);`

4.2.27 Plane3-Sphere3



Test Prefab:
Test_IntrPlane3Sphere3

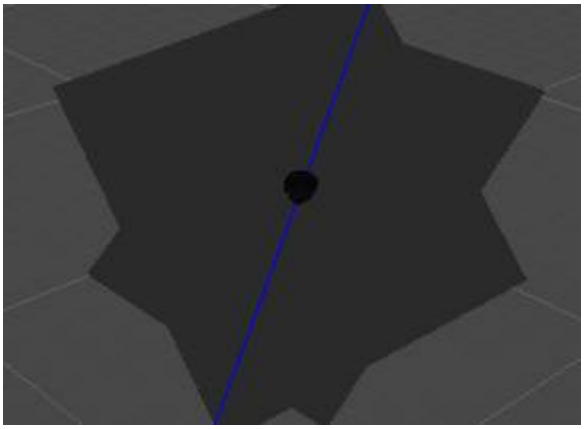
Type
`class Intersection`
Methods
`static bool TestPlane3Sphere3(ref Plane3 plane, ref Sphere3 sphere)` Tests if a plane intersects a plane. Returns true if intersection occurs false otherwise.
`static bool FindPlane3Sphere3(ref Plane3 plane, ref Sphere3 sphere, out Plane3Sphere3Intr info)` Tests if a plane intersects a plane and finds intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Plane3Sphere3Intr` Contains information about intersection of Plane3 and Sphere3
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionType.Point if a sphere is touching a plane, IntersectionType.Other if a sphere intersects a plane, otherwise IntersectionType.Empty
`Circle3 Circle` Contains intersection circle of a sphere and a plane in case of IntersectionType.Other

Example

```
bool test = Intersection.TestPlane3Sphere3(ref plane, ref sphere);
Plane3Sphere3Intr info;
bool find = Intersection.FindPlane3Sphere3(ref plane, ref sphere, out info);
```

4.2.28 Plane3-Plane3



Test Prefab:
Test_IntrPlane3Plane3

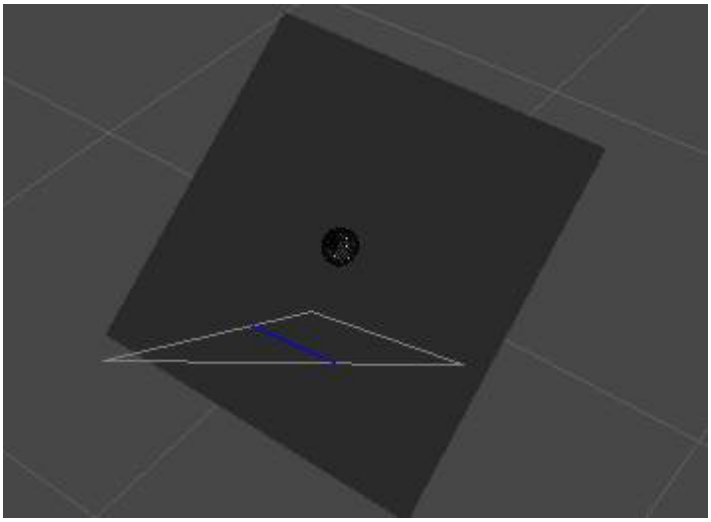
Type
`class Intersection`
Methods
`static bool TestPlane3Plane3(ref Plane3 plane0, ref Plane3 plane1)` Tests if a plane intersects another plane. Returns true if intersection occurs false otherwise (also returns false when planes are the same)
`static bool FindPlane3Plane3(ref Plane3 plane0, ref Plane3 plane1, out Plane3Plane3Intr info)` Tests if a plane intersects another plane and finds intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Plane3Plane3Intr` `Contains information about intersection of Plane3 and Plane3`
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Line or IntersectionTypes.Plane (planes are the same) if intersection occured otherwise IntersectionTypes.Empty.
`Line3 Line` Intersection line (in case of IntersectionTypes.Line)

Example

```
bool test = Intersection.TestPlane3Plane3(ref plane0, ref plane1);
Plane3Plane3Intr info;
bool find = Intersection.FindPlane3Plane3(ref plane0, ref plane1, out info);
```

4.2.29 Plane3-Triangle3



Test Prefab:
Test_IntrPlane3Triangle3

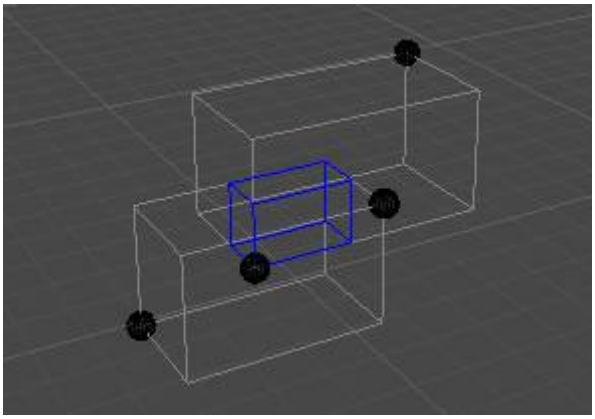
Type
`class Intersection`
Methods
`static bool TestPlane3Triangle3(ref Plane3 plane, ref Triangle3 triangle)` Tests if a plane intersects a triangle. Returns true if intersection occurs false otherwise.
`static bool FindPlane3Triangle3(ref Plane3 plane, ref Triangle3 triangle, out Plane3Triangle3Intr info)` Tests if a plane intersects a triangle and finds intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Plane3Triangle3Intr` Contains information about intersection of Plane3 and Triangle3
Fields
`IntersectionTypes IntersectionType` Equals to IntersectionTypes.Point (a triangle is touching a plane by a vertex) or IntersectionTypes.Segment (a triangle is touching a plane by an edge or intersecting the plane) or IntersectionTypes.Polygon (a triangle is lying in a plane), otherwise IntersectionTypes.Empty (no intersection).
`int Quantity` Number of intersection points. 0 - IntersectionTypes.Empty; 1 - IntersectionTypes.Point; 2 - IntersectionTypes.Segment; 3 - IntersectionTypes.Polygon;
`Vector3 Point0` First intersection point
`Vector3 Point1` Second intersection point
`Vector3 Point2` Third intersection point

Example

```
bool test = Intersection.TestPlane3Triangle3(ref plane, ref triangle);
Plane3Triangle3Intr info;
bool find = Intersection.FindPlane3Triangle3(ref plane, ref triangle, out info);
```

4.2.30 AAB3-AAB3

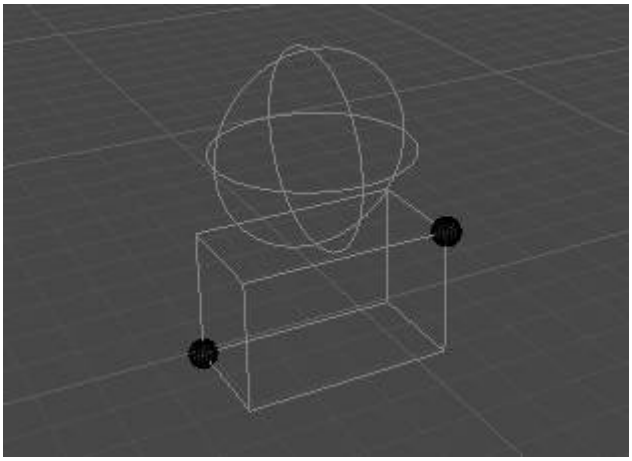


Test Prefab:
Test_IntrAAB3AAB3

Type
`class Intersection`
Methods
`static bool TestAAB3AAB3(ref AAB3 box0, ref AAB3 box1)` Tests whether two AAB intersect. Returns true if intersection occurs false otherwise.
`static bool FindAAB3AAB3(ref AAB3 box0, ref AAB3 box1, out AAB3 intersection)` Tests whether two AAB intersect and finds intersection which is AAB itself. Returns true if intersection occurs false otherwise.
`static bool TestAAB3AAB3OverlapX(ref AAB3 box0, ref AAB3 box1)` Checks whether two aab has x overlap
`static bool TestAAB3AAB3OverlapY(ref AAB3 box0, ref AAB3 box1)` Checks whether two aab has y overlap
`static bool TestAAB3AAB3OverlapZ(ref AAB3 box0, ref AAB3 box1)` Checks whether two aab has z overlap

Example
`AAB3 intr;` `bool find = Intersection.FindAAB3AAB3(ref box0, ref box1, out intr);`

4.2.31 AAB3-Sphere3

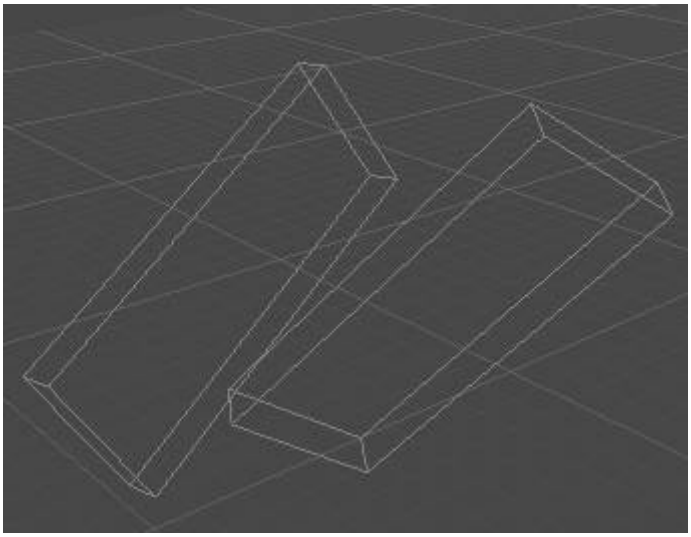


Test Prefab:
Test_IntrAAB3Sphere3

Type
`class Intersection`
Methods
`static bool TestAAB3Sphere3(ref AAB3 box, ref Sphere3 sphere)` Tests if an axis aligned box intersects a sphere. Returns true if intersection occurs false otherwise.

Example
`bool test = Intersection.TestAAB3Sphere3(ref box, ref sphere);`

4.2.32 Box3-Box3

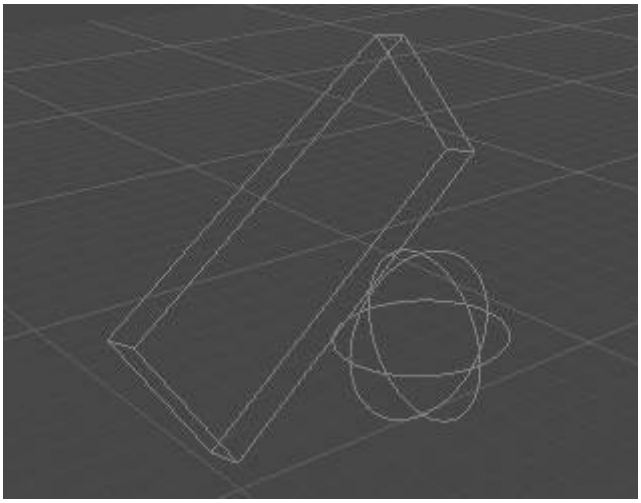


Test Prefab:
Test_IntrBox3Box3

Type
`class Intersection`
Methods
`static bool TestBox3Box3(ref Box3 box0, ref Box3 box1)` Tests if a box intersects another box. Returns true if intersection occurs false otherwise.

Example
`bool test = Intersection.TestBox3Box3(ref box0, ref box1);`

4.2.33 Box3-Sphere3

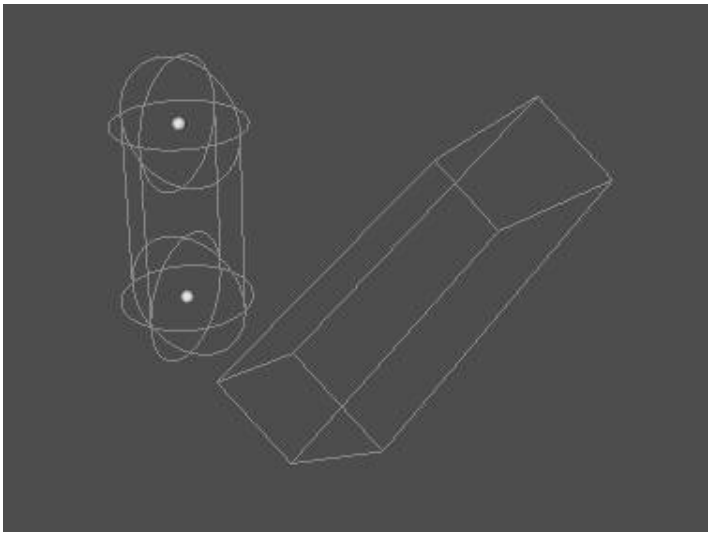


Test Prefab:
Test_IntrBox3Sphere3

Type
`class Intersection`
Methods
`static bool TestBox3Sphere3(ref Box3 box, ref Sphere3 sphere)` Tests if a box intersects a sphere. Returns true if intersection occurs false otherwise.

Example
`bool test = Intersection.TestBox3Sphere3(ref box, ref sphere);`

4.2.34 Box3-Capsule3

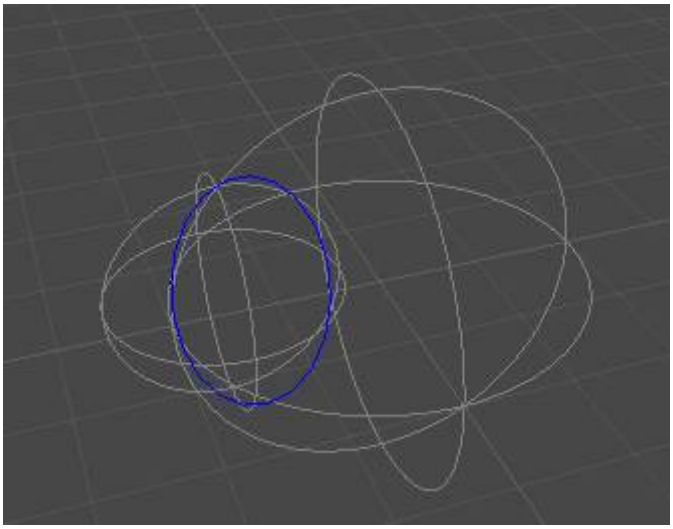


Test Prefab:
Test_IntrBox3Capsule3

Type
`class Distance`
Methods
`static float TestBox3Capsule3(ref Box3 box, ref Capsule3 capsule)` Tests if a box intersects a capsule. Returns true if intersection occurs false otherwise.

Example
`bool intr = Intersection.TestBox3Capsule3(ref box, ref capsule);`

4.2.35 Sphere3-Sphere3



Test Prefab:
Test_IntrSphere3Sphere3

Notice that intersection data uses special enumeration for the intersection type.

Type
`class Intersection`
Methods
`static bool TestSphere3Sphere3(ref Sphere3 sphere0, ref Sphere3 sphere1)` Tests if a sphere intersects another sphere. Returns true if intersection occurs false otherwise.
`static bool FindSphere3Sphere3(ref Sphere3 sphere0, ref Sphere3 sphere1, out Sphere3Sphere3Intr info)` Tests if a sphere intersects another sphere and finds intersection parameters. Returns true if intersection occurs false otherwise.

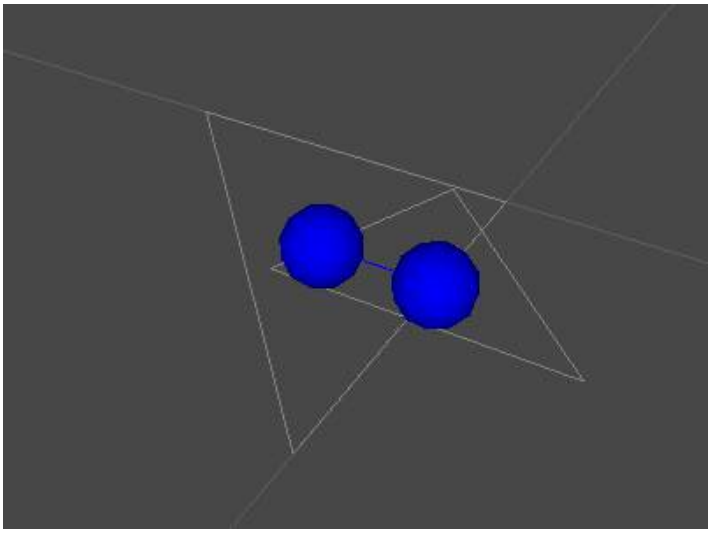
Type
`struct Sphere3Sphere3Intr` Contains information about intersection of Sphere3 and Sphere3
Fields
`Sphere3Sphere3IntrTypes IntersectionType` Equals to: Sphere3Sphere3IntersectionTypes.Empty if no intersection occurs; Sphere3Sphere3IntersectionTypes.Point if spheres are touching in a point and outside of each other; Sphere3Sphere3IntersectionTypes.Circle is spheres intersect (common case); Sphere3Sphere3IntersectionTypes.Sphere0 or Sphere3Sphere3IntersectionTypes.Sphere1 if sphere0 is strictly contained inside sphere1, or sphere1 is strictly contained in sphere0 respectively; Sphere3Sphere3IntersectionTypes.Sphere0Point or Sphere3Sphere3IntersectionTypes.Sphere1Point if sphere0 is contained inside sphere1 and share common point or sphere1 is contained inside sphere0 and share common point; Sphere3Sphere3IntersectionTypes.Same if spheres are esssentialy the same.
`Circle3 Circle` Circle of intersection in case of Sphere3Sphere3IntersectionTypes.Circle
`Vector3 ContactPoint` Contact point in case of Sphere3Sphere3IntersectionTypes.Point, Sphere3Sphere3IntersectionTypes.Sphere0Point, Sphere3Sphere3IntersectionTypes.Sphere1Point

Type
`enum Sphere3Sphere3IntrTypes`
Members
`Empty` Spheres are disjoint/separated
`Point` Spheres touch at point, each sphere outside the other
`Circle` Spheres intersect in a circle
`Sphere0` Sphere0 strictly contained in sphere1
`Sphere0Point` Sphere0 contained in sphere1, share common point
`Sphere1` Sphere1 strictly contained in sphere0
`Sphere1Point` Sphere1 contained in sphere0, share common point
`Same` Spheres are the same

Example

```
bool test = Intersection.TestSphere3Sphere3(ref sphere0, ref sphere1);
Sphere3Sphere3Intr info;
bool find = Intersection.FindSphere3Sphere3(ref sphere0, ref sphere1, out info);
```

4.2.36 Triangle3-Triangle3



Test Prefab:
Test_IntrTriangle3Triangle3

Type
`class Intersection`
Methods

`static bool TestTriangle3Triangle3(ref Triangle3 triangle0, ref Triangle3 triangle1, out IntersectionTypes intersectionType)`
`static bool TestTriangle3Triangle3(ref Triangle3 triangle0, ref Triangle3 triangle1)`
Tests if a triangle intersects another triangle. Returns true if intersection occurs false otherwise.

`static bool FindTriangle3Triangle3(ref Triangle3 triangle0, ref Triangle3 triangle1, out Triangle3Triangle3Intr info, bool reportCoplanarIntersections = false)` Tests if a triangle intersects another triangle and finds intersection parameters. Returns true if intersection occurs false otherwise.

Type
`struct Triangle3Triangle3Intr` Contains information about intersection of Triangle3 and Triangle3
Fields
`IntersectionTypes IntersectionType` Equals to: IntersectionTypes.Empty if no intersection occurs; IntersectionTypes.Point if non-coplanar triangles touch in a point, see Touching member for the description; IntersectionTypes.Segment if non-coplanar triangles intersect or are touch in a segment, see Touching member for the description; IntersectionTypes.Plane if reportCoplanarIntersections is specified to true when calling Find method and triangles are coplanar and intersect, if reportCoplanarIntersections is specified to false, coplanar triangles are reported as not intersecting.
`IntersectionTypes CoplanarIntersectionType` If triangles are non-coplanar equals to IntersectionType.Empty. If triangles are coplanar, equals to the following options: IntersectionTypes.Empty if coplanar triangles do not intersect; IntersectionTypes.Point is coplanar triangles touch in a point; IntersectionTypes.Segment if coplanar triangles touch in a segment; IntersectionTypes.Polygon if coplanar triangles intersect.

`bool Touching`
Equals to true if triangles are non-coplanar and touching in a point (IntersectionTypes.Point; touch variants are: a vertex lies in the plane of a triangle and contained by a triangle (including border), two non-collinear edges touch) or if triangles are not coplanar and touching in a segment (IntersectionTypes.Segment; an edge lies in the plane of a triangle and intersects triangle in more than one point). Generally speaking, touching is true when non-coplanar triangles touch each other by some parts of their borders. Otherwise false.

`int Quantity`
Number of intersection points.
IntersectionTypes.Empty: 0;
IntersectionTypes.Point: 1;
IntersectionTypes.Segment: 2;
IntersectionTypes.Polygon: 1 to 6.

`Vector3 Point0`
Intersection point 0

`Vector3 Point1`
Intersection point 1

`Vector3 Point2`
Intersection point 2

`Vector3 Point3`
Intersection point 3

`Vector3 Point4`
Intersection point 4

`Vector3 Point5`
Intersection point 5

Properties

`Vector3 this[int i] { get; }`
Gets intersection point by index (0 to 5). Points could be also accessed individually using Point0,...,Point5 fields.

Example

```
bool test = Intersection.TestTriangle3Triangle3(ref triangle0, ref triangle1);
Triangle3Triangle3Intr info;
bool find = Intersection.FindTriangle3Triangle3(ref triangle0, ref triangle1, out info, true);
```

5 Distance and Projection

Chapter discusses various methods for calculating distances between a point and a primitive and between two primitives. By their nature distance methods project given points onto primitives thus we can also get projection information which can be useful sometimes.

Distance methods are located inside static class `Distance`. Methods are subdivided into two categories: those which return actual distance (e.g. `Point2Line2`) and those which return squared distance (e.g. `SqrPoint2Line2`). Furthermore, every type could subdivided into methods which return closest point[s] and methods which do not return closest point[s] (in case of finding distance from a point and a primitive, closest point is the projection of a given point onto a primitive; in case of finding distance between two primitives, closest points are points on the primitives between which primitives distance is minimal, obviously in this case there are two closest points, one for each primitive). In most cases methods which calculate squared distance and do not return closest points are fastest, conversely those which return distance and closest points are slowest, other two methods are in-between. Although this does not hold true for every single operation (sometimes methods are equivalent and in very rare cases the order of methods performance is different), thus user is encouraged to choose methods depending on their task.

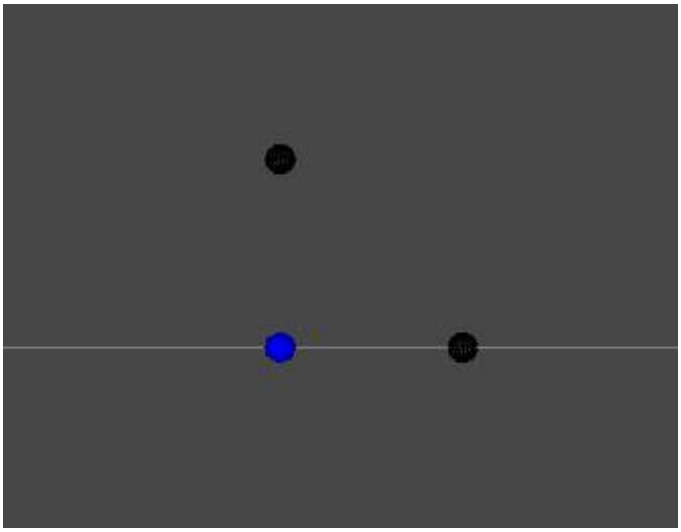
For convenience many primitives contain `DistanceTo` methods which accepts point and return distance from the point to a primitive. These methods just call into Distance class. Primitives also have `Project` method which is a call to Distance class as well, they return input point which is projected onto a primitive. If user needs full control over input and ouput and also needs maximum speed it's advisable to use Distance class directly.

Distance returned by methods is non-negative, however there are few primitives which also have method for calculating signed distance to a point. These methods are named `SignedDistanceTo` and available only on primitives.

Following sub-sections contain tables of distance methods available for primitives and the name of the respective test prefab.

5.1 2D Distance and Projection

5.1.1 Point2-Line2



Test Prefab:
Test_DistPoint2Line2

Type
`class Distance`
Methods
`static float Point2Line2(ref Vector2 point, ref Line2 line)` `static float SqrPoint2Line2(ref Vector2 point, ref Line2 line)`

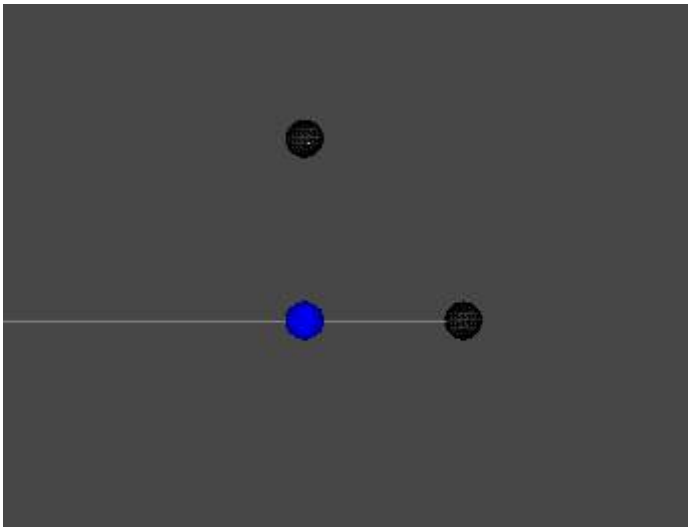
`static float Point2Line2(ref Vector2 point, ref Line2 line, out Vector2 closestPoint)`
`static float SqrPoint2Line2(ref Vector2 point, ref Line2 line, out Vector2 closestPoint)`
closestPoint - Point projected on a line

Type
`struct Line2`
Methods
`float SignedDistanceTo(Vector2 point)`
`float DistanceTo(Vector2 point)`
`Vector2 Project(Vector2 point)`

Example

`Vector2 closestPoint;`
`float dist0 = Distance.Point2Line2(ref point, ref line, out closestPoint);`
`float dist1 = line.DistanceTo(point);`

5.1.2 Point2-Ray2



Test Prefab:
Test_DistPoint2Ray2

Type
`class Distance`
Methods
`static float Point2Ray2(ref Vector2 point, ref Ray2 ray)` `static float SqrPoint2Ray2(ref Vector2 point, ref Ray2 ray)`

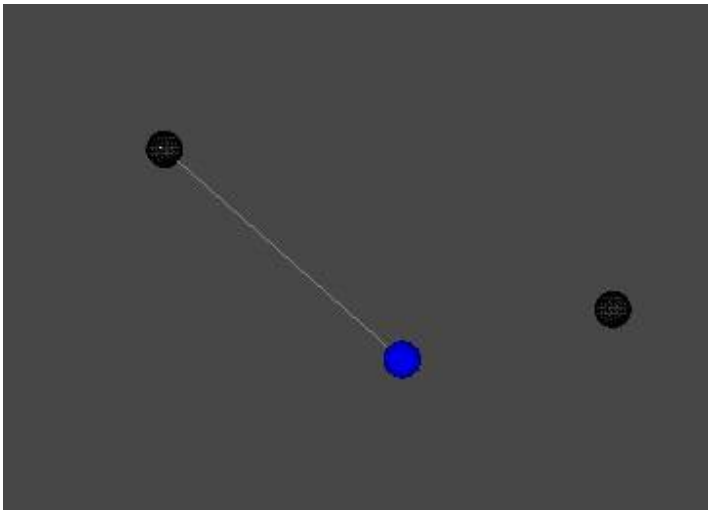
`static float Point2Ray2(ref Vector2 point, ref Ray2 ray, out Vector2 closestPoint)`
`static float SqrPoint2Ray2(ref Vector2 point, ref Ray2 ray, out Vector2 closestPoint)`
closestPoint - Point projected on a ray and clamped by ray origin

Type
`struct Ray2`
Methods
`float DistanceTo(Vector2 point)`
`Vector2 Project(Vector2 point)`

Example

```
Vector2 closestPoint;
float dist0 = Distance.Point2Ray2(ref point, ref ray, out closestPoint);
float dist1 = ray.DistanceTo(point);
```

5.1.3 Point2-Segment2



Test Prefab:
Test_DistPoint2Segment2

Type
`class Distance`
Methods
`static float Point2Segment2(ref Vector2 point, ref Segment2 segment)` `static float SqrPoint2Segment2(ref Vector2 point, ref Segment2 segment)`

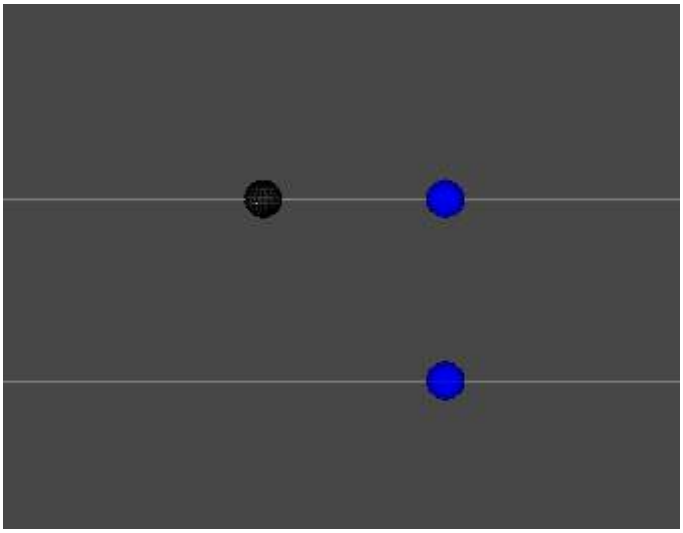
`static float Point2Segment2(ref Vector2 point, ref Segment2 segment, out Vector2 closestPoint)`
`static float SqrPoint2Segment2(ref Vector2 point, ref Segment2 segment, out Vector2 closestPoint)`
closestPoint - Point projected on a segment and clamped by segment endpoints

Type
`struct Segment2`
Methods
`float DistanceTo(Vector2 point)`
`Vector2 Project(Vector2 point)`

Example

`Vector2 closestPoint;`
`float dist0 = Distance.Point2Segment2(ref point, ref segment, out closestPoint);`
`float dist1 = segment.DistanceTo(point);`

5.1.4 Line2-Line2



Test Prefab:
Test_DistLine2Line2

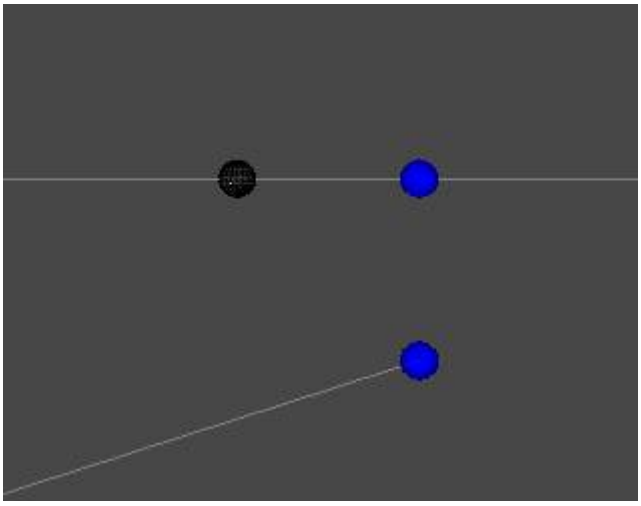
When lines are not intersecting closest points are found, when they are intersecting obviously closest point is intersection point.

Type
`class Distance`
Methods
`static float Line2Line2(ref Line2 line0, ref Line2 line1)` `static float SqrLine2Line2(ref Line2 line0, ref Line2 line1)`

`static float Line2Line2(ref Line2 line0, ref Line2 line1, out Vector2 closestPoint0, out Vector2 closestPoint1)`
`static float SqrLine2Line2(ref Line2 line0, ref Line2 line1, out Vector2 closestPoint0, out Vector2 closestPoint1)`
closestPoint0 - Point on line0 closest to line1
closestPoint1 - Point on line1 closest to line0

Example
`Vector2 closestPoint0, closestPoint1;` `float dist = Distance.Line2Line2(ref line0, ref line1, out closestPoint0, out closestPoint1);`

5.1.5 Line2-Ray2



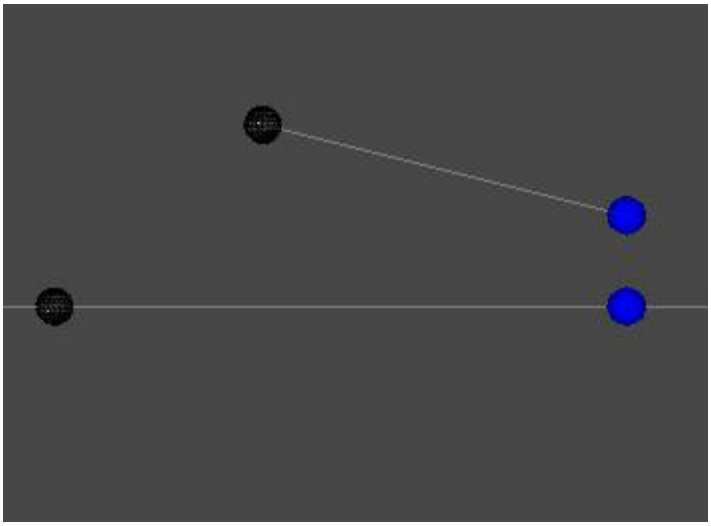
Test Prefab:
Test_DistLine2Ray2

Type
`class Distance`
Methods
`static float Line2Ray2(ref Line2 line, ref Ray2 ray)` `static float SqrLine2Ray2(ref Line2 line, ref Ray2 ray)`

`static float Line2Ray2(ref Line2 line, ref Ray2 ray, out Vector2 closestPoint0, out Vector2 closestPoint1)`
`static float SqrLine2Ray2(ref Line2 line, ref Ray2 ray, out Vector2 closestPoint0, out Vector2 closestPoint1)`
closestPoint0 - Point on line closest to ray
closestPoint1 - Point on ray closest to line

Example
`Vector2 closestPoint0, closestPoint1;` `float dist = Distance.Line2Ray2(ref line, ref ray, out closestPoint0, out closestPoint1);`

5.1.6 Line2-Segment2



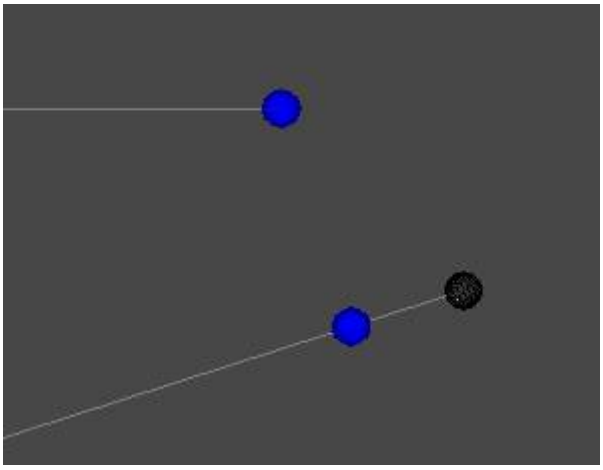
Test Prefab:
Test_DistLine2Segment2

Type
`class Distance`
Methods
`static float Line2Segment2(ref Line2 line, ref Segment2 segment)` `static float SqrLine2Segment2(ref Line2 line, ref Segment2 segment)`

`static float Line2Segment2(ref Line2 line, ref Segment2 segment, out Vector2 closestPoint0, out Vector2 closestPoint1)`
`static float SqrLine2Segment2(ref Line2 line, ref Segment2 segment, out Vector2 closestPoint0, out Vector2 closestPoint1)`
closestPoint0 - Point on line closest to segment
closestPoint1 - Point on segment closest to line

Example
`Vector2 closestPoint0, closestPoint1;` `float dist = Distance.Line2Segment2(ref line, ref segment, out closestPoint0, out closestPoint1);`

5.1.7 Ray2-Ray2



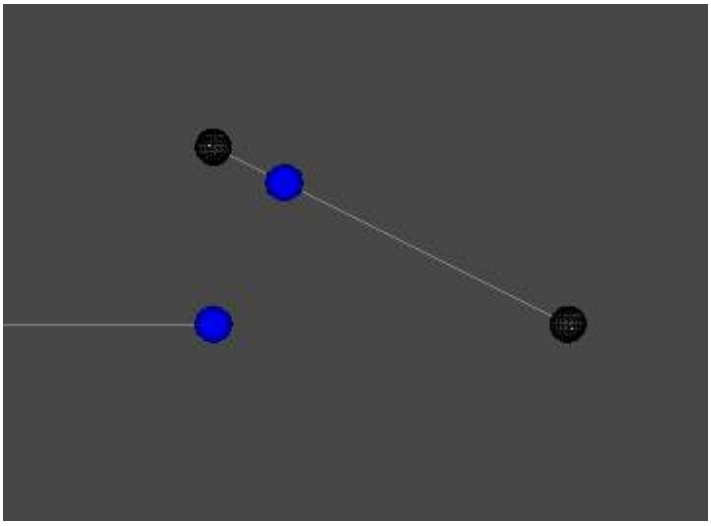
Test Prefab:
Test_DistRay2Ray2

Type
`class Distance`
Methods
`static float Ray2Ray2(ref Ray2 ray0, ref Ray2 ray1)` `static float SqrRay2Ray2(ref Ray2 ray0, ref Ray2 ray1)`

`static float Ray2Ray2(ref Ray2 ray0, ref Ray2 ray1, out Vector2 closestPoint0, out Vector2 closestPoint1)`
`static float SqrRay2Ray2(ref Ray2 ray0, ref Ray2 ray1, out Vector2 closestPoint0, out Vector2 closestPoint1)`
closestPoint0 - Point on ray0 closest to ray1
closestPoint1 - Point on ray1 closest to ray0

Example
`Vector2 closestPoint0, closestPoint1;` `float dist = Distance.Ray2Ray2(ref ray0, ref ray1, out closestPoint0, out closestPoint1);`

5.1.8 Ray2-Segment2



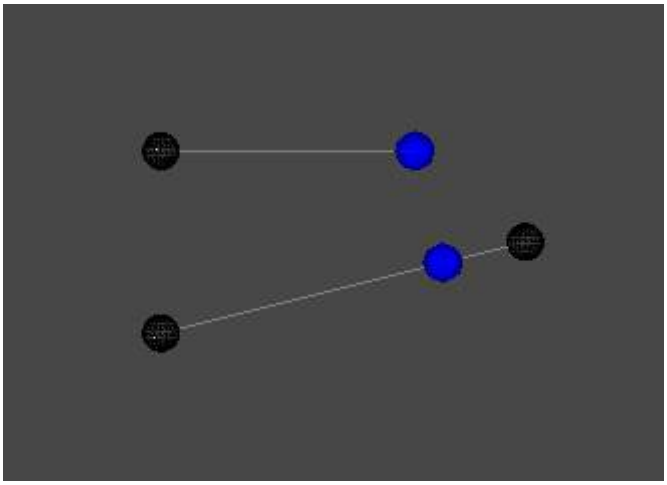
Test Prefab:
Test_DistRay2Segment2

Type
`class Distance`
Methods
`static float Ray2Segment2(ref Ray2 ray, ref Segment2 segment)` `static float SqrRay2Segment2(ref Ray2 ray, ref Segment2 segment)`

`static float Ray2Segment2(ref Ray2 ray, ref Segment2 segment, out Vector2 closestPoint0, out Vector2 closestPoint1)`
`static float SqrRay2Segment2(ref Ray2 ray, ref Segment2 segment, out Vector2 closestPoint0, out Vector2 closestPoint1)`
closestPoint0 - Point on ray closest to segment
closestPoint1 - Point on segment closest to ray

Example
`Vector2 closestPoint0, closestPoint1;` `float dist = Distance.Ray2Segment2(ref ray, ref segment, out closestPoint0, out closestPoint1);`

5.1.9 Segment2-Segment2



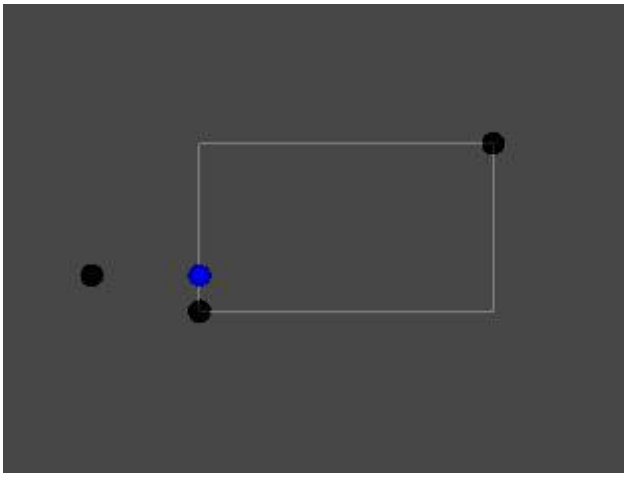
Test Prefab:
Test_DistSegment2Segment2

Type
`class Distance`
Methods
`static float Segment2Segment2(ref Segment2 segment0, ref Segment2 segment1)` `static float SqrSegment2Segment2(ref Segment2 segment0, ref Segment2 segment1)`

`static float Segment2Segment2(ref Segment2 segment0, ref Segment2 segment1, out Vector2 closestPoint0, out Vector2 closestPoint1)`
`static float SqrSegment2Segment2(ref Segment2 segment0, ref Segment2 segment1, out Vector2 closestPoint0, out Vector2 closestPoint1)`
closestPoint0 - Point on segment0 closest to segment1
closestPoint1 - Point on segment1 closest to segment0

Example
`Vector2 closestPoint0, closestPoint1;` `float dist = Distance.Segment2Segment2(ref segment0, ref segment1, out closestPoint0, out closestPoint1);`

5.1.10 Point2-AAB2



Test Prefab:
Test_DistPoint2AAB2

Box considered to be solid, if a point is inside a box then distance is zero and closest point is input point.

Type
`class Distance`
Methods
`static float Point2AAB2(ref Vector2 point, ref AAB2 box)` `static float SqrPoint2AAB2(ref Vector2 point, ref AAB2 box)`

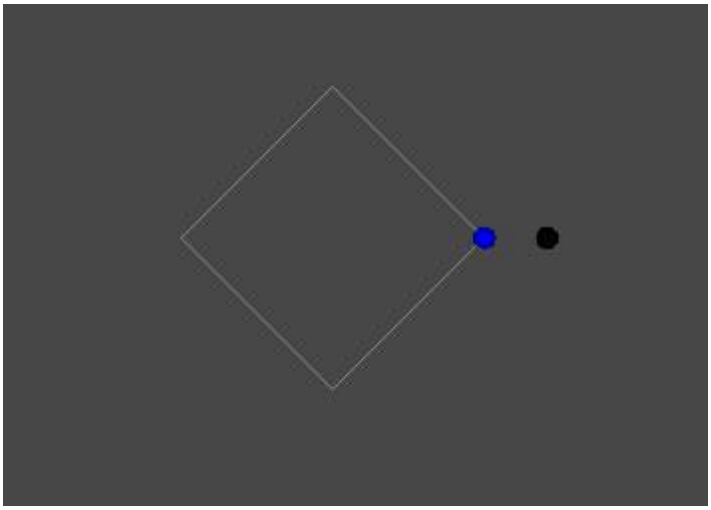
`static float Point2AAB2(ref Vector2 point, ref AAB2 box, out Vector2 closestPoint)`
`static float SqrPoint2AAB2(ref Vector2 point, ref AAB2 box, out Vector2 closestPoint)`
closestPoint - Point projected on an aab

Type
`struct AAB2`
Methods
`float DistanceTo(Vector2 point)`
`Vector2 Project(Vector2 point)`

Example

`Vector2 closestPoint;`
`float dist = Distance.Point2AAB2(ref point, ref box, out closestPoint);`
`float dist1 = Distance.SqrPoint2AAB2(ref point, ref box);`
`float dist2 = box.DistanceTo(point);`

5.1.11 Point2-Box2



Test Prefab:
Test_DistPoint2Box2

Box considered to be solid, if a point is inside a box then distance is zero and closest point is input point.

Type
`class Distance`
Methods
`static float Point2Box2(ref Vector2 point, ref Box2 box)` `static float SqrPoint2Box2(ref Vector2 point, ref Box2 box)`

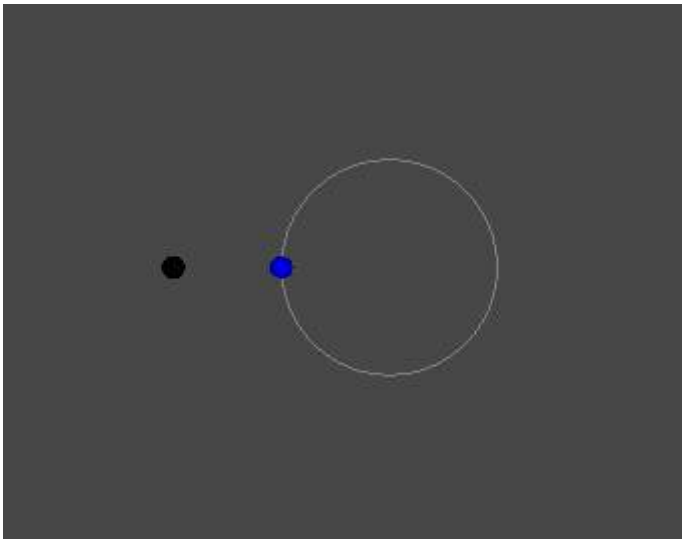
`static float Point2Box2(ref Vector2 point, ref Box2 box, out Vector2 closestPoint)`
`static float SqrPoint2Box2(ref Vector2 point, ref Box2 box, out Vector2 closestPoint)`
closestPoint - Point projected on a box

Type
`struct Box2`
Methods
`float DistanceTo(Vector2 point)`
`Vector2 Project(Vector2 point)`

Example

```
Vector2 closestPoint;
float dist = Distance.Point2Box2(ref point, ref box, out closestPoint);
```

5.1.12 Point2-Circle2



Test Prefab:
Test_DistPoint2Circle2

Circle considered to be solid, if a point is inside a circle then distance is zero and closest point is input point.

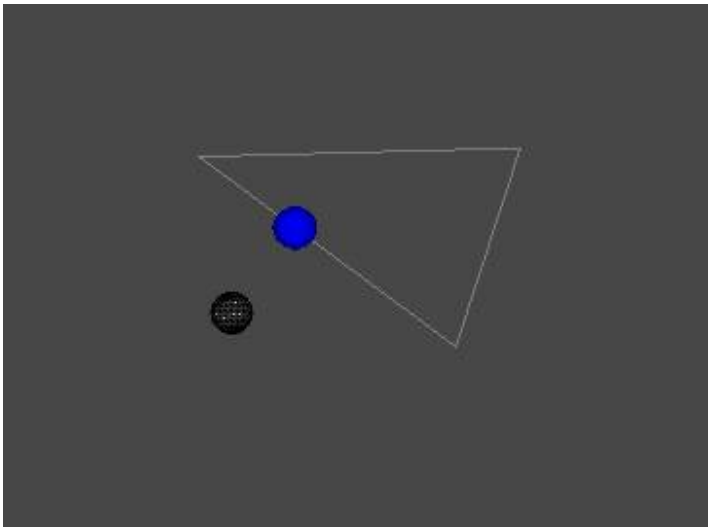
Type
`class Distance`
Methods
`static float Point2Circle2(ref Vector2 point, ref Circle2 circle)` `static float SqrPoint2Circle2(ref Vector2 point, ref Circle2 circle)`

`static float Point2Circle2(ref Vector2 point, ref Circle2 circle, out Vector2 closestPoint)`
`static float SqrPoint2Circle2(ref Vector2 point, ref Circle2 circle, out Vector2 closestPoint)`
closestPoint - Point projected on a circle

Type
`struct Circle2`
Methods
`float DistanceTo(Vector2 point)`
`Vector2 Project(Vector2 point)`

Example
`Vector2 closestPoint;` `float dist = Distance.Point2Circle2(ref point, ref circle, out closestPoint);`

5.1.13 Point2-Triangle2



Test Prefab:
Test_DistPoint2Triangle2

Triangle considered to be solid, if a point is inside a triangle then distance is zero and closest point is input point.

Type
`class Distance`
Methods
`static float Point2Triangle2(ref Vector2 point, ref Triangle2 triangle)` `static float SqrPoint2Triangle2(ref Vector2 point, ref Triangle2 triangle)`

`static float Point2Triangle2(ref Vector2 point, ref Triangle2 triangle, out Vector2 closestPoint)`
`static float SqrPoint2Triangle2(ref Vector2 point, ref Triangle2 triangle, out Vector2 closestPoint)`
closestPoint - Point projected on a triangle

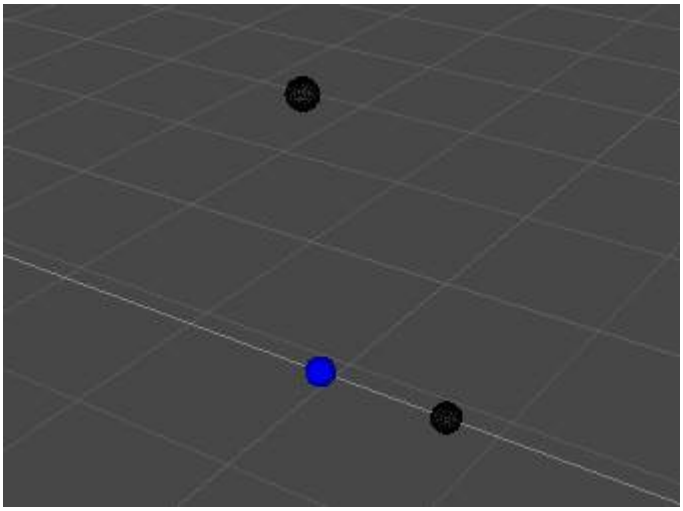
Type
`struct Triangle2`
Methods
`float DistanceTo(Vector2 point)`
`Vector2 Project(Vector2 point)`

Example

`Vector2 closestPoint;`
`float dist = Distance.Point2Triangle2(ref point, ref triangle, out closestPoint);`
`float dist1 = Distance.SqrPoint2Triangle2(ref point, ref triangle, out closestPoint);`

5.2 3D Distance and Projection

5.2.1 Point3-Line3



Test Prefab:
Test_DistPoint3Line3

Type
`class Distance`
Methods
`static float Point3Line3(ref Vector3 point, ref Line3 line)` `static float SqrPoint3Line3(ref Vector3 point, ref Line3 line)`

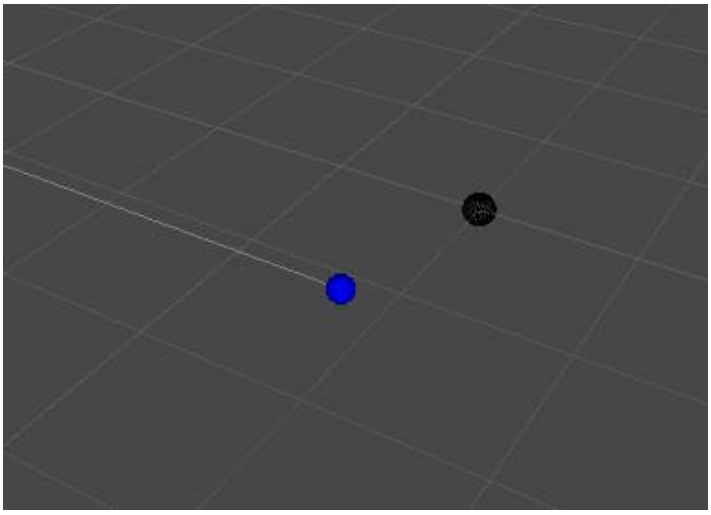
`static float Point3Line3(ref Vector3 point, ref Line3 line, out Vector3 closestPoint)`
`static float SqrPoint3Line3(ref Vector3 point, ref Line3 line, out Vector3 closestPoint)`
closestPoint - Point projected on a line

Type
`struct Line3`
Methods
`float DistanceTo(Vector3 point)`
`Vector3 Project(Vector3 point)`

Example

`Vector3 closestPoint;`
`float dist0 = Distance.Point3Line3(ref point, ref line, out closestPoint);`
`float dist1 = line.DistanceTo(point);`

5.2.2 Point3-Ray3



Test Prefab:
Test_DistPoint3Ray3

Type
`class Distance`
Methods
`static float Point3Ray3(ref Vector3 point, ref Ray3 ray)` `static float SqrPoint3Ray3(ref Vector3 point, ref Ray3 ray)`

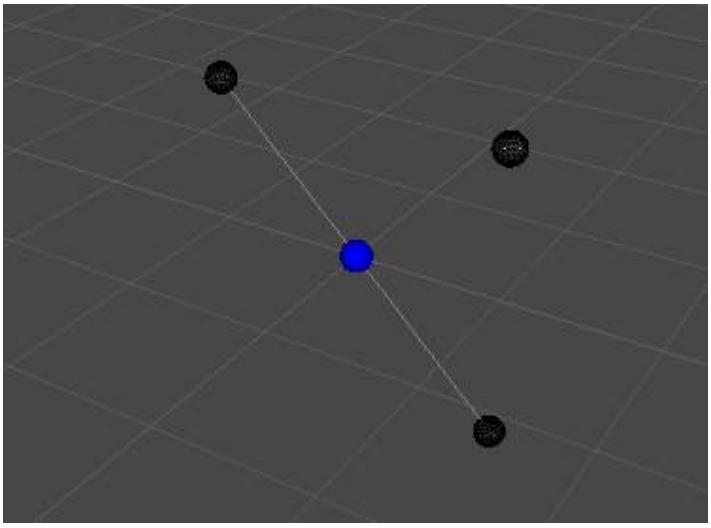
`static float Point3Ray3(ref Vector3 point, ref Ray3 ray, out Vector3 closestPoint)`
`static float SqrPoint3Ray3(ref Vector3 point, ref Ray3 ray, out Vector3 closestPoint)`
closestPoint - Point projected on a ray and clamped by ray origin

Type
`struct Ray3`
Methods
`float DistanceTo(Vector3 point)`
`Vector3 Project(Vector3 point)`

Example

```
Vector3 closestPoint;
float dist0 = Distance.Point3Ray3(ref point, ref ray, out closestPoint);
float dist1 = ray.DistanceTo(point);
```

5.2.3 Point3-Segment3



Test Prefab:
Test_DistPoint3Segment3

Type
`class Distance`
Methods
`static float Point3Segment3(ref Vector3 point, ref Segment3 segment)` `static float SqrPoint3Segment3(ref Vector3 point, ref Segment3 segment)`

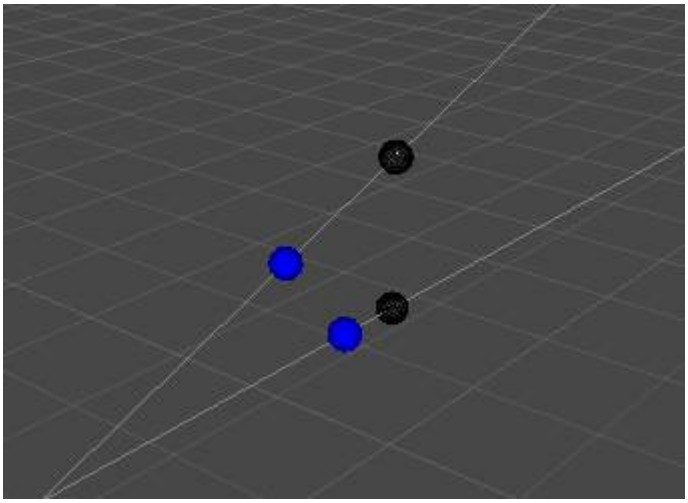
`static float Point3Segment3(ref Vector3 point, ref Segment3 segment, out Vector3 closestPoint)`
`static float SqrPoint3Segment3(ref Vector3 point, ref Segment3 segment, out Vector3 closestPoint)`
closestPoint - Point projected on a segment and clamped by segment endpoints

Type
`struct Segment3`
Methods
`float DistanceTo(Vector3 point)`
`Vector3 Project(Vector3 point)`

Example

`Vector3 closestPoint;`
`float dist0 = Distance.Point3Segment3(ref point, ref segment, out closestPoint);`
`float dist1 = segment.DistanceTo(point);`

5.2.4 Line3-Line3



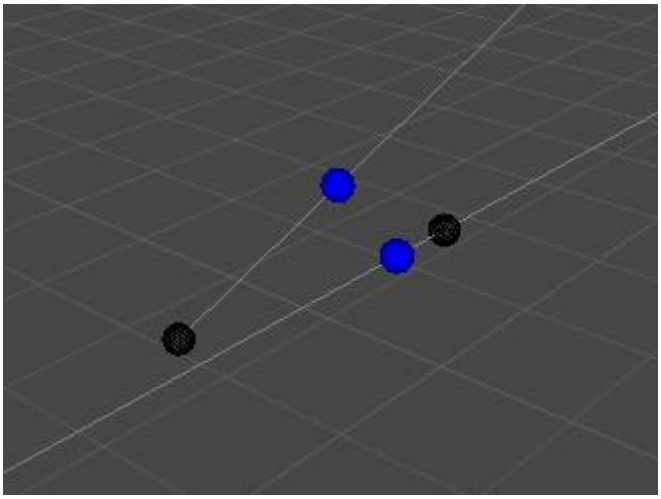
Test Prefab:
Test_DistLine3Line3

Type
`class Distance`
Methods
`static float Line3Line3(ref Line3 line0, ref Line3 line1)` `static float SqrLine3Line3(ref Line3 line0, ref Line3 line1)`

`static float Line3Line3(ref Line3 line0, ref Line3 line1, out Vector3 closestPoint0, out Vector3 closestPoint1)`
`static float SqrLine3Line3(ref Line3 line0, ref Line3 line1, out Vector3 closestPoint0, out Vector3 closestPoint1)`
closestPoint0 - Point on line0 closest to line1
closestPoint1 - Point on line1 closest to line0

Example
`Vector3 closestPoint0, closestPoint1;` `float dist = Distance.Line3Line3(ref line0, ref line1, out closestPoint0, out closestPoint1);`

5.2.5 Line3-Ray3



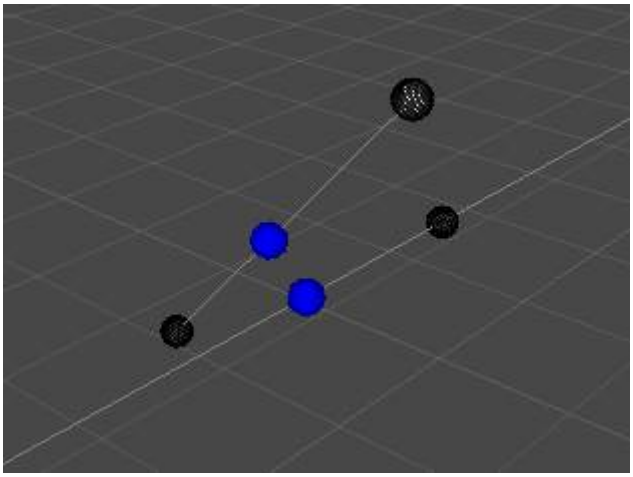
Test Prefab:
Test_DistLine3Ray3

Type
`class Distance`
Methods
`static float Line3Ray3(ref Line3 line, ref Ray3 ray)` `static float SqrLine3Ray3(ref Line3 line, ref Ray3 ray)`

`static float Line3Ray3(ref Line3 line, ref Ray3 ray, out Vector3 closestPoint0, out Vector3 closestPoint1)`
`static float SqrLine3Ray3(ref Line3 line, ref Ray3 ray, out Vector3 closestPoint0, out Vector3 closestPoint1)`
closestPoint0 - Point on line closest to ray
closestPoint1 - Point on ray closest to line

Example
`Vector3 closestPoint0, closestPoint1;` `float dist = Distance.Line3Ray3(ref line, ref ray, out closestPoint0, out closestPoint1);`

5.2.6 Line3-Segment3



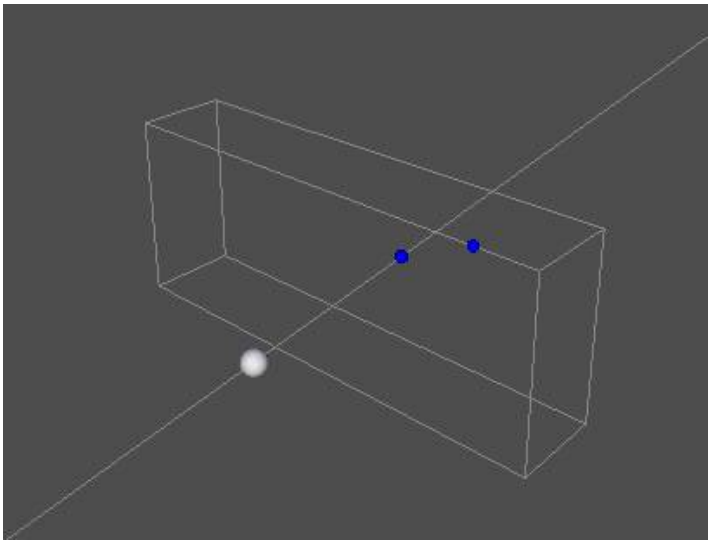
Test Prefab:
Test_DistLine3Segment3

Type
`class Distance`
Methods
`static float Line3Segment3(ref Line3 line, ref Segment3 segment)` `static float SqrLine3Segment3(ref Line3 line, ref Segment3 segment)`

`static float Line3Segment3(ref Line3 line, ref Segment3 segment, out Vector3 closestPoint0, out Vector3 closestPoint1)`
`static float SqrLine3Segment3(ref Line3 line, ref Segment3 segment, out Vector3 closestPoint0, out Vector3 closestPoint1)`
closestPoint0 - Point on line closest to segment
closestPoint1 - Point on segment closest to line

Example
Vector3 closestPoint0, closestPoint1; float dist = Distance.Line3Segment3(ref line, ref segment, out closestPoint0, out closestPoint1);

5.2.7 Line3-Box3



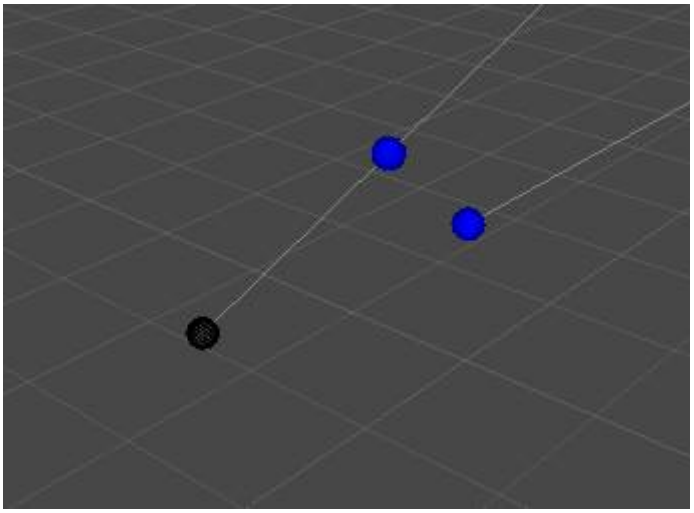
Test Prefab:
Test_DistLine3Box3

Type
`class Distance`
Methods
`static float Line3Box3(ref Line3 line, ref Box3 box)` `static float SqrLine3Box3(ref Line3 line, ref Box3 box)`

`static float Line3Box3(ref Line3 line, ref Box3 box, out Line3Box3Dist info)`
`static float SqrLine3Box3(ref Line3 line, ref Box3 box, out Line3Box3Dist info)`
info – Contains closest points and line parameter

Example
Line3Box3Dist info; float dist = Distance.Line3Box3(ref line, ref box, out info);

5.2.8 Ray3-Ray3



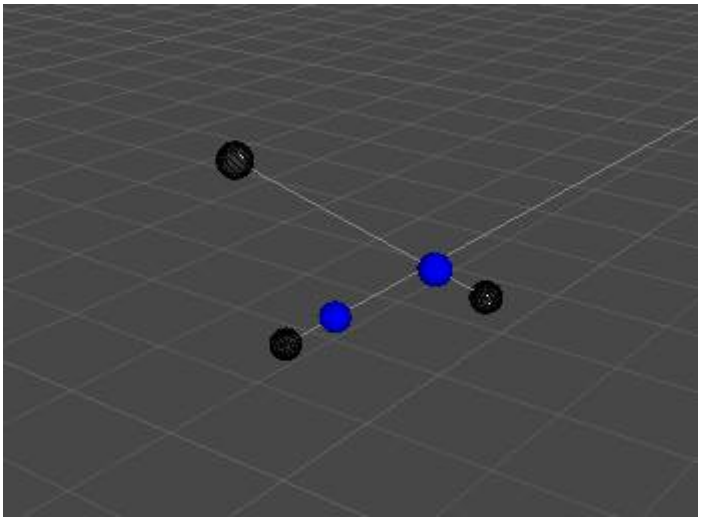
Test Prefab:
Test_DistRay3Ray3

Type
`class Distance`
Methods
`static float Ray3Ray3(ref Ray3 ray0, ref Ray3 ray1)` `static float SqrRay3Ray3(ref Ray3 ray0, ref Ray3 ray1)`

`static float Ray3Ray3(ref Ray3 ray0, ref Ray3 ray1, out Vector3 closestPoint0, out Vector3 closestPoint1)`
`static float SqrRay3Ray3(ref Ray3 ray0, ref Ray3 ray1, out Vector3 closestPoint0, out Vector3 closestPoint1)`
closestPoint0 - Point on ray0 closest to ray1
closestPoint1 - Point on ray1 closest to ray0

Example
`Vector3 closestPoint0, closestPoint1;` `float dist = Distance.Ray3Ray3(ref ray0, ref ray1, out closestPoint0, out closestPoint1);`

5.2.9 Ray3-Segment3



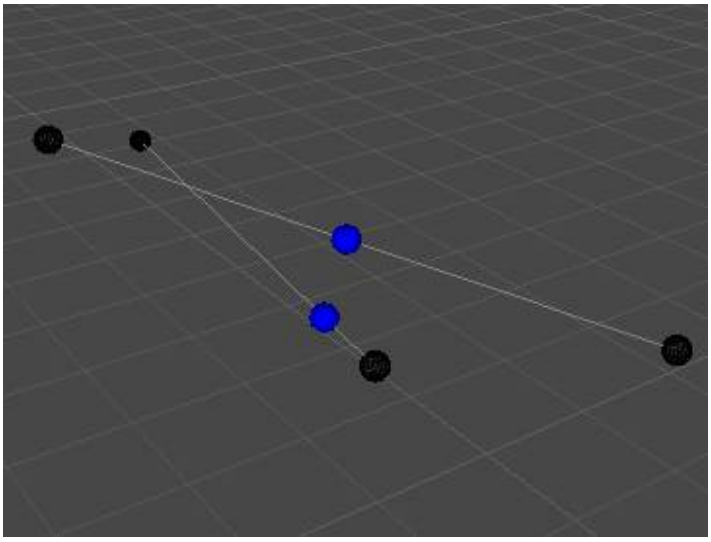
Test Prefab:
Test_DistRay3Segment3

Type
`class Distance`
Methods
`static float Ray3Segment3(ref Ray3 ray, ref Segment3 segment)` `static float SqrRay3Segment3(ref Ray3 ray, ref Segment3 segment)`

`static float Ray3Segment3(ref Ray3 ray, ref Segment3 segment, out Vector3 closestPoint0, out Vector3 closestPoint1)`
`static float SqrRay3Segment3(ref Ray3 ray, ref Segment3 segment, out Vector3 closestPoint0, out Vector3 closestPoint1)`
closestPoint0 - Point on ray closest to segment
closestPoint1 - Point on segment closest to ray

Example
`Vector3 closestPoint0, closestPoint1;` `float dist = Distance.Ray3Segment3(ref ray, ref segment, out closestPoint0, out closestPoint1);`

5.2.10 Segment3-Segment3



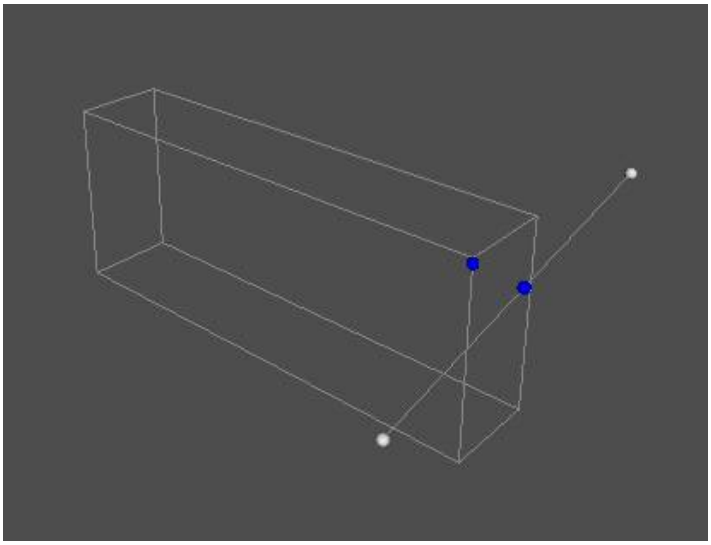
Test Prefab:
Test_DistSegment3Segment3

Type
`class Distance`
Methods
`static float Segment3Segment3(ref Segment3 segment0, ref Segment3 segment1)` `static float SqrSegment3Segment3(ref Segment3 segment0, ref Segment3 segment1)`

`static float Segment3Segment3(ref Segment3 segment0, ref Segment3 segment1, out Vector3 closestPoint0, out Vector3 closestPoint1)`
`static float SqrSegment3Segment3(ref Segment3 segment0, ref Segment3 segment1, out Vector3 closestPoint0, out Vector3 closestPoint1)`
closestPoint0 - Point on segment0 closest to segment1
closestPoint1 - Point on segment1 closest to segment0

Example
`Vector3 closestPoint0, closestPoint1;` `float dist = Distance.Segment3Segment3(ref segment0, ref segment1, out closestPoint0, out closestPoint1);`

5.2.11 Segment3-Box3



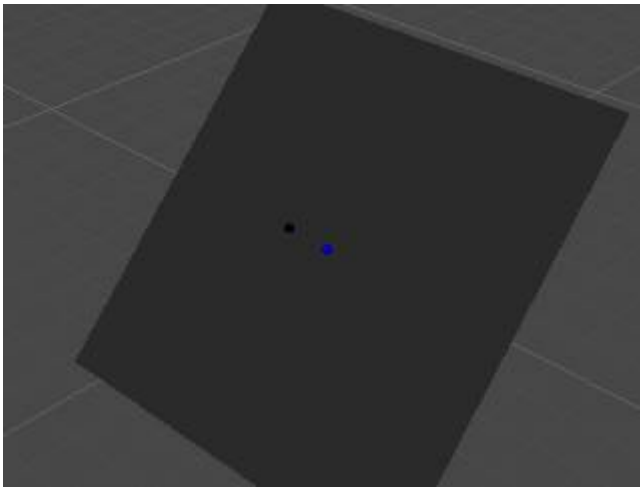
Test Prefab:
Test_DistSegment3Box3

Type
`class Distance`
Methods
`static float Segment3Box3(ref Segment3 segment, ref Box3 box)` `static float SqrSegment3Box3(ref Segment3 segment, ref Box3 box)`

`static float Segment3Box3(ref Segment3 segment, ref Box3 box, out Vector3 closestPoint0, out Vector3 closestPoint1)`
`static float SqrSegment3Box3(ref Segment3 segment, ref Box3 box, out Vector3 closestPoint0, out Vector3 closestPoint1)`
closestPoint0 - Point closest to segment
closestPoint1 - Point closest to box

Example
`Vector3 closestPoint0, closestPoint1;` `float dist = Distance.Segment3Box3(ref segment, ref box, out closestPoint0, out closestPoint1);`

5.2.12 Point3-Plane3



Test Prefab:
Test_DistPoint3Plane3

Type
`class Distance`
Methods
`static float Point3Plane3(ref Vector3 point, ref Plane3 plane)` `static float SqrPoint3Plane3(ref Vector3 point, ref Plane3 plane)`

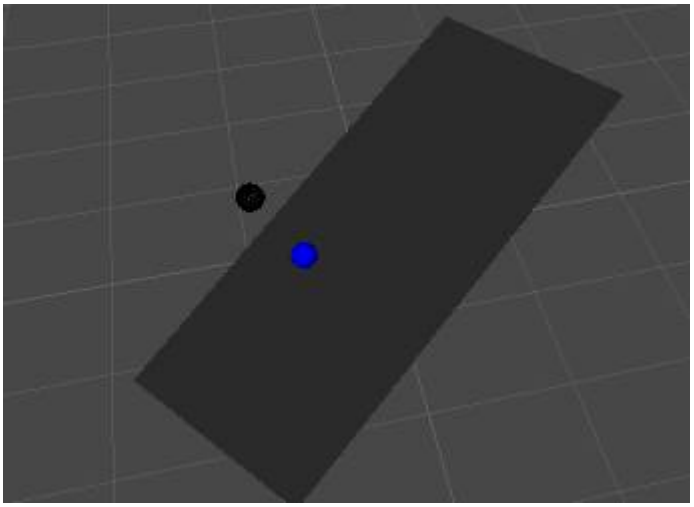
`static float Point3Plane3(ref Vector3 point, ref Plane3 plane, out Vector3 closestPoint)`
`static float SqrPoint3Plane3(ref Vector3 point, ref Plane3 plane, out Vector3 closestPoint)`
closestPoint - Point projected on a plane

Type
`struct Plane3`
Methods
`float SignedDistanceTo(Vector3 point)`
`float DistanceTo(Vector3 point)`
`Vector3 Project(Vector3 point)`
`Vector3 ProjectVector(Vector3 vector)` Projects a vector onto the plane

Example

`Vector3 closestPoint;`
`float dist0 = Distance.Point3Plane3(ref point, ref plane, out closestPoint);`
`float dist1 = plane.DistanceTo(point);`

5.2.13 Point3-Rectangle3



Test Prefab:
Test_DistPoint3Rectangle3

Rectangle considered to be solid.

