Draw call batching

Leave feedback Other Versions

To draw a GameObject on the screen, the engine has to issue a draw call to the graphics API (such as OpenGL or Direct3D). Draw calls are often resource-intensive, with the graphics API doing significant work for every draw call, causing performance overhead on the CPU side. This is mostly caused by the state changes done between the draw calls (such as switching to a different Material), which causes resource-intensive validation and translation steps in the graphics driver.

Unity uses two techniques to address this:

- **Dynamic batching**: for small enough Meshes, this transforms their vertices on the CPU, groups many similar vertices together, and draws them all in one go.
- **Static batching**: combines static (not moving) GameObjects into big Meshes, and renders them in a faster way.

Built-in batching has several benefits compared to manually merging GameObjects together; most notably, GameObjects can still be culled individually. However, it also has some downsides; static batching incurs memory and storage overhead, and dynamic batching incurs some CPU overhead.

Note: Dynamic batching is not compatible with graphics jobs (see Player Settings). If graphics jobs are enabled, dynamic batching are disabled in Standalone builds.

Material set-up for batching

Only GameObjects sharing the same Material can be batched together. Therefore, if you want to achieve good batching, you should aim to share Materials among as many different GameObjects as possible.

If you have two identical Materials which differ only in Texture, you can combine those Textures into a single big Texture. This process is often called Texture atlasing (see the Wikipedia page on Texture atlases for more information). Once Textures are in the same atlas, you can use a single Material instead.

If you need to access shared Material properties from the scripts, then it is important to note that modifying Renderer.material creates a copy of the Material. Instead, use Renderer.sharedMaterial to keep Materials shared.

Textures on them, but for the shadow caster rendering the textures are not relevant, so in this case they can be batched together.

Dynamic batching

Unity can automatically batch moving GameObjects into the same draw call if they share the same Material and fulfill other criteria. Dynamic batching is done automatically and does not require any additional effort on your side.

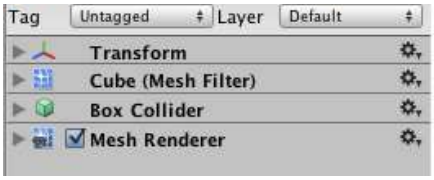
- Batching dynamic GameObjects has certain overhead per vertex, so batching is applied only to Meshes containing fewer than 900 vertex attributes in total.
  - If your Shader is using Vertex Position, Normal and single UV, then you can batch up to 300 verts, while if your Shader is using Vertex Position, Normal, UV0, UV1 and Tangent, then only 180 verts.
  - **Note**: attribute count limit might be changed in future.
- GameObjects are not batched if they contain mirroring on the transform (for example GameObject A with +1 scale and GameObject B with –1 scale cannot be batched together).
- Using different Material instances causes GameObjects not to batch together, even if they are essentially the same. The exception is shadow caster rendering.
- GameObjects with lightmaps have additional renderer parameters: lightmap index and offset/scale into the lightmap. Generally, dynamic lightmapped GameObjects should point to exactly the same lightmap location to be batched.
- Multi-pass Shaders break batching.
  - Almost all Unity Shaders support several Lights in forward rendering, effectively doing additional passes for them. The draw calls for “additional per-pixel lights” are not batched.
  - The Legacy Deferred (light pre-pass) rendering path has dynamic batching disabled, because it has to draw GameObjects twice.

Because it works by transforming all GameObject vertices into world space on the CPU, it is only an advantage if that work is smaller than doing a draw call. The resource requirements of a draw call depends on many factors, primarily the graphics API used. For example, on consoles or modern APIs like Apple Metal, the draw call overhead is generally much lower, and often dynamic batching cannot be an advantage at all.

Static batching

Static batching allows the engine to reduce draw calls for geometry of any size provided it shares the same material, and does not move. It is often more efficient than dynamic batching (it does not transform vertices on the CPU), but it uses more memory.

In order to take advantage of static batching, you need to explicitly specify that certain GameObjects are static and do not move,



Using static batching requires additional memory for storing the combined geometry. If several GameObjects shared the same geometry before static batching, then a copy of geometry is created for each GameObject, either in the Editor or at runtime. This might not always be a good idea; sometimes you have to sacrifice rendering performance by avoiding static batching for some GameObjects to keep a smaller memory footprint. For example, marking trees as static in a dense forest level can have serious memory impact.

Internally, static batching works by transforming the static GameObjects into world space and building a big vertex and index buffer for them. Then, for visible GameObjects in the same batch, a series of simple draw calls are done, with almost no state changes in between. Technically it does not save 3D API draw calls, but it saves on state changes between them (which is the resource-intensive part). Batches are limited to 64k vertices and 64k indices on most platforms (48k indices on OpenGLES, 32k indices on macOS).

Tips

Currently, only Mesh Renderers, Trail Renderers, Line Renderers, Particle Systems and Sprite Renderers are batched. This means that skinned Meshes, Cloth, and other types of rendering components are not batched.

Renderers only ever batch with other Renderers of the same type.

Semi-transparent Shaders usually require GameObjects to be rendered in back-to-front order for transparency to work. Unity first orders GameObjects in this order, and then tries to batch them, but because the order must be strictly satisfied, this often means less batching can be achieved than with opaque GameObjects.

Manually combining GameObjects that are close to each other can be a very good alternative to draw call batching. For example, a static cupboard with lots of drawers often makes sense to just combine into a single Mesh, either in a 3D modeling application or using Mesh.CombineMeshes.

- 2017–10–26 Page amended with limited editorial review
- Added note on dynamic batching being incompatible with graphics jobs in 2017.2

Did you find this page useful? Please give it a rating: