Thanks for choosing DX11Metaballs.

The **only** supported render API is **DirectX 11**(for now.)

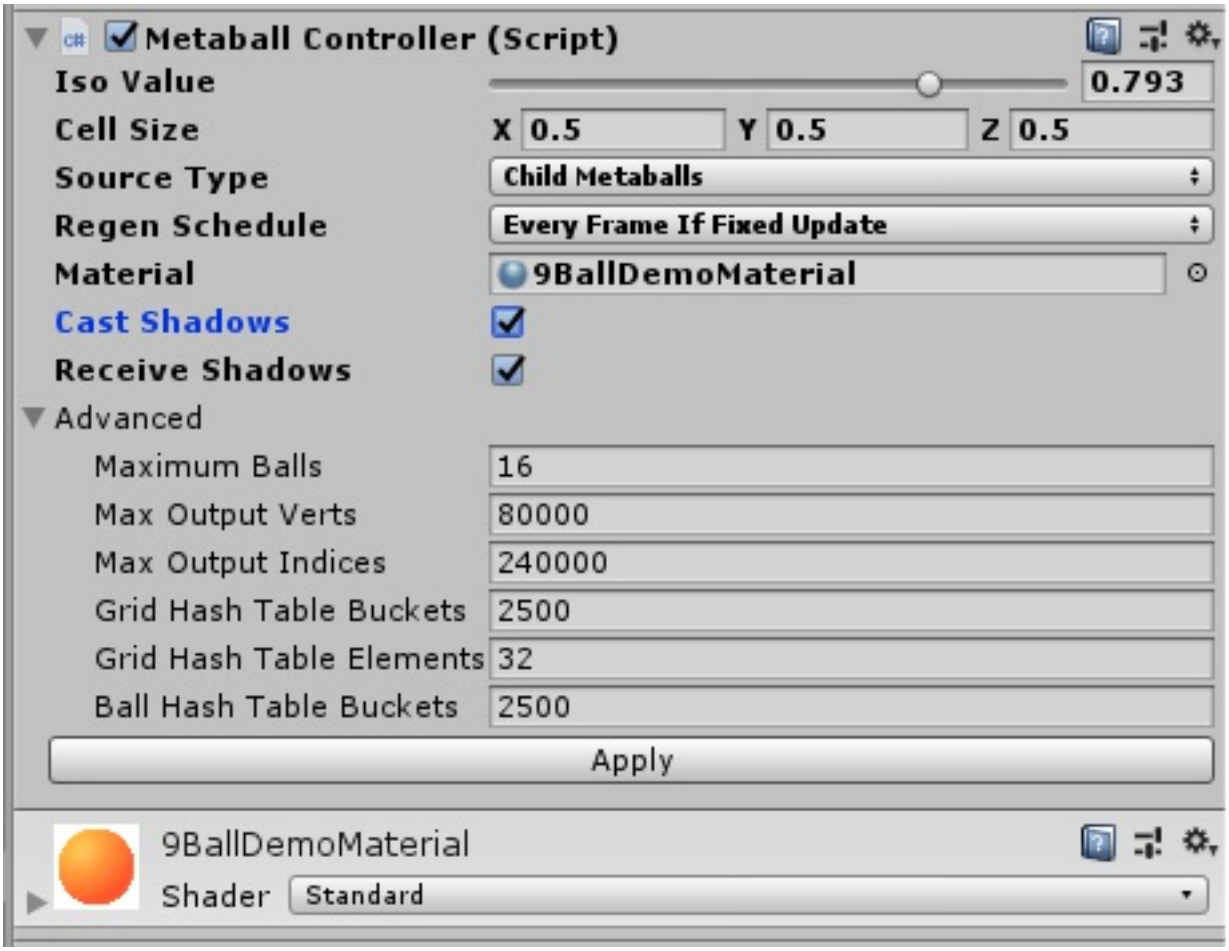
For feedback and support email dx11metaballs@gmail.com

The latest documentation can be found here

QuickStart

1. Add a new **MetaballController** from the **DX11Metaballs/Prefabs** Directory
2. Add some **Metaballs** as children of the **MetaballController** (**Dx11Metaballs/Prefabs/Metaball**)
3. Select a Material. The metaball surface should now be visible.

Using the UI



Basic

- **IsoValue** – This option alters the isoValue of the generated mesh. You can think of it as the 'blobbiness' value. A smaller value produces a more rigid surface where the individual Metaballs can be made out. Larger values lead to a surface where the balls will appear to 'stretch' to coalesce with each other.

- **Cell Size** – This option determines the Cell Size used for the generated mesh. The balls are split up into cells, and the **marching cubes algorithm** is applied to each cell. Smaller cells produce more detailed meshes, with corresponding increased in GPU usage. A good starting cellSize is to set (x,y,z) each to 1/3 of the average

ball radius.

- **Source Type** – This option determine the source of Metaball data. *'Child Metaballs'* selects all **Metaball** components that are children of the **MetaballController**. *'Particle System'* extracts the particle data from a selected **ParticleSystem**.