Type
`class Distance`
Methods
`static float Point3Rectangle3(ref Vector3 point, ref Rectangle3 rectangle)` `static float SqrPoint3Rectangle3(ref Vector3 point, ref Rectangle3 rectangle)`

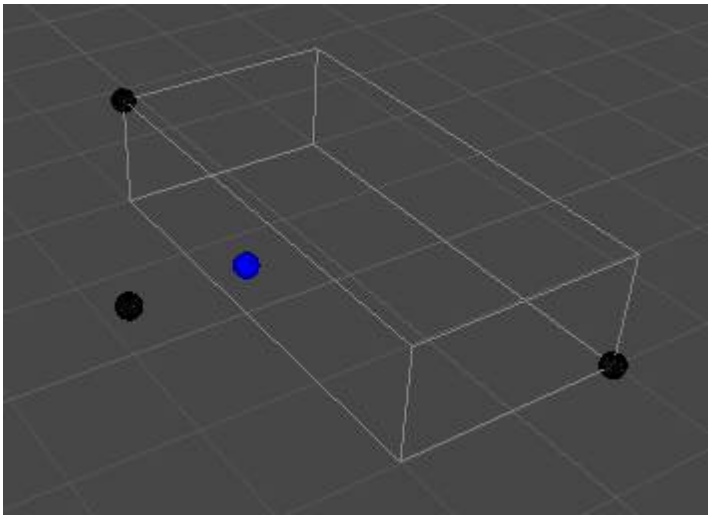
`static float Point3Rectangle3(ref Vector3 point, ref Rectangle3 rectangle, out Vector3 closestPoint)`
`static float SqrPoint3Rectangle3(ref Vector3 point, ref Rectangle3 rectangle, out Vector3 closestPoint)`
closestPoint - Point projected on a rectangle

Type
`struct Rectangle3`
Methods
`float DistanceTo(Vector3 point)`
`Vector3 Project(Vector3 point)`

Example

`Vector3 closestPoint;`
`float dist = Distance.Point3Rectangle3(ref point, ref rectangle, out closestPoint);`
`float dist1 = Distance.SqrPoint3Rectangle3(ref point, ref rectangle, out closestPoint);`
`float dist2 = rectangle.DistanceTo(point);`

5.2.14 Point3-AAB3



Test Prefab:
Test_DistPoint3AAB3

Box considered to be solid, if a point is inside a box then distance is zero and closest point is input point.

Type
`class Distance`
Methods
`static float Point3AAB3(ref Vector3 point, ref AAB3 box)` `static float SqrPoint3AAB3(ref Vector3 point, ref AAB3 box)`

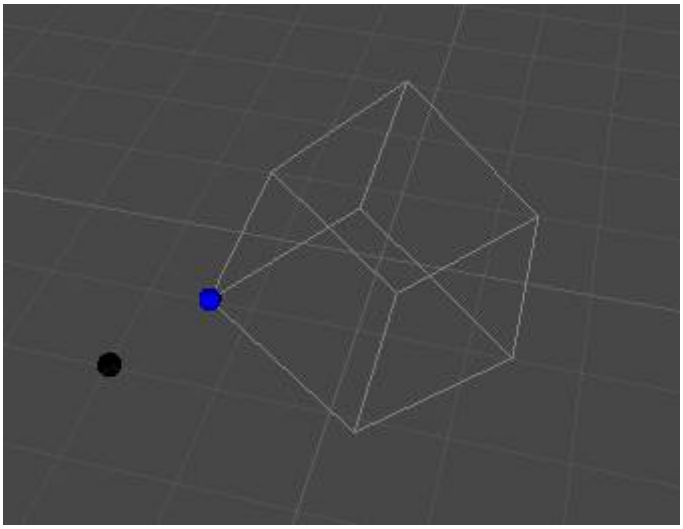
`static float Point3AAB3(ref Vector3 point, ref AAB3 box, out Vector3 closestPoint)`
`static float SqrPoint3AAB3(ref Vector3 point, ref AAB3 box, out Vector3 closestPoint)`
closestPoint - Point projected on an aab

Type
`struct AAB3`
Methods
`float DistanceTo(Vector3 point)`
`Vector3 Project(Vector3 point)`

Example

`Vector3 closestPoint;`
`float dist = Distance.Point3AAB3(ref point, ref box, out closestPoint);`
`float dist1 = Distance.SqrPoint3AAB3(ref point, ref box);`
`float dist2 = box.DistanceTo(point);`

5.2.15 Point3-Box3



Test Prefab:
Test_DistPoint3Box3

Box considered to be solid, if a point is inside a box then distance is zero and closest point is input point.

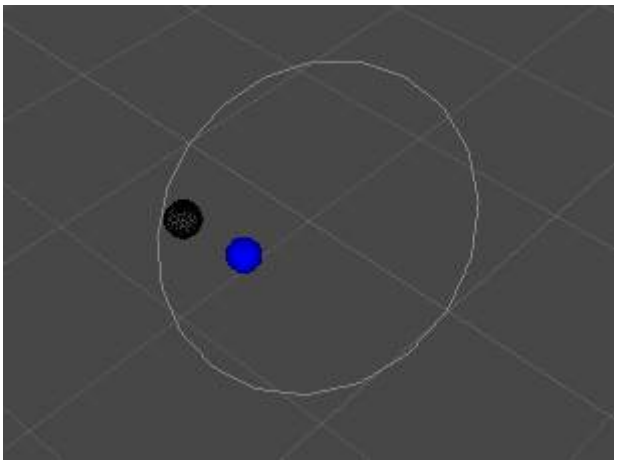
Type
`class Distance`
Methods
`static float Point3Box3(ref Vector3 point, ref Box3 box)` `static float SqrPoint3Box3(ref Vector3 point, ref Box3 box)`

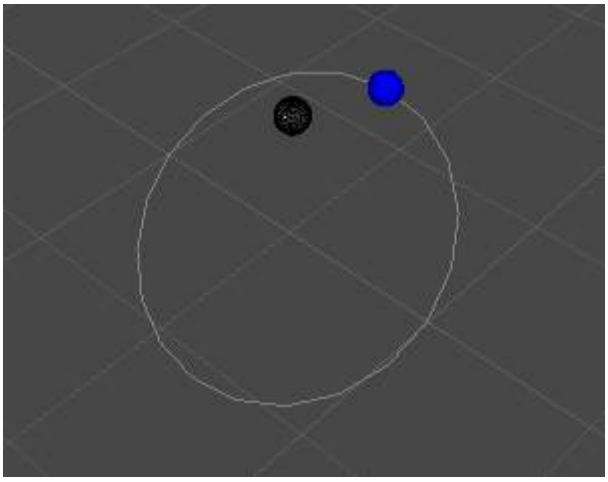
`static float Point3Box3(ref Vector3 point, ref Box3 box, out Vector3 closestPoint)`
`static float SqrPoint3Box3(ref Vector3 point, ref Box3 box, out Vector3 closestPoint)`
closestPoint - Point projected on a box

Type
`struct Box3`
Methods
`float DistanceTo(Vector3 point)`
`Vector3 Project(Vector3 point)`

Example
`Vector3 closestPoint;` `float dist = Distance.Point3Box3(ref point, ref box, out closestPoint);`

5.2.16 Point3-Circle3





Test Prefab:
Test_DistPoint3Circle3
Test_DistPoint3HollowCircle3

Circle methods have option specifying whether or not a circle should be treated solid. Default parameter value is true, meaning that by default circle considered to be solid.

When a circle is solid then it has area and when circle is hollow it has only border (i.e. it's like a ring). When a circle is hollow then distance is calculated to the border a circle.

Compare two images. Upper image uses solid circle. You can see that the point is projected to the inner area of the circle while on the lower image the point is projected onto the border.

Type
`class Distance`
Methods
`static float Point3Circle3(ref Vector3 point, ref Circle3 circle, bool solid = true)` `static float SqrPoint3Circle3(ref Vector3 point, ref Circle3 circle, bool solid = true)`
`static float Point3Circle3(ref Vector3 point, ref Circle3 circle, out Vector3 closestPoint, bool solid = true)` `static float SqrPoint3Circle3(ref Vector3 point, ref Circle3 circle, out Vector3 closestPoint, bool solid = true)`

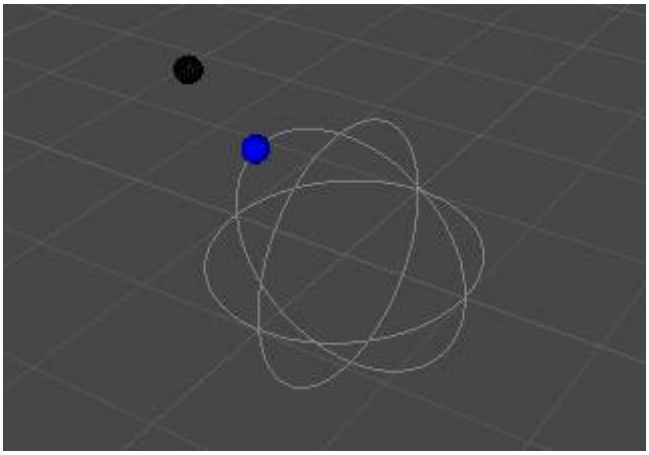
Type
`struct Ray3`
Methods
`float DistanceTo(Vector3 point, bool solid = true)`
`Vector3 Project(Vector3 point, bool solid = true)`

Example

```
// Example for solid circle
Vector3 closestPoint;
float dist = Distance.Point3Circle3(ref point, ref circle, out closestPoint);
float dist1 = Distance.SqrPoint3Circle3(ref point, ref circle);
```

```
float dist2 = circle.DistanceTo(point);
```

5.2.17 Point3-Sphere3



Test Prefab:
Test_DistPoint3Sphere3

Sphere considered to be solid, if a point is inside a sphere then distance is zero and closest point is input point.

Type
`class Distance`
Methods
`static float Point3Sphere3(ref Vector3 point, ref Sphere3 sphere)` `static float SqrPoint3Sphere3(ref Vector3 point, ref Sphere3 sphere)`

`static float Point3Sphere3(ref Vector3 point, ref Sphere3 sphere, out Vector3 closestPoint)`
`static float SqrPoint3Sphere3(ref Vector3 point, ref Sphere3 sphere, out Vector3 closestPoint)`
Point projected on a sphere

Type
`struct Sphere3`
Methods
`float DistanceTo(Vector3 point)`
`Vector3 Project(Vector3 point)`

Example
`Vector3 closestPoint;` `float dist = Distance.Point3Sphere3(ref point, ref sphere, out closestPoint);`

6 Point Containment and Bounding Objects

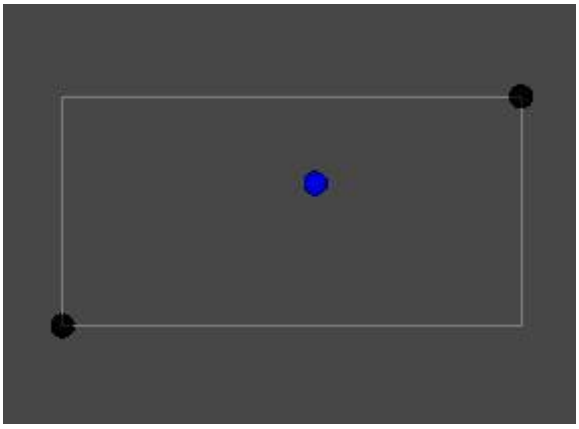
Test whether a point is contained by an object is very common. The library provides methods for testing 2D and 3D points for various primitives. It should be noted that only closed primitives can define containment test (thus, for example, we cannot define containment test for a line). Containment methods are available on primitives, such methods are named `Contains`, accept point and return boolean value stating whether a point is inside a primitive.

Constructing bounding objects usually involve finding an object which contains all the given points. Many ways exists to construct some of the primitives. Library contains some of them (mostly those which are fastest in terms of performance). Although this is not the only way to construct bounding objects.

To create bounding area of volume from a given set of points one should use static methods available on primitives. Their name starts with `CreateFrom***` (e.g. `CreateFromPoints`). Second way is to create empty object (by using ordinary constructor or special static method) and then grow it by including points or similar objects. Different primitives handle this problem using slightly different ways. Methods which grow current primitive are named `Include` and accept a point or a primitive of the same kind. Note that only construction of axis-aligned boxes, oriented boxes and circles/spheres is described in the chapter as these primitives are usually used as bounding objects.

6.1 2D Point Containment

6.1.1 Point2 in AAB2

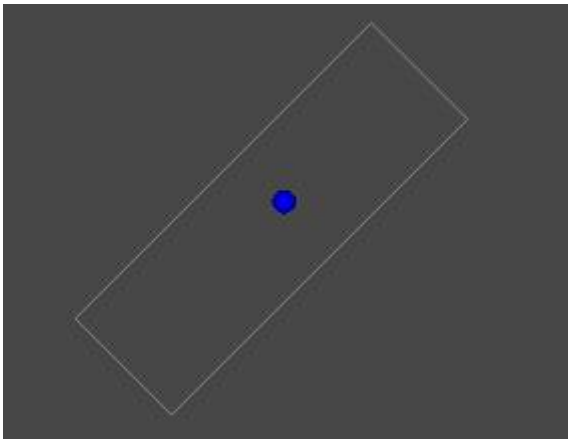


Test Prefab:
Test_ContPoint2AAB2

Type
`struct AAB2`
Methods
`bool Contains(ref Vector2 point)`
`bool Contains(Vector2 point)`

Example
`bool cont = aab.Contains(point);`

6.1.2 Point2 in Box2

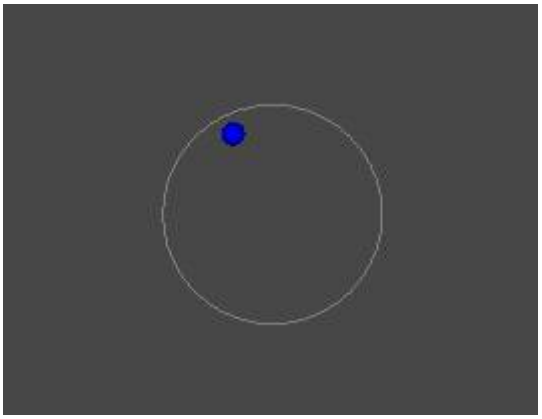


Test Prefab:
Test_ContPoint2Box2

Type
`struct Box2`
Methods
`bool Contains(ref Vector2 point)`
`bool Contains(Vector2 point)`

Example
`bool cont = box.Contains(point);`

6.1.3 Point2 in Circle2

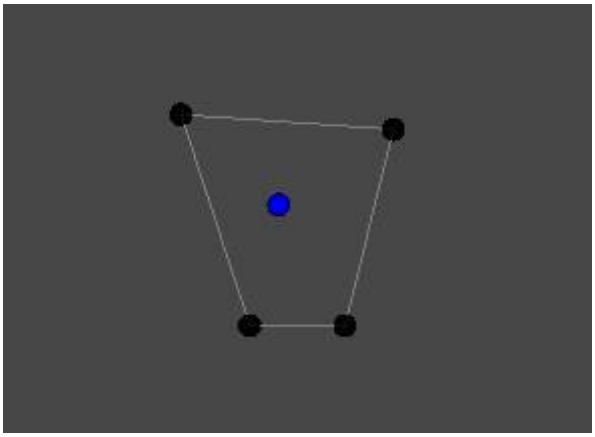


Test Prefab:
Test_ContPoint2Circle2

Type
`struct Circle2`
Methods
`bool Contains(ref Vector2 point)`
`bool Contains(Vector2 point)`

Example
`bool cont = circle.Contains(point);`

6.1.4 Point2 in Convex Polygon2



Test Prefab:
Test_ContPoint2ConvexPolygon2

For polygons there are several ways to determine whether a point is included in a polygon. These methods depend on whether a polygon is convex or concave. It's also important for convex polygons to know their orientation for test to be processed. This sub-section describes only methods for convex polygons, next sub-section also describes methods for any polygons.

To test whether a polygon is convex and to find its orientation (clockwise or counterclockwise) one should use `IsConvex` method of the `Polygon2` primitive which also returns orientation. If user knows orientation beforehand then there is no need to call IsConvex method, containment method can be called directly.

Note that for convex polygons which have exactly 4 vertices (quadrilaterals) there are optimized versions of containment tests available.

Type
`struct Polygon2`
Methods

`bool ContainsConvexQuadCCW(ref Vector2 point)`
`bool ContainsConvexQuadCCW(Vector2 point)`
Tests whether a point is contained by the convex CCW 4-sided polygon
(the caller must ensure that polygon is indeed CCW ordered)

`bool ContainsConvexQuadCW(ref Vector2 point)`
`bool ContainsConvexQuadCW(Vector2 point)`
Tests whether a point is contained by the convex CW 4-sided polygon
(the caller must ensure that polygon is indeed CW ordered)

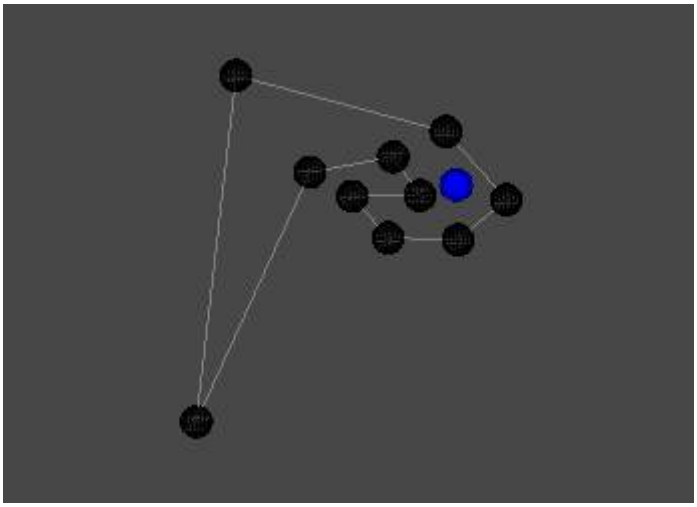
`bool ContainsConvexCCW(ref Vector2 point)`
`bool ContainsConvexCCW(Vector2 point)`
Tests whether a point is contained by the convex CCW polygon
(the caller must ensure that polygon is indeed CCW ordered)

`bool ContainsConvexCW(ref Vector2 point)`
`bool ContainsConvexCW(Vector2 point)`
Tests whether a point is contained by the convex CW polygon
(the caller must ensure that polygon is indeed CW ordered)

Example

```
Orientations orientation;
bool convex = convexPolygon.IsConvex(out orientation);
if (convex)
{
	bool cont;
	if (orientation == Orientations.CCW)
		cont = convexPolygon.ContainsConvexCCW(point);
	else // CW
		cont = convexPolygon.ContainsConvexCW(point);
}
```

6.1.5 Point2 in Polygon2



Test Prefab:
Test_ContPoint2Polygon2

For general polygon special containment method exists. The only restriction to a polygon is that it must be simple (i.e. without self-intersections, for example, such as on the image).

The library currently does not provide methods for checking whether a polygon is simple, therefore it's user's responsibility to ensure this restriction.

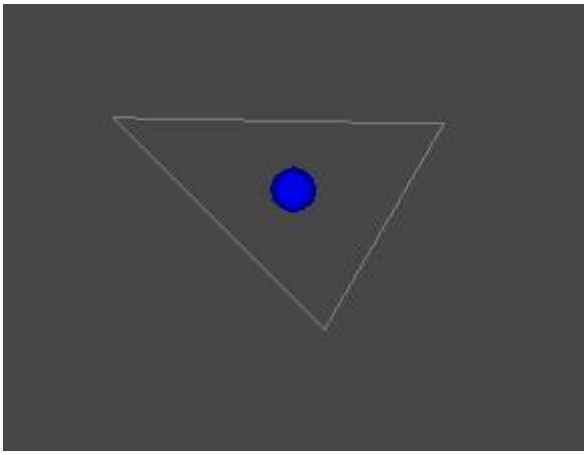
Due to nature of the algorithm points that are situated on the border of a polygon may be reported as inside or outside depending on the point position. It should be noted that intersection of 3D linear primitives (line, ray, segment) with Polygon3 uses this containment test and therefore has the same issue with classifying border points when a primitive intersects polygon border.

Type
`struct Polygon2`
Methods

`bool ContainsSimple(ref Vector2 point)`
`bool ContainsSimple(Vector2 point)`
Tests whether a point is contained by the simple polygon (i.e. without self intersection). Non-convex polygons are allowed, orientation is irrelevant.
Note that points which are on border may be classified differently depending on the point position.

Example
`bool cont = polygon.ContainsSimple(point);`

6.1.6 Point2 in Triangle2



Test Prefab:
Test_ContPoint2Triangle2

Triangle2 has three containment methods. For differences and characteristics see comments in the table below.

Type
`struct Triangle2`
Methods

`bool Contains(ref Vector2 point)`
`bool Contains(Vector2 point)`
Tests whether a point is contained by the triangle (CW or CCW ordered).
Note however that if the triangle is CCW then points which are on triangle border considered inside, but if the triangle is CW then points which are on triangle border considered outside.
For consistent (and faster) test use appropriate overloads for CW and CCW triangles.

`bool ContainsCCW(ref Vector2 point)`
`bool ContainsCCW(Vector2 point)`
Tests whether a point is contained by the CCW triangle

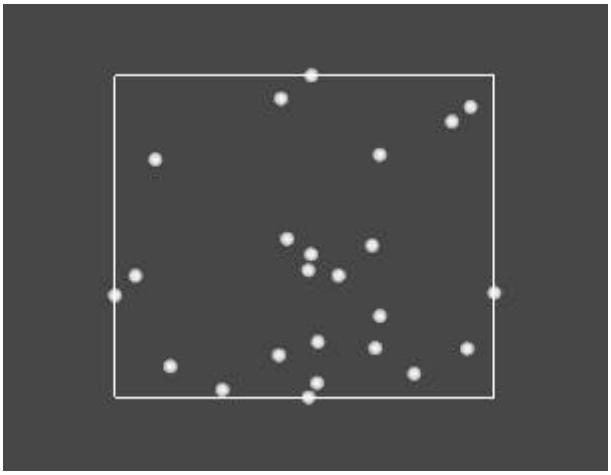
`bool ContainsCW(ref Vector2 point)`
`bool ContainsCW(Vector2 point)`
Tests whether a point is contained by the CW triangle

Example

```
Orientations orientation = triangle.CalcOrientation();

// Use this if you know triangle orientation
if (orientation == Orientations.CCW)
{
	bool cont = triangle.ContainsCCW(point);
}
else if (orientation == Orientations.CW)
{
	bool cont = triangle.ContainsCW(point);
}

// Use this if orientation is unknown
bool cont = triangle.Contains(point);
```

6.2 Bounding Areas

6.2.1 Constructing AAB2



Test Prefab:
Test_ContCreateAAB2

Test prefab contains “Toggle To Generate” item in the inspector. Press it to generate point set and build aab from it.

Type
`struct AAB2`
Methods

`static AAB2 CreateFromPoint(ref Vector2 point)`
`static AAB2 CreateFromPoint(Vector2 point)`
Creates AAB from single point. Min and Max are set to point. Use Include() method to grow the resulting AAB.

`static AAB2 CreateFromTwoPoints(ref Vector2 point0, ref Vector2 point1)`
`static AAB2 CreateFromTwoPoints(Vector2 point0, Vector2 point1)`
Computes AAB from two points extracting min and max values. In case min and max points are known, use constructor instead.

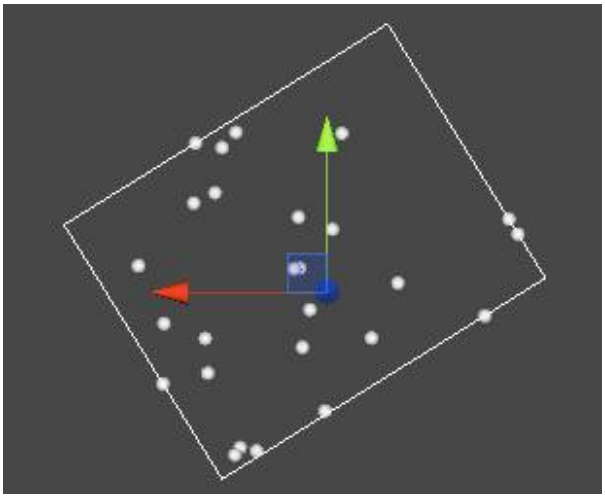
`static AAB2 CreateFromPoints(IEnumerable<Vector2> points)`
`static AAB2 CreateFromPoints(IList<Vector2> points)`
`static AAB2 CreateFromPoints(Vector2[] points)`
Computes AAB from the a of points. Method includes points from a set one by one to create the AAB. If a set is empty, returns new AAB2().

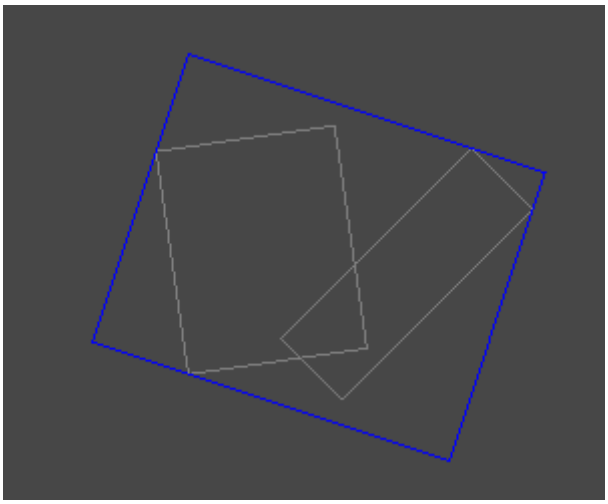
`void Include(ref Vector2 point)`
`void Include(Vector2 point)`
Enlarges the aab to include the point. If the point is inside the AAB does nothing.

`void Include(ref AAB2 box)`
`void Include(AAB2 box)`
Enlarges the aab so it includes another aab.

Example
`Vector2[] points = GenerateRandomSet2D(GenerateRadius, GenerateCountMin, GenerateCountMax);` `AAB2 aab = AAB2.CreateFromPoints(points);`

6.2.2 Constructing Box2





Test Prefab:
Test_ContCreateBox2
Test_ContBox2IncludeBox2

First test prefab contains “Toggle To Generate” item in the inspector. Press it to generate point set and build box from it.

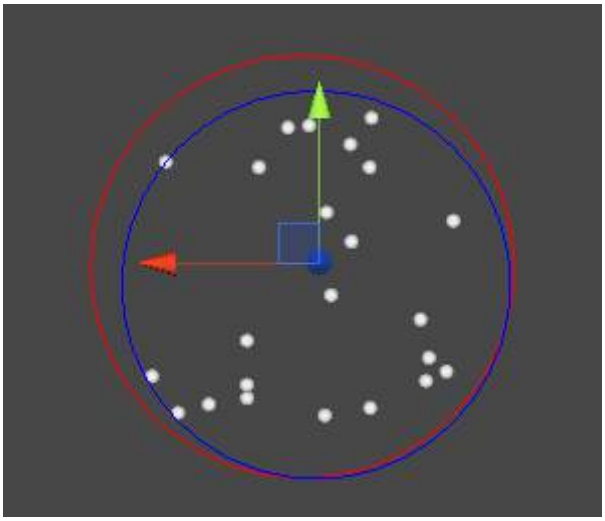
Type
`struct Box2`
Methods
`static Box2 CreateFromPoints(IList<Vector2> points)` Computes oriented bounding box from a set of points. If a set is empty returns new Box2().

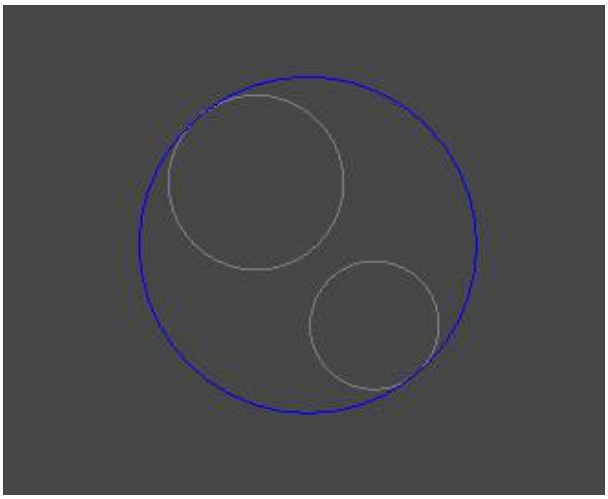
`void Include(ref Box2 box)`
`void Include(Box2 box)`
Enlarges the box so it includes another box.

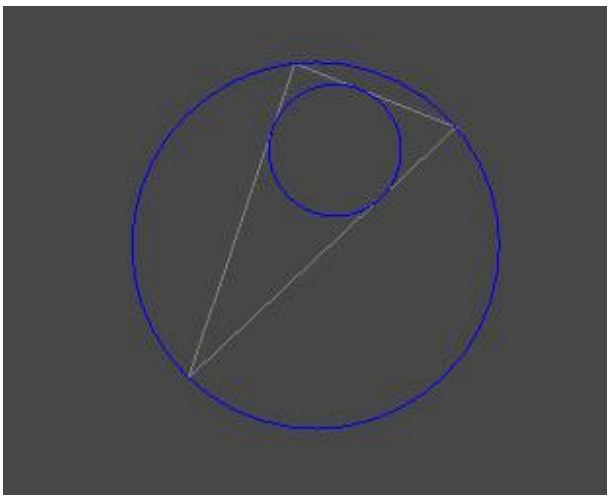
Example

```
// First method
Vector2[] points = GenerateRandomSet2D(GenerateRadius, GenerateCountMin, GenerateCountMax);
Box2 box = Box2.CreateFromPoints(points);

// Second method
box1.Include(box2);
```

6.2.3 Constructing Circle2







Test Prefab:
Test_ContCreateCircle2
Test_ContCircle2IncludeCircle2
Test_ContCreateScribeCircle2

First test prefab contains “Toggle To Generate” item in the inspector. Press it to generate point set and build circle from it.

In addition to similar methods available in aab/box primitives circle offers methods to create circle circumscribed or inscribed into triangle.

Note that there are two types of methods to create a circle from a point set. First method is faster and constructs aab from a set and then creates a circle containing aab (red circle on the upper image). Second method is slower and using set average point as the center (blue circle on the upper image). Note that blue circle fits the set better than red circle. However if point set is rather small (~5 points) the situation may change. User should learn test prefabs and try different generation configurations.

Type
`struct Circle2`
Methods

`static Circle2 CreateFromPointsAAB(IEnumerable<Vector2> points)`
`static Circle2 CreateFromPointsAAB(IList<Vector2> points)`
Computes bounding circle from a set of points. First compute the axis-aligned bounding box of the points, then compute the circle containing the box. If a set is empty returns new Circle2().

`static Circle2 CreateFromPointsAverage(IEnumerable<Vector2> points)`
`static Circle2 CreateFromPointsAverage(IList<Vector2> points)`
Computes bounding circle from a set of points. Compute the smallest circle whose center is the average of a point set. If a set is empty returns new Circle2().

`static bool CreateCircumscribed(Vector2 v0, Vector2 v1, Vector2 v2, out Circle2 circle)` Creates circle which is circumscribed around triangle. Returns 'true' if circle has been constructed, 'false' otherwise (input points are linearly dependent).
`static bool CreateInscribed(Vector2 v0, Vector2 v1, Vector2 v2, out Circle2 circle)` Creates circle which is insribed into triangle. Returns 'true' if circle has been constructed, 'false' otherwise (input points are linearly dependent).

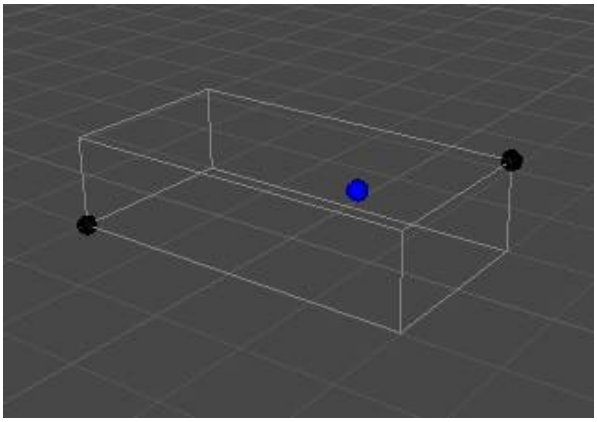
`void Include(ref Circle2 circle)`
`void Include(Circle2 circle)`
Enlarges the circle so it includes another circle.

