GIMME
DOTS Geometry

Manual

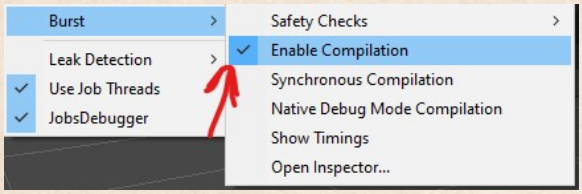
Contents

1 Setup, Dependencies and Support

In order for the package to work, the following dependencies need to be installed either manually or via the package manager:

- **Burst 1.8.0** or above (might work with lower versions; untested)
- **Collections 1.3.0** or above
- **Mathematics 1.2.0** or above (might work with lower versions; untested)

Additionally, depending on how you use the code, you might want to enable /unsafe code in the **Project Settings→Player**. Also don't forget to enable burst compilation in the Editor for performance:



Most features described in this manual are used in the sample scenes. Looking at the code of them may be the fastest way of understanding the package.

1.1 Extensions and Interoperability

The following packages, that can be found on the Asset Store, extend the functionality of Gimme DOTS Geometry:

- **Gimme GPU Geometry**

Gimme DOTS Geometry is interoperable with the following packages:

- **Entities 1.0 or above**

1.1.1 Entities / ECS

Most algorithms and data structures can be used inside an **ISystem** of Unity's Entity Package. You can find ***additional*** example scenes on how one might integrate DOTS Geometry within the package in:

`ECS_Samples.unitypackage`

In order for those ***additional*** sample scenes to work, the following items are required:

- Unity 2022.3 or above
- URP 14 or above
- Entities 1.0 or above
- Enities Graphics 1.0 or above

In general, integrating the data structures into an **ISystem** is relatively straightforward. For generic trees however, it might be necessary to tell Burst to compile the jobs for specific types in advance. This is done by registering them within an assembly:

```
[assembly: RegisterGenericJobType(typeof(GimmeDOTSGeometry
.NativeSparseQuadtree<int>.GetCellsInRadiusJob))]

[assembly: RegisterGenericJobType(typeof(GimmeDOTSGeometry
.Native2DRStarTree<MyCustomComponent>.GetRectanglesInPolygonJob))]
```

All jobs are public members of the data structures (for exactly this reason). You can find more information on why this is necessary here: Generic Jobs (the newer documentations do not include this page).

2 Settings

Safety Checks The number of safety checks that will be made depends on this settings, with **strict** being the highest safety setting. The idea is to keep the mode to **strict** in production and turn it to **normal** for a release build.

Degeneracy Handling Should be kept to **safe**. Only change to **unsafe** if you have a solid understanding of the algorithms in use and you can guarantee that your data is free of degenerate cases. If so, the performance might be further improved by removing the handling of them.

A **degenerate case** in geometry is a special case of a shape or its position (e.g. a triangle where all three points lie on a line, a circle with radius 0, etc.)

3 Lexicon

Quick Reference for some of the words used in this document in case you are unfamiliar with them. I will try to maintain and expand this list.

Collinear Three or more points are collinear if they lie on a single straight line.

Convex A boundary (for example a polygon) is convex if from every point inside it, all other points making up the boundary are visible (you can connect them with a straight line without ever intersecting the boundary)

Degenerate Case A special case of the geometric structures encountered during the execution of an algorithm that usually occurs in some form of limit. Examples are triangles where the points are **collinear**, a zero-length line segment, a circle with an infinite radius, etc.

Dynamic Data Structures in computational geometry that act on a changing set of data are **dynamic data structures**. In video game development both types of problems (dynamic and non-dynamic / static) occur, which is the reason why different data structures are often necessary.

Euclidean Distance It is the most common or "regular" way of measuring the distance between two points in space. For 2D, in mathematics, it looks like so: $\sqrt{(a_x - b_x)^2 + (a_y - b_y)^2}$. In Unity Code it looks like this: **Vector2.Distance(a, b)**. However, there are other metrics (like for example the distances on a grid)

Manhatten Distance Also called taxicab distance or city block distance. As the names suggest, it is a measurement of distances in a "grid", a space where you can travel only along the cardinal axes and not along any diagonal. In 2D, in mathematics, it looks like so: $|a_x - b_x| + |a_y - b_y|$. In Unity code it would be expressed like this: **Mathf.Abs(a.x - b.x) + Mathf.Abs(a.y - b.y)**.

Nearest Neighbor Search Abbreviated NN Search or NNS. It is a type of query that, given an input position, will return the nearest / closest point that is inside of a spatial data structure. A variant called kNN Search returns the k closest points to a given position.

Sweep line algorithm A category of algorithms where a line is "swept" through a geometric data set. It can be done in numerous ways (but usually from a "highest" point to a "lowest" point) and can speed up the calculation if used in conjunction with the right data structures.

4 Data Structures

When using the Jobs in this package, you will sometimes encounter an out-parameter of type **JobAllocations**. This is always the case when the method called is allocating some additional memory. They are then stored into this class, which has to be disposed at an appropriate point in time (e.g. when the Allocator is TempJob within 4 Frames etc.).

For some of the more advanced algorithms, some data structures which are generally useful had to be implemented as to be used with the Job System. The more interesting ones are:

- **Native Priority Queue**
- **Native Sorted List**
- and a **Native AVL Tree**

All data structures require a Comparer for determining the way the values have to be sorted (e.g. ascending, descending etc.).

If you do not want to write your own Comparer, you can use default Comparers! For example:

```
NativePriorityQueue<int, DefaultComparer<int>>(default, Allocator.TempJob);
NativeSortedList<float, DefaultComparer<float>>(default, Allocator.TempJob);
```

In addition to these structures, there are of course various classes, methods and extensions for different shapes, such as lines, triangles, tetrahedra, etc.

4.1 Native Priority Queue

A standard implementation of a Priority Queue with a Binary Heap as underlying data structure (as it is the most cache-friendly, for it can be put into a single array). It has the interface of a regular queue, i.e. methods like **Enqueue()**, **Dequeue** etc.

However, dequeuing always returns the **largest** element (depending on your definition of large in your comparer)! In other words, if a regular queue is "first in, first out", a priority queue is "first in, largest out". This is useful in many cases where you are only interested in the single-most extreme value of a data set (i.e. you want to find the enemy with the greatest health etc.)

4.2 Native Sorted List

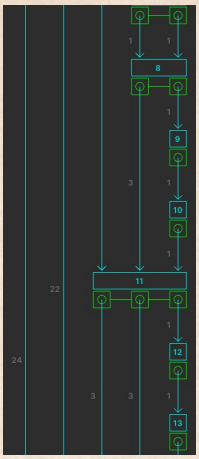


Figure 1: Editor GUI

A regular list, that keeps itself sorted with random access (like accessing elements of an array). The underlying data structure is a probabilistic skip list. This is in some way similar to a balanced Binary Search Tree, except without the tree and the recursiveness, but with more pointers.

Both **Insert()** and **Remove()** require only $\mathcal{O}(log(n))$ steps (in the limit). Accessing an element of the list is also an $\mathcal{O}(log(n))$ operation. Searching for an element with a specific value in the list (**Search()** or **Contains()**) is also an $\mathcal{O}(log(n))$ operation. In other words: Everything is $\mathcal{O}(log(n))$, which makes it an incredibly fast and versatile data structure for complex tasks.

This structure is useful if a list has to be kept sorted and elements are added and / or removed on a regular basis. As an example, you might want to sort all items based on their value (for an inventory). If items get removed or added quickly, a sorted list will be faster (and makes the code more maintainable as well).

Iterating through **ALL** elements can be done in $\mathcal{O}(n)$ using an **Enumerator**! Therefore try using the **foreach-pattern** instead of a **for-loop**. Its faster in this specific case

4.3 Native AVL Tree

A balanced binary search tree. In principle it has the same qualities as the sorted list / skip list (it has the same interface as well). However, an AVL Tree is more complicated computationally (and conceptionally as well). This gives it an higher performance overhead.

Therefore, in 95% of all cases the sorted list should be preferred. However, in some cases a tree can be more flexible (for example when it makes sense to apply special meaning to internal nodes). In addition, the memory footprint of AVL Trees is lower.

5 Polygons

An implementation of 2D non-simple Polygons can be found in **NativePolygon2D**. They can be convex or concave, contain holes or not. However, self-intersections are not allowed.

As it is more important for performance reasons and otherwise, holes are separated in the vertices list by a second separator list, containing their start

indices. Each, the polygon and the holes, should be in **counter-clockwise** order!

For adding a hole to a polygon, simply create the boundary first, and then call **AddHole()**. Holes are not allowed to overlap and they are also not allowed to lie outside the polygon (as it would not make sense).

When **Area()** is called, the holes are considered and their area subtracted from the outer boundary area (as you would expect).

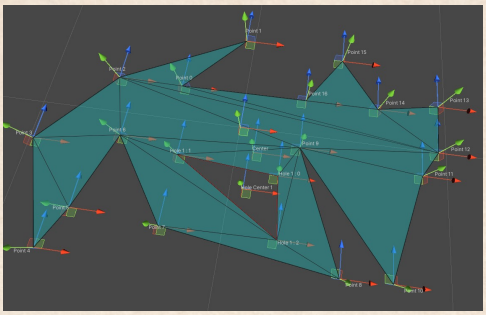


Figure 2: Polygon Handle

By making bridges from holes to the border, each polygon can be made "simple", i.e. containing no holes if you want (**MakeSimple()**). There are also methods for testing if a polygon is simple or convex.

For editing polygons, you can create and use a **NativePolygon2DHandle** and call **OnSceneGUI** from an Editor. The handle allows for adding and removing vertices and holes as desired (however the triangulation will fail when it is self-intersecting). You can also call **OnInspectorGUI** from the same handle to display all the options (show labels, show center handles, etc.).

Polygons are not serializable by default as the structs are using unsafe lists. When the **NativePolygon2DHandle** changes the polygon therefore, you'll have to read the modified polygon back with **GetModifiedPolygon()**. The handle marks the owner object as dirty when changes are present.

Polygons have methods for saving and loading them as binary files. Otherwise, you can always save the unsafe lists to regular arrays in private serialized fields of a MonoBehaviour. It should be relatively painless to combine these two saving methods with Addressables or a Resource Folder.

5.1 Location Queries

Underlying the location queries is a modified Winding Number Algorithm (using double the actual winding to be able to use integers instead of floats). You can also watch my tutorial about them here: Youtube: Winding Number Algorithm.

Other than that, it is only slightly adjusted for holes and put into two Jobs (single or parallel). Because of its simplicity, it is extremely fast. The necessary methods are in **Polygon2DPointLocation**.

5.2 Sampling

You can also create points that are distributed within the polygon. This is useful if you want to spawn objects only inside the boundary.

At the moment only points that are either evenly or distance-field distributed are supported (with Jobs). However, it should not be too difficult to implement your own sampling distributions if necessary.

In practice, you create an instance of the class **NativePolygon2DSampler** (which needs a NativePolygon2D for the constructor) and then call either **SamplePoint** (without Jobs) or **SamplePoints** (with Jobs).

5.3 Triangulation

Two different methods for triangulating **NativePolygon2D** are provided in this package (**Polygon2DTriangulation**):

- Ear-Clipping Triangulation
- Y-Monotone Triangulation

Ear-Clipping is an $\mathcal{O}(n^2)$ algorithm, but is faster for smaller polygons as it is less complex. Additionally, it is very stable even for small values. However, it cannot handle holes, and therefore each polygon using this method has to be made simple first (**NativePolygon2D.MakeSimple()**).

On the other hand, Y-Monotone or Monotone Triangulation is an $\mathcal{O}(n \log n)$ algorithm and scales very well for complex polygons and can deal with holes by default. However, as it uses a sweep line, it is prone to make errors when having small input values or positions that are too close to each other (on the local *Y-Axis*... which might be different global axis, depending on how you position your polygon).

Each triangulation job can return back the triangles in **clockwise** winding or **counter-clockwise** winding. Use clockwise for mesh-generation (as it is the default winding order of Unity)

6 Spatial Acceleration Structures

6.1 Query Types

Each spatial acceleration structure (**SAS**) supports some basic query types. This includes each quadtree variant, each octree variant, KD-Trees and Ball* Trees (2D + 3D).

For the 2D Case, these queries are:

- Circle-Query (with position and radius)
- Rectangle-Query (with Unity's Rect-Struct)

And for the 3D Case, they are:

- Sphere-Query (with position and radius)
- Cuboid-Query (with Unity's Bounds-Struct)

As you can see, each query takes a shape as input. However, you may also create parallel queries with multiple shapes of the same form. The work is then distributed with the Job System (ParallelFor-Job).

E.g. a native 3D KD-Tree has a method called **GetPointsInRadii**, taking NativeArrays as input, that contain the centers and radii. It will return a JobHandle that you can then Complete() at any moment of your choice. Each **SAS** supports parallel queries.

Note, that because overlapping shapes results in multiple CPU cores trying to write the same data to a hash set, the performance can degrade. In other words: Avoid overlapping shapes as best as you can.

6.1.1 Polygon Queries

The 2D KD-Tree, the 2D Ball* Tree and the 2D R* Tree support polygon queries. As the polygons, as always, support holes, this basically allows you to query any shape in 2D.

The methods in both trees require a **NativePolygon2D** as well as a transform in the form of a **Matrix4x4**. The points of the polygon are transformed internally before the query (as expected).

Internally, the bounding rectangle of the polygon is first used to find the approximate region. Afterwards, the points are checked against the polygon using a fast version of the Winding Number Algorithm.

6.2 Quadtree and Octree

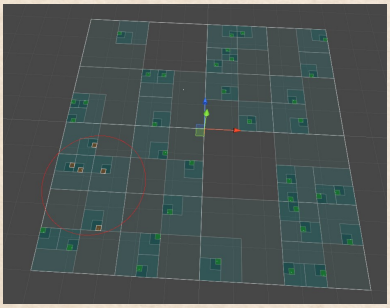


Figure 3: Quadtree Handle

There are two variants of Quadtrees and Octrees, **sparse** and **dense**. All trees implement either **IQuadtree** or **IOctree** and are spatial-hashed. The hash function is the Z-Order Curve / Morton Code in its respective dimension (Quadtree = 2, Octree = 3). Each individual hash then represents a cell or a bucket of the tree. In addition to that, only the paths to the hashes are stored (the trees are not complete).

