# GIMME
# GPU Geometry

Manual

## Contents

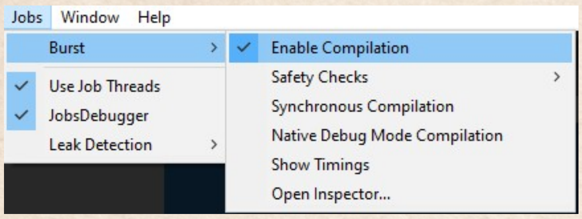
## 1 Dependencies and Setup

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

- **Gimme DOTS Geometry 2.0.0** or above
- **Burst 1.8.0** or above (might work with lower versions; untested)
- **Collections 2.0.0** or above
- **Mathematics 1.2.0** or above (might work with lower versions; untested)

Additionally, the platform you are building for needs to support Compute Shaders. You can always check if that is the case by examining the **SystemInfo.supportsComputeShaders** flag (most platforms support them).

Furthermore, 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.

## 2 Introduction

**Gimme GPU Geometry**, an extension of **Gimme DOTS Geometry**, is a collection of algorithms in geometry that can run on the GPU. It has the capability to vastly exceed the speed and performance of similar algorithms on the CPU, provided they can be sufficiently parallelized.

If used with care, this might give you the options to render and manage far more entities as would be possible otherwise, generate particle / fluid simulations or it might generate lookup tables for nearest neighbor searches faster, etc. It can be either used on the GPU alone or as a hybrid between CPU and GPU calculations.

I want however take this introduction as an opportunity to give you a fair warning. While the algorithms in this package run faster than what is possible otherwise, because of CPU-GPU communication, you will only get the most out of the package if you have a solid knowledge of **shaders**!

Why is that so? Because, by being able to write your own shaders you will be able to use the results of the algorithms directly on the GPU as well, without touching the CPU. That said, even if you do not know HLSL well enough yet (ShaderGraph is not going to work directly, as it cannot use **StructuredBuffers** yet), there are plenty of shader examples and sample scenes in the package to learn from.

So how much performance is gained? This is a surprisingly difficult question to answer. As of now, Unity does not have a way to to profile the individual compute shaders (although it is planned). Therefore, it is advised to use an external tool like **RenderDoc** for now. Additionally the performance varies a lot between different GPUs, Vendors and of course platforms.

But to no longer avoid an answer: I have checked each compute shader individually with RenderDoc on my GTX 1070 and found the performance compared to DOTS Geometry (with a 12-core CPU) somewhere in the range of **3- to a 100-times faster** or more, depending on the type of problem. This tremendous speedup is of course biased as I purposefully selected the algorithms that are easily parallelizable, gaining the most from outsourcing calculations to the GPU, compared to DOTS Geometry.

## 3 About Compute Shaders and the GPU

As compute shaders are still somewhat shrouded in mysticism and considered occult knowledge, I'll give some brief explanations about the basic concepts. This information is not necessary to *use* the package, but it will help in *understanding* it.

Compute Shader, aptly named, compute things instead of rendering. Their main advantage is that they can use the thousands of SIMD-Processors on the GPU, that are otherwise used for the fragment-stage in a regular shader. To use them simultaneously, work is split up into threads and thread groups (and wavefronts). This process is done by dispatching the shader.

Each thread then gets assigned an **ID**, that is used to determine what data it is allowed to access and write to (usually, in most cases).

The results of a calculation are then written into either a **texture** or a **buffer** in parallel (you can think of it as an array). To also then return those to the CPU-side, there are a multitude of ways, synchronous and asynchronous.

Typically, this involves **GraphicsBuffer** and a call to **GetData()** of the same (synchronous), or an **AsyncGPUReadbackRequest** with **AsyncGPUReadback.RequestIntoNativeArray** (asynchronous). It should be noted that reading back data from the GPU (and writing to it as well) is a slow memory operation and should be used as sparingly as possible. In the best case scenario, any results of a compute shader are passed on to another subsequent shader.

It should also be noted that reading back asynchronously has a delay attached that can span multiple frames. Usually, as everything involved in this package is not affecting the visuals directly, this is of minor concern though.

## 3.1 Graphics Fence

Sometimes, if you want to check if a compute shader has finished executing on the GPU after dispatching it, you can create and store a **GraphicsFence**. This is done by calling **Graphics.CreateGraphicsFence()** immediately after dispatching a compute shader. Make sure the stage of the method is set to **ComputeProcessing**. The struct that is returned has a field called **passed** that you can check (each frame).

There is another option when creating a graphics fence, and that is whether it should be used to synchronize behaviour on the CPU or the GPU. However, very important, GPU synchronization is not yet widely supported for different platforms and graphics APIs (DirectX 11 does not support it). You can check for the support by examining the **SystemInfo.supportsAsyncComputeQueue**. The package does not assume (yet) that it is supported and never depends on it (for now).

## 3.2 Buffers

Buffers, in their simplest form, are essentially arrays (except, in this particular case, having 128 bits for each element). They come in three forms:

- Readable (also referred to as **Resources**)
- Readable and Writable (also referred to as **UAV**)

The first form is the default, while the second is denoted with an **RW** in front of the name. E.g. there exists a simple **Buffer** in HLSL that is only readable. Then there also exists a **RWBuffer** that can also be written to.

This is of course somewhat simplified (for example, there are also constant buffers), but in essence this is all they are. Additionally, there are multiple useful flavours of buffers. The most commonly used versions in this package are **Structured Buffers** and **Append Buffers**.

Ok, so why are they needed in the first place? In principle we could store and sample again everything with textures, however... if we accidentally filter the texture, then depending on our results this might produce errors. Additionally, not every texture format is supported on every platform.

Therefore, first and foremost, they are used for the same reasons as in any other programming language: To store arbitrary data and access it again.

Creating a buffer is quite simple, by just making a new **GraphicsBuffer**. To then use and transfer it to a shader you have to call **SetBuffer()** on it.

### 3.2.1 Structured Buffers

Structured Buffers are the most common type of buffers you can find in Gimme GPU Geometry. They literally are capable of storing **structs**. There is also of course a **RWStructuredBuffer**, allowing you to even write (not quite) arbitrary structs to an output array!

As with any buffer, care should be taken to not write or read outside the bounds of it. This necessitates either sending an additional integer about the size of the array to the GPU or calling **GetDimensions** (in the compute shader).

How does the GPU know the structs sent from the CPU (by calling **SetData()** on a GraphicsBuffer for example)? The answer is: The GPU does simply not know. It has to be told in code, by recreating the same struct with the same number of bytes. In this document I will refer to structs that have a counterpart on the GPU as **Mirror Structures**.

### 3.2.2 Append Buffers

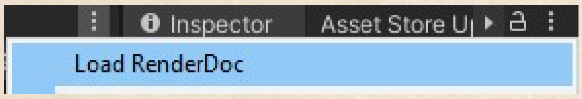
In addition to be able to hold structs (similar to the **StructuredBuffer**) an AppendBuffer (as well as its counterpart **ConsumeBuffer**) has a hidden counter variable. Instead of writing directly to it, the GPU will attempt to **Append** structs at the position of that counter. After that, the counter is increased (atomically) by one.

This is is an extremely useful type of buffer when the number of results is unknown but the order is not important (e.g. the number of lines intersecting spheres). Because the counter value is unknown to the CPU, you first have to (or should) get the same (which is a little bit elaborate and requires an additional counter buffer to hold the integer - you can find some examples of that scheme in the code) before you read back the values into a **NativeArray** or **NativeList**.

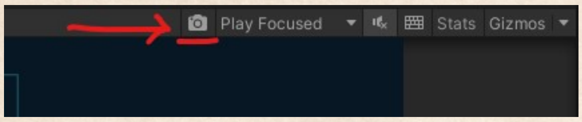
Clearing an Append Buffer is simply done by calling **SetCounterValue(0)** of the **GraphicsBuffer** in question. In this way, no new buffer has to be allocated again (which would be expensive).

## 3.3 Measuring Performance

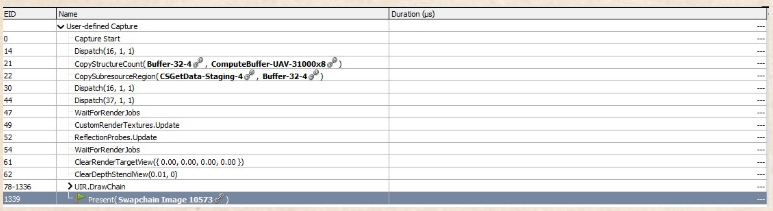
If you want to measure the performance of the compute shaders yourself, first load RenderDoc in Unity.



