How To Plan Optimizations with Unity*

Abstract

Unity provides a number of tools and settings to help make games perform smoothly. For this project, we chose ones we thought could prove to be troublesome and analyzed how they affected game performance on Intel® graphics processors.

We put ourselves in the shoes of a game developer learning how to use Unity. We wanted to stumble into performance pitfalls and then determine how to work through issues with Unity's built-in performance mechanisms. One of Unity's strengths is the ability to create content quickly, but when considering performance, especially on mobile and tablet devices, the developer needs to slow down and plan out how to utilize the built in performance mechanisms. This paper prepares new and existing Unity users with performance considerations when building your levels/games, and offers new ways to build.

Introduction

Creating games within Unity is fairly simple. Unity offers a store where you can purchase items like meshes, pre-written scripts, game demos, or even full games. For the purposes of my testing, I was concerned with manipulating an existing game to find areas where performance gains could or could not be achieved. I dove into the Unity Tech Demo called Boot Camp, free for download in the assets store, to see what kind of trouble I could get into.

I used Unity 3.0 to create the game settings and run all of the scenes. The testing was performed on a 3rd generation Intel® Core™ processor-based computer with Intel® HD Graphics 4000. The test results are not applicable to mobile devices.

Quality Manager

Unity has extra render settings for games found in: Edit->Project Settings->Quality menu (Figure 1). They are customizable render settings that can be modified for individual needs. Unity has helpful online documentation for explaining what the Quality Settings are and how to modify these settings through Unity's scripting API.

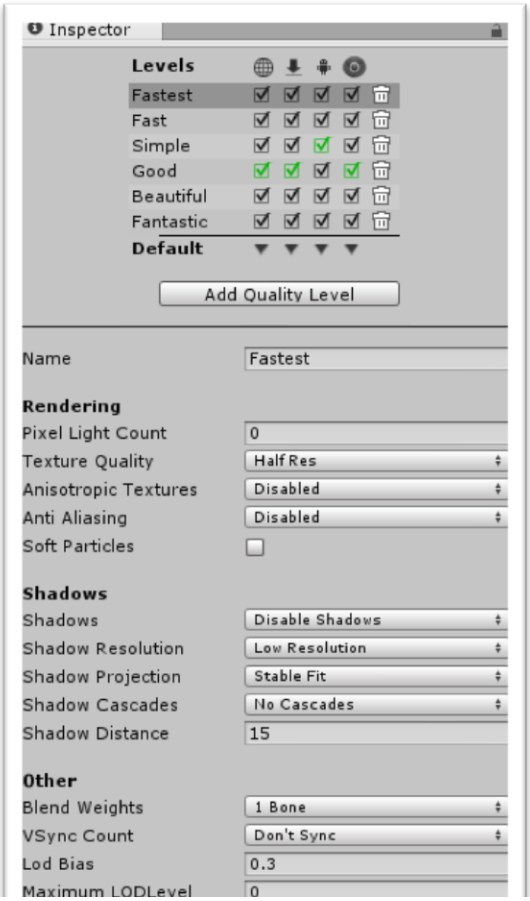

Figure 1: *The Tags and Layers available through the Edit->Project Settings->Tag inspector*

As for my task to find optimizations with Unity, I decided to mess around with some of the Quality Settings to see what kind of gains or losses I could find, although I did not test all of the different options available.

Texture Quality

The Quality Settings Inspector has a drop down menu where you select render resolutions for your textures. You can choose from 1/8, ¼, ½, or Full Resolution. To see the performance gains/losses between different texture resolutions, I took frame rate captures of a sample scene, testing all of Unity's default Quality Settings (Fastest, Fast, Good, etc.), while adjusting only the Texture Quality between each capture. Figures 2 and 3 show a comparison between a scene with 1/8 Texture Resolution and Full Resolution.

Figure 2: *Unity* Scene Boot Camp running at 1/8 resolution*

Figure 3: *Unity* Scene Boot Camp running at full resolution*

We took a frames per second (FPS) capture using Intel® Graphics Performance Analyzers (Intel® GPA) after changing the texture resolution. Looking at the Fantastic setting (Table 1), you can see the performance did not change much by varying the texture sizes.