The classes are called **NativeSparseQuadtree**, **NativeDenseQuadtree**, **NativeSparseOctree** and **NativeDenseOctree**.

Each tree has its own handle class (**NativeQuadtreeHandle** or **NativeOctreeHandle**) which similar to the **NativePolygon2DHandle** can be integrated into custom editors (don't forget to call OnSceneGUI of the Handle).

There are two types of queries (that were put into Jobs) implemented for each variant. You can either search the trees in a 2D Circle/ 3D Sphere around a world position or a 2D Rectangle / 3D Box. The query job then returns a list of hashes (Morton Codes) which you can then process how you like (i.e. you can either use it for vector calculations or getting the corresponding data buckets out of the tree).

There are also methods to update or remove values again from the tree.

6.2.1 Sparse Trees

Sparse trees have faster query times compared to their dense counterparts and, depending on your data, occupy far less memory. However, they are less dynamic, meaning that inserting, updating and removing objects from the trees will take longer.

6.2.2 Dense Trees

Dense trees have faster insert-, update- and removal times. However, the query times are slower (though not asymptotically) and they take a lot more memory. A dense quadtree has to allocate $\sum_{i=0}^{log_2 x} \frac{x^2}{4^i} = \frac{4x^2-1}{3}$ nodes in advance on creation, while a dense octree has to allocate $\sum_{i=0}^{log_2 x} \frac{x^3}{8^i} = \frac{8x^3-1}{7}$ nodes, where x is the number of cells along an axis. In other words, the memory grows quadratically and cubically respectively (as you would expect).

The first limitation therefore comes in form of the NativeArrays not being able to allocate more than 2GB (overflow) at a time. Which in the case of an octree is already reached with a 256x256x256 tree.

6.3 2D/3D KD Trees

A 2D/3D KD-Tree implemented in DOTS (**Native2DKDTree** or **Native3DKDTree**). It allows you to make very fast range queries for a static dataset (without moving points). Both, rectangle / bounds and radius search, which work similar to the quadtree/octree jobs, take about $\mathcal{O}(\sqrt{n}+k)$ time for the 2D case and $\mathcal{O}(n^{2/3}+k)$ for the 3D case, where k is the number of reported points (the size of the search result). This means,

that there is a large overhead for small point sets, that gets smaller and smaller with size.

KD-Trees, in addition to radius and rectangle queries, can search for the nearest neighbor to a given position (or multiple nearest neighbors to multiple given positions). The method, that schedules the job for it, is called **GetNearestNeighbors()**.

The 2D KD-Tree can be created in each plane of 3D Space i.e. you can sort the points along the XY-, XZ- or YZ-axis. You do this by providing the correct sorting mode to the constructor.

There is also, of course, a **Native2DKDTreeHandle** and a **Native3DKDTreeHandle**, which you can use in a custom editor in any way you see fit. Don't forget to call **OnSceneGUI**!

6.4 Bounding Volume Hierarchies

Bounding Volume Hierarchies (**BVH**) are commonly found in physics engines, ray tracing structures or machine learning. As the name implies, they save a bounding area or volume instead of point data, which enables one to do **Raycasts** or **Frustum Queries**!

Depending on your game, this will give you an additional performance boost, because it may enable you to remove some of your colliders!

In addition to that, they are also adjusting to the underlying distribution. This means that querying less densely populated areas is cheaper. On the flip side, this also means, that it is somewhat slower for a while when the distribution changes quickly.

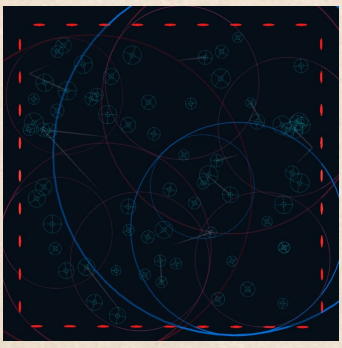


Figure 4: Dynamic 2D Ball* Tree

The literature on BVH is often about making optimal trees. However, rebuilding the structure can be very costly, especially when you have a lot of moving objects in your game. For this reason, the BVH in this package were made to optimize themselves with a heuristic over time, rather than being optimal (hence the *dynamic* and the "*").

This heuristic is its own method / job (**Optimize()**), as it can be paused without any problems if you need some additional performance for a while. However, this degrades the query performance over time and is not recommended.

Non-optimality trades in query performance for update cost. That said, the queries are still lightning fast, and most of the time faster than the simpler Quadtrees and Octrees.

BVH in this package do not have editor handles, as they are rather chaotic when drawn (You can see their behaviour in the test scenes)

6.4.1 Dynamic Ball*-Tree

Encapsulates bounding circles or spheres into a hierarchy. Which means that the objects or data inserted into the tree has two requirements:

- They implement **IBoundingCircle** or **IBoundingSphere**
- They implement **IIdentifiable** (to avoid expensive equality tests)

Queries do return those circles or spheres again, which is why they are named slightly differently:

- **GetCirlcesInRadius() / GetCirclesInRectangle()**
- **GetCirclesInPolygon()**
- **GetSpheresInRadius() / GetSpheresInBounds()**
- **etc.**

Parallel queries are available, so are queries that allow you to get the overlapping circles or spheres as well.