Then run the scene that dispatches the compute shader you want to measure. Press the button in the picture to create a snapshot.

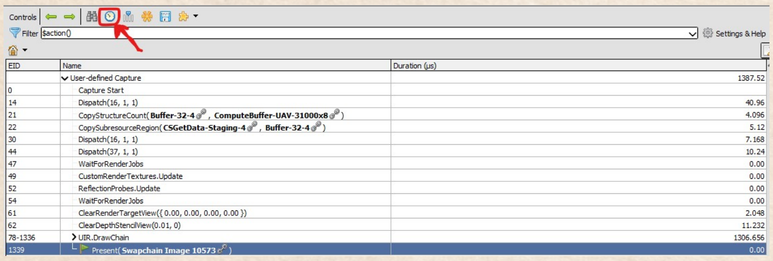


RenderDoc will open and with a double click you will be able to see a list of all the events that have happened on the GPU in that frame, including all Dispatches of Compute Shaders.



| EID | Name | Duration (µs) |
|---|---|---|
| | User-defined Capture | --- |
| 0 | Capture Start | --- |
| 14 | Dispatch(16, 1, 1) | --- |
| 21 | CopyStructureCount( **Buffer-32-4** , **ComputeBuffer-UAV-31000x8** ) | --- |
| 22 | CopySubresourceRegion( **CSGetData-Staging-4** , **Buffer-32-4** ) | --- |
| 30 | Dispatch(16, 1, 1) | --- |
| 44 | Dispatch(37, 1, 1) | --- |
| 47 | WaitForRenderJobs | --- |
| 49 | CustomRenderTextures.Update | --- |
| 52 | ReflectionProbes.Update | --- |
| 54 | WaitForRenderJobs | --- |
| 61 | ClearRenderTargetView({ 0.00, 0.00, 0.00, 0.00 }) | --- |
| 62 | ClearDepthStencilView(0.01, 0) | --- |
| 78-1336 | UIR.DrawChain | --- |
| 1339 | Present( **Swapchain Image 10573** ) | --- |

By pressing the clock, you can time the actions in this list, giving you (accurate) estimates of how long each step took in the frame.



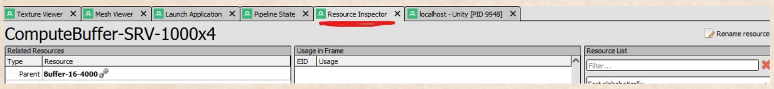
Filter: $action()

| EID | Name | Duration (µs) |
|---|---|---|
| | User-defined Capture | 1387.52 |
| 0 | Capture Start | |
| 14 | Dispatch(16, 1, 1) | 40.96 |
| 21 | CopyStructureCount( **Buffer-32-4** , **ComputeBuffer-UAV-31000x8** ) | 4.096 |
| 22 | CopySubresourceRegion( **CSGetData-Staging-4** , **Buffer-32-4** ) | 5.12 |
| 30 | Dispatch(16, 1, 1) | 7.168 |
| 44 | Dispatch(37, 1, 1) | 10.24 |
| 47 | WaitForRenderJobs | 0.00 |
| 49 | CustomRenderTextures.Update | 0.00 |
| 52 | ReflectionProbes.Update | 0.00 |
| 54 | WaitForRenderJobs | 0.00 |
| 61 | ClearRenderTargetView({ 0.00, 0.00, 0.00, 0.00 }) | 2.048 |
| 62 | ClearDepthStencilView(0.01, 0) | 11.232 |
| 78-1336 | UIR.DrawChain | 1306.656 |
| 1339 | Present( **Swapchain Image 10573** ) | 0.00 |

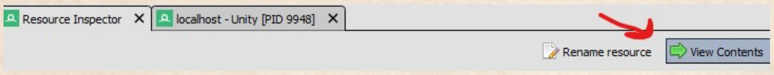
This covers the basics of measuring the performance. RenderDoc also allows you to go deeper though as well, and I recommend checking out the website and documentation for that.

## 3.4 Debugging

Debugging on the GPU is notoriously difficult when it comes to the behaviour. RenderDoc, again, can help in that regard. Sometimes it suffices to have a look at a specific compute shader and the contents of its buffers. Once you have a snapshot (see previous section), you can easily access those by clicking on the **Resource Inspector**.