Texture Quality:	1/8	¼	½	Full
Fantastic	72	73	72	69

Texture Quality:	1/8	¼	½	Full
Fastest	151	151	150	150
Fast	165	164	163	161
Simple	155	153	151	151
Good	130	130	130	128
Beautiful	113	114	114	113
Fantastic	72	73	72	69

Table 1: *Illustrates the change in FPS while switching between Unity's* provided texture qualities*

Although an Intel® graphics-based PC's performance is not affected by texture size changes, there are other things to consider, like the total amount of memory on the device and its usage by the application.

Shadow Distance

Shadow distance is a setting that changes the culling distance of the camera being used for the shadows of game objects. Game objects within the shadow distance's value from the camera have their shadows sent for rendering, whereas objects that are not within the shadow distance value do not have their shadows drawn.

Depending on the settings used, shadows can adversely affect performance due the amount of processing they require. To test the impact of Shadow Distance:

- Set up a sample scene
- Set scene to a Unity default quality setting
- Adjust the shadow distance incrementally and take FPS captures using Intel GPA
- Select different Unity default quality settings and repeat shadow distance captures

This test did not use the Fastest and Fast Quality Levels because those default to turning shadows off.

Figure 4: *This is a setting found under the Inspector menu of Edit->Project Settings->Quality*

Figure 5: *Unity* Tech Demo Boot Camp*

Shadow Distance:	*0*	*1*	*5*	*10*	*15*	*25*	*50*
Simple	124	114	96	92	82	77	73
Good	79	63	56	55	52	50	46
Beautiful	39	35	34	33	32	30	28
Fantastic	35	32	31	30	29	28	26

Table 2: *FPS results from changing the Shadow Distance of Unity* Tech Demo, Boot Camp*

Shadows significantly impact performance. The data shows the FPS dropped by almost half when going from a distance of 0 to 50 Simple mode. It is important to consider if game objects can actually be seen and to make sure you are not drawing shadows unnecessarily. The shadow distance and other shadow settings can be controlled during gameplay via Unity scripting and can accommodate numerous situations. Although we only tested the effects of shadow distance, we expect similar performance deltas occur when changing the other settings under Shadow in the Quality settings.

Layers

All game objects inside Unity are assigned to a layer upon creation. They are initially assigned to the default layer, as show in Figure 6, but you can create your own unique layers. There are two ways to do this. You can simply click on the box next to Layer and select Add New Layer. You can also go to Edit->Project Settings->Tags.