In addition to that, there is a method called **Raycast**, which takes either a **Ray2D**- or **Ray**-struct as parameter (same as Unity) and returns all circles or spheres on the line segment defined by them. The y are returned in a sorted list within a **IntersectionHit2D**- or **IntersectionHit3D**-struct. These structs also include a list of the intersection points (at most two).

In contrast to the **dynamic R*-Tree**, a 3D Ball*-Tree is capable of efficiently culling spheres from a given camera frustum. You can find a **FrustumQuery()**-method in the class which will return only the visible spheres in the provided camera.
Updating a Ball*-Tree is a two-step process:

- Update the circle positions and radii, either each one individually (with **Update**) or with the **UpdateAll**-Job
- Optimize the structure by scheduling a job with **Optimize()**

6.4.2 Dynamic R*-Tree

Encapsulates bounding rectangles or axis-aligned boxes into a hierarchy. Which means that the objects or data inserted into the tree has two requirements:

- They implement **IBoundingRect** or **IBoundingBox**
- They implement **IIdentifiable** (to avoid expensive equality tests)

Queries do return those rectangles or boxes again, which is why they are named slightly differently:

- **GetRectanglesInRadius() / GetRectanglesInRectangle()**
- **GetRectanglesInPolygon()**
- **GetBoundsInRadius() / GetBoundsInBounds()**
- **etc.**

Parallel queries are available, so are queries that allow you to get the overlapping rectangles or boxes as well.

R*-Trees are of course also capable of **Raycasts**, in the same way the Ball*-Trees are. However, they do not support **Frustum Queries**.

Updating is done in a similar fashion to the Ball*-Tree as well (first **Update()** then **Optimize()**).

7 Mesh Operations

7.1 Mesh Slicing

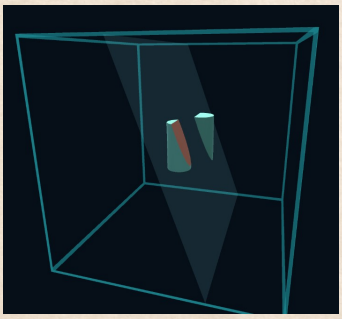


Figure 5: Sliced Cylinder

Slicing enables you to cut a mesh into pieces. The static class that implements the necessary Jobs and Methods is called **MeshSlicing**. To make a cut you will need a **Mesh** and define a **Plane** (Unity Struct).

The mesh has the additional requirement, that it is not allowed to be self-intersecting at the place of the cut (it is allowed to self-intersect in the "unaffected" parts though).

If the plane misses the mesh entirely (cutting at the wrong position), the **Slice**-Method of the class will return **null**. Otherwise, it will return two meshes representing the elements to each side of

the cutting plane. Each of them will have a number of submeshes corresponding to each unconnected surface lying on the plane (i.e. each submesh is a polygon).

The first submesh is reserved for the "shell" i.e. the parts of the original mesh that remained. Normalized UV coordinates are automatically generated for each other submesh (the polygons), allowing you to assign different materials to each surface at your leisure.

The algorithm internally is a custom design, using other algorithms from the package. It works by finding connected edge loops around the plane, which then form polygons, which are then triangulated and converted into meshes.

7.2 Mesh Primitives

The class **MeshUtil** has numerous methods for creating the primitives and shapes you can find in the table below. If the a cell in the column **Outline** is marked with "Yes", then there is also an additional method for creating the outline of the shape.

Mesh Primitives			
2D	Outline	3D	Outline
2D Grid	-	3D Grid	-
Arrow	-	Arrow	-
Circle	Yes	Box	Yes
Line	-	Cone	-
Polygon	Yes	Cylinder	-
Triangle	Yes	Line	-
Rectangle	Yes	Prism	-
		Tetrahedron	Yes
		Torus	-
		Triangle	Yes

8 Tessellations

Tessellations are divisions of surfaces into smaller parts such that the space is filled and there are no holes or gaps. Stretching the meaning, I use this terminology in this document to refer to Delaunay-Triangulation, Voronoi-Diagrams with different metrics etc. collectively (The Delaunay Triangulation is a tessellation of the convex hull of an arbitrary set of point. So in this way it checks out).

8.1 Delaunay Triangulation

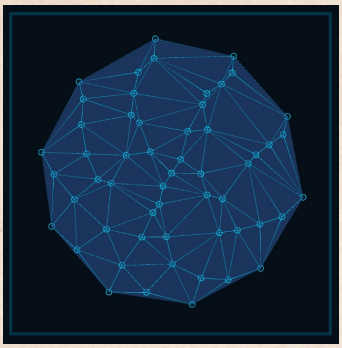


Figure 6: Delaunay

Given an arbitrary set of 2D points, the Delaunay Triangulation will form triangles in the convex hull of these points such that the minimum angle of each triangle is maximized. The package contains a randomized incremental implementation of this algorithm that runs in $\mathcal{O}(n \log n)$.

