Optimizing Unity Games for Mobile Platforms

Angelo Theodorou

Software Engineer

Brains Eden, 28th June 2013

Agenda

- Introduction
  - The author
  - ARM Ltd.
  - What do you need to have
  - What do you need to know
- Identify the bottleneck
  - CPU
  - Vertex processing
  - Fragment processing
  - Bandwidth
- The Unity Profiler (Pro only)
- Example application overview
  - Geometry, LOD, Particles, Texture, Overdraw, Lights scenes

The Author

- Angelo Theodorou
- Software Engineer
- Media Processing Division, ARM® Ltd.
  - Mali Ecosystem Use-Cases team
- B.Sc. In Computer Science
- Studied at Università degli Studi di Napoli “*Federico II*”, Italy

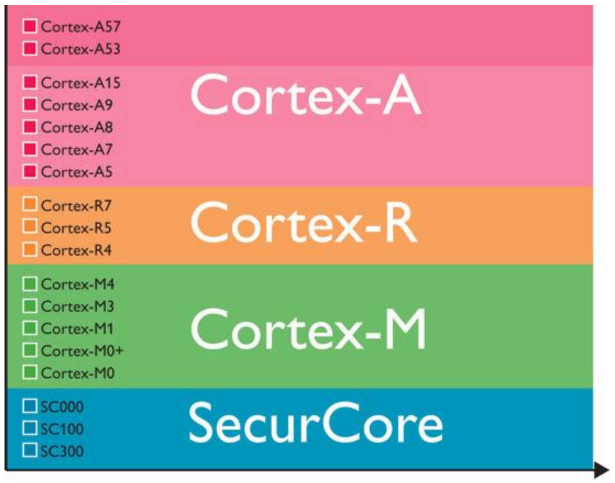
- In my previous company I have worked on From Cheese
  - Website: http://www.fromcheese.com
  - Grand Prize winner at the Samsung Smart App Challenge 2012
  - Made with Unity 3.5
  - Responsible for SPen integration and additional programming

ARM Ltd.

- Founded at the end of 1990, 13 engineers and a CEO
- Main goal was to design low power embedded 32bit processors, but to never build them
- Now more than 2500 employees
  - More than 1000 in Cambridge, the headquarters
- Hundreds of partners, a fundamental part of our business
- The business model is based on licenses and royalties
- ARM technology is everywhere: mobile, embedded, enterprise, home

ARM – Cortex™ CPUs

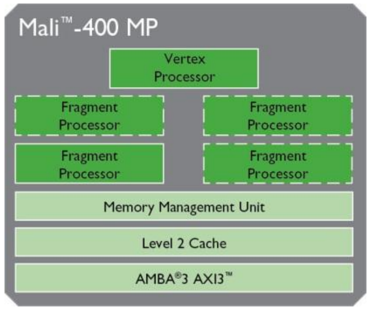
- Cortex-A
  - High performance application processors
  - Cortex-A50 series enables 64bit processing
- Cortex-R
  - Real-time embedded processors
- Cortex-M
  - Embedded processors for deterministic microcontroller applications
- SecurCore™
  - Specialist processors utilized within the security markets



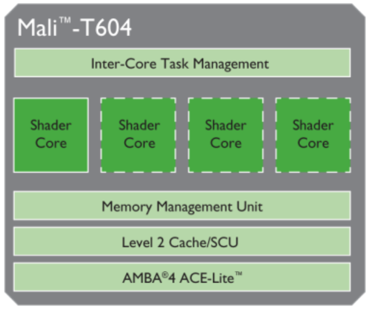
Cortex™ Low-Power Leadership from ARM

ARM – Mali™ GPUs

- Mali-400 MP
  - OpenGL® ES 2.0
  - 1 vertex processor
  - Up to 4 fragment processors
  - Deployed in many devices, like Galaxy S2 and S3



- Mali-T604
  - OpenGL ES 3.0, DirectX® 11
  - OpenCL™ 1.1 full profile compute capability
  - Up to four unified cores
  - Inside the Nexus 10, the Samsung Chromebook, the Arndale board

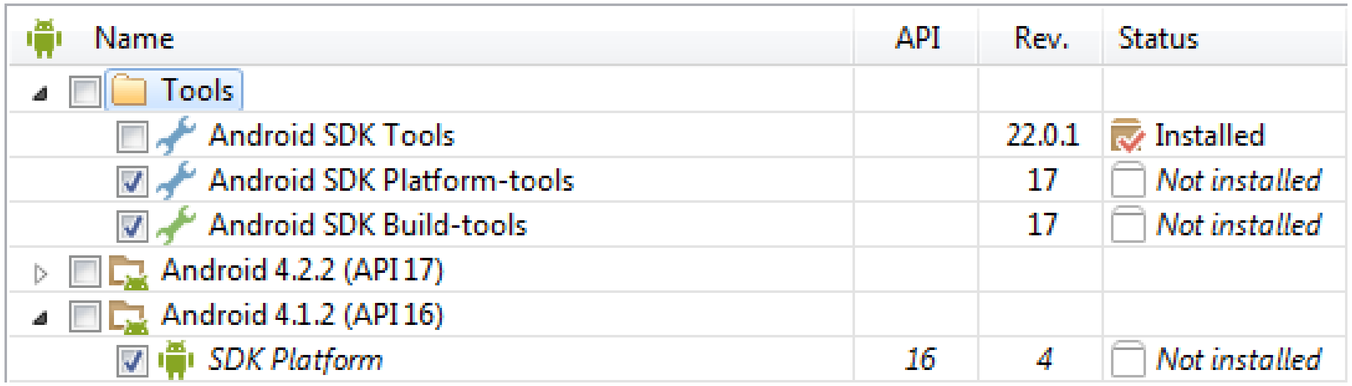


What do you need to have

- A computer with:
  - Java SE Development Kit (JDK)
  - Android SDK
  - ADB drivers (u20gt_adb.zip)
  - Unity Pro
- An Android device connected to the computer via ADB
- The example application APK

Android SDK

- Website: http://developer.android.com/sdk
- You don't need Eclipse + ADT for Unity
  1. Click on "Use an Existing IDE"
  2. Click on "Download the SDK Tools for Windows"
  3. Run the "Android SDK Manager" as Administrator
  4. Install the Android SDK platform and build tools
  5. Install at least one SDK Platform (API16 if you want to match the Android version on the Cube U18GT-S tablet)
- Unity supports r22 only since version 4.1.4



Name	API	Rev.	Status
☐ Tools			
☐ Android SDK Tools		22.0.1	Installed
✓ Android SDK Platform-tools		17	*Not installed*
✓ Android SDK Build-tools		17	*Not installed*
☐ Android 4.2.2 (API 17)			
☐ Android 4.1.2 (API 16)			
✓ *SDK Platform*	*16*	*4*	*Not installed*

Bringing Visual Computing to Life

ADB Driver

- The Android Debug Bridge (ADB) consists of a server running on the device and a client accessed through a CLI
- Add the platform-tools directory of the SDK to your path
- Extract the u20gt_adb.zip archive
  1. Enable USB debugging on the device (Settings -> Developer options -> USB debugging)
  2. Copy the ".android" directory in your home directory
  3. Go to the Windows Device Manager
  4. Locate the U18GT-S in "Other devices"
  5. Install the driver for it pointing to the "usb_driver" subdirectory
  6. After a correct installation you would see an "Android Composite ADB Interface" in "Android Device"
  7. Restart ADB and check the connection (`adb kill-server` and `adb devices` from the command line)

Unity Pro

- Unity is a cross-platform game engine
  - You can deploy on PC, consoles, mobile devices, web browsers
- Scripting based on Mono, supported languages:
  - JavaScript-like, Boo, C#
- Pro version has support for:
  - Profiler
  - Level of Detail
  - Occlusion Culling
  - Light Probes
  - Static Batching
- Latest version: 4.1.5 (June 2013)
- Website: http://unity3d.com/

Android Device

Cube U18GT-S (http://www.51cube.com/en)

- Rockchip RK3066
  - Dual-core Cortex™-A9 at 1.6GHz
  - Quad-core Mali™-400 at 250MHz
- 1GB DDR3 RAM
- 8GB Internal Storage
- 1024x600 7” 5-points capacitive screen
- HDMI 1.4 interface
- Android™ Jelly Bean 4.1.1

What do you need to know

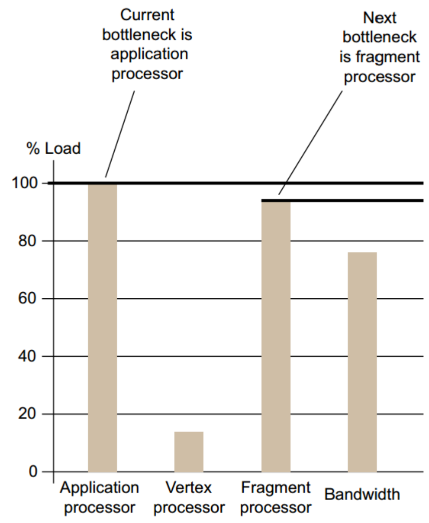
- What is a *draw call*
  - A call to a function of the underlying API (e.g. OpenGL ES) to draw something on the screen
- What is a *fragment*
  - A candidate pixel, it may or may not end up on screen for different reasons (e.g. being discarded, being overwritten, etc.)
- Differences between opaque and transparent render queue
  - Objects in the first queue are rendered in a front to back order to minimize the overdraw thanks to the depth test, transparent ones are rendered afterwards and in a back to front order
- What does *batching* mean
  - Group similar draw calls in a single one operating on the whole data
- Why mobile platforms are different
  - *Immediate* vs *deferred* rendering

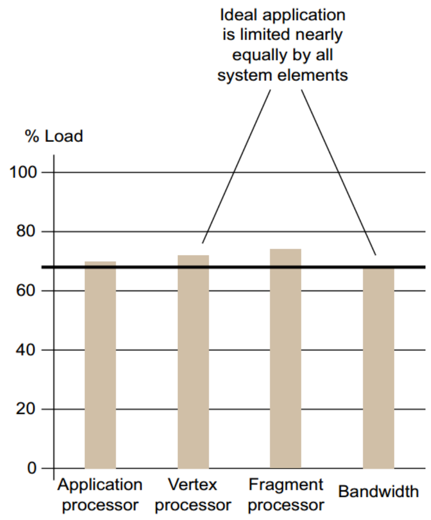
Why mobile platforms are different

- Desktop platforms
  - *Immediate mode*: graphics commands are executed when issued
  - Huge amount of bandwidth available between GPU and dedicated video memory (>100 GB/s)
  - No strict limits in size, power consumption or heat generation
- Mobile platforms
  - *Deferred mode*: graphics commands are collected by the driver and issued later
  - *Tile based*: rendering occurs in a small on-chip buffer before being written to memory
  - Bandwidth is severely reduced (~5 GB/s) and transferring data needs a great amount of power
  - Memory is unified and shared between CPU and GPU

Identify the bottleneck

- CPU
  - Too many draw calls
  - Complex scripts or physics
- Vertex processing
  - Too many vertices
  - Too much computation per vertex
- Fragment processing
  - Too many fragments, overdraw
  - Too much computation per fragment
- Bandwidth
  - Big and uncompressed textures
  - High resolution framebuffer