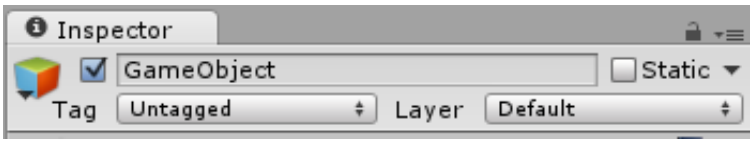

Figure 6: *The Layer menu found inside the Inspector of a game object*

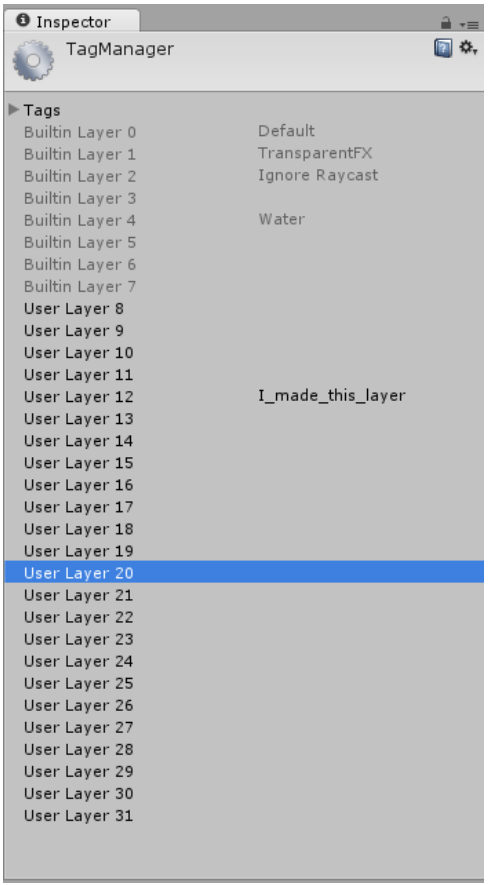

Figure 7: *The Tag Manager via the inspector menu*

From the inspector window (Figure 7) you can create a new layer and specify which layer number you want it to belong to. Both methods lead you to the same Tag Manager window. Once a layer is created, game objects can be assigned to them by choosing the desired layer under the options box next to Layer under that game object's inspector window. This way, you can group objects in common layers for later use and manipulation. Keep in mind what layers are and how to create and modify them for when I talk about a few other layer features later in the paper.

Layer Cull Distances

Your camera will not render game objects beyond the camera's clipping plain in Unity. There is a way, through Unity scripting, to have certain layers set to a shorter clipping plane.

```
function Start () {
    var distances = new float[32];
    // Set up layer 10 to cull at 15 meters distance.
    // All other layers use the far clip plane distance.
    distances[10] = 15;
    camera.layerCullDistances = distances;
}
```

Figure 8*: Sample script taken from Unity*'s Online Documentation showing how to modify a layer's culling distance*

It takes a bit of work to set up game objects so they have a shorter culling distance. First, place the objects onto a layer. Then, write a script to modify an individual layer's culling distance and attach it to the camera. The sample script in Figure 8 shows how a float array of 32 is created to correspond to the 32 possible layers available for creation under the Edit->Project Settings->Tags. Modifying a value for an index in your array and assigning it to camera.layerCullDistances will change the culling distance for the corresponding layer. If you do not assign a number for an index, the corresponding layer will use the camera's far clip plane.

To test performance gains from layerCullDistances, I set up three scenes filled with small, medium, and large objects in terms of complexity. The scenes were arranged with a number of identical game objects grouped together and placed incrementally further and further away from the camera. I used Intel GPA to take FPS captures while incrementing the layer culling distance each time, adding another group of objects to the capture, i.e., the first capture had one group of objects, whereas the sixth capture had six groups of objects.

Figures 9, 10, and 11 show the scenes I used for testing with the different types of objects.

Boots: Poly – 278 Vertices – 218

Figure 9: *Test scene filled with low polygon and vertices count boot objects*

T-Rex's: Poly – 4398 Vertices – 4400

Figure 10: *Test scene filled with medium polygon and vertices count dinosaur objects*

Airplane: Poly - 112,074 Vertices - 65,946

Figure 11: *Test scene filled with large polygon and vertices count airplane objects*

Tables 3, 4, and 5 show the change in FPS for each of the scenes tested.

# Simple models:	50	100	150	200	250	300
Fastest	355	308	302	274	225	209
Fast	310	288	279	279	283	276
Simple	295	273	262	266	265	261
Good	143	140	138	135	133	131
Beautiful	79	77	76	75	74	73
Fantastic	71	69	68	68	67	66

Table 3: *Data collected from the scene with boots (Figure 9)*

# Moderate Models:	5	10	15	20	25	30
Fastest	329	285	277	263	243	212
Fast	295	256	246	236	210	202
Simple	288	250	240	230	214	191
Good	142	134	130	121	119	111
Beautiful	82	77	73	68	66	60
Fantastic	77	72	68	66	63	57

Table 4: *Data collected from the scene with dinosaurs (Figure 10)*

# Complex models:	1	2	3	4	5	6
Fastest	213	145	116	93	76	67
Fast	213	157	124	100	83	73
Simple	207	154	121	98	81	73
Good	133	120	102	84	73	64
Beautiful	69	49	39	32	26	23
Fantastic	66	47	37	31	25	22

Table 5: *Data collected from the scene with airplanes (Figure 11)*

Number of obj's drawn:	50/5/1	100/10/2	150/15/3	200/20/4	250/25/5	300/30/6
Fantastic Mode:						
Simple	71	69	68	68	67	66
Moderate	77	72	68	66	63	57
Complex	66	47	37	31	25	22

Table 6: *Fantastic mode data from all the test scenes*

This data shows that performance gains can be achieved from using the layerCullDistances feature within Unity.

Table 6 illustrates how having more objects on the screen impacts performance, especially with complex objects. As a game developer, using the layerCullDistances proves to be very beneficial for performance if utilized properly. For example, smaller objects with a complex mesh that are farther away from the camera can be set up to only draw when the camera is close enough for the objects to be distinguished. While planning and designing a level, the developer needs to consider things like mesh complexity and the visibility of objects at a greater distance from the camera. By planning ahead, you can achieve greater benefits from using layerCullDistances.

Camera

I explored Unity's camera, focusing on its settings and features. I toyed with some of the options under its GUI and examined other features and addons.

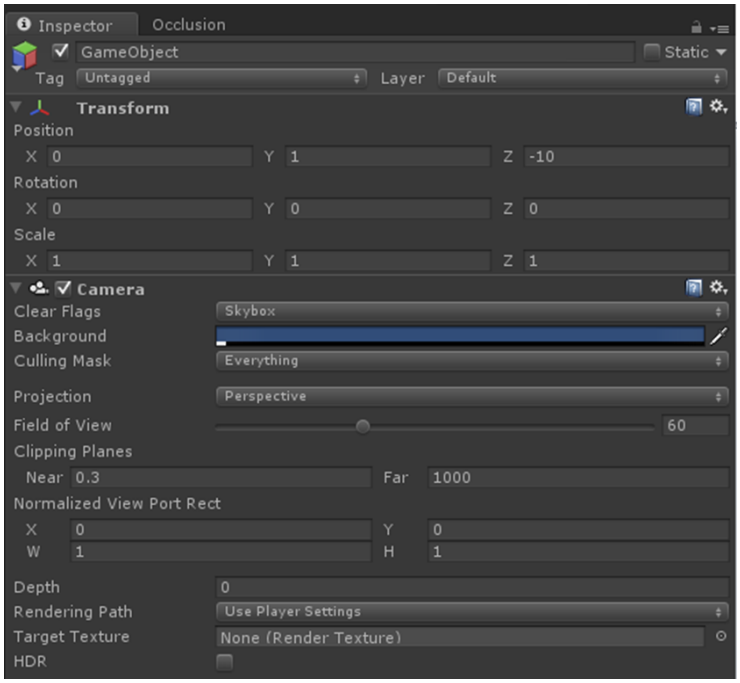

Figure 12: *The Inspector menu that appears while having a camera selected*

When creating a new scene, by default, there is only one camera game object labeled *Main Camera*. To create or add another camera, first create an empty game object by going to: Game Object->Create Empty. Then select the newly created empty object and add the camera component: Components->Rendering->Camera.

Unity's camera comes with a host of functionality inside its GUI, as shown in Figure 12. The features I chose to explore were: Rendering Path and HDR.

Render Path

The Render Path tells Unity how to handle light and shadow rendering in the game. Unity offers three render types, listed from highest cost to least; Deferred (Pro Only), Forward, and Vertex Lit rendering. Each renderer handles light and shadow a little bit differently, and they require different amounts of CPU and GPU processing power. It's important to understand the platform and hardware you want to develop for so you can choose a renderer and build your scene or game accordingly. If you pick a renderer that is not supported by the graphics hardware, Unity will automatically lower the rendering path to a lower fidelity.

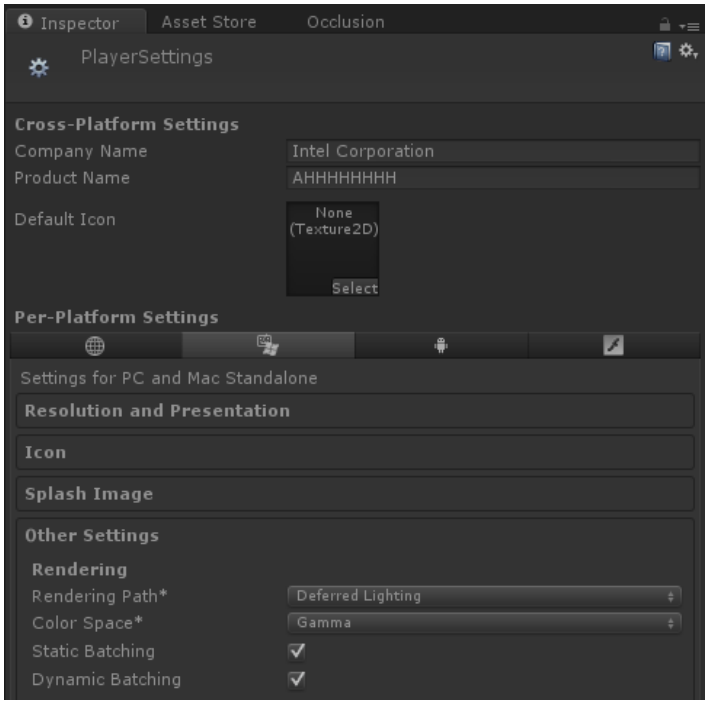

Figure 13: *Player Settings Inspector window*

The Rendering Path can be set in two different ways. The first is under the Edit->Project Settings->Player (Figure 13). You will find the Rendering Path drop down box under the Others Settings tab. The second is from the Camera Inspector GUI (Figure 14). Choosing something other then 'Use Player Settings' will override the rendering path set in your player settings, but only for that camera. So it is possible to have multiple cameras using different rendering buffers to draw the lights and shadows.

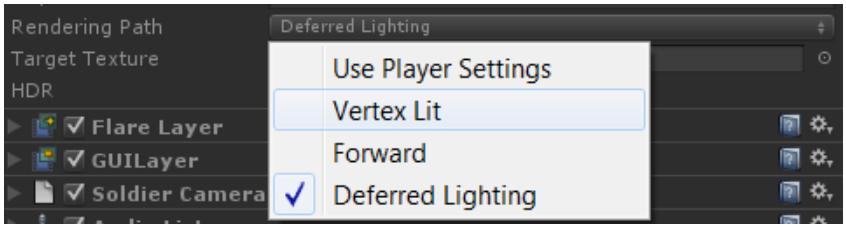

Figure 14: *The drop down box from selecting the Rendering Path under the Camera GUI*

Developers should know that these different light rendering paths are included in Unity and how each handles rendering. The reference section at the end of this document has links to Unity's online documentatoin. Make sure you know your target audience and what type of platform they expect their game to be played on. This knowledge will help you select a rendering path appropriate to the platform. For example, a game designed with numerous light sources and image effects that uses deferred rendering could prove to be unplayable on a computer with a lower end graphics card. If the target audience is a casual gamer, who may not possess a graphics card with superior processing power, this could also be a problem. It is up to developers to know the target platform on which they expect their game to be played and to choose the lights and rendering path accordingly.

HDR (High Dynamic Range)

In normal rendering, each pixel's red, blue, and green values are represented by a decimal number between 0 and 1. By limiting your range of values for the R, G, and B colors, lighting will not look realistic. To achieve a more naturalistic lighting effect, Unity has an option called HDR, which when activated, allows the number values representing the R, G, and B of a pixel to exceed their normal range. HDR creates an image buffer that supports values outside the range of 0 to 1, and performs post-processing image effects, like bloom and flares. After completing the post-processing effects, the R, G, and B values in the newly created image buffer are reset to values within the range of 0 to 1 by the Unity Image Effect Tonemapping. If Tonemapping is not executed when HDR is included, the pixels could be out of the normal accepted range and cause some of the colors in your scene to look wrong in comparison to others.

Pay attention to a few performance issues when using HDR. If using Forward rendering for a scene, HDR only will be active if image effects are present. Otherwise, turning HDR on will have no effect. Using Deferred rendering supports HDR regardless.

If a scene is using Deferred rendering and has Image Effects attached to a camera, HDR should be activated. Figure 15 compares the draw calls for a scene with image effects and deferred

rendering while HDR is turned on and HDR is off. With HDR off and image effects included, you see a larger number of draw calls then if you include image effects with HDR turned on. In Figure 15, the number of draw calls are represented by the individual blue bars, and the height of each blue bar reveals the amount of GPU time each draw call took.

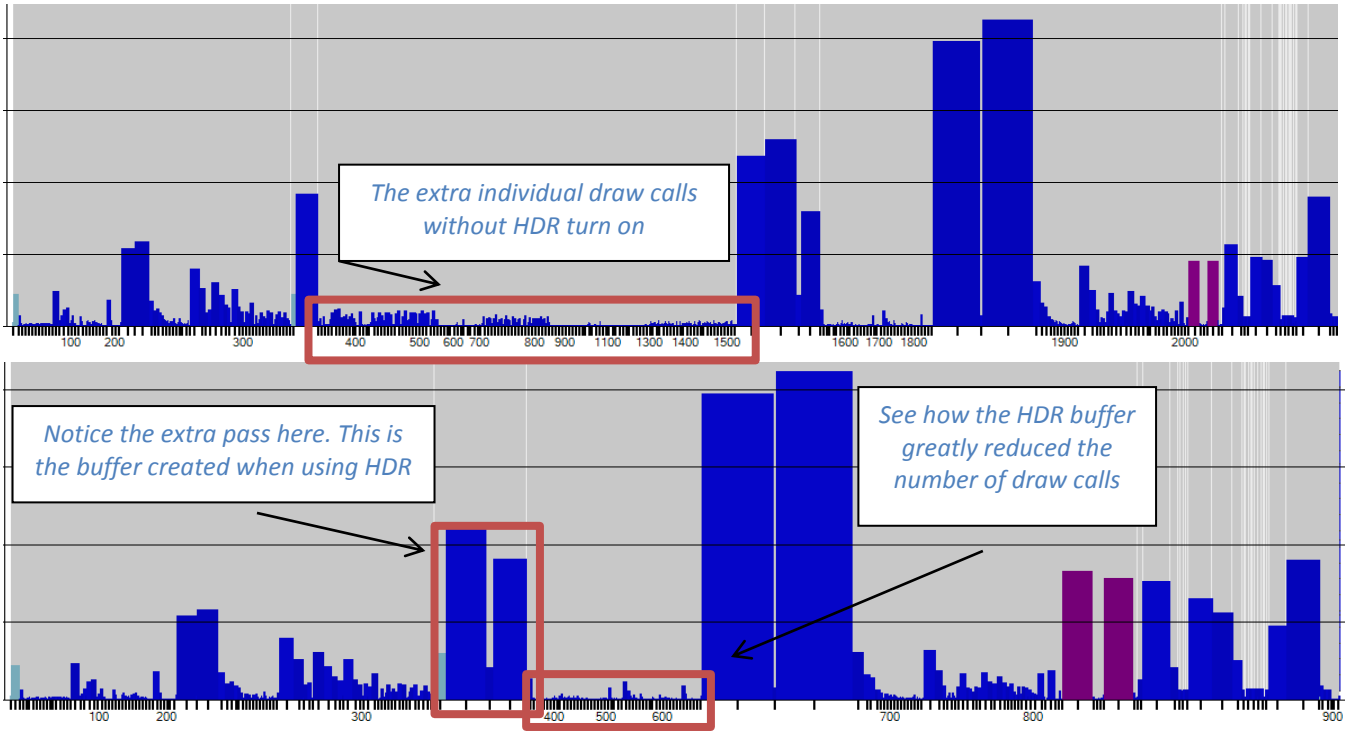

Figure 15: *The capture from Intel® Graphics Performance Analyzers with HDR OFF shows over 2000 draw call, whereas the capture with HDR ON has a little over 900 draw calls.*

Read over Unity's HDR documentation and understand how it affects game performance. You should also know when it makes sense to use HDR to ensure you are receiving its full benefits.

Image Effects

Unity Pro comes with a range of image effects that enhance the look of a scene. Add Image Effects assets, even after creating your project, by going to Assets->Import Package->Image Effects. Once imported, there are two ways to add an effect to the camera. Click on your camera game object, then within the camera GUI, select Add Component, then Image Effects. You can also click on your camera object from the menu system by going to Component->Image Effect.

SSAO – Screen Space Ambient Occlusion

Screen space ambient occlusion (SSAO) is an image effect included in Unity Pro's Image Effect package. Figure 16 shows the difference between a scene with SSAO off and on. The images look similar, but performance is markedly different. The scene without SSAO ran at 32 FPS and the scene with SSAO ran at 24 FPS, a 25% decrease.