This type of triangulation can be used for various purposes as for example terrain data, point clouds, 2D NavMeshes etc. The dual is the **Voronoi Diagram** (next section). To calculate it, call the **CalculateDelaunay()**-method from the **Delaunay2D**-class. Because of floating-point precision, issues will arise under these two circumstances:

- Two points are very close together (and therefore forming a very small-angled triangle)
- Three points lying on the convex hull are almost collinear (and therefore forming a very small-angled triangle)

Otherwise, the algorithm will run stable, even with single-precision arithmetic.

8.2 Voronoi Diagrams

A Voronoi Diagram is a tessellation of the plane using sites (a site just being a position on the plane or in space). Each region or cell of the resulting diagram is associated with one and only one of these sites. And each of them has the property that each point inside the cell is closer to the associated site than to any other site (by some form of distance metric. In game development this is almost always the euclidean distance).

Figure 7: Voronoi

However, that does not explain too much. Here is the reverse programmer-friendly version: Given an array of polygons (e.g. **NativePolygon2D**), an array of sites (e.g. **float2** or **Vector2**) and a dictionary that uniquely maps each polygon to one of the sites that make up

a Voronoi Diagram; if an arbitrary query position is inside one of these polygons, then the closest site / point is the one that the polygon is associated with in the dictionary.

These queries can be precomputed as a lookup table (or texture) for non-moving objects (next section: **Voronoi Lookup Table**) to speed up problems like "Find the closest quest NPC (site) to the current player position (query position) out of a 100 NPCs (sites)" tremendously (it is a lookup, so it is just $\mathcal{O}(1)$!) at the cost of memory.

Another use case, that may also be implemented in the future in this package, would be **2D mesh fracturing**. They can also be used to define map and control regions in certain games. Or for decoration (procedural mosaics for example).

Calculating a Voronoi Diagram is simply done by calling the **CalculateVoronoi()**-method in the **Voronoi2D**-class. As the algorithm uses the dual of the **Delaunay**-Triangulation, it has the same $\mathcal{O}(n \log n)$ complexity.

8.2.1 Voronoi Lookup Tables (VoLT)

Although the table and a voronoi texture are the same in principle, their use cases are different. For this reason, and because **Voronoi Lookup Table** is long to write, I use the acronym **VoLT** (also in the code).

A **VoLT** can be used to speed up nearest-neighbour queries of non-moving objects to $\mathcal{O}(1)$ at the cost of memory and precision. It achieves this by pre-computing and then storing the answer to the nearest-neighbour query for each point in a grid.

For the curious: Such tables might not only be created for 2D, but also 3D and for other metrics as well. Even shapes are possible!

You can calculate a **VoLT** by calling **CalculateVoronoiLookupTable()** from the **Voronoi2D**-class. The returned array is stored in one dimension but contains all the two-dimensional data. Each element of it points to the closest site.

To find the closest site with a **VoLT**, we simply have to convert regular coordinates to a table index by calling **CalculateVoronoiLookupTableIndex()** which is also found in the same **Voronoi2D**-class.

Depending on the performance-requirement of the calculation of this table (and also its size), it can also be computed via the GPU (e.g. with a compute shader). However, this would be outside the scope of this package.

You can however find a tutorial on my channel here where I show how to do this calculation (with compute shaders): Youtube: How Voronoi Diagrams Can Speed Up Your Game

9 Hulls

Finding a hull is the process of enclosing a set of points with a pre-determined shape. Currently, all algorithms are contained in the **HullAlgorithms-Class**.

9.1 Convex Hull

You can create a convex hull (consisting of a polygon) of an arbitrary set of 2D points by using one of the methods in the **HullAlgorithms-Class**. The output of one call to **CreateConvexHull()** is always a **NativePolygon2D**. If you have a lot of points (say more than a thousand) you might want to consider filtering the points (an option in the method parameters). Doing this applies the Akl-Toussaint-Heuristic to the set of points first, which will significantly speed up the calculation (but might be slower when having few points).

9.2 Minimum Enclosing Disc and Sphere

You can find the minimum disc or minumum sphere around a set of points (2D or 3D) by calling **FindMinimumEnclosingDisc** or **FindMinimumEnclosingSphere**.

The method used internally is an adaptation of Welzl. As such, it is an incremental randomized algorithm, which means that the performance varies depending on the order of your points. But in principle, it does scale with $\mathcal{O}(n)$ (where n is the number of points)

The area of enclosing discs and spheres is larger than that of convex hulls, however they are easier to work with. In addition to that, you might want to use these algorithms to calculate the minimum bounding sphere of procedurally generated meshes.

