Optimizing graphics rendering in Unity games

For every frame that is rendered, the CPU does the following work:

The CPU checks every object in the scene to determine whether it should be rendered.

The CPU gathers information about every object that will be rendered and sorts this data into commands known as draw calls. A draw call contains data about a single mesh and how that mesh should be rendered; for example, which textures should be used. Under certain circumstances, objects that share settings may be combined into the same draw call. Combining data for different objects into the same draw call is known as batching.

The CPU creates a packet of data called a batch for each draw call. Batches may sometimes contain data other than draw calls, but these situations are unlikely to contribute to common performance issues and we therefore won’t consider these in this article.

For every batch that contains a draw call, the CPU now must do the following:

The CPU may send a command to the GPU to change a number of variables known collectively as the render state. This command is known as a SetPass call. A SetPass call tells the GPU which settings to use to render the next mesh. A SetPass call is sent only if the next mesh to be rendered requires a change in render state from the previous mesh.

The CPU sends the draw call to the GPU. The draw call instructs the GPU to render the specified mesh using the settings defined in the most recent SetPass call.

Under certain circumstances, more than one pass may be required for the batch. A pass is a section of shader code and a new pass requires a change to the render state. For each pass in the batch, the CPU must send a new SetPass call and then must send the draw call again.

Meanwhile, the GPU does the following work:

The GPU handles tasks from the CPU in the order that they were sent.

If the current task is a SetPass call, the GPU updates the render state.

If the current task is a draw call, the GPU renders the mesh. This happens in stages, defined by separate sections of shader code. This part of rendering is complex and we won’t cover it in great detail, but it’s useful for us to understand that a section of code called the vertex shader tells the GPU how to process the mesh’s vertices and then a section of code called the fragment shader tells the GPU how to draw the individual pixels.

This process repeats until all tasks sent from the CPU have been processed by the GPU.

We will use two tools to help us understand and fix our rendering performance problems: the Profiler window and the Frame Debugger.

Once we have established that our problems relate to rendering, we must also understand whether our game is CPU bound or GPU bound.

There are three types of thread involved in Unity’s rendering process: the main thread, the render thread and worker threads. The main thread is where the majority of CPU tasks for our game take place, including some rendering tasks. The render thread is a specialised thread that sends commands to the GPU. Worker threads each perform a single task, such as culling or mesh skinning. Which tasks are performed by which thread depends on our game’s settings and the hardware on which our game runs. For example, the more CPU cores our target hardware has, the more worker threads can be spawned. For this reason, it is very important to profile our game on target hardware; our game may perform very differently on different devices.

The Graphics jobs option in Player Settings determines whether Unity uses worker threads to carry out rendering tasks that would otherwise be done on the main thread and, in some cases, the render thread.

Sending commands to the GPU

The time taken to send commands to the GPU is the most common reason for a game to be CPU bound. This task is performed on the render thread on most platforms, although on certain platforms (for example, PlayStation 4) this may be performed by worker threads.

The most costly operation that occurs when sending commands to the GPU is the SetPass call. If our game is CPU bound due to sending commands to the GPU, reducing the number of SetPass calls is likely to be the best way to improve performance.

Reducing the number of batches and/or making more objects share the same render state will, in most cases, reduce the number of SetPass calls.

Reducing the number of SetPass calls will, in most cases, improve CPU performance.

There are, broadly, three ways of reducing the number of batches and SetPass calls. We will look more in-depth at each one of these:

Reducing the number of objects to be rendered will likely reduce both batches and SetPass calls.

Reducing the number of times each object must be rendered will usually reduce the number of SetPass calls.

Combining the data from objects that must be rendered into fewer batches will reduce the number of batches.

Reducing the number of objects being rendered

Simply reducing the number of visible objects in our scene can be an effective solution.

We can reduce our camera’s draw distance using the camera’s Far Clip Plane property.

For a more fine-grained approach to hiding objects based on distance, we can use our camera’s Layer Cull Distances property to provide custom culling distances for objects that are on separate layers. This approach can be useful if we have lots of small foreground decorative details; we could hide these details at a much shorter distance than large terrain features.

We can use a technique called occlusion culling to disable the rendering of objects that are hidden by other objects. Unity’s occlusion culling is not suitable for all scenes, can lead to additional CPU overhead and can be complex to set up, but it can greatly improve performance in some scenes. n addition to using Unity’s occlusion culling, we can also implement our own form of occlusion culling by manually deactivating objects that we know cannot be seen by the player.

Reducing the number of times each object must be rendered

As a general rule, Deferred Rendering is likely to be a better choice if our game runs on higher-end hardware and uses a lot of realtime lights, shadows and reflections. Forward Rendering is likely to be more suitable if our game runs on lower-end hardware and does not use these features.

(渲染路径研究)

（优化：光照，阴影Shadow Distance，光照探针）

Combining objects into fewer batches

A batch can contain the data for multiple objects when certain conditions are met. To be eligible for batching, objects must:

Share the same instance of the same material

Have identical material settings (i.e., texture, shader and shader parameters)

There are a few different techniques for batching eligible objects:

Static batching is a technique that allows Unity to batch nearby eligible objects that do not move. A good example of something that could benefit from static batching is a pile of similar objects, such as boulders. This page of the Unity Manual contains instructions on setting up static batching in our game. Static batching can lead to higher memory usage so we should bear this cost in mind when profiling our game.

Dynamic batching is another technique that allows Unity to batch eligible objects, whether they move or not. There are a few restrictions on the objects that can be batched using this technique. These restrictions are listed, along with instructions, on this page of the Unity Manual. Dynamic batching has an impact on CPU usage that can cause it to cost more in CPU time than it saves. We should bear this cost in mind when experimenting with this technique and be cautious with its use.

Batching Unity’s UI elements is a little more complex, as it can be affected by the layout of our UI. This video from Unite Bangkok 2015 gives a good overview of the subject and this guide to optimizing Unity UI provides in-depth information on how to ensure that UI batching works as we intend it to.

GPU instancing is a technique that allows large numbers of identical objects to be very efficiently batched. There are limitations to its use and it is not supported by all hardware, but if our game has many identical objects onscreen at once we may be able to benefit from this technique. This page of the Unity Manual contains an introduction to GPU instancing in Unitywith details of how to use it, which platforms support it and the circumstances under which it may benefit our game.

Texture atlasing is a technique where multiple textures are combined into one larger texture. It is commonly used in 2D games and UI systems, but can also be used in 3D games. If we use this technique when creating art for our game, we can ensure that objects share textures and are therefore eligible for batching. Unity has a built-in texture atlasing tool called Sprite Packer for use with 2D games.

It is possible to manually combine meshes that share the same material and texture, either in the Unity Editor or via code at runtime. When combining meshes in this way, we must be aware that shadows, lighting and culling will still operate on a per-object level; this means that a performance increase from combining meshes could be counteracted by no longer being able to cull those objects when they would otherwise not have been rendered. If we wish to investigate this approach, we should examine the the Mesh.CombineMeshes function. The CombineChildren script in Unity’s Standard Assets package is an example of this technique.

We must be very careful when accessing Renderer.material in scripts. This duplicates the material and returns a reference to the new copy. Doing so will break batching if the renderer was part of a batch because the renderer no longer has a reference to the same instance of the material. If we wish to access a batched object’s material in a script, we should use Renderer.sharedMaterial.