Figure 16: *A same level comparison with SSAO off (top) vs. SSAO on (bottom)*

Be careful when adding image effects because they can negatively affect performance. For this document we only tested the SSAO image effect but expect to see similar results with the other image effects.

Occlusion Culling

Occlusion Culling disables object rendering not only outside of the camera's clipping plane, but for objects hidden behind other objects as well. This is very beneficial for performance because it cuts back on the amount of information the computer needs to process, but setting up occlusion culling is not straightforward. Before you set up a scene for occlusion culling, you need to understand the terminology.

Occluder – An object marked as an occluder acts as a barrier that prevents objects marked as occludees from being rendered.

Occludee – Marking a game object as an occludee will tell Unity not to render the game object if blocked by an occluder.

For example, all of the objects inside a house could be tagged as occludees and the house could be tagged as an occluder. If a player stands outside of that house, all the objects inside marked as occludees will not be rendered. This saves CPU and GPU processing time.

Unity documents Occlusion Culling and its setup. You can find the link for setup information in the references section.

To show the performance gains from using Occlusion Culling, I set up a scene that had a single wall with highly complex meshed objects hidden behind. I took FPS captures of the scene while using Occlusion Culling and then without it. Figure 17 shows the scene with the different frame rates.

Figure 17: *The image on the left has no Occlusion Culling so the scene takes extra time to render all the objects behind the wall resulting in an FPS of 31. The image on the right takes advantage of Occlusion Culling so the objects hidden behind the wall will be rendered resulting in an FPS of 126.*

Occlusion culling requires developers to do a lot of manual setup. They need to also consider occlusion culling during game design as to make the game's configuration easier and performance gains greater.

Level of Detail (LOD)

Level of Detail (LOD) allows multiple meshes to attach to a game object and provides the ability to switch between meshes the object uses based on camera distance. This can be beneficial for complex game objects that are really far away from the camera. The LOD can automatically simplify the mesh to compensate. To see how to use and setup LOD, check out Unity's online documentation. The link to it is in the reference section.

To test the performance gains from LOD, I built a scene with a cluster of houses with 3 different meshes attached to them. While standing in the same place, I took an FPS capture of the houses when the most complex mesh was attached. I then modified the LOD distance so the next lesser mesh appeared, and took another FPS capture. I did this for the three mesh levels and recorded my findings as shown in Table 5.

Figures 18, 19, and 20 show the three varying levels of mesh complexity as well as the number of polygons and vertices associated with each mesh.

Best Quality – LOD 0

Building A
- Vert – 7065
- Poly – 4999

Building B
- Vert - 5530
- Poly – 3694

Figure 18: *LOD level 0. This is the highest LOD level that was set with the more complex building meshes*

Medium Quality – LOD 1

Building A
- Vert– 6797
- Poly – 4503

Building B
- Poly – 5476
- Vert - 3690

Figure 19: *LOD level 1. The next step on the LOD scale; this level was set with the medium complexity meshes*

Low Quality – LOD 2

Building A
- Vert– 474
- Poly – 308

Building B
- Poly – 450
- Vert - 320

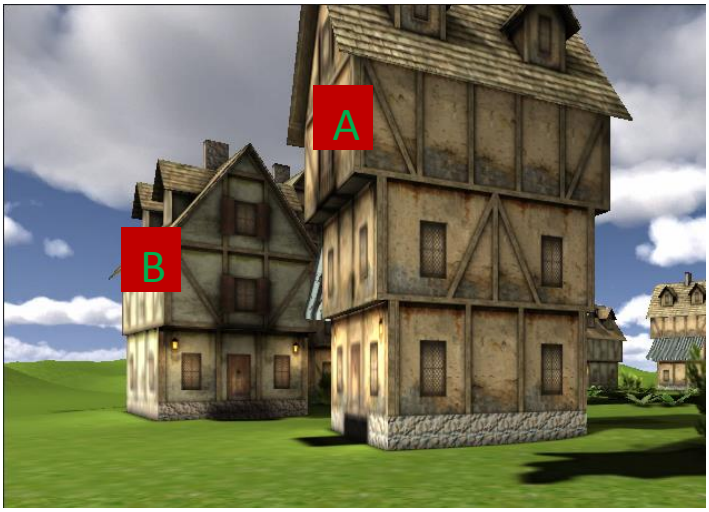

Figure 20: *LOD level 2.This LOD level was the last one used and contained the least complex meshes for the buildings*

As I switched between the different LOD models, I took FPS captures for comparison (Table 7).