9.3 Bounding Rectangle and Box

For completion's sake, there are also Jobs provided for calculating the minimum enclosing rectangle and box, also known as bounding rectangle and box. The methods are called **CalculateBoundingRect** and **CalculateBoundingBox** respectively.

10 Intersection, Overlap and Containment

10.1 Shape-Shape Tables

Although not the primary focus of this package, many intersection-, overlap- and containment-procedures have been written for various algorithms. The tables should act as quick reference.

Legend - Table Symbols:

X ... Implemented
D ... Dimension of a Shape has to be changed (e.g. 2D-Line to 3D-Line)
U ... Already implemented by Unity
- ... Does not have meaning

Shape Abbreviations:
LS2 ... Line Segment 2D
LS3 ... Line Segment 3D
L2 ... Line 2D
L3 ... Line 3D
P ... Plane
R ... Rectangle
B2 ... 2D Ball / Circle
B3 ... 3D Ball / Sphere
Poly... Polygon (2D)

10.1.1 Shape Intersections

Intersections between Shapes									
Shape	LS2	LS3	L2	L3	P	R	Bound	B2	B3
LS2	X	D	X		D	X	D	X	D
LS3	D	X			X		X	D	X
L2	X		X	D	D	X			
L3			D	X	X				
P	D	X	D	X	X		X		
R	X		X						
Bound	D	X			X		U		
B2	X	D							
B3	D	X							

10.1.2 Shape Overlap

Overlap between Shapes						
Shape	LS2	LS3	R	Bound	B2	B3
LS2	–	–	X	D	X	D
LS3	–	–	–	X	D	X
R	X	–	U		X	
Bound	D	X	–	X	–	X
B2	X	D	X		X	
B3	D	X	–	X	–	X

10.1.3 Shape Containment

Containment between Shapes								
Shape	Point	LS2	LS3	R	Bound	B2	B3	Poly
Point	–	–	–	U	U	X	X	X
LS2	–	–	–					
LS3	–	–	–	–				
R	–	–	–	X		X	D	
Bound	–	–	–	–	X	–	X	
B2	–	–	–	X		X		
B3	–	–	–	–	X	–	X	
Poly	–	–	–					

10.2 Line Intersections

Detecting all intersections between a set of n lines (or line segments) scales n^2 in the worst-case (each segment crosses each other segment). Therefore the optimal algorithm also is at best $\mathcal{O}(n^2)$.

Therefore, a simple detection algorithm just checks all combinations and returns the intersections it found. This is used when calling **FindLineSegmentIntersectionsCombinatorial** in **LineIntersection**.

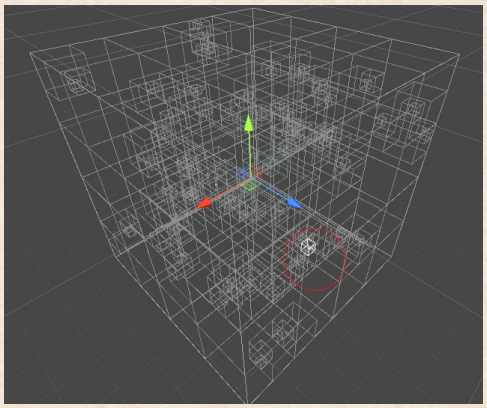


Figure 8: Octree Handle

However, in most cases, the number of intersections i is far lower than n^2 and one can create an algorithm that is output-dependent using a sweep line. The runtime is something of the likes of $\mathcal{O}(n \log n + i \log n)$. That means, the algorithm can perform better or worse than checking all combinations, depending on the shape of the input.

Moreover, as it is using a sweep line, it can be unstable because of floating-point precision. To remedy the worst problems, the Job can restore its status to allow it to continue after it detects an "error" (when an element can't be found in a tree). Restarting adds some additional performance penalty (if there are floating-point problems).

I'd recommend to use the combinatorial version for line intersections (it is very fast, as it is very simple), except if you have thousands of lines that are relatively spaced out. There is a sample scene included that shows a situation where the sweep line is faster.

If you're wondering why the sweep line algorithm is even included → it is the precursor for computing map overlays.

11 Special Queries

11.1 All Radius Query

The **All Radius Query** is well suited for finding all points within a fixed radius r for each point in a set of points. You can find it in the class **SpecialQuery**. A parallel version of the algorithm exists as well.

Why choose it over querying a SAS n-times with radius r? Because, for this particular type of query, there are some improvements that can be made in terms of performance.

To my knowledge, there is no available description of this algorithm yet (but I might have simply not found it because I do not know the name), so I'll explain the concept a little bit more in detail, so it can be understood by the curious (it is very, very simple compared to other sweep line algorithms).

First, notice that if the radius is fixed, then as soon as point a is within the radius of point b, then so is point b within the radius of point a, as both radii are equal. This halfs the number of comparisons we need to do.

Also, very loosely speaking, in a SAS when doing a query, first the "general region" is determined and then only when we get closer to the leaves of the (usually) trees we get some actual results. This is more than fine for a single query. However, using a sweep line algorithm, as we now traverse the set of points in an ordered fashion, we can update the "general region" accordingly depending on where we are right now. Which removes additional comparisons.