- **Regen Schedule** – This option determines how often the Mesh is regenerated. Select *'Every Frame'* if the metaball source data changes every frame. Choose *'Every Frame If Fixed Update'* if the metaballs are transformed by physics or only updated in **FixedUpdate()** callbacks.

- **Material** – Sets the Material applied to the produced mesh.

- **Cast Shadows** – Check to enable shadow casting

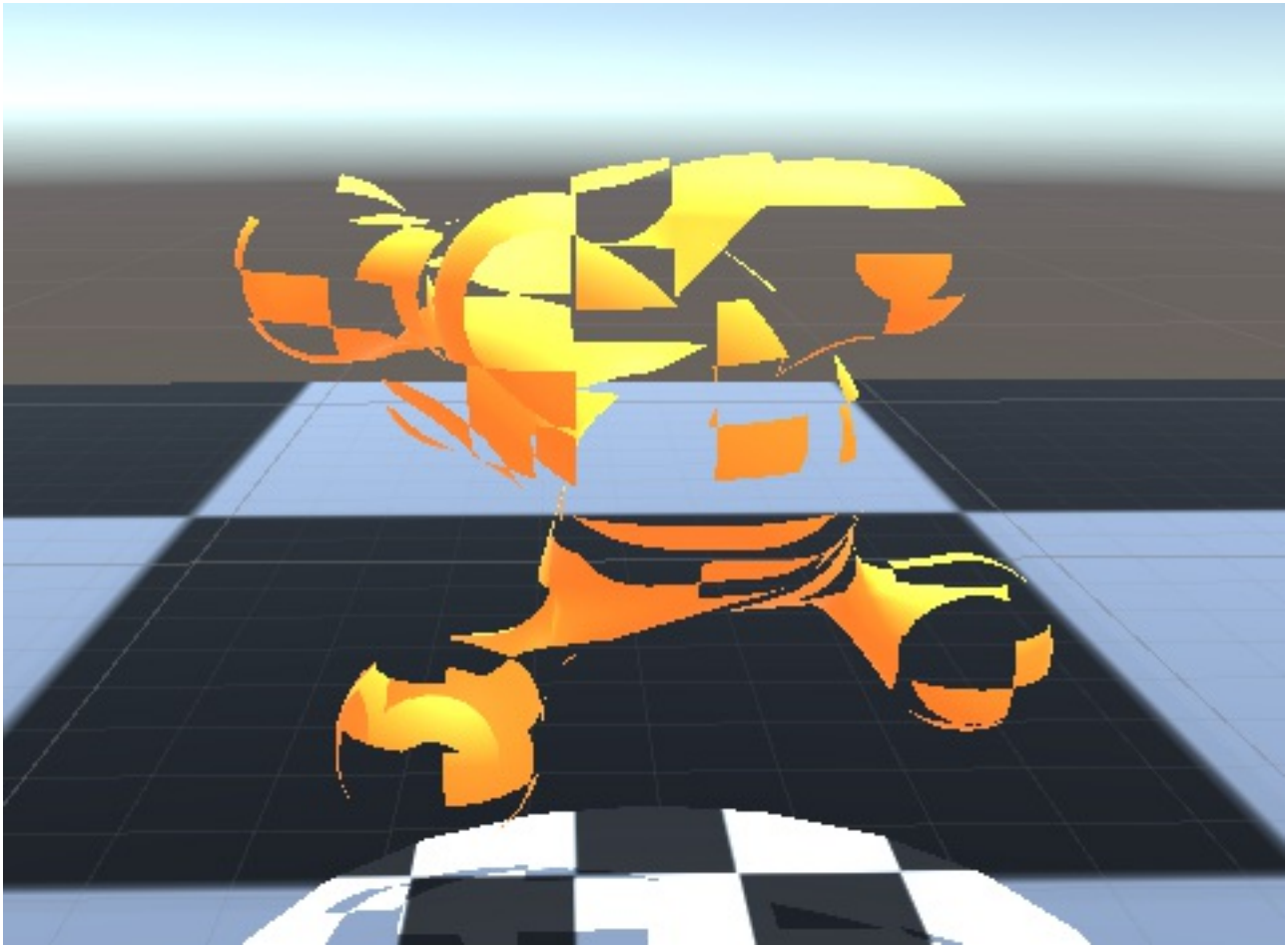
- **Receive Shadows** – Check to enable receiving shadows.