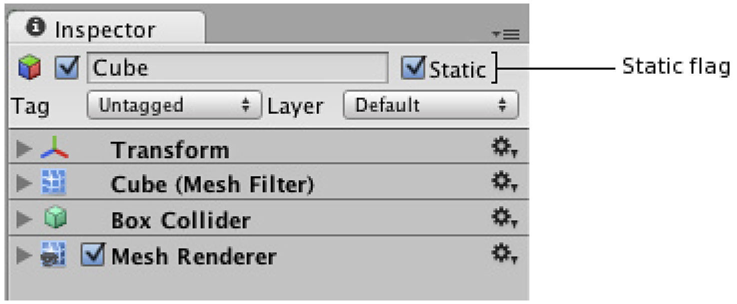


CPU Bound

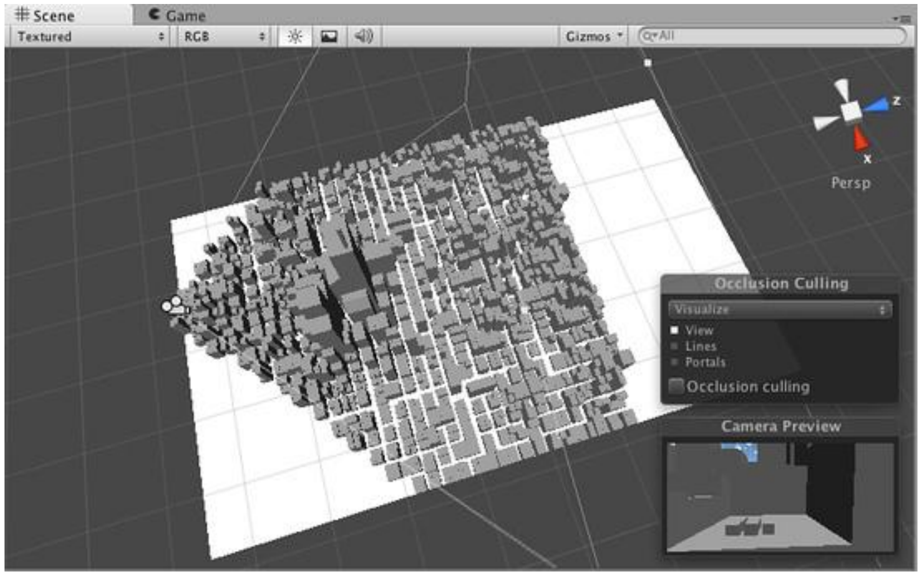
- Too many draw calls
  - Static batching (Unity Pro only)
  - Dynamic batching (automatic)
  - Frustum culling (automatic)
  - Occlusion culling (Unity Pro only)
- Complex scripts
  - Component caching
  - Pool of objects
  - Reduce Unity GUI calls
- Complex physics
  - Compound colliders instead of mesh ones

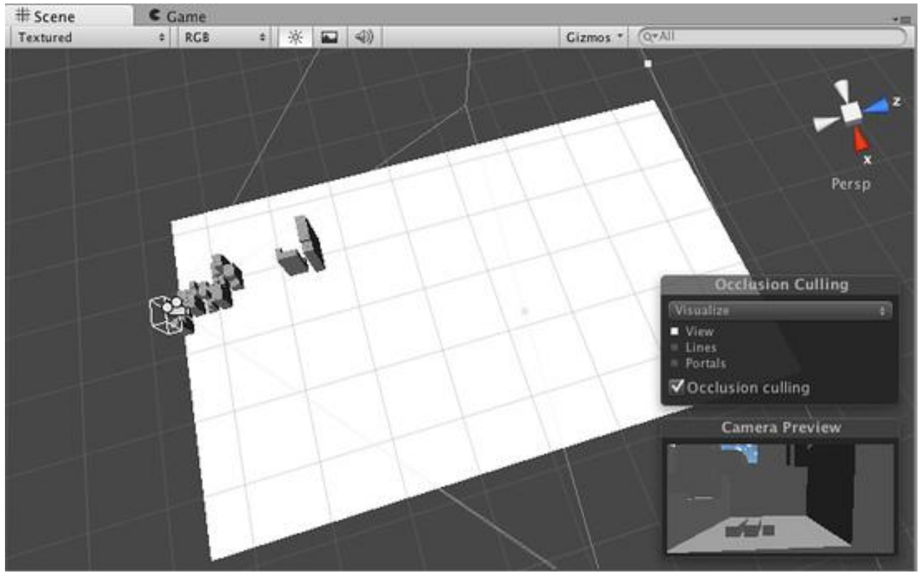
CPU Bound – Batching and Culling

- Make an object static to enable static batching (Unity Pro)