You can select a buffer (as they do not have proper names sometimes, it is recommended to look at the pipeline stage first to find the right compute shader). Afterwards, if you press the **View Contents** button, a window will open that shows the contents of that particular buffer (in bytes)!



Converting bytes to structs again is maybe not very user-friendly, but it beats the alternative of *guessing* problems any day.

## 4 Classes and Data Structures

### 4.1 GPU Job Manager

The GPU Job Manager has to be part of every scene that wants to use Gimme GPU Geometry!

Next to some simple functions, the **GPUJobManager** (MonoBehaviour) is expected to be part of the scene to start various coroutines for awaiting certain tasks on the GPU (that have to be executed in order).

Note that not all algorithms technically require it, but to be safe, always add it first to a scene when using the package.

### 4.2 GPU Jobs

GPU Jobs (class: **GPUJob**) are intended to be the pardon to regular **Jobs**. In contrast to **burst-compiled** code that is executed, here it is HLSL and compute shaders.

Similar to Jobs, you first schedule them (with optional dependencies) to get back a **GPUJobHandle**. Automatically, the compute shader will be dispatched based on the parameters.

The dependencies can either be other GPU Jobs or regular Burst Jobs (in that case, the compute shader is dispatched after the regular Job has finished).

You can also pair the Job with an asynchronous read back into a NativeArray. For that, use the optional **GPUReadbackContainer** parameter of the constructor of the **GPUJob** class.

To query if a GPU Job has completed, call **IsCompleted()** of the GPU-JobHandle. The method will return true if the shader itself and all its dependencies are completed.

As of now, the system is yet incomplete and not very well tested. It will work as intended with the given API and the example scenes are using them, but many things are not yet supported (i.e. the only GraphicsBuffers types supported right now for read back are **StructuredBuffers** and **AppendBuffers**). As the functionality of this class is just intended to ease the ordering and dispatching of compute shaders, it is always possible to order them yourself should the class fail.

## 4.3 Mirror Structures

Mirror Structures are **structs** that exist on both the CPU and GPU. In other words, these are data types that are supported by the compute shader library provided with this package.

These include:

- LineSegment2D (DOTS Geometry)
- LineSegment3D (DOTS Geometry)
- Triangle2D (DOTS Geometry)
- Triangle3D (DOTS Geometry)
- Plane (Unity)
- Rectangles (Unity - use extension ToVector4())
- Bounds (Unity - use extension ToGPUBounds())

Spheres and circles, similar to DOTS Geometry, do not have their own structs. If they are sent to the GPU, they have the following format (float4):

- Circle: $(X, Y, Radius^2, 0)$
- Sphere: $(X, Y, Z, Radius^2)$

### 4.3.1 GPU Polygons

**NativePolygon2D** can also be used on the GPU. However, as they can be arbitrarily large, they are stored in buffers (which have to be disposed again). To manage polygons on the GPU, use the class **GPUPolygon2D**.

To generate a GPU polygon from an existing one, call the static method **GPUPolygon2D.ConstructFrom()**. Currently, GPU Polygons are not supporting holes. This is for now more a performance consideration and the uses of the class in the package. Once it is necessary to have polygons with holes, the support for them will follow.

## 5 Spatial Acceleration Structures

### 5.1 Bins

Bins, more commonly known as grids, provide a way to do queries on points faster than without any structure. It should be noted though, that for the most common use-cases, brute-force queries are more than enough.

Bins do only become necessary when having an incredible amount of points (far more than 10k, like in simulations) and a vast amount of queries (all-radius queries for example). Otherwise, for simplicity, brute-force queries should be preferred. If in doubt, profile both and weigh the advantages and disadvantages.

The two classes are called **GPU2DSpatialBins** and **GPU3DSpatialBins**. The reason they are not named "grids" is for a user (that did not read the manual) to not make the assumption that each cell can only hold a single value. However, in this document the terms bin and cell, and bins and grids, are used interchangeably.