Example

```
// First method
Vector2[] points = GenerateRandomSet2D(GenerateRadius, GenerateCountMin, GenerateCountMax);
Circle2 circle0 = Circle2.CreateFromPointsAAB(points);
Circle2 circle1 = Circle2.CreateFromPointsAverage(points);

// Second method
circle2.Include(circle3);

// Third method
Circle2 circumscribed;
bool b0 = Circle2.CreateCircumscribed(v0, v1, v2, out circumscribed);
Circle2 inscribed;
bool b1 = Circle2.CreateInscribed(v0, v1, v2, out inscribed);
```

6.3 3D Point Containment

6.3.1 Point3 in AAB3

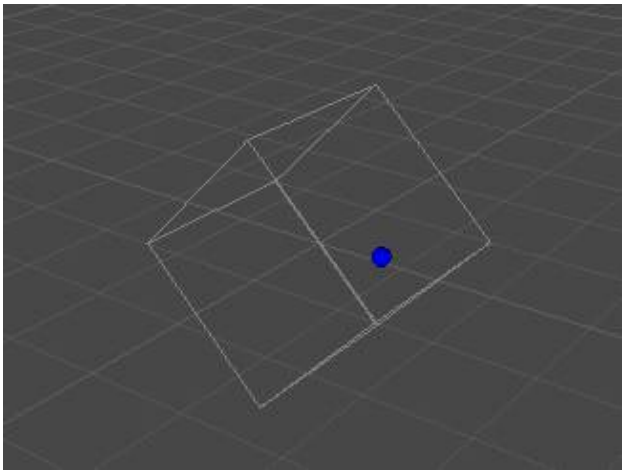


Test Prefab:
Test_ContPoint3AAB3

Type
`struct AAB3`
Methods
`bool Contains(ref Vector3 point)`
`bool Contains(Vector3 point)`

Example
`bool cont = box.Contains(point);`

6.3.2 Point3 in Box3

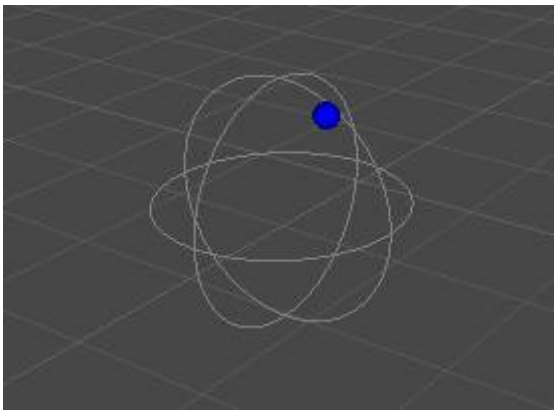


Test Prefab:
Test_ContPoint3Box3

Type
`struct Box3`
Methods
`bool Contains(ref Vector3 point)`
`bool Contains(Vector3 point)`

Example
`bool cont = box.Contains(point);`

6.3.3 Point3 in Sphere3



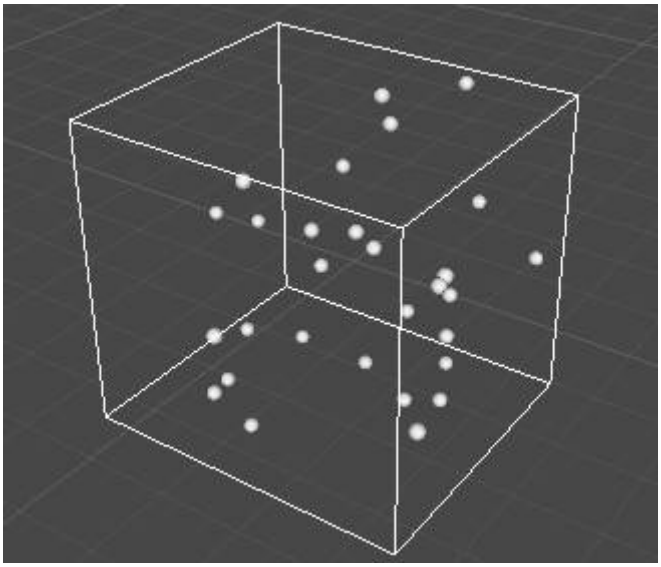
Test Prefab:
Test_ContPoint3Sphere3

Type
`struct Sphere3`
Methods
`bool Contains(ref Vector3 point)`
`bool Contains(Vector3 point)`

Example
`bool cont = sphere.Contains(point);`

6.4 Bounding Volumes

6.4.1 Constructing AAB3



Test Prefab:
Test_ContCreateAAB3

Test prefab contains “Toggle To Generate” item in the inspector. Press it to generate point set and build aab from it.

Type
`struct AAB3`
Methods

`static AAB3 CreateFromPoint(ref Vector3 point)`
`static AAB3 CreateFromPoint(Vector3 point)`
Creates AAB from single point. Min and Max are set to point. Use Include() method to grow the resulting AAB.

`static AAB3 CreateFromTwoPoints(ref Vector3 point0, ref Vector3 point1)`
`static AAB3 CreateFromTwoPoints(Vector3 point0, Vector3 point1)`
Computes AAB from two points extracting min and max values. In case min and max points are known, use constructor instead.

`static AAB3 CreateFromPoints(IEnumerable<Vector3> points)`
`static AAB3 CreateFromPoints(IList<Vector3> points)`
`static AAB3 CreateFromPoints(Vector3[] points)`
Computes AAB from a set of points. Method includes points from a set one by one to create the AAB. If a set is empty returns new AAB3().

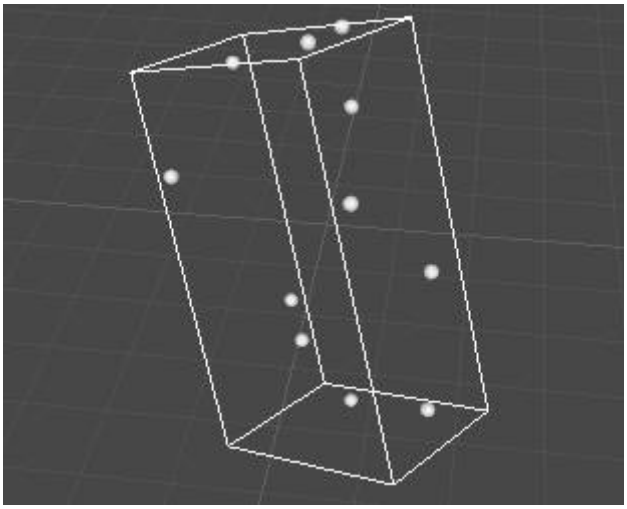
`void Include(ref Vector3 point)`
`void Include(Vector3 point)`
Enlarging the aab to include the point. If the point is inside the AAB does nothing.

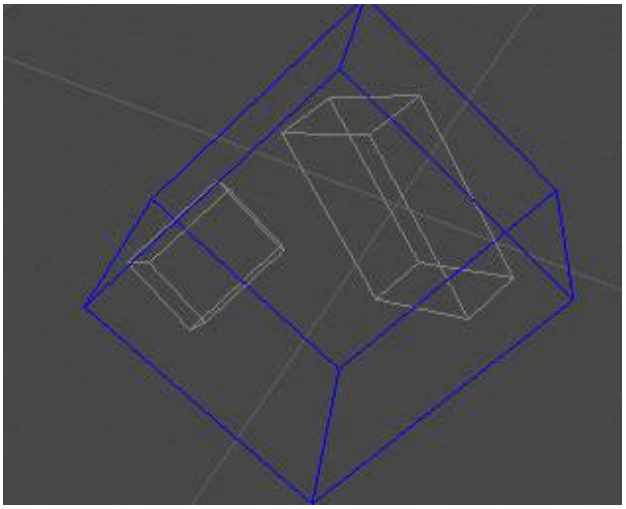
`void Include(ref AAB3 box)`
`void Include(AAB3 box)`
Enlarges the aab so it includes another aab.

Example

```
Vector3[] points = GenerateRandomSet3D(GenerateRadius, GenerateCountMin, GenerateCountMax);
AAB3 aab = AAB3.CreateFromPoints(points);
```

6.4.2 Constructing Box3





Test Prefab:
Test_ContCreateBox3
Test_ContBox3IncludeBox3

First test prefab contains “Toggle To Generate” item in the inspector. Press it to generate point set and build box from it.

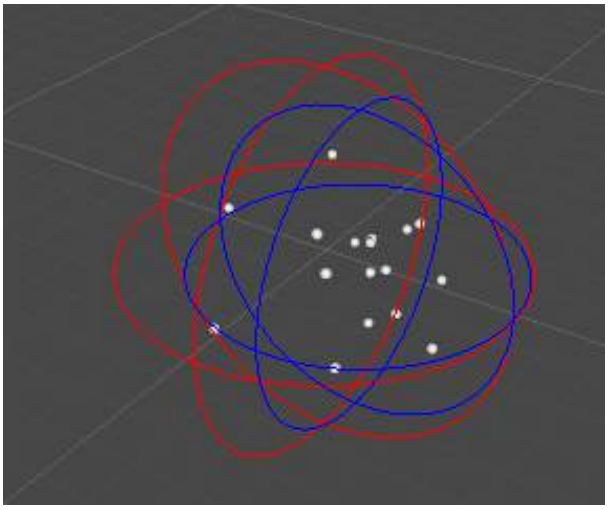
Type
`struct Box3`
Methods
`static Box3 CreateFromPoints(IList<Vector3> points)` Computes oriented bounding box from a set of points. If a set is empty returns new Box3().

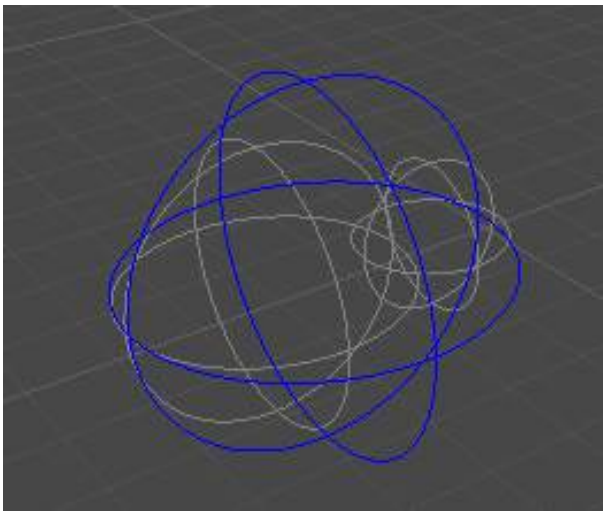
`void Include(ref Box3 box)`
`void Include(Box3 box)`
Enlarges the box so it includes another box.

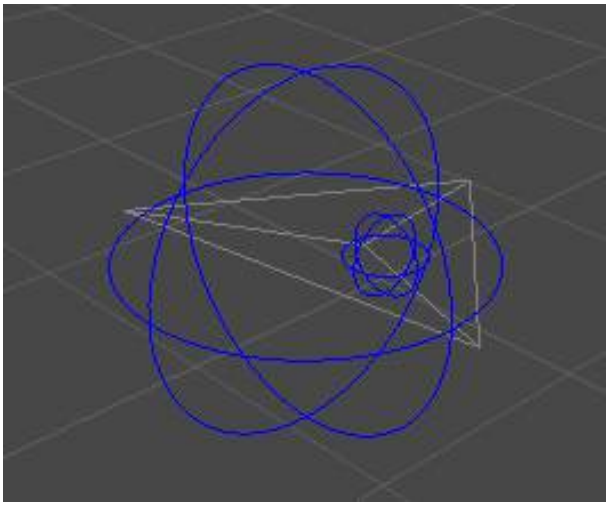
Example

```
// First method
Vector3[] points = GenerateRandomSet3D(GenerateRadius, GenerateCountMin, GenerateCountMax);
Box3 box = Box3.CreateFromPoints(points);

// Second method
box1.Include(box2);
```

6.4.3 Constructing Sphere3







Test Prefab:

Test_ContCreateSphere3
Test_ContSphere3IncludeSphere3
Test_ContCreateScribeSphere3

First test prefab contains “Toggle To Generate” item in the inspector. Press it to generate point set and build sphere from it.

In addition to similar methods available in aab/box primitives sphere offers methods to create sphere circumscribed or inscribed into tetrahedron.

Note that there are two types of methods to create a sphere from a point set. First method is faster and constructs aab from a set and then creates a sphere containing aab (red sphere on the upper image). Second method is slower and using set average point as the center (blue sphere on the upper image). Note that blue sphere fits the set better than red sphere. However if point set is rather small (~5 points) the situation may change. User should learn test prefabs and try different generation configurations.

Type
`struct Sphere3`
Methods

`static Sphere3 CreateFromPointsAAB(IEnumerable<Vector3> points)`
`static Sphere3 CreateFromPointsAAB(IList<Vector3> points)`
Computes bounding sphere from a set of points. First compute the axis-aligned bounding box of the points, then compute the sphere containing the box. If a set is empty returns new Sphere3().

`static Sphere3 CreateFromPointsAverage(IEnumerable<Vector3> points)`
`static Sphere3 CreateFromPointsAverage(IList<Vector3> points)`
Computes bounding sphere from a set of points. Compute the smallest sphere whose center is the average of a point set. If a set is empty returns new Sphere3().

`static bool CreateCircumscribed(Vector3 v0, Vector3 v1, Vector3 v2, Vector3 v3, out Sphere3 sphere)` Creates sphere which is circumscribed around tetrahedron. Returns 'true' if sphere has been constructed, 'false' otherwise (input points are linearly dependent).
`static bool CreateInscribed(Vector3 v0, Vector3 v1, Vector3 v2, Vector3 v3, out Sphere3 sphere)` Creates sphere which is insribed into tetrahedron. Returns 'true' if sphere has been constructed, 'false' otherwise (input points are linearly dependent).

`void Include(ref Sphere3 sphere)`
`void Include(Sphere3 sphere)`
Enlarges the sphere so it includes another sphere.

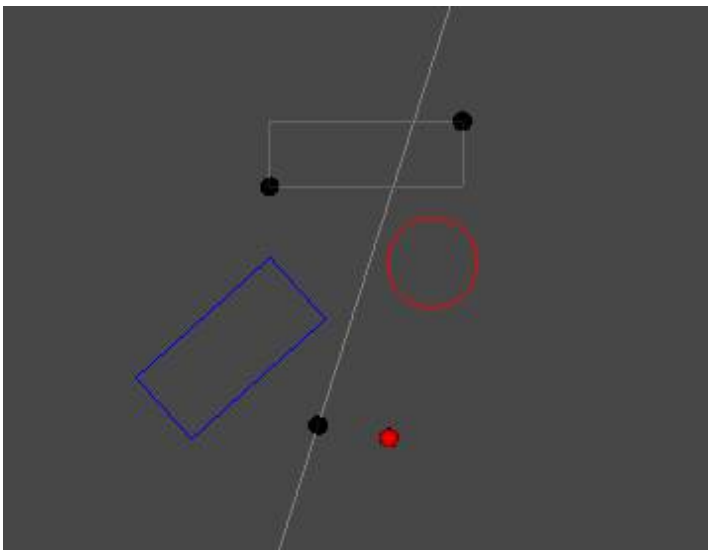
Example

```
// First method
Vector3[] points = GenerateRandomSet3D(GenerateRadius, GenerateCountMin, GenerateCountMax);
Sphere3 sphere0 = Sphere3.CreateFromPointsAAB(points);
Sphere3 sphere1 = Sphere3.CreateFromPointsAverage(points);

// Second method
sphere2.Include(sphere3);

// Third method
Sphere3 circumscribed;
bool b0 = Sphere3.CreateCircumscribed(v0, v1, v2, v3, out circumscribed);
Sphere3 inscribed;
bool b1 = Sphere3.CreateInscribed(v0, v1, v2, v3, out inscribed);
```

7 Side Testing

This chapter covers methods for querying point position against some primitive. In practice such tests may arise in culling algorithms where one should know whether an object should be, for example, rendered (this may ends in asking whether an object is on the positive side of a plane?). Thus side testing is mainly useful against lines in 2D and planes in 3D (there are also methods on triangle in 2D). Tested primitives include point and bounding objects. All these methods are available on primitive objects and start with `QuerySide***`. They return flag which tells the caller where is the tested primitive lies.

7.1 Line2 Tests



Test Prefab:
Test_Line2Sides

Test prefab has examples for quering side for point, AAB, box and circle in 2D.

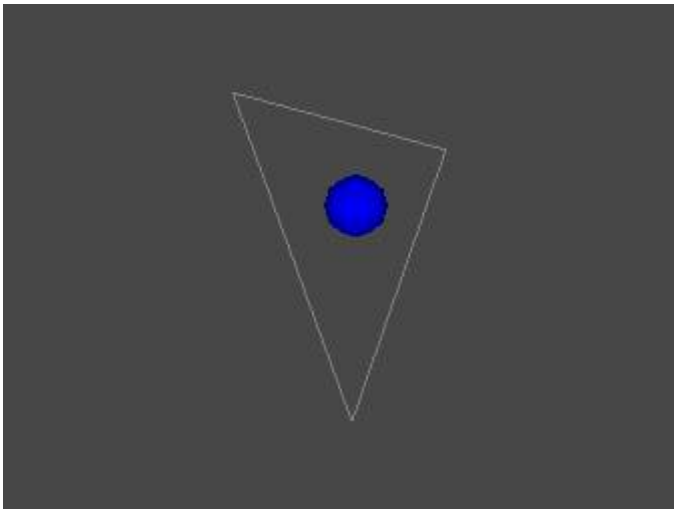
Type
`struct Line2`
Methods
`int QuerySide(Vector2 point, float epsilon = MathfEx.ZeroTolerance)` Determines on which side of the line a point is. Returns +1 if a point is to the right of the line, 0 if it's on the line, -1 if it's on the left.
`bool QuerySideNegative(Vector2 point, float epsilon = MathfEx.ZeroTolerance)` Returns true if a point is on the negative side of the line, false otherwise.
`bool QuerySidePositive(Vector2 point, float epsilon = MathfEx.ZeroTolerance)` Returns true if a point is on the positive side of the line, false otherwise.
`int QuerySide(ref Box2 box, float epsilon = MathfEx.ZeroTolerance)` Determines on which side of the line a box is. Returns +1 if a box is to the right of the line, 0 if it's intersecting the line, -1 if it's on the left.
`bool QuerySideNegative(ref Box2 box, float epsilon = MathfEx.ZeroTolerance)` Returns true if a box is on the negative side of the line, false otherwise.
`bool QuerySidePositive(ref Box2 box, float epsilon = MathfEx.ZeroTolerance)` Returns true if a box is on the positive side of the line, false otherwise.

`int QuerySide(ref AAB2 box, float epsilon = MathfEx.ZeroTolerance)` Determines on which side of the line a box is. Returns +1 if a box is to the right of the line, 0 if it's intersecting the line, -1 if it's on the left.
`bool QuerySideNegative(ref AAB2 box, float epsilon = MathfEx.ZeroTolerance)` Returns true if a box is on the negative side of the line, false otherwise.
`bool QuerySidePositive(ref AAB2 box, float epsilon = MathfEx.ZeroTolerance)` Returns true if a box is on the positive side of the line, false otherwise.
`int QuerySide(ref Circle2 circle, float epsilon = MathfEx.ZeroTolerance)` Determines on which side of the line a circle is. Returns +1 if a circle is to the right of the line, 0 if it's intersecting the line, -1 if it's on the left.
`bool QuerySideNegative(ref Circle2 circle, float epsilon = MathfEx.ZeroTolerance)` Returns true if a circle is on the negative side of the line, false otherwise.
`bool QuerySidePositive(ref Circle2 circle, float epsilon = MathfEx.ZeroTolerance)` Returns true if a circle is on the positive side of the line, false otherwise.

Example

```
// Get side information.
// -1 - on the negative side of the line
//  0 - on the line or intersecting the line
// +1 - on the positive side of the line
int pointSide  = line.QuerySide(point);
int aabSide    = line.QuerySide(ref aab);
int boxSide    = line.QuerySide(ref box);
int circleSide = line.QuerySide(ref circle);

// true when an object is on the positive side of the line
bool pointPos  = line.QuerySidePositive(point);
bool aabPos    = line.QuerySidePositive(ref aab);
bool boxPos    = line.QuerySidePositive(ref box);
bool circlePos = line.QuerySidePositive(ref circle);

// true when an object is on the negative side of the line
bool pointNeg  = line.QuerySideNegative(point);
bool aabNeg    = line.QuerySideNegative(ref aab);
bool boxNeg    = line.QuerySideNegative(ref box);
bool circleNeg = line.QuerySideNegative(ref circle);

// Note that positive/negative tests are little bit more optimized than just query,
// as they don't have separate check for 0 case.
```

7.2 Triangle2 Tests



Test Prefab:
Test_Triangle2Sides

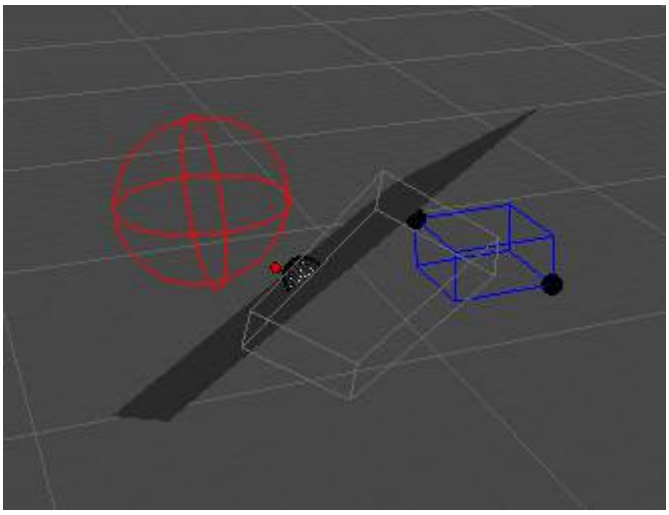
Type
`struct Triangle2`
Methods
`int QuerySideCCW(Vector2 point, float epsilon = MathfEx.ZeroTolerance)` Determines on which side of the triangle a point is. Returns +1 if a point is outside of the triangle, 0 if it's on the triangle border, -1 if it's inside the triangle. Method must be called for CCW ordered triangles.
`int QuerySideCW(Vector2 point, float epsilon = MathfEx.ZeroTolerance)` Determines on which side of the triangle a point is. Returns +1 if a point is outside of the triangle, 0 if it's on the triangle border, -1 if it's inside the triangle. Method must be called for CW ordered triangles.

Reader may ask why DistanceTo() method can not be used instead? Because it serves exactly what it should – calculates the distance. Query methods are much faster to execute and consistent with border determination for CW and CCW triangles.

Example

```
Orientations orientation = triangle.CalcOrientation();
if (orientation == Orientations.CCW)
       int ccwSide = triangle.QuerySideCCW(point);
else
       int cwSide  = triangle.QuerySideCW(point);
```

7.3 Plane3 Tests



Test Prefab:
Test_Plane3Sides

Test prefab has examples for quering side for point, AAB, box and sphere in 3D.

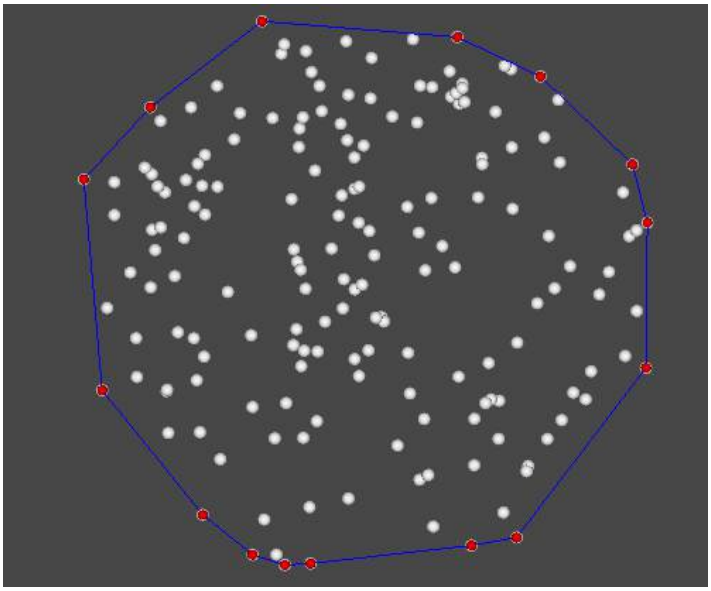
Type
`struct Plane3`
Methods
`int QuerySide(Vector3 point, float epsilon = MathfEx.ZeroTolerance)` Determines on which side of the plane a point is. Returns +1 if a point is on the positive side of the plane, 0 if it's on the plane, -1 if it's on the negative side. The positive side of the plane is the half-space to which the plane normal points.
`bool QuerySideNegative(Vector3 point, float epsilon = MathfEx.ZeroTolerance)` Returns true if a point is on the negative side of the plane, false otherwise.
`bool QuerySidePositive(Vector3 point, float epsilon = MathfEx.ZeroTolerance)` Returns true if a point is on the positive side of the plane, false otherwise.
`int QuerySide(ref Box3 box, float epsilon = MathfEx.ZeroTolerance)` Determines on which side of the plane a box is. Returns +1 if a box is on the positive side of the plane, 0 if it's intersecting the plane, -1 if it's on the negative side. The positive side of the plane is the half-space to which the plane normal points.
`bool QuerySideNegative(ref Box3 box, float epsilon = MathfEx.ZeroTolerance)` Returns true if a box is on the negative side of the plane, false otherwise.
`bool QuerySidePositive(ref Box3 box, float epsilon = MathfEx.ZeroTolerance)` Returns true if a box is on the positive side of the plane, false otherwise.
`int QuerySide(ref AAB3 box, float epsilon = MathfEx.ZeroTolerance)` Determines on which side of the plane a box is. Returns +1 if a box is on the positive side of the plane, 0 if it's intersecting the plane, -1 if it's on the negative side. The positive side of the plane is the half-space to which the plane normal points.
`bool QuerySideNegative(ref AAB3 box, float epsilon = MathfEx.ZeroTolerance)` Returns true if a box is on the negative side of the plane, false otherwise.
`bool QuerySidePositive(ref AAB3 box, float epsilon = MathfEx.ZeroTolerance)` Returns true if a box is on the positive side of the plane, false otherwise.
`int QuerySide(ref Sphere3 sphere, float epsilon = MathfEx.ZeroTolerance)` Determines on which side of the plane a sphere is. Returns +1 if a sphere is on the positive side of the plane, 0 if it's intersecting the plane, -1 if it's on the negative side. The positive side of the plane is the half-space to which the plane normal points.
`bool QuerySideNegative(ref Sphere3 sphere, float epsilon = MathfEx.ZeroTolerance)` Returns true if a sphere is on the negative side of the plane, false otherwise.
`bool QuerySidePositive(ref Sphere3 sphere, float epsilon = MathfEx.ZeroTolerance)` Returns true if a sphere is on the positive side of the plane, false otherwise.

Example

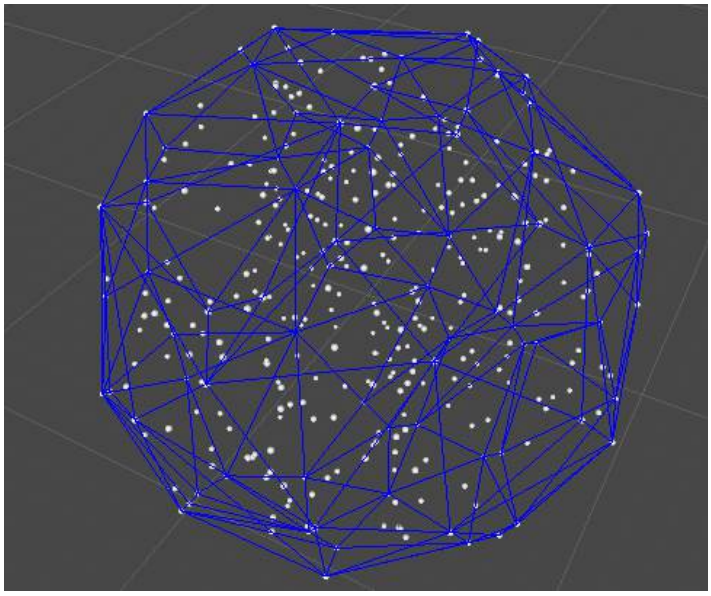
```
// Get side information.
// -1 - on the negative side of the plane
//  0 - on the plane or intersecting the plane
// +1 - on the positive side of the plane
int pointSide  = plane.QuerySide(point);
int aabSide    = plane.QuerySide(ref aab);
int boxSide    = plane.QuerySide(ref box);
int sphereSide = plane.QuerySide(ref sphere);

// true when an object is on the positive side of the plane
bool pointPos  = plane.QuerySidePositive(point);
bool aabPos    = plane.QuerySidePositive(ref aab);
bool boxPos    = plane.QuerySidePositive(ref box);
bool spherePos = plane.QuerySidePositive(ref sphere);

// true when an object is on the negative side of the plane
bool pointNeg  = plane.QuerySideNegative(point);
bool aabNeg    = plane.QuerySideNegative(ref aab);
bool boxNeg    = plane.QuerySideNegative(ref box);
bool sphereNeg = plane.QuerySideNegative(ref sphere);

// Note that positive/negative tests are little bit more optimized than just query,
// as they don't have separate check for 0 case.
```

8 Geometric Algorithms

8.1 Convex hull



Test Prefab:
Test_ConvexHull2



Test Prefab:
Test_ConvexHull3

Type
`class ConvexHull`
Methods
`static bool Create2D(IList<Vector2> points, out int[] indices, out int dimension, float epsilon = Mathfex.ZeroTolerance)` Generates 2D convex hull of the input point set. Resulting convex hull is defined by the indices parameter. Its behavior depends on the dimension parameter. If dimension is 2, then convex hull is 2D polygon and indices should be accessed as Edge=(points[indices[i]], points[indices[(i+1)%indices.Length]), for i=[0,indices.Length-1]. If dimension is 1, then input point set lie on the line and covex hull is a segment, use (points[indices[0]], points[indices[1]) to access the segment. If dimension is 0, then all points in the input set are practically the same. points - Input point set whose convex hull should be calculated. indices - Contains indices into point set (null if construction has failed). dimension - Resulting dimension of the input set: 2, 1 or 0. epsilon - Small positive number used to determine dimension of the input set. returns - True if convex hull is created, false otherwise (in case if input point set is null, contains no points or if some error has occured during the construction)

```
public static bool Create3D(IList<Vector3> points, out int[] indices, out int dimension, float epsilon = Mathfex.ZeroTolerance)
```

Generates 3D convex hull of the input point set. Resulting convex hull is defined fo the indices parameter. Its behavior depends on the dimension parameter.

If dimension is 3, then convex hull is 3D polyhedron and indices define triangles,
Triangle=(points[indices[i]], points[indices[i+1]], points[indices[i+2]]), for i=[0,indices.Length-1], i+=3.

If dimension is 2, then convex hull is 2D polygon and indices should be accessed as
Edge=(points[indices[i]], points[indices[(i+1)%indices.Length]), for i=[0,indices.Length-1].

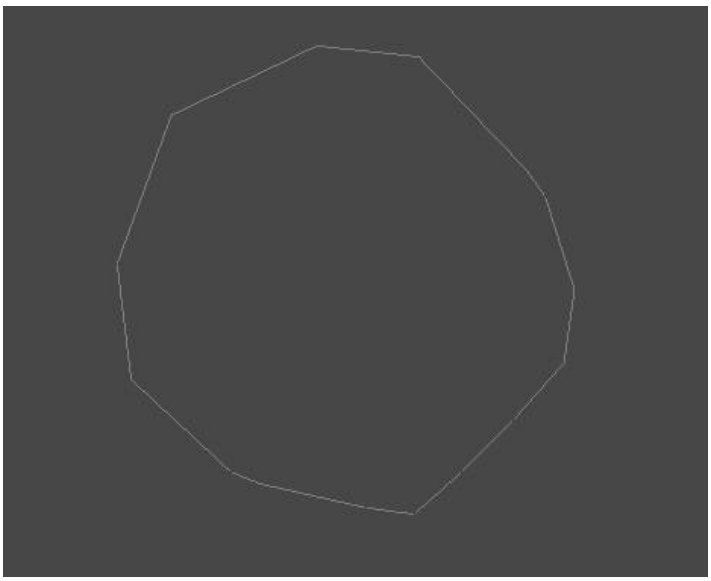
If dimension is 1, then input point set lie on the line and covex hull is a segment, use (points[indices[0]], points[indices[1]) to access the segment.

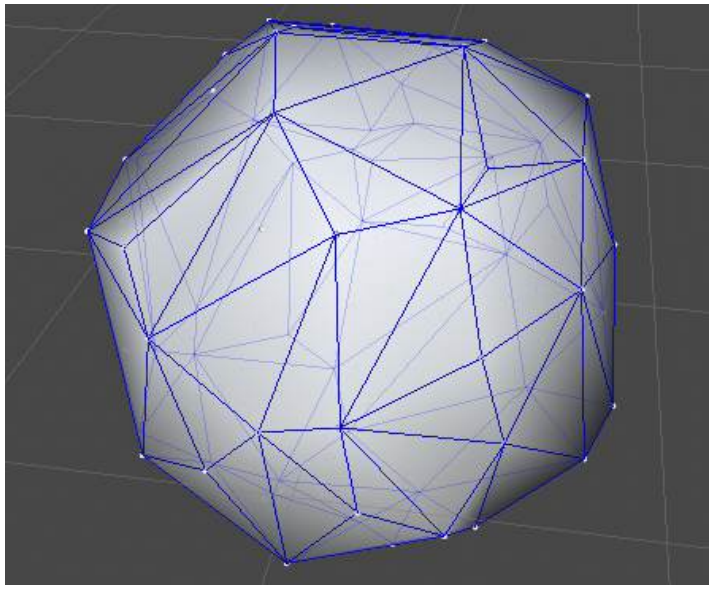
If dimension is 0, then all points in the input set are practically the same.

points - Input point set whose convex hull should be calculated.
indices - Contains indices into point set (null if construction has failed).
dimension - Resulting dimension of the input set: 3, 2, 1 or 0.
epsilon - Small positive number used to determine dimension of the input set.

returns - True if convex hull is created, false otherwise (in case if input point set is null, contains no points or if some error has occured during construction)

Both test prefabs in addition to demonstrating how to create convex hulls from the point sets also show how to construct `Mesh` objects from the resulting convex hulls. Users can enable CreateMeshObject flag in the inspector of the both prefabs before generating convex hull (see `CreateMesh()` methods). When the flag is enabled test prefabs will create gameobject with mesh made out of convex hull (`MeshTopology.LineStrip` for 2D hulls, `MeshTopology.Triangles` for 3D hulls). Two images below show the examples of meshes created by test prefabs.