And so these are precisely the things that make this algorithm. It is a sweep line checking all *semi-circles* from bottom to top. Each point has two events. The first event is located at the position of the point itself (**start event**), and the second one is where the semi-circle extending upwards ends (**end event**). A **NativeSortedList** keeps track of all the *semi-circles* we have to consider right now. Points are added to the status, when there is a **start event** and removed when there is an **end event**. This now gives us a *section* of thickness r along the Y-Axis (the "general region").

Stating the obvious, notice that two points are within their radii if the distance between their points is smaller than the radius r. But now only looking at the X-Coordinate of the points, we can also see that the distance has to be smaller than the radius r as well. This is why the **NativeSortedList** sorts the points with their X-Coordinate. If we now want to compare a point p with our current region, we first find all points in the status that are, with their X-Coordinates, in the interval $[p_x - r, p_x + r]$. This is done using a **Range Query**, returning a subset of the status.

What remains now are points within a rectangle around p. And for these points only we check the distance. When it is detected that two points a and b have a distance less than r, then b is added to the list of a and vice versa.

As for the time complexity (not rigorous): First, the points need to be sorted along their Y-Coordinates (and on their X-Coordinates when there are ties), an $\mathcal{O}(n \log n)$ operation. For the **main part** of the algorithm we have to go through a list of size $2n$. Lets denote the number of points within a *section* of height r as k. Then for n points of the list we do a range search which is a $\log k$ operation. Lets denote the number of points within the rectangle defined by a point p as previously as l. After the range search we iterate through a (sorted) list of size l doing l comparisons. The total number of comparisons of the **main part** is approximately $n(\log k + l)$ or $n \log k + nl$. The runtime complexity in total is approximately $\mathcal{O}(n \log n + n \log k + nl)$ (note that both k and l can be as big as n giving us a quadratic runtime. I omitted the $n \log l$ factor for adding and removing from the sorted list as it cannot be more dominant than nl).

11.2 All Rectangle Query

The **All Rectangle Query** can be used to find all points within a fixed rectangle r for each point in a set of points. You can find it in the class **SpecialQuery**. A parallel version of the algorithm exists as well.

It works very similar to the **All Radius Query** from the interface / API point-of-view as well as the internal workings. The only difference is that the radius check is *removed*, making it slightly faster than the **All Radius Query**. That said, it depends on the dimensions of the rectangle r.

11.3 Range-Queries / 1D-Queries

Range Queries are queries like "find all enemies with a health between 38 and 55", "return all NPCs whose quest is between 1 and 3 days old" etc. They are frequently used in databases, but they also have their use for solving geometric problems when trying to answer a question like "which map locations are between the x-coordinates 400 and 550" for example.

There is no special SAS for 1D-Queries. Instead, the **NativeSortedList** has a method called **SearchRange()** with a minimum and maximum parameter, returning a JobHandle to be completed.

12 FAQ

> 1. — I have static objects / static positions (like vertices for example) and want to query them from a known position. Which data structure should I use?

KD-Trees. They are very, very fast.

2. — I have few or rarely moving objects and want to query them from a known position. Which data structure should I use?

Native2DBallStarTree or **Native3DBallStarTree**. They are fast and adapt themselves. Otherwise, use **NativeSparseQuadtree** or **NativeSparseOctree** as they are faster than the dense version and take a lot less space!

3. — I have a lot of moving objects and want to query them from a known position. Which data structure should I use?

Native2DBallStarTree or **Native3DBallStarTree**. Definitely!

If you have a reason to use Quadtrees or Octrees (because of world chunks for example), then it depends on how many moving objects exactly. If you have roughly around 3000 moving objects or more, use the **NativeDenseQuadtree** or the **NativeDenseOctree**. Otherwise, still use **NativeSparseQuadtree** or **NativeSparseOctree**.

If you are unsure about which variant to use, you can test them both with various parameters in the **QuadtreeMovementTest-Scene** and the **OctreeMovementTest-Scene**.

4. — I have a lot of objects n and want to query from each object all objects within a radius or rectangle **r**

All Radius Query or the **All Rectangle Query** are your best choices (better than querying a SAS over and over)

13 Contact

For any questions or suggestions, you can reach me anytime by the following email-adress:

blenderfan@gmx.at

There is also a discord server, which is usually the fastest way to reach

me:

Parable Games - Discord

Alternatively, you can also find some social media links and contact information on my website:

https://parable-games.com

14 Future Plans

Some additional features I plan to implement in the near future:

- Map Overlay Algorithm
- Straight Skeleton of Polygons (for generating roofs)
- AABB Tree
- Marching Squares / Cubes
- Quickhull Disk (for fun)

Thank You!

Your purchase of **Gimme DOTS Geometry** enables me to continue developing code and techniques for game-development in an independent way!