Optimizing Graphics Performance

Good performance is critical to the success of many games. Below are some simple guidelines for maximizing the speed of your game's graphical rendering.

Where are the graphics costs

The graphical parts of your game can primarily cost on two systems of the computer: the GPU or the CPU. The first rule of any optimization is to find **where the performance problem is**; because strategies for optimizing for GPU vs. CPU are quite different (and can even be opposite - it's quite common to make GPU do more work while optimizing for CPU, and vice versa).

Typical bottlenecks and ways to check for them:

- GPU is often limited by **fillrate** or memory bandwidth.
  - Does running the game at lower display resolution make it faster? If so, you're most likely limited by fillrate on the GPU.
- CPU is often limited by the number of things that need to be rendered, also known as "**draw calls**".
  - Check "draw calls" in Rendering Statistics window; if it's more than several thousand (for PCs) or several hundred (for mobile), then you might want to optimize the object count.

Of course, these are only the rules of thumb; the bottleneck could as well be somewhere else. Less typical bottlenecks:

- Rendering is not a problem, neither on the GPU nor the CPU! For example, your scripts or physics might be the actual problem. Use Profiler to figure this out.
- GPU has too many vertices to process. How many vertices are "ok" depends on the GPU and the complexity of vertex shaders. Typical figures are "not more than 100 thousand" on mobile, and "not more than several million" on PC.
- CPU has too many vertices to process, for things that do vertex processing on the CPU. This could be skinned meshes, cloth simulation, particles etc.

CPU optimization - draw call count

In order to render any object on the screen, the CPU has some work to do - things like figuring out which lights affect that object, setting up the shader & shader parameters, sending drawing commands to the graphics driver, which then prepares the

commands to be sent off to the graphics card. All this "per object" CPU cost is not very cheap, so if you have lots of visible objects, it can add up.

So for example, if you have a thousand triangles, it will be much, much cheaper if they are all in one mesh, instead of having a thousand individual meshes one triangle each. The cost of both scenarios on the GPU will be very similar, but the work done by the CPU to render a thousand objects (instead of one) will be significant.

In order to make CPU do less work, it's good to reduce the visible object count:

- Combine close objects together, either manually or using Unity's draw call batching.
- Use less materials in your objects, by putting separate textures into a larger texture atlas and so on.
- Use less things that cause objects to be rendered multiple times (reflections, shadows, per-pixel lights etc., see below).

Combine objects together so that each mesh has at least several hundred triangles and uses only one **Material** for the entire mesh. It is important to understand that combining two objects which don't share a material does not give you any performance increase at all. The most common reason for having multiple materials is that two meshes don't share the same textures, so to optimize CPU performance, you should ensure that any objects you combine share the same textures.

However, when using many pixel lights in the Forward rendering path, there are situations where combining objects may not make sense, as explained below.

GPU: Optimizing Model Geometry

When optimizing the geometry of a model, there are two basic rules:

- Don't use any more triangles than necessary
- Try to keep the number of UV mapping seams and hard edges (doubled-up vertices) as low as possible

Note that the actual number of vertices that graphics hardware has to process is usually not the same as the number reported by a 3D application. Modeling applications usually display the geometric vertex count, i.e. the number of distinct corner points that make up a model. For a graphics card, however, some geometric vertices will need to be split into two or more logical vertices for rendering purposes. A vertex must be split if it has multiple normals, UV coordinates or vertex colors. Consequent-

ly, the vertex count in Unity is invariably higher than the count given by the 3D application.

While the amount of geometry in the models is mostly relevant for the GPU, some features in Unity also process models on the CPU, for example mesh skinning.

Lighting Performance

Lighting which is not computed at all is always the fastest! Use Lightmapping to "bake" static lighting just once, instead of computing it each frame. The process of generating a lightmapped environment takes only a little longer than just placing a light in the scene in Unity, **but**:

- It is going to run a lot faster (2-3 times for 2 per-pixel lights)
- And it will look a lot better since you can bake global illumination and the lightmapper can smooth the results

In a lot of cases there can be simple tricks possible in shaders and content, instead of adding more lights all over the place. For example, instead of adding a light that shines straight into the camera to get "rim lighting" effect, consider adding a dedicated "rim lighting" computation into your shaders directly.

Lights in forward rendering

Per-pixel dynamic lighting will add significant rendering overhead to every affected pixel and can lead to objects being rendered in multiple passes. On less powerful devices, like mobile or low-end PC GPUs, avoid having more than one **Pixel Light** illuminating any single object, and use lightmaps to light static objects instead of having their lighting calculated every frame. Per-vertex dynamic lighting can add significant cost to vertex transformations. Try to avoid situations where multiple lights illuminate any given object.

If you use pixel lighting then each mesh has to be rendered as many times as there are pixel lights illuminating it. If you combine two meshes that are very far apart, it will increase the effective size of the combined object. All pixel lights that illuminate any part of this combined object will be taken into account during rendering, so the number of rendering passes that need to be made could be increased. Generally, the number of passes that must be made to render the combined object is the sum of the number of passes for each of the separate objects, and so nothing is gained by combining. For this reason, you should not combine meshes that are far enough apart to be affected by different sets of pixel lights.

During rendering, Unity finds all lights surrounding a mesh and calculates which of those lights affect it most. The Quality Settings are used to modify how many of the lights end up as pixel lights and how many as vertex lights. Each light calculates its importance based on how far away it is from the mesh and how intense its illumination is. Furthermore, some lights are more important than others purely from the game context. For this reason, every light has a **Render Mode** setting which can be set to **Important** or **Not Important**; lights marked as **Not Important** will typically have a lower rendering overhead.