LOD	*Level 0*	*Level 1*	*Level 2*
FPS	160	186	240

Table 7: *LOD FPS comparison switching between lower model meshes*

Table 7 shows the increased performance gains from setting up and using LOD. The FPS capture shows significant performance gains when using lower quality meshes. This however, can take a lot of extra work on the 3-D artists, who must produce multiple models. It is up to the game designer to decide whether or not spending the extra time for more models is worth the performance gains.

Batching

Having numerous draws calls can cause overhead on the CPU and slow performance. The more objects on the screen, the more draw calls to be made. Unity has a feature called Batching that combines game objects in to a single draw call. Static Batching affects static objects, and Dynamic Batching is for those that move. Dynamic Batching happens automatically, if all requirements are met (see batching documentation), whereas Static Batching needs to be created.
There are some requirements for getting the objects to draw together for both Dynamic and Static Batching, all of which are covered in Unity's Batching document listed in the references section.

To test the performance gains of Static Batching, I set up a scene with complex airplane game objects (Figure 21) and took FPS captures of the airplanes both with batching and without batching (Table 8).

Figure 21: *Static Batching Test scene filled with very complex airplane meshes*

Static Batching:	***Off***	***On***
FPS	24	58
Draw Calls	5144	390

Table 8: *Showing the difference between FPS and Draw Calls while turning static batching on and off for the test scene (Figure 21)*

Unity's batching mechanism comes in two forms, Dynamic and Static. To fully see the benefits from batching, plan to have as many objects as possible batched together for single draw calls. Refer to Unity's batching documentation and know what qualifies an object for dynamic or static batching.

Conclusion

While Unity proves to be fairly simple to pick up and develop with, it can also be very easy to get yourself into performance trouble. Unity provides a number of tools and settings to help make games perform smoothly, but not all of them are as intuitive and easy to set up as others. Likewise, Unity has some settings that when turned on or used inappropriately can negatively affect game performance. An important part of developing with Unity is to have a plan before starting because some of the performance features require manual setup and can be much more challenging to implement if not planned at the project's creation.

References

Quality Settings Documentation:
http://docs.unity3d.com/Documentation/Components/class-QualitySettings.html

Quality Settings Scripting API:
http://docs.unity3d.com/Documentation/ScriptReference/QualitySettings.html

Tech Demo Bootcamp:
http://u3d.as/content/unity-technologies/bootcamp/28W

Level of Detail Documentation:
http://docs.unity3d.com/Documentation/Manual/LevelOfDetail.html

Occlusion Culling Documentation:
http://docs.unity3d.com/Documentation/Manual/OcclusionCulling.html

Batching Documentation:
http://docs.unity3d.com/Documentation/Manual/DrawCallBatching.html

Rendering Path Documentation:
http://docs.unity3d.com/Documentation/Manual/RenderingPaths.html

Intel GPA:
http://software.intel.com/en-us/vcsource/tools/intel-gpa

About the Author

John Wesolowski, Intern

The focus of the group that I worked for at Intel was to enable Intel® chipsets for upcoming technology, with a focus on video games. It was our task to test the latest and upcoming video games to find potential bugs or areas of improvement inside the Intel® architecture or in the video game.

Outside of work, my all-time favorite activity used to be playing Halo* 2 online with my friends but since Microsoft shut down all Xbox LIVE* service for original Xbox* games, my friends and I like to LAN Halo 2 whenever we can. I also enjoy playing poker and flying kites. I am currently attending California State University, Monterey Bay and pursuing a degree in Computer Science and Information Technology.

Notices

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to:
http://www.intel.com/design/literature.htm

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark* and MobileMark*, are measured using specific computer systems, components, software, operations, and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Any software source code reprinted in this document is furnished under a software license and may only be used or copied in accordance with the terms of that license.

*Other names and brands may be claimed as the property of others.