

---

## Unity 2020.1 Mesh API improvements examples

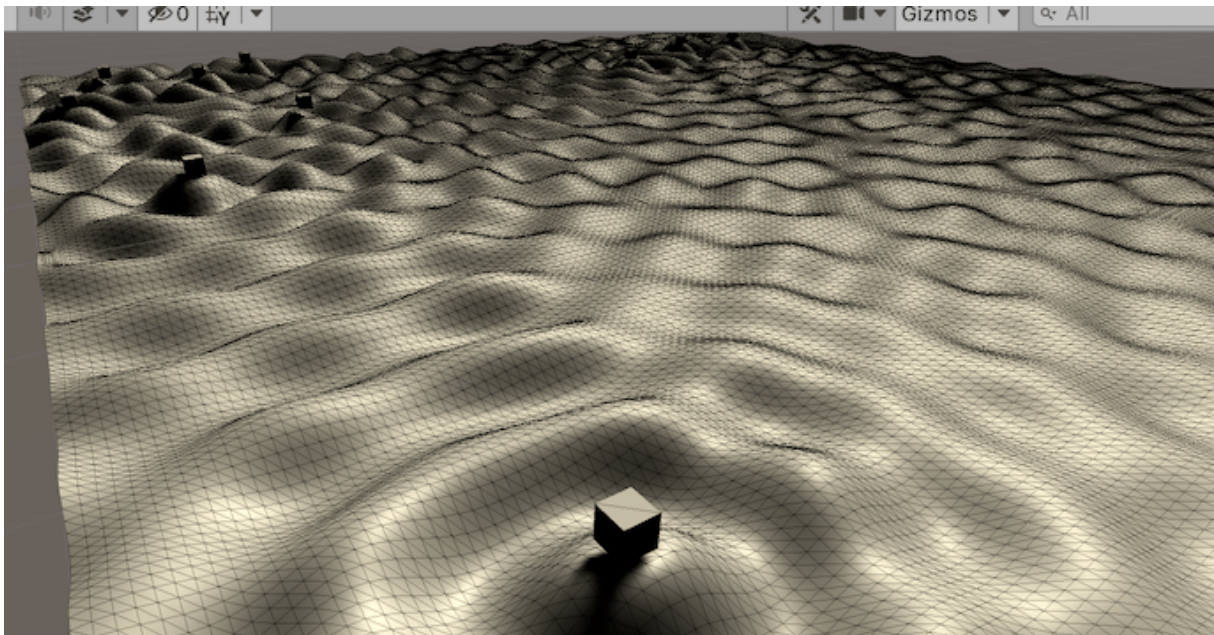
Unity 2020.1 adds [MeshData](#) APIs for C# Jobs/Burst compatible way of reading & writing Mesh data; see overview document.

This repository contains several small examples of that. Required Unity version is **2020.1** or later.

When on Unity 2021.2 or later version, the examples also show how to use GPU Compute Shaders to access and modify Mesh vertex buffers.

### Procedural Water/Wave Mesh

A simple example where a dense “water” surface mesh is updated every frame, based on positions on “wave source” objects.



[Assets/ProceduralWaterMesh](#) is the sample scene and code. Each vertex of the resulting mesh is completely independent of others, and only depends on positions of the “wave source” objects. Using C# Job System and Burst to compute all vertex positions in parallel brings some nice speedups. The sample also implements a similar computation using a GPU compute shader to modify the Mesh vertex buffer, for comparison.