As an example, consider a driving game where the player's car is driving in the dark with headlights switched on. The headlights are likely to be the most visually significant light sources in the game, so their Render Mode would probably be set to **Important**. On the other hand, there may be other lights in the game that are less important (other cars' rear lights, say) and which don't improve the visual effect much by being pixel lights. The Render Mode for such lights can safely be set to **Not Important** so as to avoid wasting rendering capacity in places where it will give little benefit.

Optimizing per-pixel lighting saves both CPU and the GPU: the CPU has less draw calls to do, and the GPU has less vertices to process and pixels to rasterize for all these additional object renders.

GPU: Texture Compression and Mipmaps

Using Compressed Textures will decrease the size of your textures (resulting in faster load times and smaller memory footprint) and can also dramatically increase rendering performance. Compressed textures use only a fraction of the memory bandwidth needed for uncompressed 32bit RGBA textures.

Use Texture Mip Maps

As a rule of thumb, always have Generate Mip Maps enabled for textures used in a 3D scene. In the same way Texture Compression can help limit the amount of texture data transfered when the GPU is rendering, a mip mapped texture will enable the GPU to use a lower-resolution texture for smaller triangles.

The only exception to this rule is when a texel (texture pixel) is known to map 1:1 to the rendered screen pixel, as with UI elements or in a 2D game.

LOD and Per-Layer Cull Distances

In some games, it may be appropriate to cull small objects more aggressively than

large ones, in order to reduce both the CPU and GPU load. For example, small rocks and debris could be made invisible at long distances while large buildings would still be visible.

This can be either achieved by Level Of Detail system, or by setting manual per-layer culling distances on the camera. You could put small objects into a separate layer and setup per-layer cull distances using the Camera.layerCullDistances script function.

Realtime Shadows

Realtime shadows are nice, but they can cost quite a lot of performance, both in terms of extra draw calls for the CPU, and extra processing on the GPU. For further details, see the Shadows page.

GPU: Tips for writing high-performance shaders

A high-end PC GPU and a low-end mobile GPU can be literally hundreds of times performance difference apart. Same is true even on a single platform. On a PC, a fast GPU is dozens of times faster than a slow integrated GPU; and on mobile platforms you can see just as large difference in GPUs.

So keep in mind that GPU performance on mobile platforms and low-end PCs will be much lower than on your development machines. Typically, shaders will need to be hand optimized to reduce calculations and texture reads in order to get good performance. For example, some built-in Unity shaders have their "mobile" equivalents that are much faster (but have some limitations or approximations - that's what makes them faster).

Below are some guidelines that are most important for mobile and low-end PC graphics cards:

Complex mathematical operations

Transcendental mathematical functions (such as **pow**, **exp**, **log**, **cos**, **sin**, **tan**, etc) are quite expensive, so a good rule of thumb is to have no more than one such operation per pixel. Consider using lookup textures as an alternative where applicable.

It is not advisable to attempt to write your own **normalize**, **dot**, **inversesqrt** operations, however. If you use the built-in ones then the driver will generate much better code for you.

Keep in mind that alpha test (**discard**) operation will make your fragments slower.

Floating point operations

You should always specify the precision of floating point variables when writing custom shaders. It is **critical** to pick the smallest possible floating point format in order to get the best performance. Precision of operations is completely ignored on many desktop GPUs, but is critical for performance on many mobile GPUs.

If the shader is written in Cg/HLSL then precision is specified as follows:

- **float** - full 32-bit floating point format, suitable for vertex transformations but has the slowest performance.
- **half** - reduced 16-bit floating point format, suitable for texture UV coordinates and roughly twice as fast as **highp**.
- **fixed** - 10-bit fixed point format, suitable for colors, lighting calculation and other high-performance operations and roughly four times faster than **highp**.

If the shader is written in GLSL ES then the floating point precision is specified specified as **highp**, **mediump**, **lowp** respectively.

For further details about shader performance, please read the Shader Performance page.

Simple Checklist to make Your Game Faster

- Keep vertex count below 200K..3M per frame when targetting PCs, depending on the target GPU
- If you're using built-in shaders, pick ones from Mobile or Unlit category. They work on non-mobile platforms as well; but are simplified and approximated versions of the more complex shaders.
- Keep the number of different materials per scene low - share as many materials between different objects as possible.
- Set **Static** property on a non-moving objects to allow internal optimizations like static batching.
- Do not use **Pixel Lights** when it is not necessary - choose to have only a single (preferably directional) pixel light affecting your geometry.
- Do not use dynamic lights when it is not necessary - choose to bake lighting instead.
- Use compressed texture formats when possible, otherwise prefer 16bit textures over 32bit.

- Do not use fog when it is not necessary.
- Learn benefits of Occlusion Culling and use it to reduce amount of visible geometry and draw-calls in case of complex static scenes with lots of occlusion. Plan your levels to benefit from ccclusion culling.
- Use skyboxes to "fake" distant geometry.
- Use pixel shaders or texture combiners to mix several textures instead of a multi-pass approach.
- If writing custom shaders, always use smallest possible floating point format:
    - **fixed** / **lowp** - for colors, lighting information and normals,
    - **half** / **mediump** - for texture UV coordinates,
    - **float** / **highp** - avoid in pixel shaders, fine to use in vertex shader for position calculations.
- Minimize use of complex mathematical operations such as **pow**, **sin**, **cos** etc. in pixel shaders.
- Choose to use less textures per fragment.

See Also

- Draw Call Batching
- Modeling Characters for Optimal Performance
- Rendering Statistics Window

Page last updated: 2012-07-29