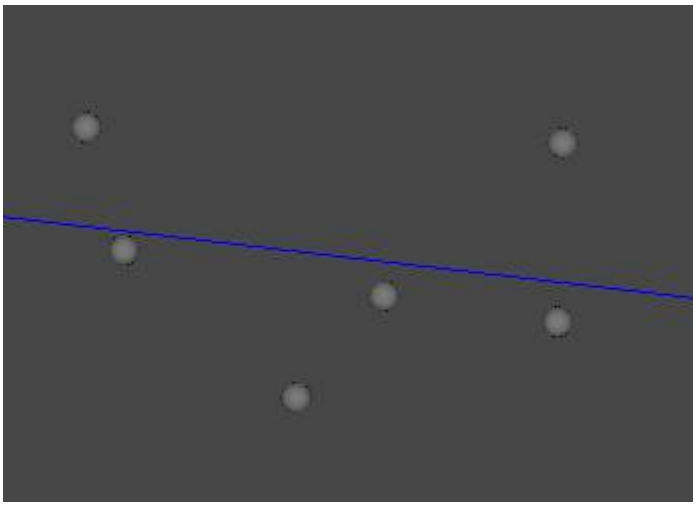


9 Approximation

Approximation serves the purpose to extract some meaningful object from a point set. The library provides several methods constructing line and plane from points, also it offers fitting points using Gaussian distribution. All methods are situated inside static class `Approximation` and accept point set as an input parameter.

9.1 2D Approximation

9.1.1 LineFit2



Test Prefab:
Test_ApprLineFit2

Type
`class Approximation`
Methods
`static Line2 LeastSquaresLineFit2(IList<Vector2> points)` Producing a line using least-squares fitting. A set must contain at least one point!

Example
`Vector2[] points = CreatePoints2(Points);` `if (points.Length > 0) Line2 line = Approximation.LeastSquaresLineFit2(points);`

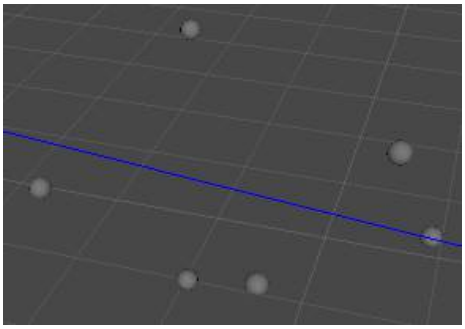
9.1.2 GaussPointsFit2

Type
`class Approximation`
Methods
`static Box2 GaussPointsFit2(IList<Vector2> points)` Fits points with a Gaussian distribution. Produces box as the result. Box center is average of a point set. Box axes are eigenvectors of the covariance matrix, box extents are eigenvalues. A set must contain at least one point!

This method is used inside of the construction of Box2 from a point set, thus its test prefab could be used to see the result.

9.2 3D Approximation

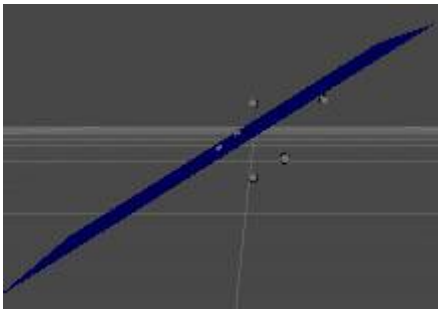
9.2.1 LineFit3



Test Prefab:
Test_ApprLineFit3

Type
`class Approximation`
Methods
`static Line3 LeastsSquaresLineFit3(IList<Vector3> points)` Producing a line using least-squares fitting. A set must contain at least one point!

9.2.2 PlaneFit3



Test Prefab:
Test_ApprPlaneFit3

Type
`class Approximation`
Methods
`static Plane3 LeastSquaresPlaneFit3(IList<Vector3> points)` Producing a plane using least-squares fitting. A set must contain at least one point!

9.2.3 GaussPointsFit3

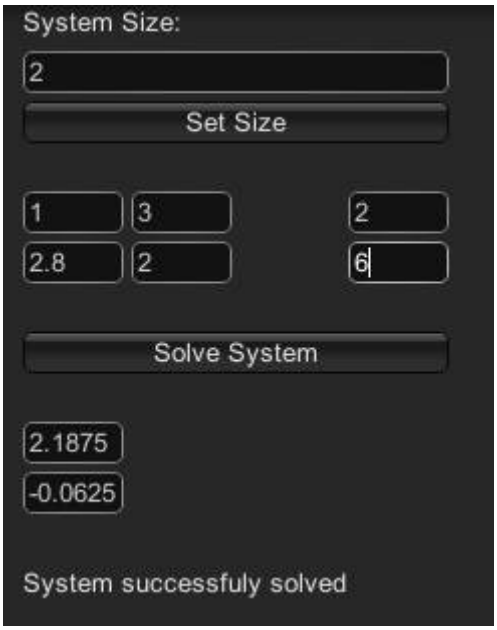
Type
`class Approximation`
Methods
`static Box3 GaussPointsFit3(IList<Vector3> points)` Fits points with a Gaussian distribution. Produces box as the result. Box center is average of a point set. Box axes are eigenvectors of the covariance matrix, box extents are eigenvalues. A set must contain at least one point!

This method is used inside of the construction of Box3 from a point set, thus its test prefab could be used to see the result.

10 Numerical Methods

Although it's not very common, sometimes developers need to solve systems, find roots or intergrate functions. Therefore library provides experimental support for several common numerical methods which may be used by the developers. “Experimental” means that this functionality is subject to change or even removal in future versions. This is due to library mostly aimed to aid game developers which rarely need numerical methods. Future of this section will depend mostly on user feedback and complexity of support.

10.1 Solving Linear Systems



Test Prefab:
Test_NumericalLinearSystem

Library provides methods for solving systems of linear equiations of various sizes.

To see the image on the left, scene with test prefab must be launched. User then can enter system size and values and press the button to solve the system. The example of calling is presented in the code of the test prefab in the `SolveSystem()` method.

Type
`class LinearSystem`
Methods
`Solve2(float[,] A, float[] B, out float[] X, float zeroTolerance)`
`Solve2(float[,] A, float[] B, out Vector2 X, float zeroTolerance)`
`Solve3(float[,] A, float[] B, out float[] X, float zeroTolerance)`
`Solve3(float[,] A, float[] B, out Vector3 X, float zeroTolerance)`
`Solve(float[,] A, float[] B, out float[] X)`
`SolveTridiagonal(float[] A, float[] B, float[] C, float[] R, out float[] U)`
`Inverse(float[,] A, out float[,] invA)`

```
static bool Solve2(float[,] A, float[] B, out float[] X, float zeroTolerance = MathfEx.ZeroTolerance)
```
Solves linear system A*X=B with two equations and two unknowns.
A - float[2,2] array containing equations coefficients
B - float[2] array containing constants
X - Out float[2] array contaning the solution or null if system has no solution
zeroTolerance - Small positive number
Returns - True if solution is found, false otherwise

```
static bool Solve2(float[,] A, float[] B, out Vector2 X, float zeroTolerance = MathfEx.ZeroTolerance)
```
Solves linear system A*X=B with two equations and two unknowns.
A - float[2,2] array containing equations coefficients
B - float[2] array containing constants
X - Out vector contaning the solution or zero vector if system has no solution
zeroTolerance - Small positive number
Returns - True if solution is found, false otherwise

```
static bool Solve3(float[,] A, float[] B, out float[] X, float zeroTolerance = MathfEx.ZeroTolerance)
```
Solves linear system A*X=B with three equations and three unknowns.
A - float[3,3] array containing equations coefficients
B - float[3] array containing constants
X - Out float[3] array contaning the solution or null if system has no solution
zeroTolerance - Small positive number
Returns - True if solution is found, false otherwise

```
static bool Solve3(float[,] A, float[] B, out Vector3 X, float zeroTolerance = MathfEx.ZeroTolerance)
```
Solves linear system A*X=B with three equations and three unknowns.
A - float[3,3] array containing equations coefficients
B - float[3] array containing constants
X - Out vector contaning the solution or zero vector if system has no solution
zeroTolerance - Small positive number
Returns - True if solution is found, false otherwise

```
static bool Solve(float[,] A, float[] B, out float[] X)
```
Solves linear system A*X=B with N equations and N unknowns.
A - float[N,N] array containing equations coefficients
B - float[N] array containing constants
X - Out float[N] array contaning the solution or null if system has no solution
Returns - True if solution is found, false otherwise

```
static bool SolveTridiagonal(float[] A, float[] B, float[] C, float[] R, out float[] U)
```
Solves linear system A*X=B, where A is tridiagonal matrix.
A - Lower diagonal float[N-1]
B - Main diagonal float[N]
C - Upper diagonal float[N-1]
R - Right-hand side float[N]
U - Out float[N] containing the solution or null if system has no solution
Returns - True if solution is found, false otherwise

```
static bool Inverse(float[,] A, out float[,] invA)
```
Inverses square matrix A. Returns inversed matrix in invA parameter (invA is null if A has no inverse).

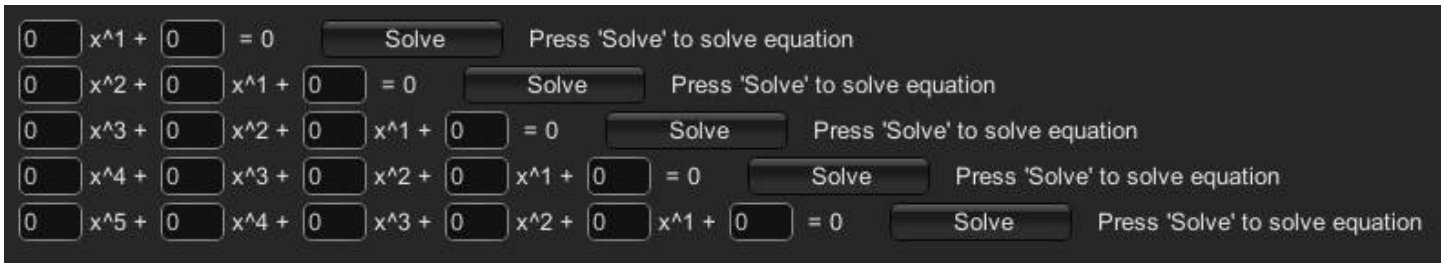
10.2 Eigen Decomposition

Having square symmetric matrix it's sometimes needed to know its eigenvalues and eigenvectors.

Type
`class EigenDecomposition`
Methods
`static EigenData Solve(float[,] symmetricSquareMatrix, bool increasingSort)` Solve the eigensystem. Set increasingSort to true when you want the eigenvalues to be sorted in increasing order (from smallest to largest); otherwise, the eigenvalues are sorted in decreasing order (from largest to smallest). symmetricSquareMatrix - Matrix must be square and symmetric. Matrix size must be >= 2. increasingSort - true for increasing sort, false for decreasing sort. Returns - Data containing eigenvalues and eigenvectors or null if matrix is non-square or size is <2.

Type
`class EigenData`
Properties
`int Size { get; }` Eigen system size
Methods
`float GetEigenvalue(int index)` Gets eigenvalue. Index must be 0<=index<Size.
`Vector2 GetEigenvector2(int index)` Gets eigenvector. Use this only if eigen system was of 2x2 size. Index must be 0<=index<Size.
`Vector3 GetEigenvector3(int index)` Gets eigenvector. Use this only if eigen system was of 3x3 size. Index must be 0<=index<Size.
`float[] GetEigenvector(int index)` Gets eigenvector. Size of the resulting array is equal to eigen system size. Index must be 0<=index<Size.
`void GetEigenvector(int index, float[] out_eigenvector)` Gets eigenvector. Size of the array must match eigen system size. Method will fill in components of eigenvector into the array. Index must be 0<=index<Size.

10.3 Finding Polynomial Roots



Test Prefab:
Test_NumericalPolyRoots
Test_NumericalBrentsMethod

Finding roots of polynomial equations can be done via RootFinder static class. It provides methods for solving low-order equations and any dimension equations.

Type
`class RootFinder`
Methods
`Linear(float c0, float c1, out float root, float epsilon = MathfEx.ZeroTolerance)`
`Quadratic(float c0, float c1, float c2, out QuadraticRoots roots, float epsilon)`
`Cubic(float c0, float c1, float c2, float c3, out CubicRoots roots, float epsilon)`
`Quartic(float c0, float c1, float c2, float c3, float c4, out QuarticRoots roots, float epsilon)`
`PolynomialBound(Polynomial poly, float epsilon)`
`Polynomial(Polynomial poly, float xMin, float xMax, out float[] roots, int digits, float epsilon)`
`Polynomial(Polynomial poly, out float[] roots, int digits, float epsilon)`

`BrentsMethod(Func<float, float> function, float x0, float x1, out BrentsRoot root,`
`int maxIterations, float negativeTolerance, float positiveTolerance, float stepTolerance,`
`float segmentTolerance)`

```
static bool Linear(float c0, float c1, out float root, float epsilon = MathfEx.ZeroTolerance)
```
Linear equations: c1*x+c0 = 0

```
static bool Quadratic(float c0, float c1, float c2, out QuadraticRoots roots, float epsilon =
MathfEx.ZeroTolerance)
```
Quadratic equations: c2*x^2+c1*x+c0 = 0

```
static bool Cubic(float c0, float c1, float c2, float c3, out CubicRoots roots, float epsilon =
MathfEx.ZeroTolerance)
```
Cubic equations: c3*x^3+c2*x^2+c1*x+c0 = 0

```
static bool Quartic(float c0, float c1, float c2, float c3, float c4, out QuarticRoots roots, float epsilon
= MathfEx.ZeroTolerance)
```
Quartic equations: c4*x^4+c3*x^3+c2*x^2+c1*x+c0 = 0

```
static float PolynomialBound(Polynomial poly, float epsilon = MathfEx.ZeroTolerance)
```
Gets roots bound of the given polynomial or -1 if polynomial is constant.

```
static bool Polynomial(Polynomial poly, float xMin, float xMax, out float[] roots, int digits = 6, float
epsilon = MathfEx.ZeroTolerance)
```
General polynomial equation: Σ(c_i * x^i), where i=[0..degree]. Finds roots in the interval [xMin..xMax].
poly - Polynomial whose roots to be found
xMin - Interval left border
xMax - Interval right border
roots - Roots of the polynomial
digits - Accuracy
epsilon - Small positive number

```
static bool Polynomial(Polynomial poly, out float[] roots, int digits = 6, float epsilon =
MathfEx.ZeroTolerance)
```
General polynomial equation: Σ(c_i * x^i), where i=[0..degree].
poly - Polynomial whose roots to be found
roots - Roots of the polynomial
digits - Accuracy
epsilon - Small positive number

```
static bool BrentsMethod(Func<float, float> function, x0, float x1, out BrentsRoot root,
int maxIterations = 128, float negativeTolerance = MathfEx.NegativeZeroTolerance,
float positiveTolerance = MathfEx.ZeroTolerance, float stepTolerance = MathfEx.ZeroTolerance,
float segmentTolerance = MathfEx.ZeroTolerance)
```
Implementation of Brent's Method for calculating a root of a function on an interval [x0,x1] for which f(x0)*f(x1)<0 (i.e. values of the function must have different signs on interval ends). The method uses inverse quadratic interpolation to generate a root estimate but falls back to inverse linear interpolation (secant method) if necessary. Also, based on previous iterates, the method will fall back to bisection when it appears the interpolated estimate is not of sufficient quality. The methid will compute a root (if any) on the interval [x0,x1]. The function returns true when the root is found, returns false when the interval is invalid x1<x0 or when f(x0)*f(x1)>0.
function - The function whose root must be found. The function accepts real number and returns real number.
x0 - Interval left border
x1 - Interval right border
root - Out parameter containing root of the function.
maxIterations - The maximum number of iterations used to locate a root. Should be positive number.
negativeTolerance - The root approximation x is accepted when the function value satisfies negativeTolerance <= f(x) <= positiveTolerance. negativeTolerance must be non-positive, positiveTolerance must be non-negative.
positiveTolerance - See negativeTolerance.
stepTolerance - The method requires some tests before an approximated x-value is accepted as the next root estimate. One of these tests compares the difference of consequent iterates and requires it to be larger than a stepTolerance to make sure the progress is made.
segmentTolerance - The root search is allowed to terminate when length of the subinterval is less than this tolerance.
Returns - True if root is found, false otherwise.

Type
`struct QuadraticRoots`

Properties
`float this[int rootIndex] { get; }`
Fields
`float X0`
`float X1`
`int RootCount`

Type
`struct CubicRoots`
Properties
`float this[int rootIndex] { get; }`
Fields
`float X0`
`float X1`
`float X2`
`int RootCount`

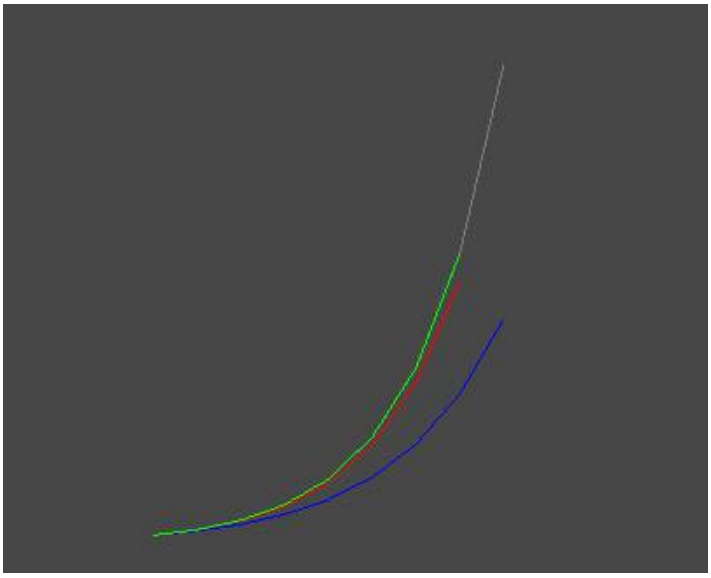
Type
`struct QuarticRoots`
Properties
`float this[int rootIndex] { get; }`
Fields
`float X0`
`float X1`
`float X2`
`float X3`
`int RootCount`

Type
`struct BrentsRoot`
Fields
`float X` Function root
`int Iterations` Number of cycles in the inner loop which were performed to find the root.
`bool ExceededMaxIterations` True when inner loop exceeds maxIterations variable (in which case root is assigned current approximation), false otherwise.

Type
`class Polynomial`

Represents n-degree polynomial of one variable
Properties
`int Degree { get; set; }` Gets or sets polynomial degree (0 - constant, 1 - linear, 2 - quadratic, etc). When set, recreates coefficient array thus all coefficients become 0.
`float this[int index] { get; set; }` Gets or sets polynomial coefficient. index - Valid index is 0<=index<=Degree
Construction
`Polynomial(int degree)` Creates polynomial of specified degree. Use indexer to set coefficients. Coefficients order is from smallest order to highest order, e.g for quadratic equation it's: c0 + c1*x + c2*x^2, coefficients array will be [c0,c1,c2]. degree - Must be >= 0!
`Polynomial DeepCopy()` Copies the polynomial
Methods
`Polynomial CalcDerivative()` Returns derivative of the current polynomial. Formula is: p (x) = c0 + c1*x + c2*x^2 + ... + cn*x^n p'(x) = c1 + 2*c2*x + 3*c3*x^2 + ... + n*cn*x^(n-1)
`Polynomial CalcInversion()` Computes inversion of the current polynomial (invpoly[i] = poly[degree-i] for 0<=i<=degree).
`void Compress(float epsilon = MathfEx.ZeroTolerance)` Reduce the degree by eliminating all (nearly) zero leading coefficients and by making the leading coefficient one. The input parameter is the threshold for specifying that a coefficient is effectively zero.
`float Eval(float t)` Evaluates the polynomial

10.4 Solving ODEs



Test Prefab:
Test_NumericalOdeSolver

Test prefab shows the examples of solving several ODEs using different methods (gray is exact solution, blue is Euler's method, red is midpoint method, green is Runge-Kutta method).

ODE solvers are organized slightly differently than other functionality in the library. Rather being static methods, each ODE solver is a separate class deriving from the common `OdeSolver` class. Starting with some initial values, user should call `Update` function on the solver consecutively. There is the example of this in the test prefab. Summary, library

provides several methods for solving ODEs. They are: Euler's method, midpoint method and Runge-Kutta 4th order method.

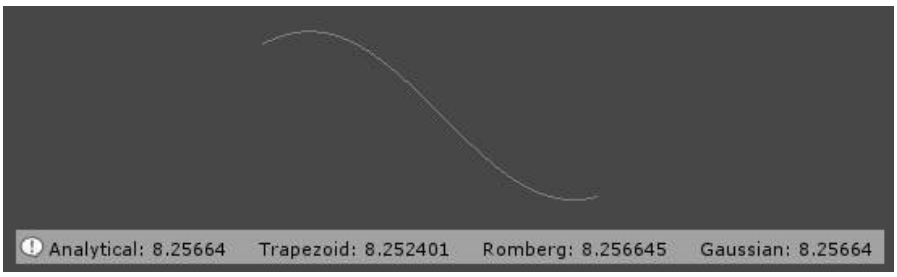
Type
`abstract class OdeSolver`
Properties
`virtual float Step { get; set; }`
Construction
`OdeSolver(int dim, float step, OdeFunction function)`
Methods
`abstract void Update(float tIn, float[] yIn, ref float tOut, float[] yOut)`

Type

`delegate void OdeFunction(`
`  float  t,  // t`
`  float[] y,  // y`
`  float[] F);  // F(t,y)`
The system is y'(t) = F(t,y). The dimension of y is passed to the constructor of OdeSolver.

Classes implementing the solver are `OdeEuler`, `OdeMidpoint`, `OdeRungeKutta4`. All have the same construction as the base class. Most accurate method is Runge-Kutta method. Other methods present mostly for comparison reasons.

10.5 Integrating



Test Prefab:
Test_NumericalIntegrator

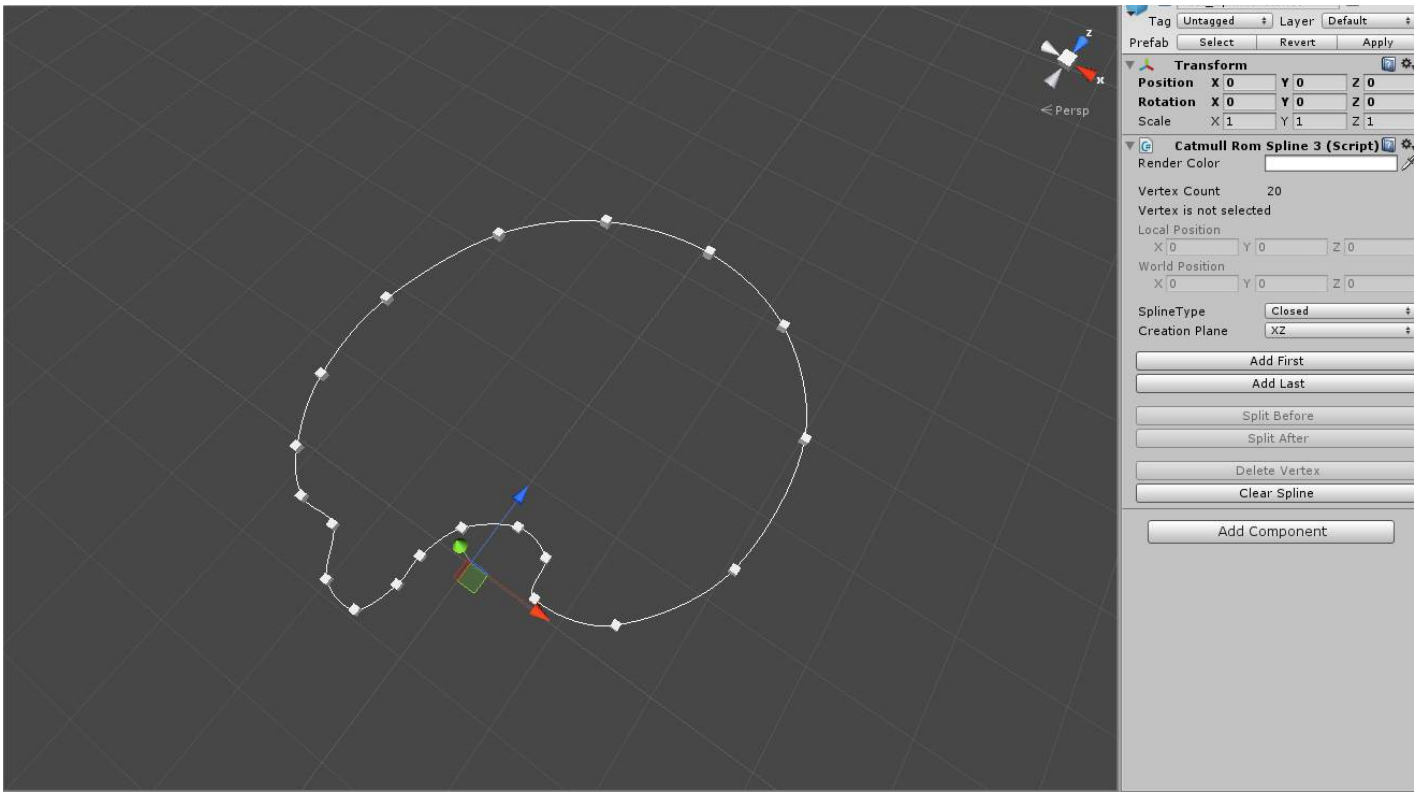
Test prefab shows the examples of intergrating several functions using several methods (trapezoidal rule, Romberg rule and gaussian quadrature rule).

Type
`class Integrator`
Methods
`static float TrapezoidRule(Func<float, float> function, float a, float b, int sampleCount)` Evaluates integral ∫f(x)dx on [a,b] interval using trapezoidal rule. sampleCount must be greater or equal to 2.
`static float RombergIntegral(Func<float, float> function, float a, float b, int order)` Evaluates integral ∫f(x)dx on [a,b] interval using Romberg's method. Integration order must be positive (order > 0).
`static float GaussianQuadrature(Func<float, float> function, float a, float b)` Evaluates integral ∫f(x)dx on [a,b] interval using Gaussian quadrature rule (five Legendre polynomials).

11 Curves

Many types of curves exist nowadays. Different curves have different properties. Among them cubic Hermite splines are pretty famous. Library provides implementation of Catmull-Rom splines which are one type of Hermite splines. In the future more spline types could be added (e.g. natural cubic splines, Bezier splines).

11.1 Catmull-Rom Spline

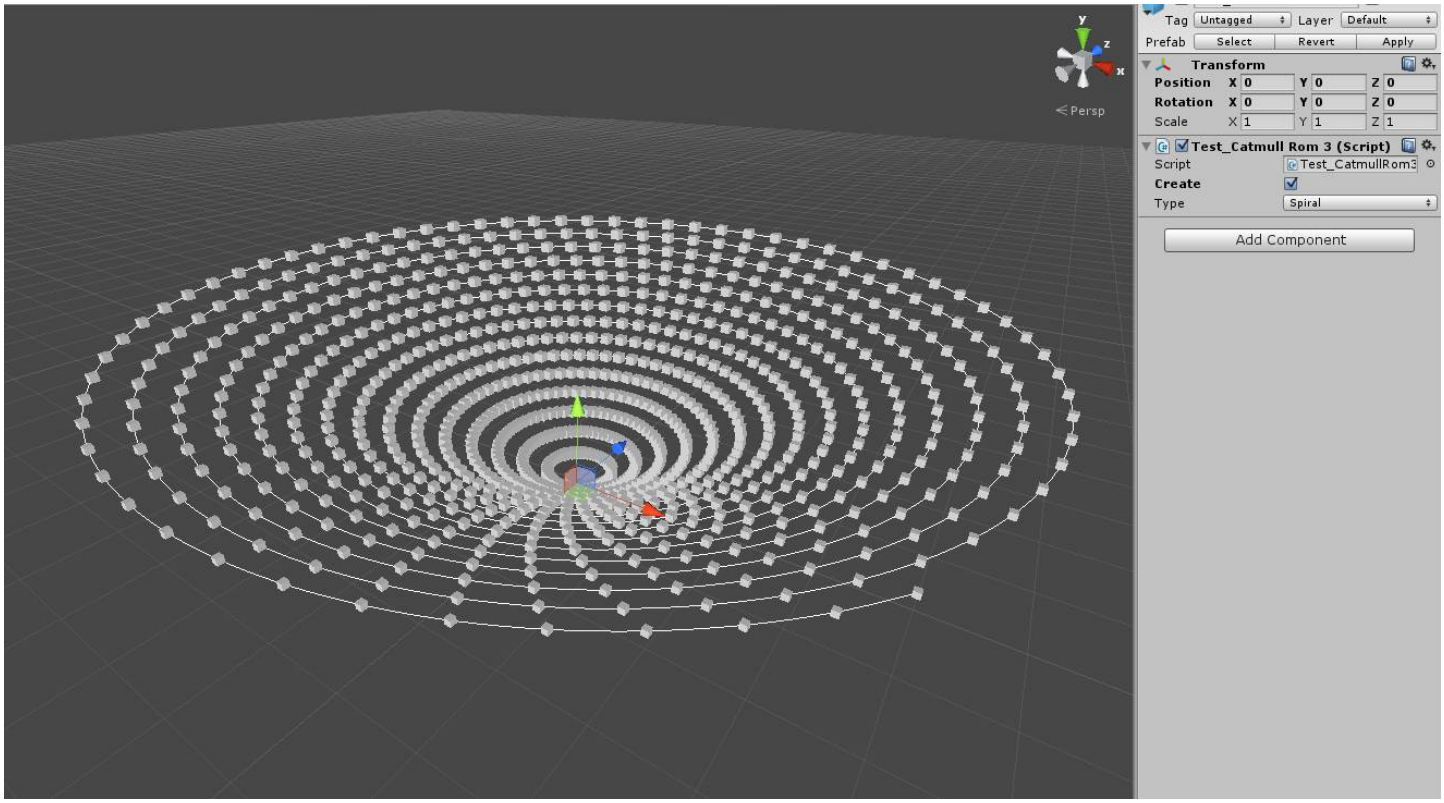


Test Prefab:
Test_SplineInstance
Test_CatmullRom3
Test_FollowSpline

The section describes Catmull-Rom spline class. It mostly serves as a reference. Detailed explanation of how to use and construct splines can be found in tutorials chapter of the documentation and next section about editor extension.

Catmull-Rom splines are piece-wise cubic polynomials which are C^1 curves (continious derivative). They are famous for that a spline goes through every control point, thus it's possible, for example, to create exact way for a camera in a scene. They also have extreme locality property, that is if one changes the vertex of a spline, only small area around a vertex must be updated (this is not true for all types of splines, for example, for natural cubic splines it's required to recalculate whole spline every time any vertex is changed). Locality allowed to

create very performant implementation in the code. For example, in Test_CatmullRom3 prefab user can create spiral spline consisting of 1000 vertices. On authors notebook (with Phenom II X4 N970 processor, HD 6650M video card) editing of this spline visually gives no lag at all. Thus it's possible to construct very large splines with this implementation. An example of spiral spline is on the image below.



`CatmullRomSpline3` class is derived from `MonoBehaviour` and has editor extension. It is serializable, thus users can make prefabs out of it (see Test_SplineInstance test prefab). It also supports Undo/Redo system (Unity 4.3 is required). Test_CatmullRom3 prefab shows examples of creating several splines. To create a spline click on the “Create” button in the inspector. Of course, spline can be created from the scratch as any other script, by adding the component to a game object.

Note that spline and spline editor classes are provided as source code and situated in Source\Curves and Source\Editor folders.

Type
`class CatmullRomSpline3 : MonoBehaviour`
Properties
`VertexCount { get; }`
`bool Valid { get; }`
`CatmullRomType SplineType { get; set; }`
`RenderColor { get; set; }`
Construction
`CatmullRomSpline3 Create()`
`CatmullRomSpline3 Create(IList<Vector3> points, CatmullRomType type)`

Methods
AddVertexFirst(Vector3 position)
AddVertexLast(Vector3 position)
RemoveVertex(int index)
Clear()
InsertBefore(int vertexIndex, Vector3 position)
InsertAfter(int vertexIndex, Vector3 position)
GetVertex(int vertexIndex)
SetVertex(int vertexIndex, Vector3 position)
EvalPosition(float time)
EvalTangent(float time)
EvalPositionTangent(float time)
EvalPosition(float time, out Vector3 position)
EvalTangent(float time, out Vector3 tangent)
EvalPositionTangent(float time, out PositionTangent positionTangent)
EvalFrame(float time, out CurveFrame frame)
EvalCurvature(float time)
EvalTorsion(float time)
EvalPositionParametrized(float length)
EvalTangentParametrized(float length)
EvalPositionTangentParametrized(float length)
EvalPositionParametrized(float length, out Vector3 position)
EvalTangentParametrized(float length, out Vector3 tangent)
EvalPositionTangentParametrized(float length, out PositionTangent positionTangent)
EvalFrameParametrized(float length, out CurveFrame frame)
EvalCurvatureParametrized(float length)
EvalTorsionParametrized(float length)
CalcTotalLength()
LengthToTime(float length, int iterations, float tolerance)
LengthToTime(float length)
ParametrizeByArcLength(int pointCount)

`int VertexCount  { get; }`
Gets spline vertex count

`bool Valid { get; }`
Returns true if spline is valid (i.e. contains 2 or more points) false otherwise

`CatmullRomType SplineType { get; set; }`
Gets or set spline type.

`Color RenderColor { get; set; }`
Gets or sets spline color in the editor.

`static CatmullRomSpline3 Create()`
Creates empty spline.

```
static CatmullRomSpline3 Create(IList<Vector3> points, CatmullRomType type)
```
Creates spline from supplied points.

```
void AddVertexFirst(Vector3 position)
```
Adds vertex in the beginning of the spline.

```
void AddVertexLast(Vector3 position)
```
Adds vertex to the end of the spline.

```
void RemoveVertex(int index)
```
Removes vertex from the spline. Valid index is [0..VertexCount-1].

```
void Clear()
```
Removes all vertices.

```
void InsertBefore(int vertexIndex, Vector3 position)
```
Inserts vertex before specified index. Valid index is [0..VertexCount].

```
void InsertAfter(int vertexIndex, Vector3 position)
```
Inserts vertex after specified index. Valid index is [-1..VertexCount-1]

```
Vector3 GetVertex(int vertexIndex)
```
Gets vertex position. Valid index is [0..VertexCount-1].

```
void SetVertex(int vertexIndex, Vector3 position)
```
Sets vertex position. Valid index is [0..VertexCount-1].

```
Vector3 EvalPosition(float time)
```
Evaluates position on the spline. Valid time is [0..1], where 0 is spline first point, 1 - spline end point.
Caller must ensure that spline is valid before calling this method!

```
Vector3 EvalTangent(float time)
```
Evaluates tangent on the spline. Valid time is [0..1], where 0 is spline first point, 1 - spline end point.
Caller must ensure that spline is valid before calling this method!

```
PositionTangent EvalPositionTangent(float time)
```
Evaluates position and tangent on the spline. Valid time is [0..1], where 0 is spline first point, 1 - spline end point.
Caller must ensure that spline is valid before calling this method!

```
void EvalPosition(float time, out Vector3 position)
```
Evaluates position on the spline. Valid time is [0..1], where 0 is spline first point, 1 - spline end point.
Caller must ensure that spline is valid before calling this method!

```
void EvalTangent(float time, out Vector3 tangent)
```
Evaluates tangent on the spline. Valid time is [0..1], where 0 is spline first point, 1 - spline end point.
Caller must ensure that spline is valid before calling this method!

```
void EvalPositionTangent(float time, out PositionTangent positionTangent)
```
Evaluates position and tangent on the spline. Valid time is [0..1], where 0 is spline first point, 1 - spline end point.
Caller must ensure that spline is valid before calling this method!

`void EvalFrame(float time, out CurveFrame frame)`
Evaluates Frene frame (orthogonal basis) and position on the spline. Valid time is [0..1], where 0 is spline first point, 1 - spline end point. Caller must ensure that spline is valid before calling this method!