Frame times on 400x400 water mesh, with 10 wave source objects, on 2019 MacBookPro (Core i9 2.4GHz, Radeon Pro 5500M); note that these are full frame times including rendering:

- Single threaded C#: 155ms

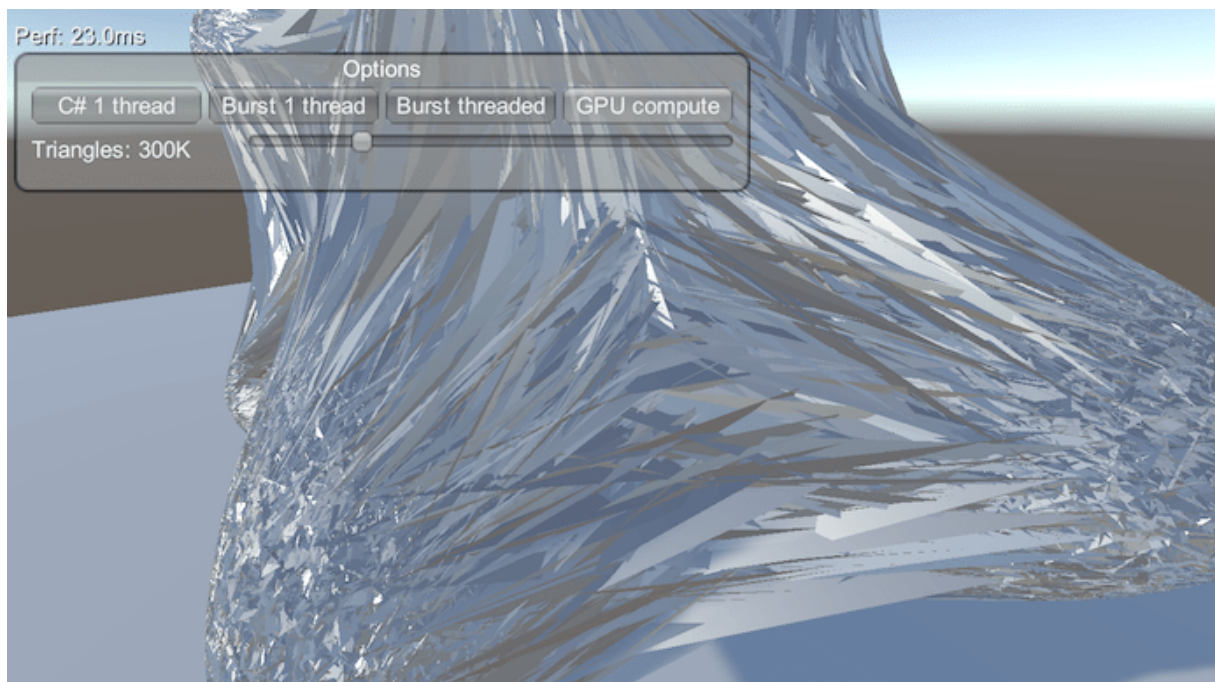
- 
- Single threaded Burst: 38ms
  - Multi threaded Burst: 9ms
  - GPU compute shader: 4ms

Same scene on Windows, AMD ThreadRipper 1950X 3.4GHz w/ 16 threads, GeForce GTX 1080Ti, DX11:

- Single threaded C#: 208ms
- Single threaded Burst: 45ms
- Multi threaded Burst: 11ms
- GPU compute shader: 2ms

## Noise Ball

A mesh with procedural simplex noise driven mesh. The mesh positions and normals are updated every frame, using either CPU code or a GPU compute shader. Based on NoiseBall2 by Keijiro Takahashi.



[Assets/NoiseBall](#) is the sample scene and code. Implementation is very similar to the water sample above.

Frame times on a 300 thousand triangle mesh, on 2019 MacBookPro; note that these are full frame times including rendering:

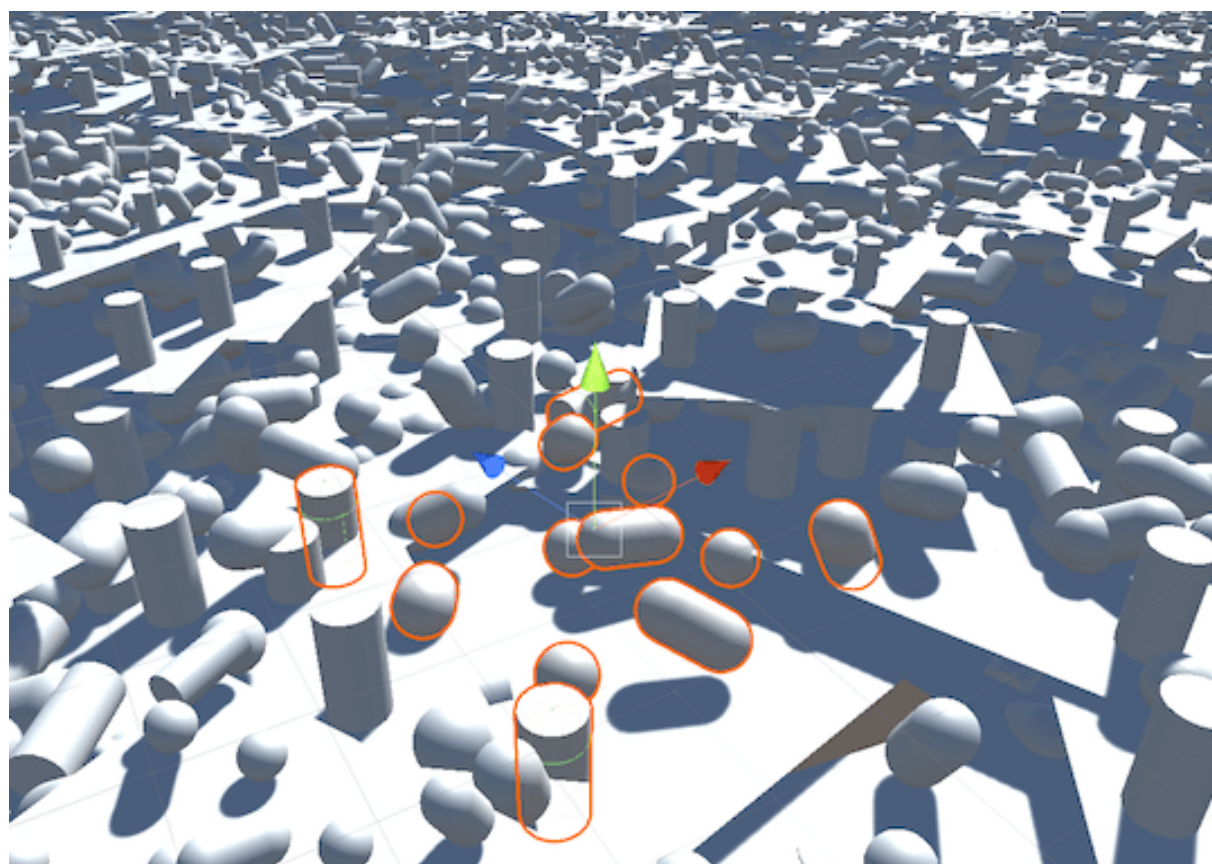
- 
- Single threaded C#: 2723ms
  - Single threaded Burst: 187ms
  - Multi threaded Burst: 22ms
  - GPU compute shader: 14ms

Same scene on Windows, AMD ThreadRipper 1950X 3.4GHz w/ 16 threads, GeForce GTX 1080Ti, DX11:

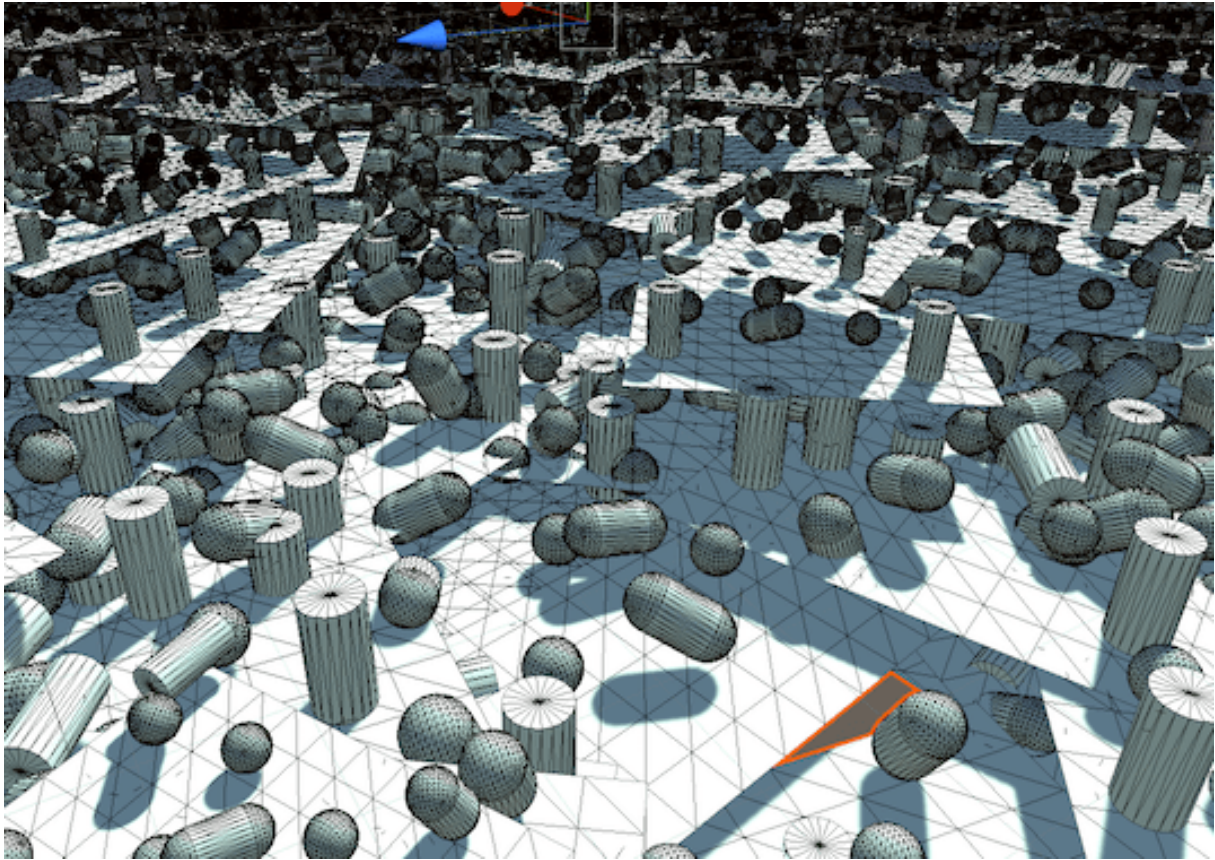
- Single threaded C#: 3368ms
- Single threaded Burst: 184ms
- Multi threaded Burst: 22ms
- GPU compute shader: 6ms

### **Combine Many Input Meshes Into One**

A more complex example, where for some hypothetical tooling there's a need to process geometry of many input Meshes, and produce an output Mesh. Here, all input meshes are transformed into world space, and a giant output mesh is created that is the union of all input meshes. This is very similar to how Static Batching in Unity works.







[Assets/CreateMeshFromAllSceneMeshes](#) is the sample scene and code. The script registers two menu items under **Mesh API Test** top-level menu; both do the same thing just one uses “traditional” Mesh API and does everything on the main thread, whereas the other uses 2020.1 new APIs to do it in C# Jobs with Burst.

Numbers for 11466 input objects, total 4.6M vertices, on 2018 MacBookPro (Core i9 2.9GHz):

- Regular API: 760ms (and 23k GC allocations totaling 640MB)
- Jobs+Burst: 60ms (0.3MB GC allocations)

Same scene on Windows, AMD ThreadRipper 1950X 3.4GHz w/ 16 threads:

- Regular API: 920ms
- Jobs+Burst: 70ms