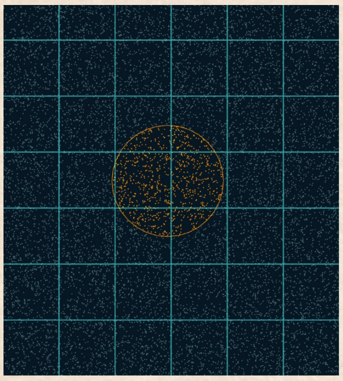


Figure 1: 2D GPU Grid - Radius Query

To create a grid, simply call the (code-documented) constructor, where you can provide the size and capacities of the bins, queries and results. Additionally, you have to pass a GraphicsBuffer that holds all the positions in the grid. It is allowed to change in size (i.e. it can be an Append Buffer) and each entry should contain three floats for the position (float3).

Each bin has a fixed maximum amount of entries it can hold, which is provided by the **binCapacity** of the constructor. Should it be exceeded, then the additional entries will simply be ignored - which in turn means that the query results will always be correct, but may be incomplete should any capacity be insufficient.

Once it is created, the process of doing queries in a grid on the GPU from the CPU is the same in 2D and 3D, and is done with the following four steps:

- Queuing an arbitrary amount of queries with the provided methods (e.g. **QueueRadiusQuery()**, etc.)
- Calling **SendQueryQueueToGPU()** to actually send the data to the GPU
- Starting the coroutine **DoQueries()** and awaiting its completion

- Calling **GetQueryResult()** to optionally read back the results (and the number of results) to the CPU

Each entry in the results buffer consists of two integers. The first points to the index of the query - for example, if you executed 40 radius queries, then an index of 15 means that the result was gathered on the GPU while executing the 15th query. The second integer points to the index in the position buffer you provided (with the constructor) - this gives you a way to find the actual position again that were inside the query (if needed - commonly the indices are enough for further computations).

You can also use these results in further shaders (and it is recommended to do so by simply adding a **StructuredBuffer<int2>** to a regular shader).

#### 5.1.1 2D Grids / Bins

The following queries are supported:

- Radius Query (**QueueRadiusQuery** / **QueueRadiusQueries**)
- Rectangle Query (**QueueRectangleQuery** / **QueueRectangleQueries**)

Additionally, All-Queries can be queued (see DOTS Geometry manual for an explanation). However, as sending thousands of queries to the GPU would be expensive, they are also scheduled via a compute shader. Make sure that the query capacity vastly exceeds the number of positions. Additional care has to be taken when querying All-Queries and regular queries (which should be rare). In that case, the regular queries should be queued and sent before the All-Query compute shader dispatches.

The current All-Queries are:

- All-Radius Query (**QueueAllRadiusQuery**)
- All-Varying-Radius Query (**QueueAllVaryingRadiusQuery**)

The All-Varying Radius Query is a variant, where each point in the position buffer may search for a different radius (also stored in a Graphics-Buffer).

#### 5.1.2 3D Grids / Bins

The following queries are supported:

- Radius Query (**QueueRadiusQuery** / **QueueRadiusQueries**)
- Box Query (**QueueBoxQuery** / **QueueBoxQueries**)

Additionally, All-Queries can be queued (see DOTS Geometry manual for an explanation). However, as sending thousands of queries to the GPU would be expensive, they are also scheduled via a compute shader. Make sure that the query capacity vastly exceeds the number of positions. Additional care has to be taken when querying All-Queries and regular queries (which should be rare). In that case, the regular queries should be queued and sent before the All-Query compute shader dispatches.

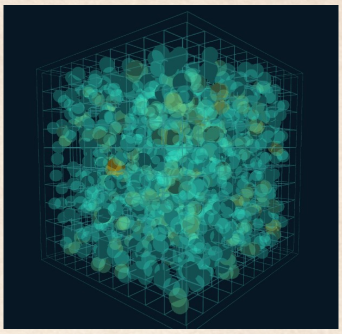


Figure 2: 3D All-Radius Query, warmer colors = more results

The current All-Queries are:

- All-Radius Query (**QueueAllRadiusQuery**)
- All-Varying-Radius Query (**QueueAllVaryingRadiusQuery**)

The All-Varying Radius Query is a variant, where each point in the position buffer may search for a different radius (also stored in a Graphics-Buffer).

### 5.2 KD-Trees

Similar to GPU Polygons, GPU KD-Trees may be created by using an existing **Native2DKDTree** or **Native3DKDTree** from DOTS Geometry. To do that, simply call the static methods **GPU2DKDTree.ConstructFrom()** or **GPU3DKDTree.ConstructFrom()**.