`float EvalCurvature(float time)`
Evaluates curvature on the spline. Valid time is [0..1], where 0 is spline first point, 1 - spline end point. Caller must ensure that spline is valid before calling this method!

`float EvalTorsion(float time)`
Evaluates torsion on the spline. Valid time is [0..1], where 0 is spline first point, 1 - spline end point. Caller must ensure that spline is valid before calling this method!

`Vector3 EvalPositionParametrized(float length)`
Evaluates position on the spline using curve distance from the start. Valid length is [0..TotalLength].
Caller must ensure that spline is valid before calling this method!

`Vector3 EvalTangentParametrized(float length)`
Evaluates tangent on the spline using curve distance from the start. Valid length is [0..TotalLength].
Caller must ensure that spline is valid before calling this method!

`PositionTangent EvalPositionTangentParametrized(float length)`
Evaluates position and tangent on the spline using curve distance from the start. Valid length is [0..TotalLength].
Caller must ensure that spline is valid before calling this method!

`void EvalPositionParametrized(float length, out Vector3 position)`
Evaluates position on the spline using curve distance from the start. Valid length is [0..TotalLength].
Caller must ensure that spline is valid before calling this method!

`void EvalTangentParametrized(float length, out Vector3 tangent)`
Evaluates tangent on the spline using curve distance from the start. Valid length is [0..TotalLength].
Caller must ensure that spline is valid before calling this method!

`void EvalPositionTangentParametrized(float length, out PositionTangent positionTangent)`
Evaluates position and tangent on the spline using curve distance from the start. Valid length is [0..TotalLength].
Caller must ensure that spline is valid before calling this method!

`void EvalFrameParametrized(float length, out CurveFrame frame)`
Evaluates Frenet frame on the spline using curve distance from the start. Valid length is [0..TotalLength]. Caller must ensure that spline is valid before calling this method!

`float EvalCurvatureParametrized(float length)`
Evaluates curvature on the spline using curve distance from the start. Valid length is [0..TotalLength]. Caller must ensure that spline is valid before calling this method!

`float EvalTorsionParametrized(float length)`
Evaluates torsion on the spline using curve distance from the start. Valid length is [0..TotalLength].
Caller must ensure that spline is valid before calling this method!

`float CalcTotalLength()`
Returns total length of the spline.

`float LengthToTime(float length, int iterations, float tolerance)`
Converts length parameter [0..TotalLength] to time parameter [0..1]
length - Distance parameter to convert into time parameter
iterations - Number of iterations used in internal calculations, default is 32
tolerance - Small positive number, e.g. 1e-5f

`float LengthToTime(float length)`
Converts length parameter [0..TotalLength] to time parameter [0..1]
length - Distance parameter to convert into time parameter

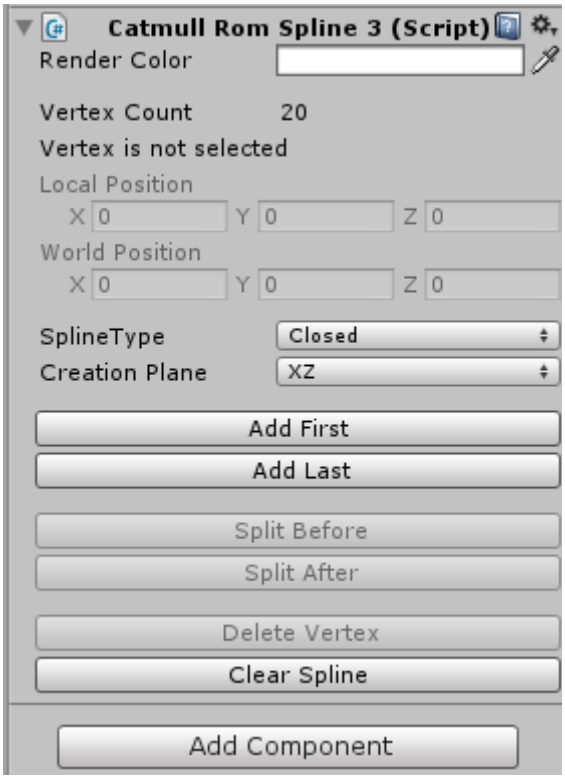
`float ParametrizeByArcLength(int pointCount)`
Parametrizes the spline using arc length. This method must be called before calling any parametrized evaluation methods, otherwise they will throw an exception. Returns spline total length.
pointCount - Number of points which will be used to divide total length of the spline into equal intervals when parametrizing.

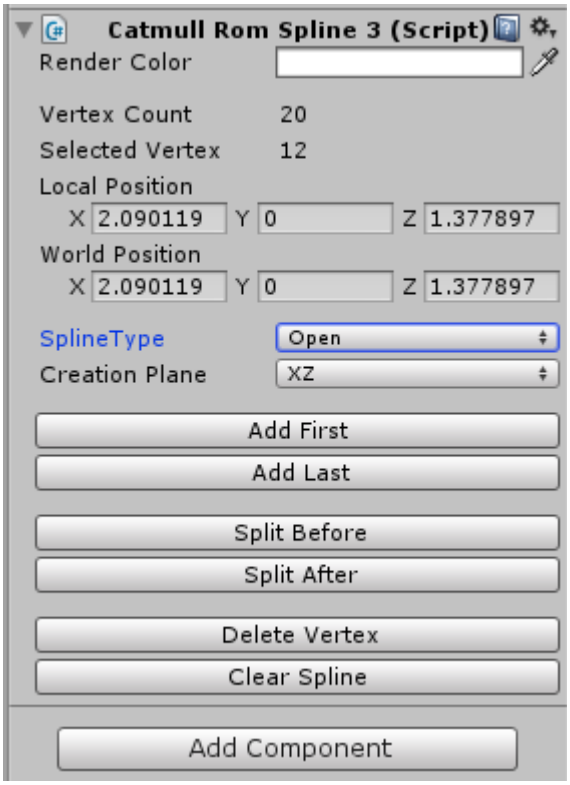
11.2 Catmull-Rom Spline Editor Extension

Editor extension `CatmullRomSpline3Editor` has been implemented to help users construct Catmull-Rom splines in the editor. This is extremely useful for non-technical staff, e.g. level designers and artists. This section describes functionality available in the Inspector of `CatmullRomSpline3` class.

As spline and spline editor are given as source code, user can change or add functionality to meet his specific needs. Currently editor has several common functions which most users will require to construct their splines.

Images below shows the examples of the Inspector window when an instance of a spline is selected.





In the top of the window the first option is the color the spline. It's relevant only for the editor and is not used elsewhere.

Next the vertex information group follows. It show total amount of vertices in the spline. If

some vertex is selected then it shows the index of the selected vertex and also gives the ability to change vertex position in both local and world coordinates. Local coordinates are in the coordinate system of the spline object transform while world coordinates are global coordinates.

Further below user will find two enumerations. The first one is affecting spline closure. There are two options – open and closed splines. Closed spline will connect first and last vertex with the additional segment, thus it's possible to create splines for car tracks for example. “Creation Plane” affects projection of the new vertices which are added using intruments in the below (options are XZ, XY, YZ).

The rest of the buttons are the instruments for manupulating spline vertices. “Add First” and “Add Last” are toggle buttons. When either of them are is toggled on clicking in the Scene window will add vertices (either to the end of the spline or at the start of the spline). Here creation plane affects projection of the new vertices (e.g. if the mode is XZ then new vertices will be projected to XZ plane and has 0 y coordinate). When user wants to end vertex addition he can toggle off the button or just right-click in the scene window.

When some vertex is selected “Split Before” and “Split After” buttons are active. They will add the new vertex in front of or afterwards the selected vertex in the middle of the segment. This allows to refine the spline where necessary.

Last two buttons allow to delete selected vertex and to clear the spline completely leaving no vertices at all.

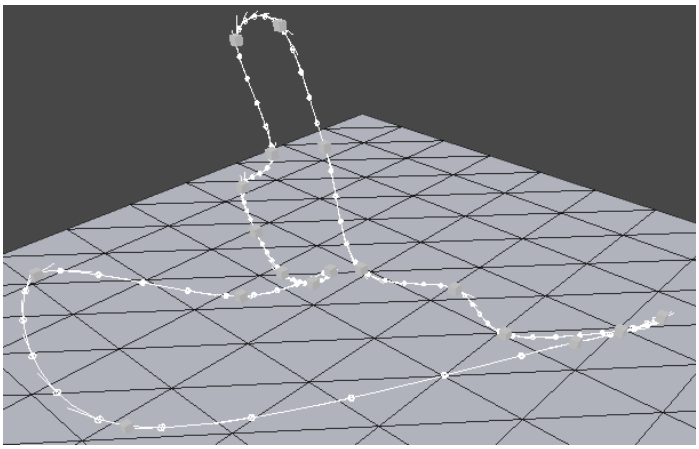
11.3 Moving Along Spline at Constant Speed

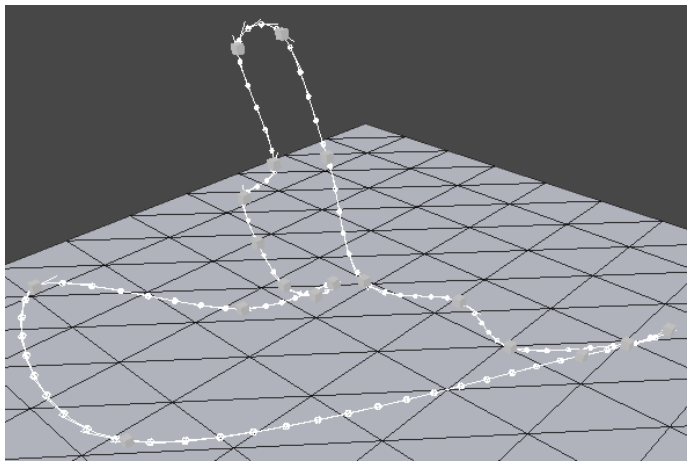
Probably splines are used the most for positioning of some objects. Having some parameter user want to get the position on the spline. On of the perametrizations is the parametrization by time where time goes from 0 (start of the spline) to 1 (end of the spline). Catmull-Rom spline implemented in the library has uniform parametrization. It means that every segment has equal amount of time. E.g., if spline has 3 vertices, then parametrization at vertex 0 is 0, at vertex 1 is 0.5 and at vertex 2 is 1. This parameterization does not depent on vertex placement. Thus when one evaluates the spline using time parameter the resulting set of vertices will not be evenly distrubuted along the spline which will result in non-constant speed movement (i.e. on long segment object will move faster than on small segments). Sometimes it's crucial for an object to move evenly. Of course one solution is to place spline vertices approximately such that segments lengths are equal, however this is no very convenient.

To overcome the problem other parameterization is required. This parameterization is called “arc-length parameterization” and the parameter is the distance from the beginning of a spline. Arc-length parameterization requires heavy computations and is barely can be done in real-time application every frame. However it is possible to precalculate some data it use it to perform approximate arc-length parameterization during real time. Catmull-Rom spline has such method and it's called `ParametrizeByArcLength(int pointCount)`. This method must be called prior to calling any of the methods which return arc-length parametrized data.

Test_FollowSpline prefab has the example of using parameterizations. Look at the images below. Image on the left show time parameterization, while the image on the right – arc-length parameterization. Notice that arc-length offers even distribution of the points along the spline no matter how long the segments are. In the test prefab user can select whether the spline should be arc-length parameterized (“Constant Speed Traversal” toggle is on) or not (toggle if off). Choose the parameter (and other parameters if required) and press play to

see the result.





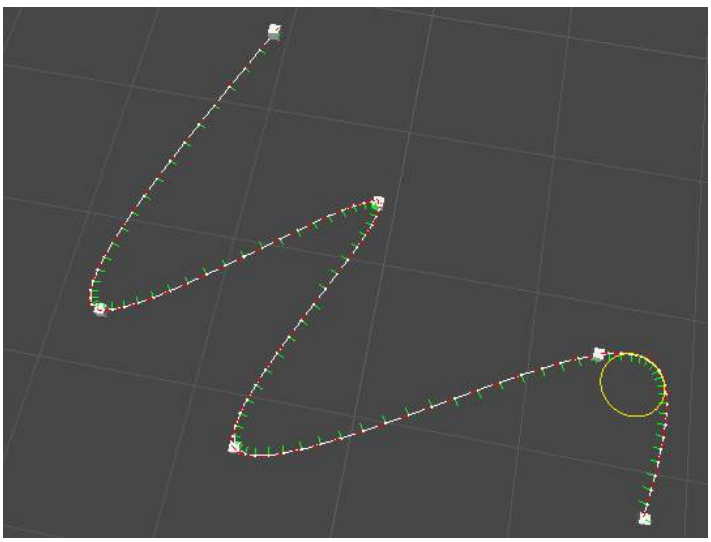
Not only the position but also tanget could be quired from the spline (small line segments on the images above are tangents). It's useful when one needs not only to position an object but also to align an object while moving. To see the examples of getting position and tangent (both usual and arc-length parametrized) from the spline, see Test_FollowSpline prefab.

11.4 Cubic Spline (natural and closed)

Although technically term “cubic spline” can be applied to most of the splines (as they are piecewise third-order polynomials), we will call a “cubic spline” one of the types of splines which have C^2 continuity of the inner points.

Two variations of these splines are presented in the library. First – natural cubic splines; they have zero second derivative at the ends of the spline. Second – closed cubic splines; the first and the last points of such spline have the same first and second derivatives.

Class which represents these splines is called `CubicSpline3`. It's similar to the Catmull-Rom spilnes in terms of the usage and editor extension, thus refer to the Catmull-rom subsection for the reference.



Test Prefab:
Test_CubicSpline3

Shows CubicSpline3 example and also shows calculation of spline curvature. Drag CurvatureIndex parameter in the inspector to get curvature of different spline points.

12 Misc

Library contains large amount of convenient extensions and additions to standard Unity types and several useful generic math methods. There are four extension static classes and they are described in following sections.

12.1 Vector2ex

Type
class Vector2ex
Fields
Zero
One
UnitX
UnitY
PositiveInfinity
NegativeInfinity
Extensions
Length(this Vector2 vector)
LengthSqr(this Vector2 vector)
DotPerp(this Vector2 vector, Vector2 value)
DotPerp(this Vector2 vector, ref Vector2 value)
Dot(this Vector2 vector, Vector2 value)
Dot(this Vector2 vector, ref Vector2 value)
Perp(this Vector2 vector)
AngleDeg(this Vector2 vector, Vector2 target)
AngleRad(this Vector2 vector, Vector2 target)
ToVector3XY(this Vector2 vector)
ToVector3XZ(this Vector2 vector)
ToVector3YZ(this Vector2 vector)
ToStringEx(this Vector2 vector)
Methods
DotPerp(ref Vector2 vector, ref Vector2 value)
Dot(ref Vector2 vector, ref Vector2 value)
Normalize(ref Vector2 vector, float epsilon)
SetLength(ref Vector2 vector, float lengthValue, float epsilon)
GrowLength(ref Vector2 vector, float lengthDelta, float epsilon)
Replicate(float value)

```
static readonly Vector2 Zero
```
Vector2(0.0f, 0.0f)

```
static readonly Vector2 One
```
Vector2(1.0f, 1.0f)

```
static readonly Vector2 UnitX
```
Vector2(1.0f, 0.0f)

```
static readonly Vector2 UnitY
```
Vector2(0.0f, 1.0f)

```
static readonly Vector2 PositiveInfinity
```
Vector2(float.PositiveInfinity, float.PositiveInfinity);

```
static readonly Vector2 NegativeInfinity
```
Vector2(float.NegativeInfinity, float.NegativeInfinity)

```
static float Length(this Vector2 vector)
```
Returns vector length

```
static float LengthSqr(this Vector2 vector)
```
Returns vector squared length

```
static float DotPerp(this Vector2 vector, Vector2 value)
static float DotPerp(this Vector2 vector, ref Vector2 value)
```
Returns x0*y1 - y0*x1

```
static float Dot(this Vector2 vector, Vector2 value)
static float Dot(this Vector2 vector, ref Vector2 value)
```
Vector dot product

```
static Vector2 Perp(this Vector2 vector)
```
Returns (y,-x)

```
static float AngleDeg(this Vector2 vector, Vector2 target)
```
Returns angle in degrees between this vector and the target vector. Method normalizes input vectors. Result lies in [0..180] range.

```
static float AngleRad(this Vector2 vector, Vector2 target)
```
Returns angle in radians between this vector and the target vector. Method normalizes input vectors. Result lies in [0..PI] range.

```
static Vector3 ToVector3XY(this Vector2 vector)
```
Converts Vector2 to Vector3, copying x and y components of Vector2 to x and y components of Vector3 respectively.

```
static Vector3 ToVector3XZ(this Vector2 vector)
```
Converts Vector2 to Vector3, copying x and y components of Vector2 to x and z components of Vector3 respectively.

```
static Vector3 ToVector3YZ(this Vector2 vector)
```
Converts Vector2 to Vector3, copying x and y components of Vector2 to y and z components of Vector3 respectively.

```
static string ToStringEx(this Vector2 vector)
```
Returns string representation (does not round components as standard Vector2.ToString() does)

```
static float DotPerp(ref Vector2 vector, ref Vector2 value)
```
Returns x0*y1 - y0*x1

```
static float Dot(ref Vector2 vector, ref Vector2 value)
```
Vector dot product

```
static float Normalize(ref Vector2 vector, float epsilon = MathfEx.ZeroTolerance)
```
Normalizes given vector and returns it's length before normalization.

```
static float SetLength(ref Vector2 vector, float lengthValue, float epsilon = Mathfex.ZeroTolerance)
```
Sets vector length to the given value. Returns new vector length or 0 if vector's initial length is less than epsilon.

```
static float GrowLength(ref Vector2 vector, float lengthDelta, float epsilon = Mathfex.ZeroTolerance)
```
Changes vector length adding given value. Returns new vector length or 0 if vector's initial length is less than epsilon.

```
static Vector2 Replicate(float value)
```
Creates a vector with all components equal to value

12.2 Vector3ex

Type
class Vector3ex
Fields
Zero
One
UnitX
UnitY
UnitZ
PositiveInfinity
NegativeInfinity
Extensions
Length(this Vector3 vector)
LengthSqr(this Vector3 vector)
Dot(this Vector3 vector, Vector3 value)
Dot(this Vector3 vector, ref Vector3 value)
AngleDeg(this Vector3 vector, Vector3 target)
AngleRad(this Vector3 vector, Vector3 target)
SignedAngleDeg(this Vector3 vector, Vector3 target, Vector3 normal)
SignedAngleRad(this Vector3 vector, Vector3 target, Vector3 normal)
Cross(this Vector3 vector, Vector3 value)
Cross(this Vector3 vector, ref Vector3 value)
UnitCross(this Vector3 vector, Vector3 value)
UnitCross(this Vector3 vector, ref Vector3 value)
ToVector2XY(this Vector3 vector)
ToVector2XZ(this Vector3 vector)
ToVector2YZ(this Vector3 vector)
ToVector2(this Vector3 vector, ProjectionPlanes projectionPlane)
GetProjectionPlane(this Vector3 vector)
ToStringEx(this Vector3 vector)
Methods
Dot(ref Vector3 vector, ref Vector3 value)
Cross(ref Vector3 vector, ref Vector3 value)
UnitCross(ref Vector3 vector, ref Vector3 value)
Normalize(ref Vector3 vector, float epsilon)
SetLength(ref Vector3 vector, float lengthValue, float epsilon)
GrowLength(ref Vector3 vector, float lengthDelta, float epsilon)
Replicate(float value)
CreateOrthonormalBasis(out Vector3 u, out Vector3 v, ref Vector3 w)
SameDirection(Vector3 value0, Vector3 value1)
AngleDeg(Vector3 v0, Vector3 v1, Vector3 v2)
AngleRad(Vector3 v0, Vector3 v1, Vector3 v2)

SignedAngleDeg(Vector3 v0, Vector3 v1, Vector3 v2, Vector3 normal)
SignedAngleRad(Vector3 v0, Vector3 v1, Vector3 v2, Vector3 normal)

static readonly Vector3 Zero
Vector3(0.0f, 0.0f, 0.0f)

static readonly Vector3 One
Vector3(1.0f, 1.0f, 1.0f)

static readonly Vector3 UnitX
Vector3(1.0f, 0.0f, 0.0f)

static readonly Vector3 UnitY
Vector3(0.0f, 1.0f, 0.0f)

static readonly Vector3 UnitZ
Vector3(0.0f, 0.0f, 1.0f)

static readonly Vector3 PositiveInfinity
Vector3(float.PositiveInfinity, float.PositiveInfinity, float.PositiveInfinity)

static readonly Vector3 NegativeInfinity
Vector3(float.NegativeInfinity, float.NegativeInfinity, float.NegativeInfinity)

static float Length(this Vector3 vector)
Returns vector length

static float LengthSqr(this Vector3 vector)
Returns vector squared length

static float Dot(this Vector3 vector, Vector3 value)
static float Dot(this Vector3 vector, ref Vector3 value)
Vector dot product

static float AngleDeg(this Vector3 vector, Vector3 target)
Returns angle in degrees between this vector and the target vector. Method normalizes input vectors. Result lies in [0..180] range.

static float AngleRad(this Vector3 vector, Vector3 target)
Returns angle in radians between this vector and the target vector. Method normalizes input vectors. Result lies in [0..PI] range.

static float SignedAngleDeg(this Vector3 vector, Vector3 target, Vector3 normal)
Returns signed angle in degrees between this vector and the target vector. Method normalizes input vectors. Result lies in [-180..180] range.
normal - Vector which defines world 'up'

static float SignedAngleRad(this Vector3 vector, Vector3 target, Vector3 normal)
Returns signed angle in radians between this vector and the target vector. Method normalizes input vectors. Result lies in [-PI..PI] range.
normal - Vector which defines world 'up'

```
static Vector3 Cross(this Vector3 vector, Vector3 value)
static Vector3 Cross(this Vector3 vector, ref Vector3 value)
```
Vector cross product

```
static Vector3 UnitCross(this Vector3 vector, Vector3 value)
static Vector3 UnitCross(this Vector3 vector, ref Vector3 value)
```
Returns normalized cross product of vectors

```
static Vector2 ToVector2XY(this Vector3 vector)
```
Converts Vector3 to Vector2, copying x and y components of Vector3 to x and y components of Vector2 respectively.

```
static Vector2 ToVector2XZ(this Vector3 vector)
```
Converts Vector3 to Vector2, copying x and z components of Vector3 to x and y components of Vector2 respectively.

```
static Vector2 ToVector2YZ(this Vector3 vector)
```
Converts Vector3 to Vector2, copying y and z components of Vector3 to x and y components of Vector2 respectively.

```
static Vector2 ToVector2(this Vector3 vector, ProjectionPlanes projectionPlane)
```
Converts Vector3 to Vector2 using specified projection plane.

```
static ProjectionPlanes GetProjectionPlane(this Vector3 vector)
```
Returns most appropriate projection plane considering vector as a normal (e.g. if y component is largest, then XZ plane is returned).

```
static string ToStringEx(this Vector3 vector)
```
Returns string representation (does not round components as standard Vector3.ToString() does)

```
static float Dot(ref Vector3 vector, ref Vector3 value)
```
Vector dot product

```
static Vector3 Cross(ref Vector3 vector, ref Vector3 value)
```
Vector cross product

```
static Vector3 UnitCross(ref Vector3 vector, ref Vector3 value)
```
Returns normalized cross product of vectors

```
static float Normalize(ref Vector3 vector, float epsilon = MathfEx.ZeroTolerance)
```
Normalizes given vector and returns it's length before normalization.

```
static float SetLength(ref Vector3 vector, float lengthValue, float epsilon = Mathfex.ZeroTolerance)
```
Sets vector length to the given value. Returns new vector length or 0 if vector's initial length is less than epsilon.

```
static float GrowLength(ref Vector3 vector, float lengthDelta, float epsilon = Mathfex.ZeroTolerance)
```
Changes vector length adding given value. Returns new vector length or 0 if vector's initial length is less than epsilon.

```
static Vector3 Replicate(float value)
```
Creates a vector with all components equal to value

```
static void CreateOrthonormalBasis(out Vector3 u, out Vector3 v, ref Vector3 w)
```
Input W must be a unit-length vector. The output vectors {U,V} are unit length and mutually perpendicular, and {U,V,W} is an orthonormal basis.

```
static bool SameDirection(Vector3 value0, Vector3 value1)
```
Returns true if Dot(value0,value1) > 0

```
static float AngleDeg(Vector3 v0, Vector3 v1, Vector3 v2)
```
Returns angle in degrees between v1-v0 vector and v2-v0 vector. Result lies in [0..180] range.

```
static float AngleRad(Vector3 v0, Vector3 v1, Vector3 v2)
```
Returns angle in radians between v1-v0 vector and v2-v0 vector. Result lies in [0..PI] range.

```
static float SignedAngleDeg(Vector3 v0, Vector3 v1, Vector3 v2, Vector3 normal)
```
Returns signed angle in degrees between v1-v0 vector and v2-v0 vector. Result lies in [-180..180] range.
normal - Vector which defines world 'up

```
static float SignedAngleRad(Vector3 v0, Vector3 v1, Vector3 v2, Vector3 normal)
```
Returns signed angle in radians between v1-v0 vector and v2-v0 vector. Result lies in [-PI..PI] range.
normal - Vector which defines world 'up'

12.3 Quaternionex

Type
`class Quaternionex`
Extensions
`DeltaTo(this Quaternion quat, Quaternion target)`
`ToStringEx(this Quaternion quat)`

`static Quaternion DeltaTo(this Quaternion quat, Quaternion target)`
Calculates difference from this quaternion to given target quaternion. I.e. if you have quaternions Q1 and Q2, this method will return quaternion Q such that Q2 == Q * Q1 (remember that quaternions are multiplied right-to-left).

`static string ToStringEx(this Quaternion quat)`
Returns string representation (does not round components as standard Quaternion.ToString() does)

12.4 Matrix4x4ex

This helper class offers large amount of additional methods for Unity matrices, including conversion to and from quaternion, constructing transformation matrices, constructing shadow matrices and so on. Many of the presented methods are faster than similar Unity methods.

Test Prefab:
Test_Matrix4x4Ex

Test prefab has the ability to test correctness of many methods (uncomment `TestCorrectness()` method in Start method) and also test their performance comparing to Unity counterparts where it's meaningful (uncomment `TestPerformance()` method in Start method). Also it shows shadow matrix in action if one looks at game window when player is launched.

Type
`class Matrix4x4ex`
Fields
`Identity`
Methods
`RotationMatrixToQuaternion(ref Matrix4x4 matrix, out Quaternion quaternion)`
`QuaternionToRotationMatrix(Quaternion quaternion, out Matrix4x4 matrix)`
`QuaternionToRotationMatrix(ref Quaternion quaternion, out Matrix4x4 matrix)`
`CreateTranslation(Vector3 position, out Matrix4x4 matrix)`
`CreateTranslation(ref Vector3 position, out Matrix4x4 matrix)`
`CreateScale(Vector3 scale, out Matrix4x4 matrix)`
`CreateScale(ref Vector3 scale, out Matrix4x4 matrix)`
`CreateScale(float scale, out Matrix4x4 matrix)`
`CreateRotationEuler(float eulerX, float eulerY, float eulerZ, out Matrix4x4 matrix)`
`CreateRotationEuler(Vector3 eulerAngles, out Matrix4x4 matrix)`
`CreateRotationEuler(ref Vector3 eulerAngles, out Matrix4x4 matrix)`
`CreateRotationX(float angleInDegrees, out Matrix4x4 matrix)`
`CreateRotationY(float angleInDegrees, out Matrix4x4 matrix)`
`CreateRotationZ(float angleInDegrees, out Matrix4x4 matrix)`
`CreateRotationAngleAxis(float angleInDegrees, Vector3 rotationAxis, out Matrix4x4 matrix)`
`CreateRotationAngleUnitAxis(float angleInDegrees, Vector3 normalizedAxis, out Matrix4x4 matrix)`
`CreateRotation(Vector3 rotationOrigin, Quaternion rotation, out Matrix4x4 result)`
`CreateRotation(ref Vector3 rotationOrigin, ref Quaternion rotation, out Matrix4x4 result)`
`Transpose(ref Matrix4x4 matrix)`
`Transpose(ref Matrix4x4 matrix, out Matrix4x4 transpose)`
`CalcDeterminant(ref Matrix4x4 matrix)`
`Inverse(ref Matrix4x4 matrix, float epsilon)`
`Inverse(ref Matrix4x4 matrix, out Matrix4x4 inverse, float epsilon)`
`CopyMatrix(ref Matrix4x4 source, out Matrix4x4 destination)`

Multiply(ref Matrix4x4 matrix0, ref Matrix4x4 matrix1, out Matrix4x4 result)
MultiplyRight(ref Matrix4x4 matrix0, ref Matrix4x4 matrix1)
MultiplyLeft(ref Matrix4x4 matrix1, ref Matrix4x4 matrix0)
Multiply(ref Matrix4x4 matrix, float scalar)
Multiply(ref Matrix4x4 matrix, float scalar, out Matrix4x4 result)
Multiply(ref Matrix4x4 matrix, Vector4 vector)
Multiply(ref Matrix4x4 matrix, ref Vector4 vector)
CreateSRT(Vector3 scaling, Quaternion rotation, Vector3 translation, out Matrix4x4 result)
CreateSRT(ref Vector3 scaling, ref Quaternion rotation, ref Vector3 translation, out Matrix4x4 result)
CreateSRT(float scaling, Quaternion rotation, Vector3 translation, out Matrix4x4 result)
CreateSRT(float scaling, ref Quaternion rotation, ref Vector3 translation, out Matrix4x4 result)
CreateSRT(Vector3 scaling, Vector3 rotationOrigin, Quaternion rotation, Vector3 translation, out Matrix4x4 result)
CreateSRT(ref Vector3 scaling, ref Vector3 rotationOrigin, ref Quaternion rotation, ref Vector3 translation, out Matrix4x4 result)
CreateSRT(float scaling, Vector3 rotationOrigin, Quaternion rotation, Vector3 translation, out Matrix4x4 result)
CreateSRT(float scaling, ref Vector3 rotationOrigin, ref Quaternion rotation, ref Vector3 translation, out Matrix4x4 result)
CreateRT(Quaternion rotation, Vector3 translation, out Matrix4x4 result)
CreateRT(ref Quaternion rotation, ref Vector3 translation, out Matrix4x4 result)
CreateRT(Vector3 rotationOrigin, Quaternion rotation, Vector3 translation, out Matrix4x4 result)
CreateRT(ref Vector3 rotationOrigin, ref Quaternion rotation, ref Vector3 translation, out Matrix4x4 result)
CreateST(Vector3 scaling, Vector3 translation, out Matrix4x4 result)
CreateST(ref Vector3 scaling, ref Vector3 translation, out Matrix4x4 result)
CreateRotationFromColumns(Vector3 column0, Vector3 column1, Vector3 column2, out Matrix4x4 matrix)
CreateRotationFromColumns(ref Vector3 column0, ref Vector3 column1, ref Vector3 column2, out Matrix4x4 matrix)
CreateShadowDirectional(Plane3 shadowPlane, Vector3 dirLightOppositeDirection, out Matrix4x4 result)
CreateShadowDirectional(ref Plane3 shadowPlane, ref Vector3 dirLightOppositeDirection, out Matrix4x4 result)
CreateShadowPoint(Plane3 shadowPlane, Vector3 pointLightPosition, out Matrix4x4 result)
CreateShadowPoint(ref Plane3 shadowPlane, ref Vector3 pointLightPosition, out Matrix4x4 result)
CreateShadow(Plane3 shadowPlane, Vector4 lightData, out Matrix4x4 result)
CreateShadow(ref Plane3 shadowPlane, ref Vector4 lightData, out Matrix4x4 result)

```
static readonly Matrix4x4 Identity
```
Identity matrix

```
static void RotationMatrixToQuaternion(ref Matrix4x4 matrix, out Quaternion quaternion)
```
Converts rotation matrix to quaternion

```
static void QuaternionToRotationMatrix(Quaternion quaternion, out Matrix4x4 matrix)
static void QuaternionToRotationMatrix(ref Quaternion quaternion, out Matrix4x4 matrix)
```
Converts quaternion to rotation matrix

```
static void CreateTranslation(Vector3 position, out Matrix4x4 matrix)
static void CreateTranslation(ref Vector3 position, out Matrix4x4 matrix)
```
Creates translation matrix

```
static void CreateScale(Vector3 scale, out Matrix4x4 matrix)
static void CreateScale(ref Vector3 scale, out Matrix4x4 matrix)
```
Creates non-uniform scale matrix

```
static void CreateScale(float scale, out Matrix4x4 matrix)
```
Creates uniform scale matrix

```
static void CreateRotationEuler(float eulerX, float eulerY, float eulerZ, out Matrix4x4 matrix)
```
Creates rotaion matrix using euler angles (order is the same as in Quaternion.Euler() method)

```
static void CreateRotationEuler(Vector3 eulerAngles, out Matrix4x4 matrix)
static void CreateRotationEuler(ref Vector3 eulerAngles, out Matrix4x4 matrix)
```
Creates rotaion matrix using euler angles (order is the same as in Quaternion.Euler() method)

```
static void CreateRotationX(float angleInDegrees, out Matrix4x4 matrix)
```
Creates a matrix that rotates around x-axis

```
static void CreateRotationY(float angleInDegrees, out Matrix4x4 matrix)
```
Creates a matrix that rotates around y-axis

```
static void CreateRotationZ(float angleInDegrees, out Matrix4x4 matrix)
```
Creates a matrix that rotates around z-axis

```
static void CreateRotationAngleAxis(float angleInDegrees, Vector3 rotationAxis, out Matrix4x4 matrix)
```
Creates a matrix that rotates around an arbirary axis (function will normalize axis)

```
static void CreateRotationAngleUnitAxis(float angleInDegrees, Vector3 normalizedAxis, out Matrix4x4 matrix)
```
Creates a matrix that rotates around an arbirary axis (caller must provide unit-length axis)

```
static void CreateRotation(Vector3 rotationOrigin, Quaternion rotation, out Matrix4x4 result)
static void CreateRotation(ref Vector3 rotationOrigin, ref Quaternion rotation, out Matrix4x4 result)
```
Creates a matrix that rotates around specified point

```
static void Transpose(ref Matrix4x4 matrix)
```
Transposes given matrix
matrix - Matrix to transpose (will be overriden to contain output)

```
static void Transpose(ref Matrix4x4 matrix, out Matrix4x4 transpose)
```
Transposes given matrix
matrix - Matrix to transpose
transpose - Output containing transposed matrix

```
static float CalcDeterminant(ref Matrix4x4 matrix)
```
Returns matrix determinant

```
static void Inverse(ref Matrix4x4 matrix, float epsilon = MathfEx.ZeroTolerance)
```
Inverses given matrix
matrix - Matrix to inverse (will be overriden to contain output)
epsilon - Small positive number used to compare determinant with zero

```
static void Inverse(ref Matrix4x4 matrix, out Matrix4x4 inverse, float epsilon = MathfEx.ZeroTolerance)
```
Inverses given matrix. IMPORTANT: 'matrix' and 'inverse' parameters must be different! If you want matrix to contain inverse of itself, use another overload.
matrix - Matrix to inverse
inverse - Output containing inverse matrix (Must not be the same variable as 'matrix'!)
epsilon - Small positive number used to compare determinant with zero

```
static void CopyMatrix(ref Matrix4x4 source, out Matrix4x4 destination)
```
Copies source matrix into destination matrix

```
static void Multiply(ref Matrix4x4 matrix0, ref Matrix4x4 matrix1, out Matrix4x4 result)
```
Multiplies two matrices. Result matrix is matrix0*matrix1. IMPORTANT: 'result' parameter must not be the same variable as either 'matrix0' or 'matrix1', if you want to store the result in one of the input matrices, use MultiplyLeft or MultiplyRight methods.

```
static void MultiplyRight(ref Matrix4x4 matrix0, ref Matrix4x4 matrix1)
```
Multiplies matrix0 by matrix1 on the right, i.e. the result is matrix0*matrix1. Output is written into matrix0 parameter.

```
static void MultiplyLeft(ref Matrix4x4 matrix1, ref Matrix4x4 matrix0)
```
Multiplies matrix1 by matrix0 on the left, i.e. the result is matrix0*matrix1. Output is written into matrix1 parameter.

```
static void Multiply(ref Matrix4x4 matrix, float scalar)
```
Multiplies matrix by a scalar.
matrix - Matrix to multiply (will be overriden to contain output)
scalar - Scalar to multiply

```
static void Multiply(ref Matrix4x4 matrix, float scalar, out Matrix4x4 result)
```
Multiplies matrix by a scalar.
matrix - Matrix to multiply
scalar - Scalar to multiply
result - Output containing multiplied matrix

```
static Vector4 Multiply(ref Matrix4x4 matrix, Vector4 vector)
static Vector4 Multiply(ref Matrix4x4 matrix, ref Vector4 vector)
```
Multiplies matrix by vector. Result vector is matrix*vector.
matrix - Matrix to multiply
vector - Vector to multiply

```
static void CreateSRT(Vector3 scaling, Quaternion rotation, Vector3 translation, out Matrix4x4 result)
static void CreateSRT(ref Vector3 scaling, ref Quaternion rotation, ref Vector3 translation, out Matrix4x4
result)
```
Creates a transformation matrix. Transformation order is: scaling, rotation, translation.

```
static void CreateSRT(float scaling, Quaternion rotation, Vector3 translation, out Matrix4x4 result)
static void CreateSRT(float scaling, ref Quaternion rotation, ref Vector3 translation, out Matrix4x4
result)
```
Creates a transformation matrix. Transformation order is: uniform scaling, rotation, translation.

```
static void CreateSRT(Vector3 scaling, Vector3 rotationOrigin, Quaternion rotation, Vector3 translation,
out Matrix4x4 result)
static void CreateSRT(ref Vector3 scaling, ref Vector3 rotationOrigin, ref Quaternion rotation, ref Vector3
translation, out Matrix4x4 result)
```
Creates a transformation matrix. Transformation order is: scaling, moving to rotation origin, rotation, moving to translation point.

```
static void CreateSRT(float scaling, Vector3 rotationOrigin, Quaternion rotation, Vector3 translation, out
Matrix4x4 result)
static void CreateSRT(float scaling, ref Vector3 rotationOrigin, ref Quaternion rotation, ref Vector3
translation, out Matrix4x4 result)
```
Creates a transformation matrix. Transformation order is: uniform scaling, moving to rotation origin, rotation, moving to translation point.

```
static void CreateRT(Quaternion rotation, Vector3 translation, out Matrix4x4 result)
static void CreateRT(ref Quaternion rotation, ref Vector3 translation, out Matrix4x4 result)
```
Creates a transformation matrix. Transformation order is: rotation, translation.

```
static void CreateRT(Vector3 rotationOrigin, Quaternion rotation, Vector3 translation, out Matrix4x4
result)
static void CreateRT(ref Vector3 rotationOrigin, ref Quaternion rotation, ref Vector3 translation, out
Matrix4x4 result)
```
Creates a transformation matrix. Transformation order is: moving to rotation origin, rotation, moving to translation point.

```
static void CreateST(Vector3 scaling, Vector3 translation, out Matrix4x4 result)
static void CreateST(ref Vector3 scaling, ref Vector3 translation, out Matrix4x4 result)
```
Creates a transformation matrix. Transformation includes scaling and translation (order is unimportant).

```
static void CreateRotationFromColumns(Vector3 column0, Vector3 column1, Vector3 column2, out Matrix4x4
matrix)
static void CreateRotationFromColumns(ref Vector3 column0, ref Vector3 column1, ref Vector3 column2, out
Matrix4x4 matrix)
```
Creates rotation matrix from 3 vectors (vectors are columns of the matrix)

```
static void CreateShadowDirectional(Plane3 shadowPlane, Vector3 dirLightOppositeDirection, out Matrix4x4
result)
static void CreateShadowDirectional(ref Plane3 shadowPlane, ref Vector3 dirLightOppositeDirection, out
Matrix4x4 result)
```
Creates directional light shadow matrix that flattens geometry into a plane.
shadowPlane - Projection plane
dirLightOppositeDirection - Light source is a directional light and parameter contains opposite direction of directional light (e.g. if light direction is L, caller must pass -L as a parameter)

```
static void CreateShadowPoint(Plane3 shadowPlane, Vector3 pointLightPosition, out Matrix4x4 result)
static void CreateShadowPoint(ref Plane3 shadowPlane, ref Vector3 pointLightPosition, out Matrix4x4 result)
```
Creates point light shadow matrix that flattens geometry into a plane.
shadowPlane - Projection plane
pointLightPosition - Light source is a point light and parameter contains position of a point light

```
static void CreateShadow(Plane3 shadowPlane, Vector4 lightData, out Matrix4x4 result)
static void CreateShadow(ref Plane3 shadowPlane, ref Vector4 lightData, out Matrix4x4 result)
```
Creates a generic shadow matrix that flattens geometry into a plane.
shadowPlane - Projection plane
lightData - If w component is 0.0f, then light source is directional light and x,y,z components contain opposite direction of directional light. If w component is 1.0f then source is point light and x,y,z components contain position of point light.

