Unity 2020.1 Mesh API Improvements

Unity 2019.3 added some Mesh API improvements, but not everything that we wanted :) This document describes further improvements done for Unity 2020.1.

Keep in mind that the scope of 2020.1 Mesh API improvements is "what already exists in the underlying engine, but has not been exposed to C# yet". In an ideal world we'd probably redo the whole thing about how Meshes and various GPU buffers work, but that is likely for some future full-DOTS graphics engine & API rewrite.

Scripting documentation of all the below things:
https://docs.unity3d.com/2020.1/Documentation/ScriptReference/Mesh.MeshData.html

Let us know what you think about this!

Read-only jobified data access

(when: 2020.1 alpha 17)

Taking a data snapshot of possibly multiple meshes at once:

```
Mesh.MeshDataArray data = Mesh.AcquireReadOnlyMeshData(aMesh);
// also: Mesh.AcquireReadOnlyMeshData(Mesh[] meshes);
// also: Mesh.AcquireReadOnlyMeshData(List<Mesh> meshes);
// …
data.Dispose();
```

AcquireReadOnlyMeshData takes a snapshot of input mesh(es) data. The struct has a `Length` and indexing `[]` operator, and corresponds to one or more meshes. Entries of MeshDataArray are `Mesh.MeshData` structs. It can be accessed from any thread (for example, using C# Job System) and reflects mesh data state as it was at the snapshot time. If the original Mesh is modified, the new modifications will **not** be reflected in the snapshot.

It is a fast operation if a mesh is not being modified — there are no memory allocations or data copies. However, if there are some non-disposed MeshData objects and the mesh is being modified, then the whole mesh data is copied into a new memory allocation.

The API is based on "array of things at once" primarily to reduce memory allocation tracking / job safety overhead, when you need to have a read-only access to many meshes at once.

All the functions of MeshData can be called from any thread, including C# and Burst jobs.

The MeshData has some usual info getters, similar to the ones on Mesh class:

```
int vertexCount { get; }
IndexFormat indexFormat { get; }
```

```
int subMeshCount { get; }
SubMeshDescriptor GetSubMesh(int index);
int vertexBufferCount { get; }
bool HasVertexAttribute(VertexAttribute a);
int GetVertexAttributeDimension(VertexAttribute a);
VertexAttributeFormat GetVertexAttributeFormat(VertexAttribute a);
```

If you know the exact layout of index or vertex data, direct “pointers” into mesh data can be obtained — without any memory allocations, data copies or format conversions:

```
NativeArray<T> GetIndexData<T>();
NativeArray<T> GetVertexData<T>(int stream=0);
```

That obviously needs struct T to match the exact vertex data layout, etc. In some cases you don’t know that, e.g. you are writing some code where completely arbitrary meshes in any vertex formats might be coming in. For these cases, there are copying (and optionally format converting) methods:

```
void GetVertices(NativeArray<Vector3> outVertices);
void GetNormals(NativeArray<Vector3> outNormals);
void GetTangents(NativeArray<Vector4> outTangents);
void GetColors(NativeArray<Color> outColors);
void GetColors(NativeArray<Color32> outColors);
void GetUVs(int channel, NativeArray<Vector2> outUVs);
void GetUVs(int channel, NativeArray<Vector3> outUVs);
void GetUVs(int channel, NativeArray<Vector4> outUVs);
```

In case you use custom data types (e.g. float3 from Unity.Mathematics), then NativeArray.Reinterpret works well:

```
var verts = new NativeArray<float3>(data.vertexCount, Allocator.Temp);
data.GetVertices(verts.Reinterpret<Vector3>());
// ...
verts.Dispose();
```

Here’s a snippet of actual code using the above API: **example project**. It creates a giant mesh out of all meshes in the scene (similar to static batching); first using the regular existing Mesh API, with everything on the main thread (0.75s for 4.6M vertices), and then using Jobs, Burst and this new API (0.08s for the same scene). There’s quite a bit more code for the jobified part, but it’s also much faster.

Jobified Mesh creation

(when: 2020.1 alpha 17)

Similar to read-only API above:

```
Mesh.MeshDataArray data = Mesh.AllocateWritableMeshData(meshCount);
```

```
// ...fill with actual data, jobified etc.
// ...then on the main thread:
Mesh.ApplyAndDisposeWritableMeshData(data, aMesh);
// also: Mesh.ApplyAndDisposeWritableMeshData(data, Mesh[] meshes);
// also: Mesh.ApplyAndDisposeWritableMeshData(data, List<Mesh> meshes);
```

AllocateWritableMeshData allocates an array of thread-safe mesh data storage objects. These can be filled from C# Jobs, Burst etc. using MeshData APIs:

```
void SetVertexBufferParams(
    int vertexCount, params VertexAttributeDescriptor[] attrs);
void SetIndexBufferParams(int indexCount, IndexFormat format);
int subMeshCount { set; }
void SetSubMesh(int index, SubMeshDescriptor desc, MeshUpdateFlags flags);
```

These set up vertex/index/subset information. Actual vertex data can be written into the array returned by `GetVertexData<T>()`, and similarly the index buffer data is written into `GetIndexData<T>()`.

Not Done Yet, But We Are Thinking About It

Would be nice to have **mesh buffers be usable by compute shaders**. Perhaps in one of these ways:

- Optional flags that would make mesh buffers be compute-writable/readable, and a way to get ComputeBuffer or GraphicsBuffer out of a Mesh, or
- A way to create “external” Mesh out of user supplied ComputeBuffer or GraphicsBuffer objects and some extra information. Similar to how “external” Textures can be created.

Adding “**Flags**” argument to SetVertexBuffer**Params**/SetIndexBufferParams. Could contain flags like:

- “Dynamic buffer”, replacing Mesh.MarkDynamic
- “Compute”, see above wrt Compute Shader usage
- “Keep/convert previous data”, try to convert previously existing data
- Add a flag to check if the mesh has previously been “baked” by physics.BakeMesh(int mesh_instanceID, bool convex), to not do a new bake of the mesh (Mesh.Bake()) when we set the collider on after adding the baked mesh inside sharedmesh of meshCollider.)

Misc TODO:

- NativeSlice overloads for GetFoo etc.