Figure 3: 3D KD-Tree, lines connect the input to NN

Not every operation of the CPU KD-Trees are yet supported. In fact, as of now the only queries the GPU-versions are capable of are **nearest neighbor searches**. This is due to the fact that for other queries like a simple radius-query, the amount of threads that have to be created is not known in advance, making the dispatching complicated (it would have to be done with multiple invocations). However, it is planned to at least support kNN as well in the future.

The NN Search is done simply by calling one of two methods:

- **ComputeNearestNeighbors()** (no reading back to the CPU)

- **GetNearestNeighbors()** (reading back the data to the CPU after completion)

You will need to provide a GraphicsBuffer containing the **queryPoints** and a container to store the resulting positions. Both methods will return a **GPUJobHandle**.

## 6 Tesselations

### 6.1 Voronoi Diagrams

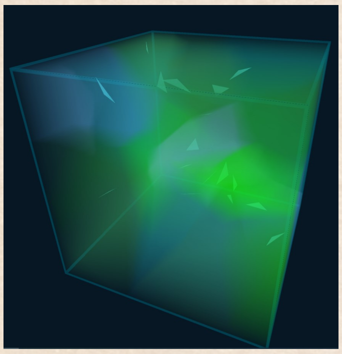


Figure 4: 3D Voronoi of Triangles - Raymarched

For an explanation of Voronoi Diagrams see the DOTS Geometry manual. All the methods for calculating them can be found in the two classes **Voronoi2DGPU** and **Voronoi3DGPU**. Each call requires the dimension of the grid or texture, the bounds in the world, a number of input sites (which can be **Points**, **Lines** or **Triangles**) and a distance metric (**Euclidean** or **Manhatten**).

For example, a 3D Voronoi Lookup Table of Triangles with an Euclidean Distance Metric, once it has been calculated, would allow you to lookup the nearest triangle in space to a given input position within the cube without any calculations, raycasts, etc. (see picture).

#### 6.1.1 Voronoi Lookup Tables

Voronoi Lookup Tables (**VoLTs**) are tables or grids containing the index to the closest site of the diagram. They can largely help with a quick approximate lookups of the nearest neighbors or sites from a given position.

Calculating this table is also possible with DOTS Geometry, yet because of the parallel nature of the computation, this version should be much preferred. In addition to the parameters mentioned above, you will also need to provide a NativeArray<int>for reading back the data.

The tables can be calculated for 2D and 3D and for the following types:

- Points of type float2 / float3 (**CalculateVoronoiLookupTable()**)
- Lines of type LineSegment2D / LineSegment3D (**CalculateLineVoronoiLookupTable()**)
- Triangles of type NativeTriangle2D / NativeTriangle3D (**CalculateTriangleVoronoiLookup()**)

### 6.1.2 Voronoi Textures

Voronoi Textures are the colored versions of the tables in texture form. Each index is assigned a color from an input array (you have to provide). The result is written to a RenderTexture of your choice, and no data is read back.

The textures can be computed for 2D and 3D and for the following types:

- Points of type float2 / float3 (**CalculateVoronoi()**)
- Lines of type LineSegment2D / LineSegment3D (**CalculateLineVoronoi()**)
- Triangles of type NativeTriangle2D / NativeTriangle3D (**CalculateTriangleVoronoi()**)

### 6.1.3 Voronoi SDFs

Signed Distance Fields store the distance to a shape for each point in a texture and, how the name suggests, with a + or **-** sign depending on if the point is **inside** a shape or **outside** a shape (e.g. inside a triangle or outside a triangle).

Similarly, a Voronoi SDF Texture stores the **closest** distance to each shape / site in the diagram and assigns a + or **-** sign to it depending if the point is inside our outside (if applicable).

They can be computed for 2D and 3D and for the following types:

- Points of type float2 / float3 (**CalculateVoronoiSDF()**)
- Lines of type LineSegment2D / LineSegment3D (**CalculateLineVoronoiSDF()**)
- Triangles of type NativeTriangle2D / NativeTriangle3D (**CalculateTriangleVoronoiSDF()**)

# 7 Special Queries

## 7.1 Brute Force Queries

Brute Force Queries are queries executed without any underlying data structure i.e. simply each point or shape is checked individually to test if it satisfies the query condition. For example, for checking which points are within a circle, each position in the complete data set is checked if it falls within.