12.5 Mathfex

Type
class Mathfex
Fields
ZeroTolerance
NegativeZeroTolerance
ZeroToleranceSqr
Pi
HalfPi
TwoPi
Methods
EvalSquared(float x)
EvalInvSquared(float x)
EvalCubic(float x)
EvalInvCubic(float x)
EvalQuadratic(float x, float a, float b, float c)
EvalSigmoid(float x)
EvalOverlappedStep(float x, float overlap, int objectIndex, int objectCount)
EvalSmoothOverlappedStep(float x, float overlap, int objectIndex, int objectCount)
EvalGaussian(float x, float a, float b, float c)
EvalGaussian2D(float x, float y, float x0, float y0, float A, float a, float b, float c)
Lerp(float value0, float value1, float factor)
LerpUnclamped(float value0, float value1, float factor)
SigmoidInterp(float value0, float value1, float factor)
SinInterp(float value0, float value1, float factor)
CosInterp(float value0, float value1, float factor)
WobbleInterp(float value0, float value1, float factor)
CurveInterp(float value0, float value1, float factor, AnimationCurve curve)
FuncInterp(float value0, float value1, float factor, System.Func<float, float> func)
InvSqrt(float value)
Near(float value0, float value1, float epsilon)
NearZero(float value, float epsilon)
CartesianToPolar(Vector2 cartesianCoordinates)
PolarToCartesian(Vector2 polarCoordinates)
CartesianToSpherical(Vector3 cartesianCoordinates)
SphericalToCartesian(Vector3 sphericalCoordinates)
CartesianToCylindrical(Vector3 cartesianCoordinates)
CylindricalToCartesian(Vector3 cylindricalCoordinates)

const float ZeroTolerance
1e-5f

```
const float NegativeZeroTolerance
```
-1e-5f

```
const float ZeroToleranceSqr
```
(1e-5f)^2

```
const float Pi
```
π

```
const float HalfPi = 0.5f * Pi
```
$\pi/2$

```
const float TwoPi = 2f * Pi
```
2π

```
static float EvalSquared(float x)
```
Evaluates x^2

```
static float EvalInvSquared(float x)
```
Evaluates $x^{1/2}$

```
static float EvalCubic(float x)
```
Evaluates x^3

```
static float EvalInvCubic(float x)
```
Evaluates $x^{1/3}$

```
static float EvalQuadratic(float x, float a, float b, float c)
```
Evaluates quadratic equation $a*x^2 + b*x + c$

```
static float EvalSigmoid(float x)
```
Evaluates sigmoid function (used for smoothing values).
Formula: $x^2 * (3 - 2*x)$

```
static float EvalOverlappedStep(float x, float overlap, int objectIndex, int objectCount)
```
Evaluates overlapped step function. Useful for animating several objects (stepIndex parameter is number of the objects), where animations follow one after another with some overlapping in time (overlap parameter).
x - Evaluation parameter, makes sence in [0,1] range
overlap - Overlapping between animations (must be greater or equal to zero), where 0 means that animations do not overlap and follow one after another.
objectIndex - Index of object beeing animated
objectCount - Number of objects beeing animated

```
static float EvalSmoothOverlappedStep(float x, float overlap, int objectIndex, int objectCount)
```
Evaluates overlapped step function and applies sigmoid to smooth the result. Useful for animating several objects (stepIndex parameter is number of the objects), where animations follow one after another with some overlapping in time (overlap parameter).
x - Evaluation parameter, makes sence in [0,1] range
overlap - Overlapping between animations (must be greater or equal to zero), where 0 means that animations do not overlap and follow one after another.
objectIndex - Index of object beeing animated
objectCount - Number of objects beeing animated

```
static float EvalGaussian(float x, float a, float b, float c)
```
Evaluates scalar gaussian function. The formula is:
$a * e^{(-(x-b)^2 / 2*c^2)}$

x - Function parameter

```
static float EvalGaussian2D(float x, float y, float x0, float y0, float A, float a, float b, float c)
```
Evaluates 2-dimensional gaussian function. The formula is:
$A * e^{(-(a*(x - x0)^2 + 2*b*(x - x0)*(y - y0) + c*(y - y0)^2))}$
x - First function parameter
y - Second function parameter

```
static float Lerp(float value0, float value1, float factor)
```
Linearly interpolates between 'value0' and 'value1'.
factor - Interpolation factor in range [0..1] (will be clamped)

```
static float LerpUnclamped(float value0, float value1, float factor)
```
Linearly interpolates between 'value0' and 'value1'.
factor - Interpolation factor in range [0..1] (will NOT be clamped, i.e. interpolation can overshoot)

```
static float SigmoidInterp(float value0, float value1, float factor)
```
Interpolates between 'value0' and 'value1' using sigmoid as interpolation function.
factor - Interpolation factor in range [0..1] (will be clamped)

```
static float SinInterp(float value0, float value1, float factor)
```
Interpolates between 'value0' and 'value1' using sine function easing at the end.
factor - Interpolation factor in range [0..1] (will be clamped)

```
static float CosInterp(float value0, float value1, float factor)
```
Interpolates between 'value0' and 'value1' using cosine function easing in the start.
factor - Interpolation factor in range [0..1] (will be clamped)

```
static float WobbleInterp(float value0, float value1, float factor)
```
Interpolates between 'value0' and 'value1' in using special function which overshoots first, then waves back and forth gradually declining towards the end.
factor - Interpolation factor in range [0..1] (will be clamped)

```
static float CurveInterp(float value0, float value1, float factor, AnimationCurve curve)
```
Interpolates between 'value0' and 'value1' using provided animation curve (curve will be sampled in [0..1] range]).
factor - Interpolation factor in range [0..1] (will be clamped)

```
static float FuncInterp(float value0, float value1, float factor, System.Func<float, float> func)
```
Interpolates between 'value0' and 'value1' using provided function (function will be sampled in [0..1] range]).
factor - Interpolation factor in range [0..1] (will be clamped)

```
static float InvSqrt(float value)
```
Returns 1/Sqrt(value) if value != 0, otherwise returns 0.

```
static bool Near(float value0, float value1, float epsilon = Mathfex.ZeroTolerance)
```
Returns abs(v0-v1)<eps

```
static bool NearZero(float value, float epsilon = Mathfex.ZeroTolerance)
```
Returns abs(v)<eps

`static Vector2 CartesianToPolar(Vector2 cartesianCoordinates)`
Converts cartesian coordinates to polar coordinates. Resulting vector contains rho (length) in x coordinate and phi (angle) in y coordinate; rho >= 0, 0 <= phi < 2pi. If cartesian coordinates are (0,0) resulting coordinates are (0,0).

`static Vector2 PolarToCartesian(Vector2 polarCoordinates)`
Converts polar coordinates to cartesian coordinates. Input vector contains rho (length) in x coordinate and phi (angle) in y coordinate; rho >= 0, 0 <= phi < 2pi.

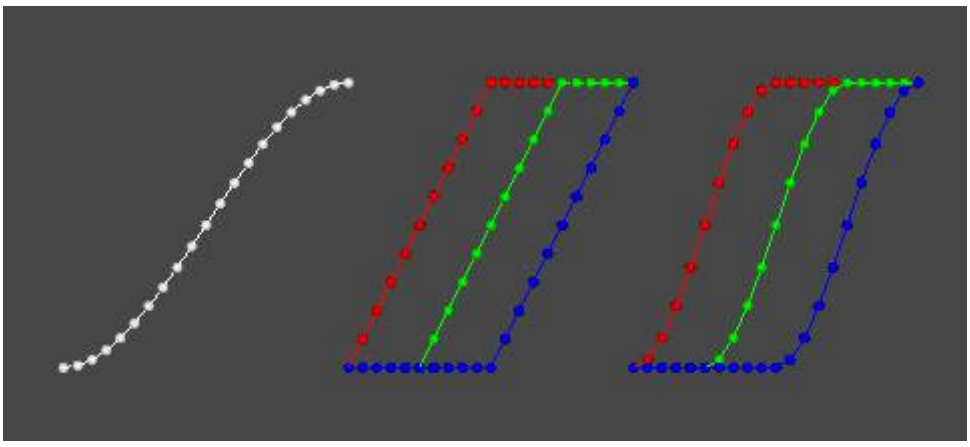
`static Vector3 CartesianToSpherical(Vector3 cartesianCoordinates)`
Converts cartesian coordinates to spherical coordinates. Resulting vector contains rho (length) in x coordinate, theta (azimutal angle in XZ plane from X axis) in y coordinate, phi (zenith angle from positive Y axis) in z coordinate; rho >= 0, 0 <= theta < 2pi, 0 <= phi < pi. If cartesian coordinates are (0,0,0) resulting coordinates are (0,0,0).

`static Vector3 SphericalToCartesian(Vector3 sphericalCoordinates)`
Converts spherical coordinates to cartesian coordinates. Input vector contains rho (length) in x coordinate, theta (azimutal angle in XZ plane from X axis) in y coordinate, phi (zenith angle from positive Y axis) in z coordinate; rho >= 0, 0 <= theta < 2pi, 0 <= phi < pi.

`static Vector3 CartesianToCylindrical(Vector3 cartesianCoordinates)`
Converts cartesian coordinates to cylindrical coordinates. Resulting vector contains rho (length) in x coordinate, phi (polar angle in XZ plane) in y coordinate, height (height from XZ plane to the point) in z coordinate.

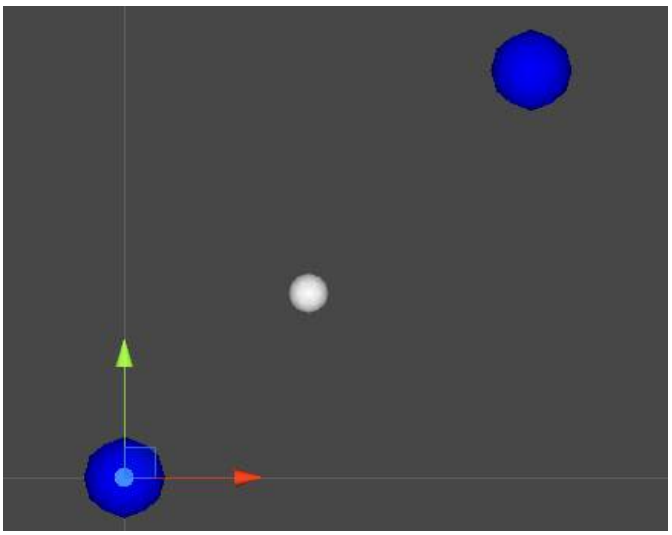
`static Vector3 CylindricalToCartesian(Vector3 cylindricalCoordinates)`
Converts cylindrical coordinates to cartesian coordinates. Input vector contains rho (length) in x coordinate, phi (polar angle in XZ plane) in y coordinate, height (height from XZ plane to the point) in z coordinate.

Among these methods there are several of special interest. They are `EvalSigmoid()`, `EvalOverlappedStep()`, `EvalSmoothOverlappedStep()`. Sigmod is very handy S-shaped curve, which can be used for smooth transitions between values. The behaviour of the curve when it's used for interpolation is that it goes slow at the ends. It can help to fade in or out elements of the GUI or smoothly move objects. Among x, x^2, $x^{1/2}$, x^3, $x^{1/3}$, sigmoid is very useful tweening function. Eval***step methods allow to stack animations not just one after another but with some overlapping in time (see test prefab for an example of fading in/out several spheres on the screen). Smooth step is just combination of step and sigmod functions.



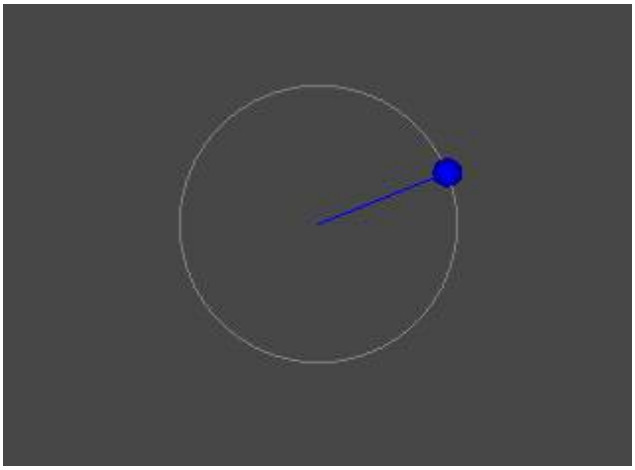
Test Prefab:
Test_MathfEx
Test_OverlappedStep

Image of the first test prefab. Left to right: sigmoid, overlapped step, smooth overlapped step. Second test prefab must be launched. It shows various tweenings in action using overlapped steps. User can choose tweening types from the inspector changing “Left Type” which is type for fading in, and “Right Type” which is type for fading out.



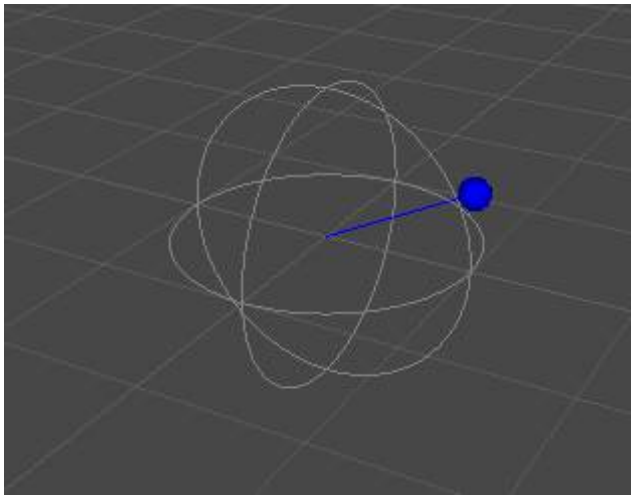
Test Prefab:
Test_Interpolation

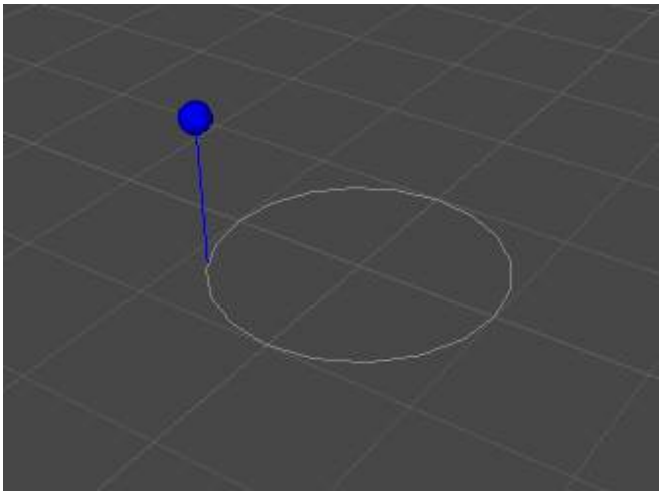
Shows different interpolation methods. Press play and then switch back to the scene window to see the result. Various interpolation methods can be selected in the inspector. Using selected method prefab will interpolate between two points (they can also be changed) using speed and time segment.



Test Prefab:
Test_PolarCoords
Test_SphericalCoords
Test_CylindricalCoords

There are 3 test prefabs showing how to convert coordinates between coordinate systems.





12.6 Pseudo Random Numbers Generator

Games often need random number to vary gameplay or create a level. Pseudo random numbers (those which are produced algoritmically rather than using special hardware which produces real random numbers) can be required in many places of the program. As pseudo random number generators (PRNG) are algorithms they require some initial values from which they generate further numbers. This intial value is called *seed*.

Unity already contains PRNG. The problem with Unity generator is that users can not create different generators with different seeds and this is often required. So there is only single stream of random numbers. Theoretically this can be fixed by using System.Random class from the Mono library, however algorithm which generates random numbers is far from being good quality generator. And instances of System.Random class which are created one after another will sometimes have same seed as their seed is taken from the current time (this is common problem with standard C# randomizer).

To overcome the problems the library contains its own PRNG class `Rand`. It is the implementation of famous Xorshift128 algorithm which is one of the fastest algorithms available and gives very good results. Users can create different streams by creating different instances of the class. By default seed is randomized, thus it's possible to create several randomizers one after another. Also the class contains numerous additional handy methods which default Unity randomizer doesn't have.

The class is released in source code. Test prefabs are described after the class description.

Type
`class Rand`
Fields
`static Rand Instance`
Construction
`Rand()`
`Rand(int seed)`
Methods
`ResetSeed(int seed)`
`GetState(out uint x, out uint y, out uint z, out uint w)`
`SetState(uint x, uint y, uint z, uint w)`
`NextInt()`
`NextInt(int max)`
`NextInt(int min, int max)`
`NextIntInclusive(int min, int max)`
`NextUInt()`
`NextDouble()`
`NextDouble(double min, double max)`
`NextFloat()`
`NextFloat(float min, float max)`
`NextBool()`
`NextByte()`

RandomColorOpaque()
RandomColorTransparent()
RandomColor32Opaque()
RandomColor32Transparent()
RandomAngleRadians()
RandomAngleDegrees()
InSquare(float side)
OnSquare(float side)
InCube(float side)
OnCube(float side)
InCircle(float radius)
InCircle(float radiusMin, float radiusMax)
OnCircle(float radius)
InSphere(float radius)
OnSphere(float radius)
InTriangle(ref Vector3 v0, ref Vector3 v1, ref Vector3 v2)
InTriangle(Vector3 v0, Vector3 v1, Vector3 v2)
RandomRotation()

`static Rand Instance`
Default stream of random numbers. Use it if you don't want to create your own streams.

`Rand()`
Creates random number generator using randomized seed.

`Rand(int seed)`
Creates random number generator using specified seed.

`void ResetSeed(int seed)`
Resets generator using specified seed.

`void GetState(out uint x, out uint y, out uint z, out uint w)`
Gets generator inner state represented by four uints. Can be used for generator serialization.

`void SetState(uint x, uint y, uint z, uint w)`
Sets generator inner state from four uints. Can be used for generator deserialization.

`int NextInt()`
Generates a random integer in the range [int.MinValue,int.MaxValue].

`int NextInt(int max)`
Generates a random integer in the range [0,max)

`int NextInt(int min, int max)`
Generates a random integer in the range [min,max). max must be >= min.

`int NextIntInclusive(int min, int max)`
Generates a random integer in the range [min,max]. max must be >= min.
The method simply calls NextInt(min,max+1), thus largest allowable value for max is int.MaxValue-1.

`int NextPositiveInt()`
Generates a random integer in the range [0,int.MaxValue].

`uint NextUInt()`
Generates a random unsigned integer in the range [0,uint.MaxValue].

`double NextDouble()`
Generates a random double in the range [0,1).

`double NextDouble(double min, double max)`
Generates a random double in the range [min,max).

`float NextFloat()`
Generates a random float in the range [0,1).

`float NextFloat(float min, float max)`
Generates a random float in the range [min,max).

`bool NextBool()`
Generates a random bool.

`byte NextByte()`
Generates a random byte.

`Color RandomColorOpaque()`
Generates a random opaque color.

`Color RandomColorTransparent()`
Generates a random color with randomized alpha.

`Color32 RandomColor32Opaque()`
Generates a random opaque color.

`Color32 RandomColor32Transparent()`
Generates a random color with randomized alpha.

`float RandomAngleRadians()`
Generates a random angle [0,2*pi)

`float RandomAngleDegrees()`
Generates a random angle [0,360)

`Vector2 InSquare(float side = 1f)`
Generates a random point inside the square with specified side size.

`Vector2 OnSquare(float side = 1f)`
Generates a random point on the border of the square with specified side size.

`Vector3 InCube(float side = 1f)`
Generates a random point inside the cube with specified side size.

```
Vector3 OnCube(float side = 1f)
```
Generates a random point on the surface of the cube with specified side size.

```
Vector2 InCircle(float radius = 1f)
```
Generates a random point inside the circle with specified radius.

```
Vector2 InCircle(float radiusMin, float radiusMax)
```
Generates a random point inside the ring with specified radia.

```
Vector2 OnCircle(float radius = 1f)
```
Generates a random point on the border of the circle with specified radius.

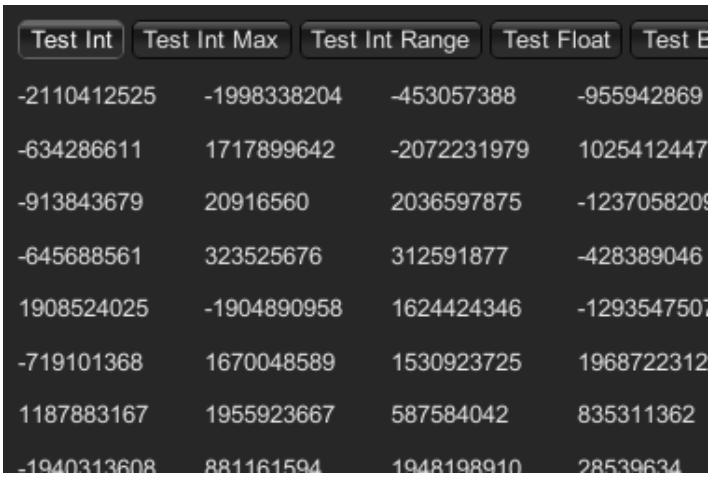
```
Vector3 InSphere(float radius = 1f)
```
Generates a random point inside the sphere with specified radius.

```
Vector3 OnSphere(float radius = 1f)
```
Generates a random point on the surface of the sphere with specified radius.

```
Vector3 InTriangle(ref Vector3 v0, ref Vector3 v1, ref Vector3 v2)
Vector3 InTriangle(Vector3 v0, Vector3 v1, Vector3 v2)
```
Generates a random point inside the triangle.

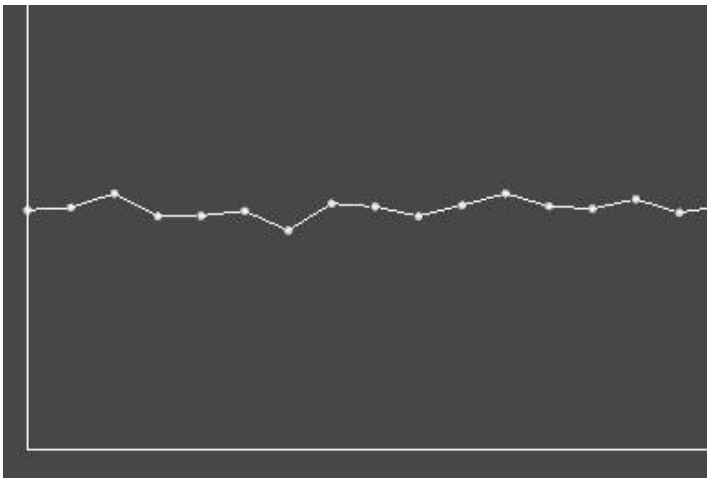
```
Quaternion RandomRotation()
```
Generates a random rotation.

There are many test prefabs demonstrating Rand class in action.



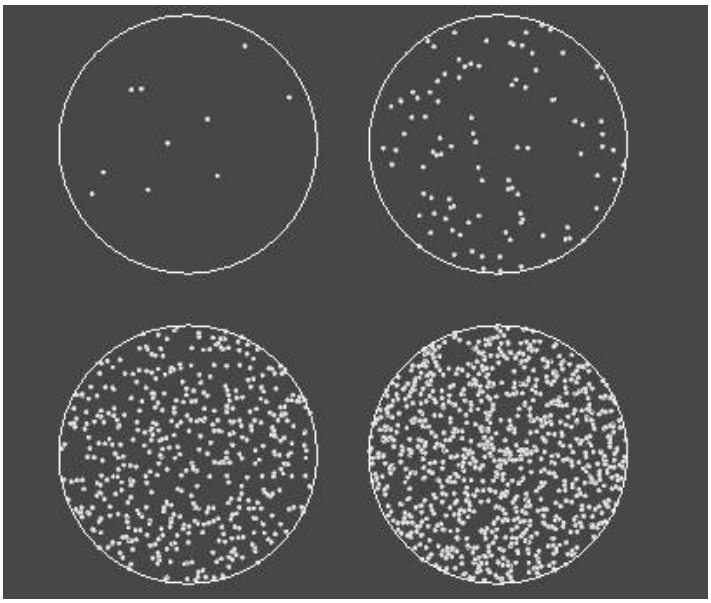
Test Prefab:
Test_Random

Simply shows number generation. Users can toggle to creating numbers with standard Unity generator to ensure the correctness.



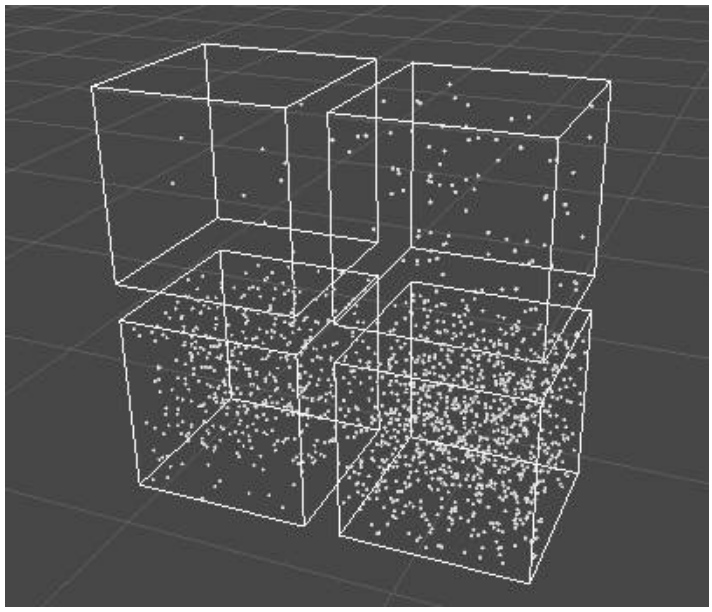
Test Prefab:
Test_RandomDistribution

Generates integer numbers in some range and plots the graph of occurrence of generated numbers. If number of generated growth the distribution becomes a line.



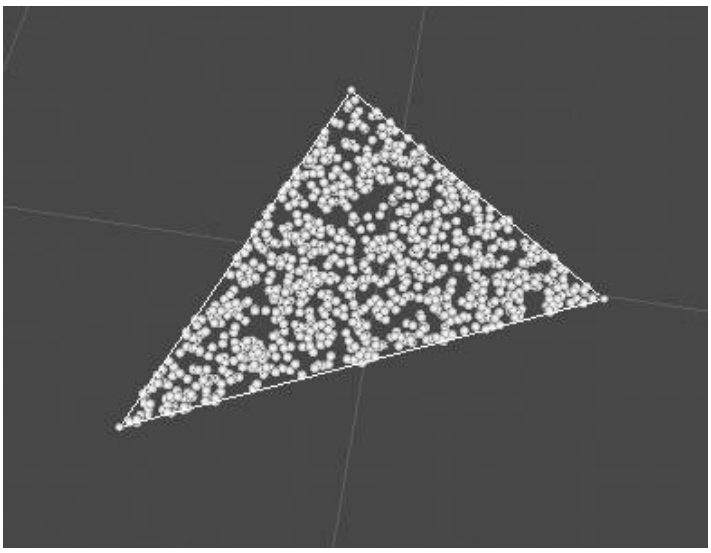
Test Prefab:
Test_Random2D

Shows examples of 2D Rand methods – in/on circle, in/on square and in ring. In each case 4 sets are generated with different number of samples.



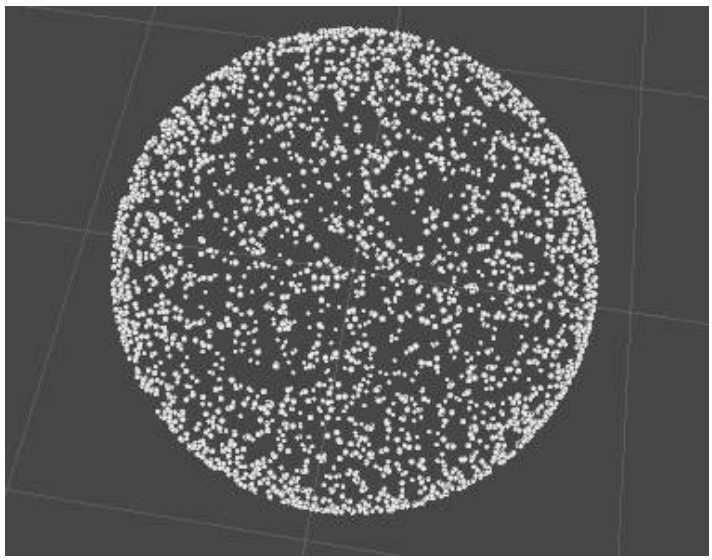
Test Prefab:
Test_Random3D

Shows examples of 3D Rand methods – in/on sphere, in/on cube. In each case 4 sets are generated with different number of samples.



Test Prefab:
Test_RandomInTriangle

Shows random picking inside the triangle (Rand.InTriangle method).



Test Prefab:
Test_RandomRotation

Shows uniform random rotation generation (Rand.RandomRotation method).

12.7 Random Samplers

Random uniform sampling of some sets is already presented in `Rand` class in previous section (e.g. sampling of cube or sphere). Sometimes more complex sampling is required where some information must be precalculated in order for sampling to be performant. This section describes several such samplers. All the classes are released in source code.

12.7.1 Weighted Sampler

Imagine the case when a player kills a mob and then loots it. Accodring to mob loot table there is 50% chance of getting a banana, 30% of getting an apple and 20% of getting an orange. This can be repsetented as an array { 0.5, 0.3, 0.2 } of weights. Using random generator game should pick one of the fruits using weights provided by the game designer. `WeightedSampler` class allows to do exactly that.

Users feed an array of weights to the constructor and then can query random index (which is integer number from 0 to weights array length minus 1). Notice that the class does not require weights to sum up to 1 because weight normalization is applied. See next section about shuffle bags for slightly different approach to this problem.