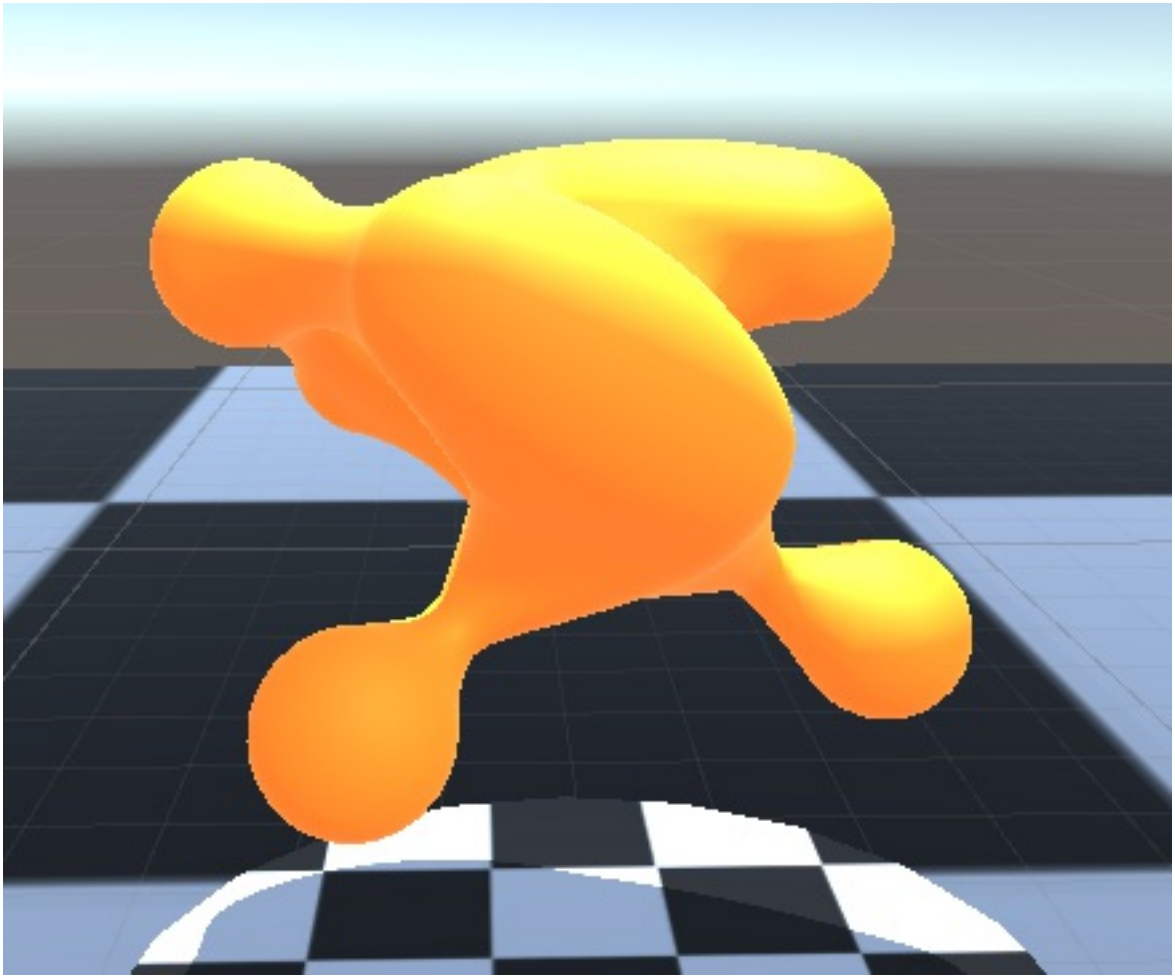
Advanced

Changes to values in the apply section only take effect once the apply button has been pressed.

- **Maximum Balls** – Sets the Maximum number of metaballs taken into account. Try and keep this value as close to your maximum value as possible as it will effect performance.

- **Max Output Verts** – Sets the maximum size of the produced Mesh's vertex buffer. If You see 'holes' in the output mesh, you probably need to increase this value.





- **Max Output Indices** – Sets the maximum size of the produce Mesh's index buffer. This should probably be around 3 times the size of **Max Output Verts**, though you can get away with less if the metaballs are densely packed.
- **Grid Hash Table Buckets** – The algorithm uses a spatial hash Buffer to store any Voxels that might be part of the output mesh. This determines the number of buckets in that buffer. This values should be around an order of magnitude (x10) larger than the maximum number of metaballs. However for particularly detailed meshes (small cell size) it may need to be increased.
- **Grid Hash Table Elements Per Bucket** – Used for handling collision in the spatial hash buffer. You Probably won't need to alter this value.

- **Ball Hash Table Buckets** – Similar to **Grid Hash Table Buckets** but for storing balls that may contribute to an area of the output mesh. This should be at least as large as Grid Hash Table Buckets, larger if many balls are close together.

How does this work anyway?

DX11Metaballs uses a modified version of the **Parrallel Marching Blocks** algorithm, which itself is a form of the Marching Cubes algorithm suited for GPUs.

- Space is split into an infinite 3D grid of cuboids or ‘buckets’.
- We take all the buckets which overlap with at least one metaball, and remove any which won’t overlap the output surface (isosurface in Marching cubes parlance).
- The remaining buckets are split up into cells (3x7x7 per bucket) which we apply Marching Cubes to. This produces two ComputeBuffers, one containing the mesh attributes (Position and Normals), and another buffer with indexes into the first.
- To draw the surface though, we need to copy this data into a Unity **Mesh**. We use the included DLL (NativeCopyBuffer) to get Directx11 to do the copy for us. (Which is why this is Dx11 only).
- Now we can call DrawMeshInstancedIndirect in Unity to draw the mesh. Hooray.