This might seem inefficient at first (compared to a **Quadtree** for example), however, its simplicity makes it very easy to parallelize and fully utilize the GPU to its full potential. With the amount of cores available, for a single query, hundreds of thousands of points can easily be managed.

As long as the number of queries is low, this option is preferable to **GPU Bins**.

The classes responsible are **BruteForceQuery2DGPU** and **BruteForceQuery3DGPU**. Each query can either be executed with or without reading back data again.

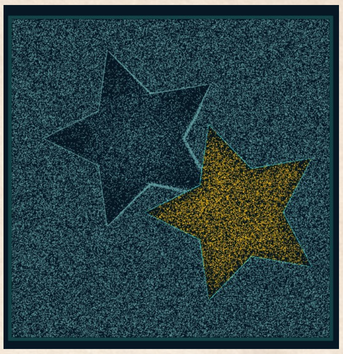


Figure 5: Brute Force Polygon Query

**2D** In the 2D case, the following three query types are supported:

- Circle (**ComputePointsInCircle()**)
- Rectangle (**ComputePointsInRectangle()**)
- Polygon (**ComputePointsInPolygon()**)

**3D** In the 3D case, the following two query types are supported:

- Sphere (**ComputePointsInSphere()**)
- Box (**ComputePointsInBox()**)

Each query will simply return the indices to the results (the same way as the **Bins**). You can then get back the position (optionally) by simply accessing it from the input buffer (you provided).

## 8 Intersections

Compute Shaders are provided in the package that allow you to intersect shapes with each other (in parallel). This can be used for example to highlight specific parts of the scene that intersect, or to do simple raycasts and/or physics calculation on the GPU.

### 8.1 Line Segment Intersections

The line intersection shaders in the package enable you to find out where a number of line segments (stored in a GraphicsBuffer) cut another shape (circles, boxes, planes, etc.). As the number of points a line segment may meet other shapes may vary, different structs are stored in the results.

The output of each query will be written either to a NativeArray (reading back) or to a GraphicsBuffer of your choice (the results can be used in a subsequent shader this way - recommended way). As the number of intersections is not known beforehand, this GraphicsBuffer has to have the target set to **Append** and sufficient capacity to hold all the intersections when creating it.

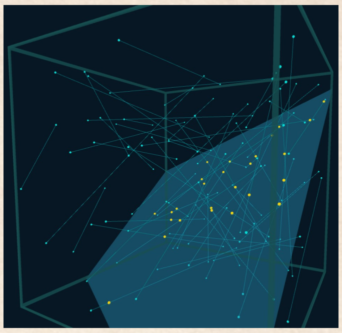


Figure 6: Line Segment - Plane Intersections

### 8.1.1 2D

In the 2D case, a GraphicsBuffer consisting of **LineSegment2D** has to be provided into method calls to the class **LineIntersection2DGPU**. Additionally, a buffer of the shapes you want to intersect has to be chosen as well. The data layout varies depending on the query (see **Mirror structures** for more information).

Three types of queries are available for the 2D case:

- Rectangles (**IntersectLineSegmentsWithRectangles()**)
- Circles (**IntersectLineSegmentsWithCircles()**)
- Line Segments (**IntersectLineSegments()**)

The last query type will simply check the intersections that exist between the 2D Line Segments and no further GraphicsBuffer holding any shapes is required.

### 8.1.2 3D

In the 3D case, a GraphicsBuffer consisting of **LineSegment3D** has to be provided into method calls to the class **LineIntersection3DGPU**. Additionally, a buffer of the shapes you want to intersect has to be chosen as well. The data layout varies depending on the query (see **Mirror structures** for more information).

Three types of queries are available for the 3D case:

- Boxes (**IntersectLineSegmentsWithBoxes()**)

- Spheres (**IntersectLineSegmentsWithSpheres()**)
- Planes (**IntersectLineSegmentsWithPlanes()**)

## 9 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**

For this package in particular, that involves shaders and geometry at the same time, please do not hesitate to ask whenever you encounter a problem or have trouble understanding something!

## 10 Future Plans

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

- Marching Squares and Cubes
- All-Rectangle Queries for Grids
- KD-Tree kNN
- Bitonic Sort
- Bindable Shader Properties

## Thank You!

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