Type
`class WeightedSampler`
Construction
`WeightedSampler(float[] weights)` Creates sampler instance from specified weights of the data collection and using Rand.Instance randomizer.
`WeightedSampler(float[] weights, Rand rand)` Creates sampler instance from specified weights of the data collection and specified randomizer.
Methods
`int SampleIndex()` Returns random index which can be used to access data collection.

12.7.2 Indexed and Non-Indexed Triangle Set Sampler

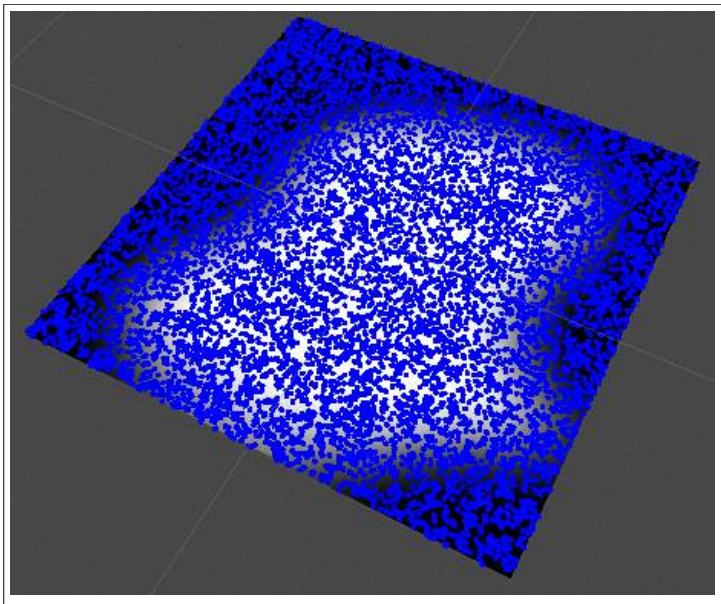
Another interesting sampling can be done by uniformly sampling a mesh or speaking generally – sampling a set of triangles. The library provides two classes which can sample sets of triangles – for indexed triangle sets and for non-indexed.

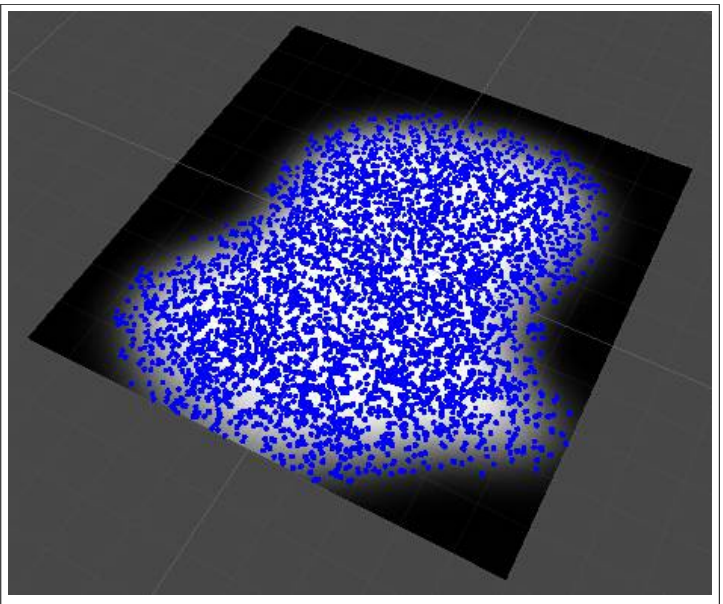
Type
`class IndexedTrianglesSampler : TrianglesSamplerBase` Sampler which uniformly samples from a surface defined by the set of indexed triangles.
Construction
`IndexedTrianglesSampler(Vector3[] vertices, int[] indices)` Creates sampler instance using specified vertices and indices and using Rand.Instance randomizer.
`IndexedTrianglesSampler(Vector3[] vertices, int[] indices, Rand rand)` Creates sampler instance using specified vertices and indices and specified randomizer.
`IndexedTrianglesSampler(Mesh mesh)` Creates sampler instance using mesh.vertices and mesh.GetIndices(0) and using Rand.Instance randomizer.
`IndexedTrianglesSampler(Mesh mesh, Rand rand)` Creates sampler instance using mesh.vertices and mesh.GetIndices(0) and specified randomizer.

Methods
`Vector3 Sample()` Samples a random point on the surface.
`Vector3[] SampleArray(int count)` Samples an array of random point on the surface.
`Vector3[] SampleArray(int count, Vector2[] uvs, Texture2D sampleMap, float min = 0f, float max = 1f)` Samples an array of random points on the surface. Users can provide additional parameters for sampling. Sample map controls the chance of sampling (texture is read from r channel), where white gives maximum chance and black gives no chance. To read the texture an array of uv coordinates is used. It's also possible to explicitly set additional range of chances (so that if sample map has gradient from 0 to 1, then adding min to 0.5 will result in no samples generated in areas with color <= 0.5). Notice that in this function count is upper bound of samples to generate, resulting array may have less samples due to some samples being rejected according to sample map.

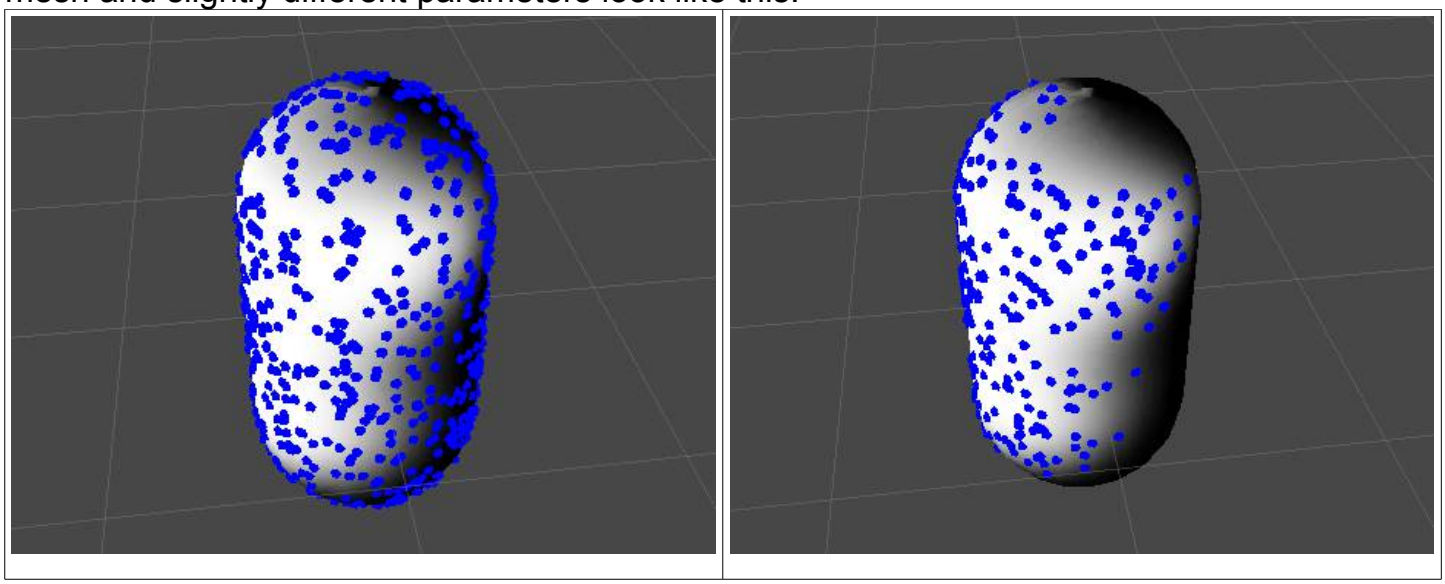
Type
`class NonIndexedTrianglesSampler : TrianglesSamplerBase` Sampler which uniformly samples from a surface defined by the set of non-indexed triangles.
Construction
`NonIndexedTrianglesSampler(Vector3[] vertices)` Creates sampler instance using specified vertices and using Rand.Instance randomizer.
`NonIndexedTrianglesSampler(Vector3[] vertices, Rand rand)` Creates sampler instance using specified vertices and specified randomizer.
`NonIndexedTrianglesSampler(Mesh mesh)` Creates sampler instance using mesh.vertices and using Rand.Instance randomizer.
`NonIndexedTrianglesSampler(Mesh mesh, Rand rand)` Creates sampler instance using mesh.vertices and specified randomizer.
Methods
`Vector3 Sample()` Samples a random point on the surface.
`Vector3[] SampleArray(int count)` Samples an array of random point on the surface.

Test Prefab:
Test_RandomOnMesh





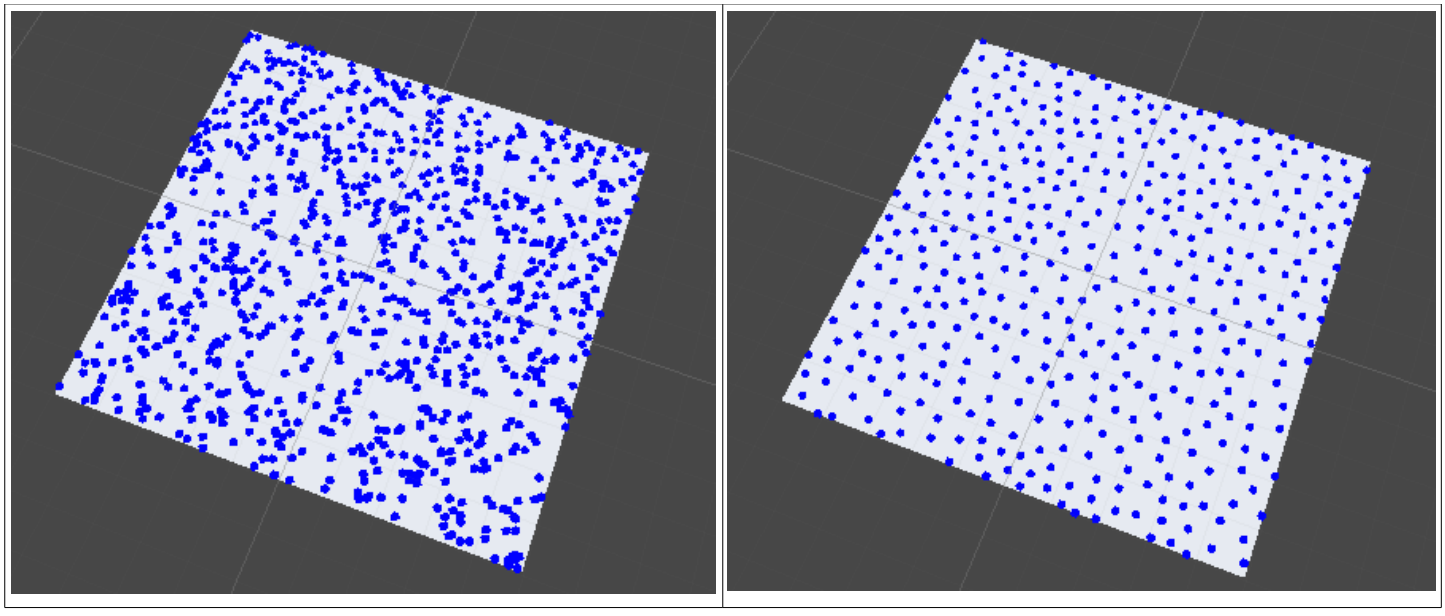
On the images above indexed sampler is used to sample plane mesh (users can choose any other mesh by setting the mesh in MeshFilter component of the test prefab). It also uses min/max parameters for the SampleArray method. For example same prefab with cylinder mesh and slightly different parameters look like this.



In both cases left images show uniform sampling from the whole mesh, while right images use sample map to control areas of sampling (this can be switched in the test prefab by toggling UseSpawnMap parameter in the inspector panel).

12.7.3 Poisson Disk Sampler

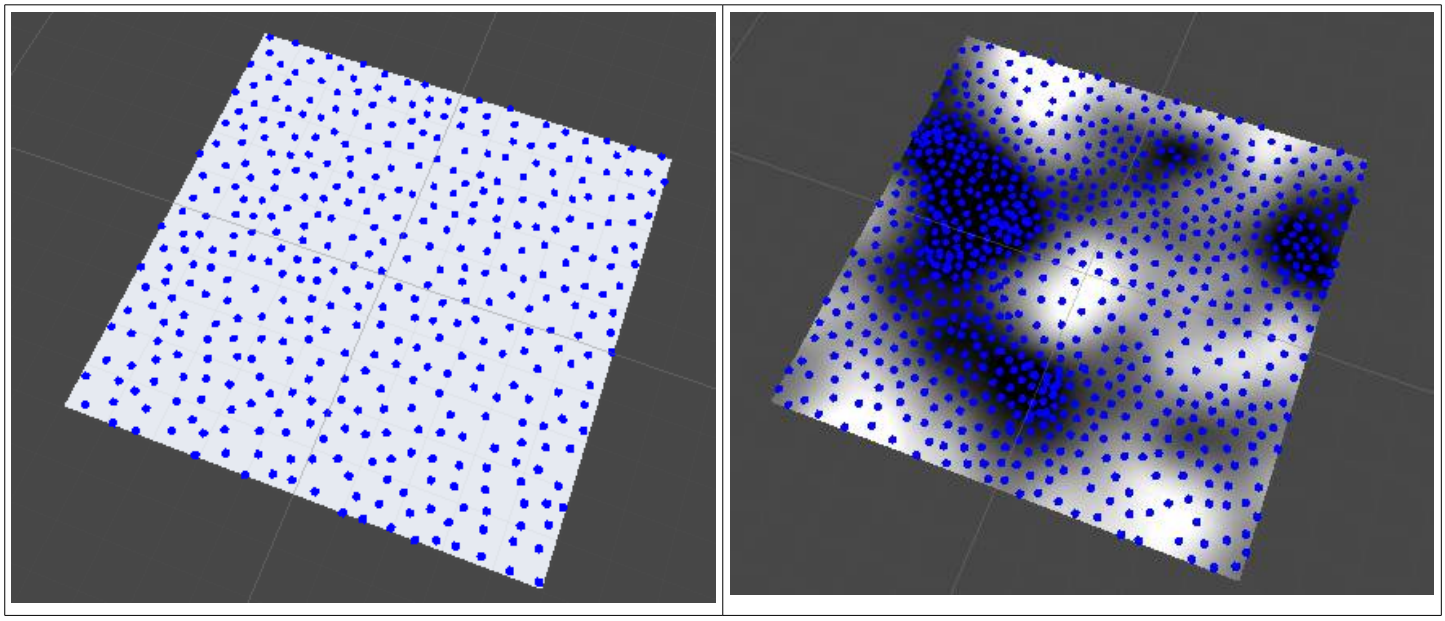
Random sampling of the domain gives somewhat chaotic results (this is what *random* should do). Sometimes it's required that sampling is not purely random, but depends on some parameters. One such case is Poisson disk sampling. It allows to generate samples insides the rectangle area (although algorithm could be generalized to higher dimensions, e.g. instance of rectangle the domain can be box in 3D). Main advantage of the algorithm is that it specifies minimum distance between samples. Resulting set of samples while still random in some sence look much less chaotic. Compare two images below – on the left purely random sampling is used, while on the right Poisson disk sampling is used.



Minimum distance between the samples is the main input parameter of the algorithm. Users can also specify maximum amount of samples generated (this can also prevent hanging if wrong input parameters are used).

Test Prefab:
Test_PoissonDiskSampler

Yet another parameter can control distance between the samples via texture thus it's possible to create clusters. Again compare two images: on the left simple Poisson disk sampling is used, on the right sampling is altered using Perlin noise texture.



This can be controlled by toggling UseDistanceMap parameter in the inspector and by specifying outer/inner distances (if distance map is not used then distance between samples is greater or equal to DistOuter, if distance map is used then distance between samples changes linearly interpolating between DistInner and DistOuter using value from r channel of the texture as lerp parameter). See test prefab script for the example of code.

One problem with Poisson disk sampling is that authors of the paper describing how to create the sampling only shown generation in the rectangular domain. This severly limits algorithm applicability as game developers often need arbitrary domains such as meshes (although it is possible to discard some samples after they have been generated, e.g. only keep samples according to some texture map, generated samples will still be in rectangular domain). This leads us to the next sub-section.

12.7.4 Point Set Filtering

To overcome the problems described in the previous sub-section, library contains the algorithm to filter arbitrary point set so the distance between samples is not less than some threshold. The algorithm is as follows: generate 3D point set from any source (e.g. randomly sampling mesh surface), filter out points of the set such that no two points are closer than specified threshold.

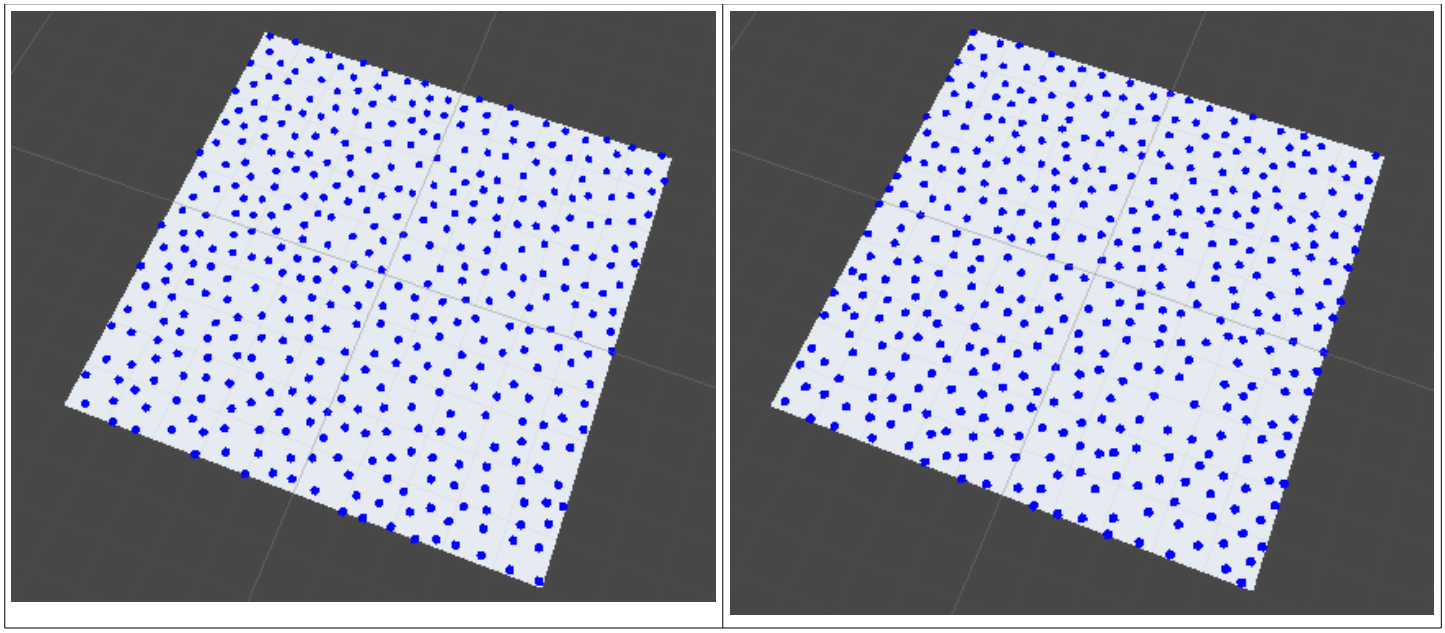
While this technique somewhat bruteforce and gives less accurate results than real Poisson disk sampling it allows to modify arbitrary point sets which is the goal. It's bruteforce because the input point set should contain really large number of samples otherwise the result will be heavily filtered and very small amount of samples will remain. Same test prefab

as for sampling the mesh is used to show distance filtering. To use the filtering, turn on DistanceFiltering in the inspector and set Distance parameter.

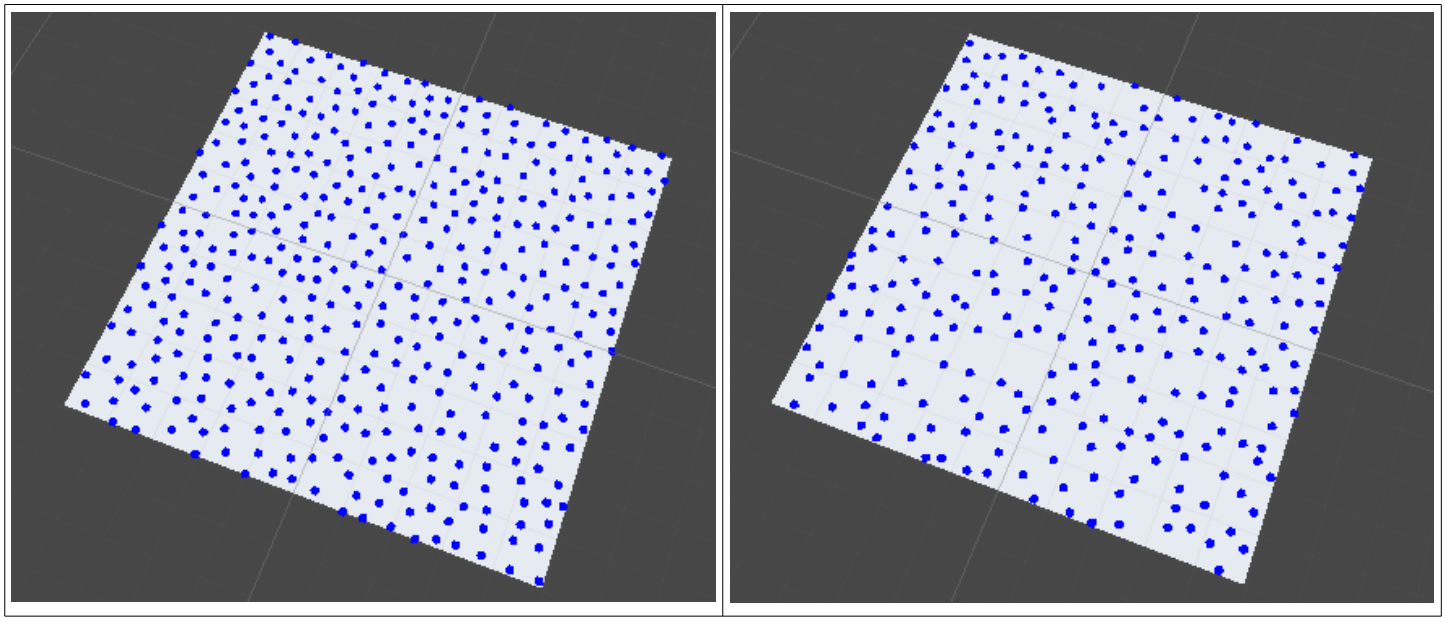
Test Prefab:

Test_RandomOnMesh

Look at two images conmparing real Poisson disk sampling (left) and fake sampling using technique from this sub-section (right).



While fake technique gives rather good results it has costs. Poisson disk sampling generates points using clever algorithm so that rejection rate is low and the process is fast. Filtering requires large input set beforehand, e.g. the produced image was created by first generating 5000 samples and then filtering was applied leaving only 417 points (while by carefully changing number of input points and distance parameter this can be optimized, still rejection rate is very high). If, for example, input point set is reduced to 1000 points, output point set can be reduced to 297 points (see images below, on the left Poisson sampling, on the right filtering from 1000 points).



Clearly image on the right is undersampled comparing to the image on the left. To use high amount of input samples authors heavily optimized filtering procedure so that input samples with several dozens of thousands still can be used almost without visible lag.

Random sampling, Poisson disk sampling and point filtering can be used by level tool programmers as it allows to create some randomization without efforts (e.g. planting bushes on the hills).

12.8 ShuffleBag

Library includes implementation of the ShuffleBag collection which can be used for generating endless random sequences of weighted values. For example, if the character in the game should do double damage once during 5 hits, developer can put 4 usual hits in the bag and one critical hit. Using shuffle bags instead of using random generators sometimes preferable as it allows game designers to control user experience more thoroughly (using pure random numbers may often frustrate players when really bad random sequences are generated, e.g. no ciritical hits during 10 hits, while crit chance is 25%). The class is released in source code.

Type
`class ShuffleBag<T> : IEnumerable<T>`
Properties
`int Count { get; }` Returns collection count.
Construction
`ShuffleBag()` Creates new instance of ShuffleBag with Rand.Instance randomizer and zero capacity of the underlying collection.
`ShuffleBag(int capacity)` Creates new instance of ShuffleBag with Rand.Instance randomizer and specified capacity of the underlying collection.
`ShuffleBag(Rand rand)` Creates new instance of ShuffleBag with specified randomizer and zero capacity of the underlying collection.
`ShuffleBag(Rand rand, int capacity)` Creates new instance of ShuffleBag with specified randomizer and specified capacity of the underlying collection.
Methods
`void Add(T item, uint count = 1)` Adds the item to the bag with specified number of entries.
`T NextItem()` Draws an item out of the bag.
`void Reset()` Resets bag traversal.
`void Clear()` Removes all items from the bag.
`IEnumerator<T> GetEnumerator()`
`IEnumerator IEnumerable.GetEnumerator()`

12.9 Util

Util class contains various useful methods which shouldn't be put into separate class. The class is released in source code.

Type
`static class Util`
Methods

`static void Shuffle<T>(this IList<T> collection)`
`static void Shuffle<T>(this T[] collection)`
Shuffles the collection using Rand.Instance.

`static void Shuffle<T>(this IList<T> collection, Rand rand)`
`static void Shuffle<T>(this T[] collection, Rand rand)`
Shuffles the collection using specified random generator.

12.10 Logger

On rare occasion some algorithms will encounter unexpected conditions and mus be aborted. In this case they may print out an error. As printing errors directly into the console using `Debug` class maybe undesirable for the user, special logging wrapper is used instead.

Type
`class Logger`
Methods
`static void LogInfo(object value)`
`static void LogWarning(object value)`
`static void LogError(object value)`
`static void SetLogger(ILogger logger)`

Type
`interface ILogger`
Methods
`void LogInfo(object value)`
`void LogWarning(object value)`
`void LogError(object value)`

Thus users can set their own loggers using `SetLogger` method. The parameter is the instance of the class derived from `ILogger` interface. There are two built-in implementations – `DefaultLogger` (which just prints into Unity console using `UnityEngine.Debug` methods) and `EmptyLogger` which does not print anything. Therefore if one wants to disable error reporting this could be done like this: `Logger.SetLogger(new EmptyLogger())`. By default Logger is using instance of the DefaultLogger class.

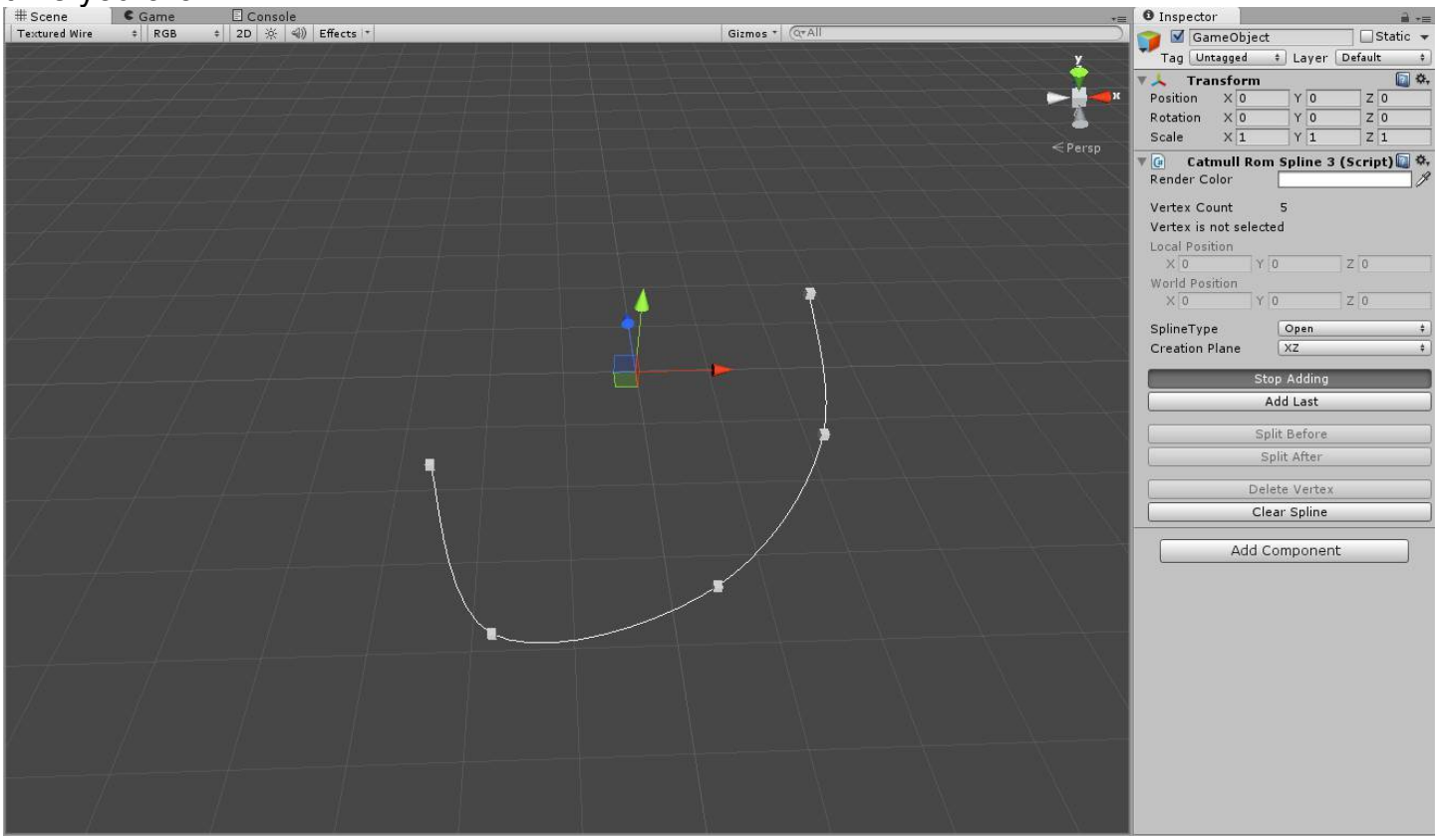
13 Tutorials

This chapter contains descriptions and links to the video tutorials which will help users of the library to get started with various parts of the library.

13.1 Working with Catmull-Rom Splines

In this tutorial we will look at common workflow when using Catmull-Rom splines. To create the spline from the editor add CatmullRomSpline3 component to the gameobject. Inspector will show several operations for spline construction.

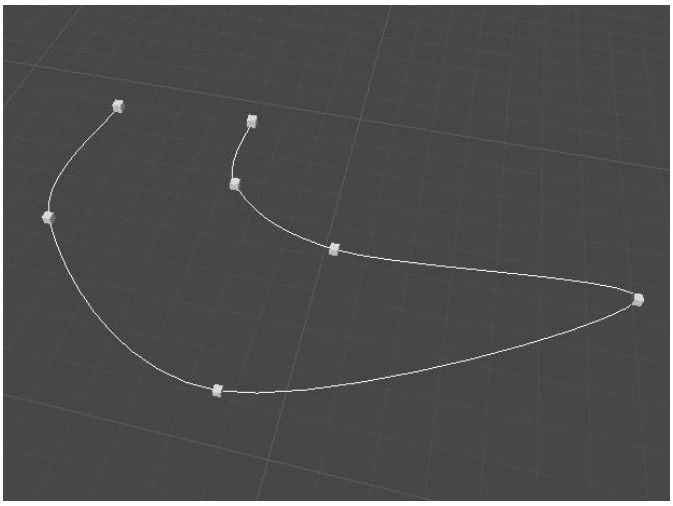
The easiest way to begin constructing the spline is to press “Add First” or “Add Last” button and then start clicking in the scene window. Vertices will be added to the spline every time you click.

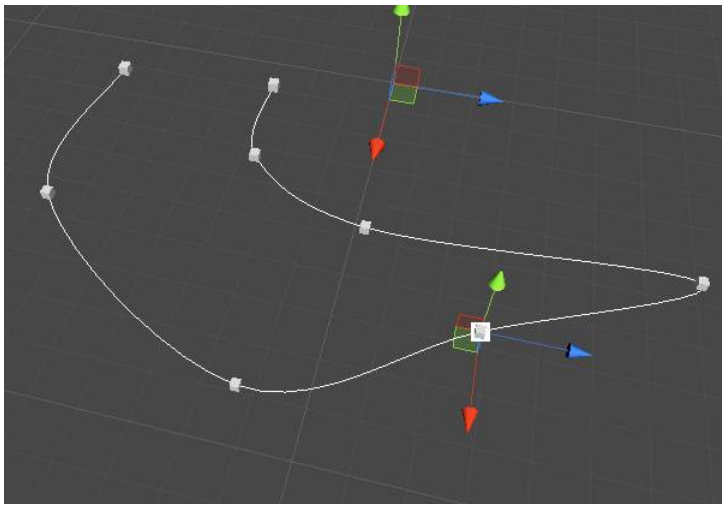


When you are happy with current spline form, turn off the button or simply right-click in the scene window.

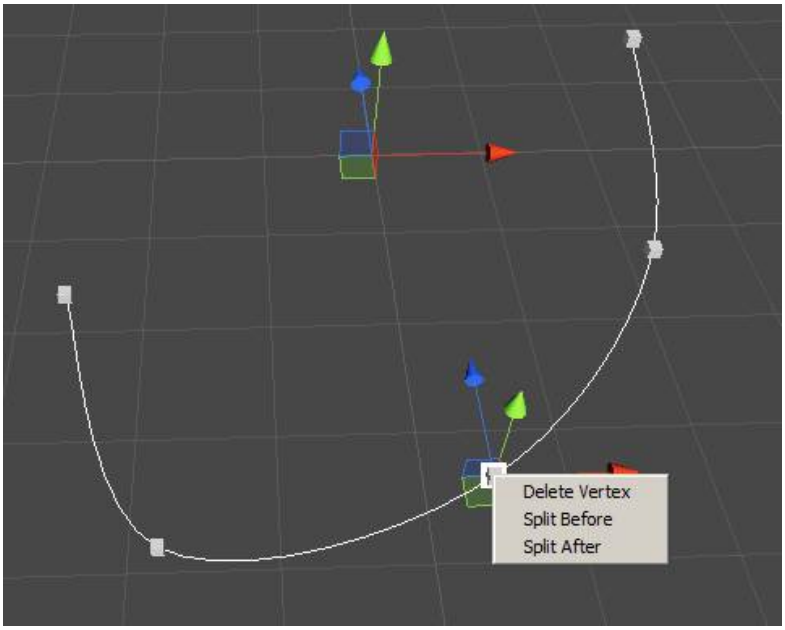
Now you can alter the spline by changing vertex positions or refining the spline by splitting it. To select the vertex just click on the small box. Selected vertices are shown in the Inspector panel. You can set exact positions by typing in numbers into vector fields. Vertices are affected by the transform of the parent spline object. You can set either local or world position in the Inspector.

If you need to insert vertices in the middle of the spline you can do that easily with “Split Before” or “Split After” buttons. Select the vertex first. Then if you click “Split After” the new vertex will be added after the selected vertex in the middle of the sub-interval. Similarly, “Split Before” adds the new vertex before the selected vertex.





You can also delete selected vertex by clicking on the button (as well as clearing whole spline). For convenience, three deleting and splitting operations has been added to the context menu. If you right-click on the vertex the menu will appear.



When you've finished constructing the spline you can save the gameobject to the prefab for later instantiation.

Splines could also be open or closed. Just select required type from the inspector.

You can create and manupulate spline from the code. Spline editor class just calls into the spline class, so you can write your own editor or simply call spline methods from the code in the similar manner spline editor class does.