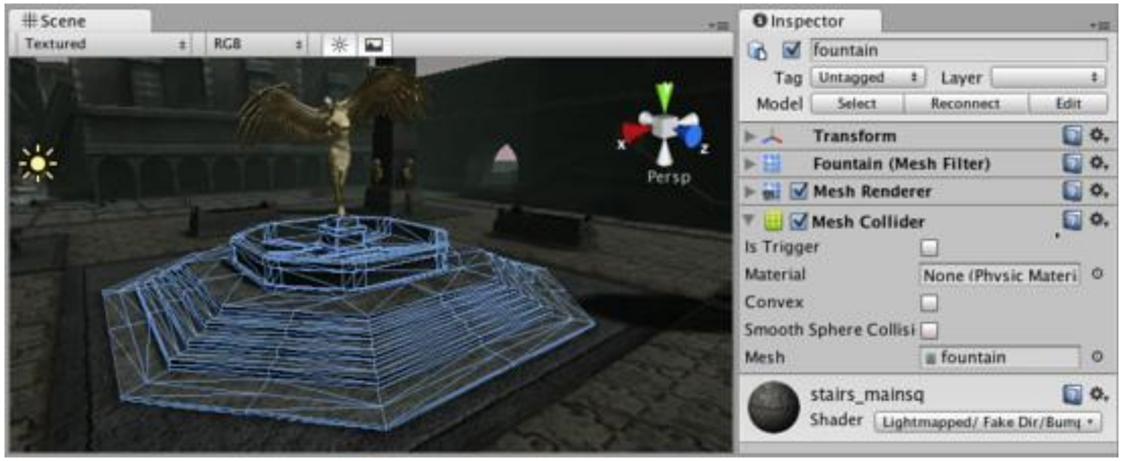
- Occlusion culling (Unity Pro only)



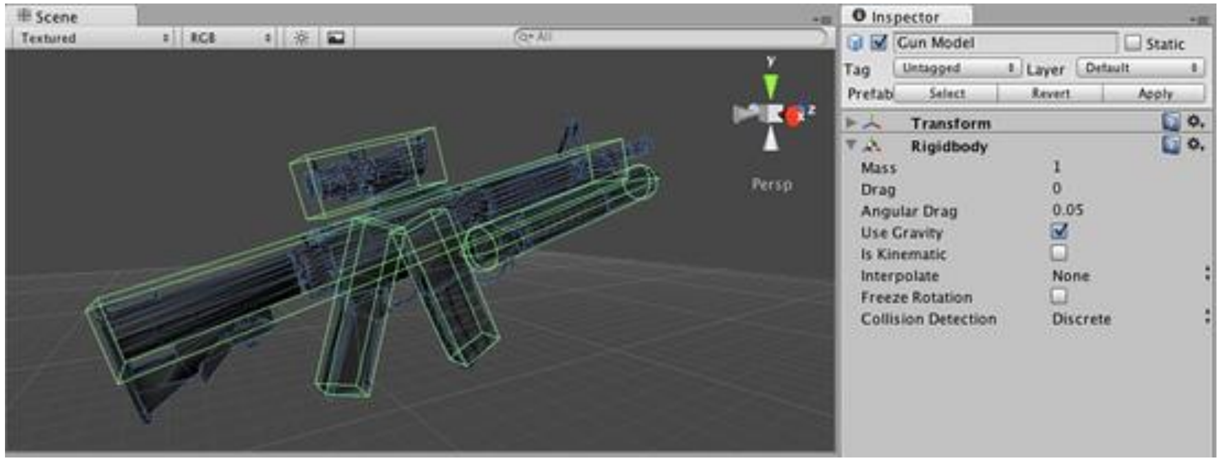


CPU Bound – Compound colliders

- Use compound colliders instead of mesh colliders if possible



fountain
Tag Untagged Layer
Model Select Reconnect Edit
Transform
Fountain (Mesh Filter)
Mesh Renderer
Mesh Collider
Is Trigger ☐
Material None (Physic Materi
Convex ☐
Smooth Sphere Collisi ☐
Mesh fountain
stairs_mainsq
Shader Lightmapped/ Fake Dir/Bump



Gun Model ☐ Static
Tag Untagged Layer Default
Prefab Select Revert Apply
Transform
Rigidbody

Mass	1
Drag	0
Angular Drag	0.05
Use Gravity	✓
Is Kinematic	☐
Interpolate	None
Freeze Rotation	☐
Collision Detection	Discrete

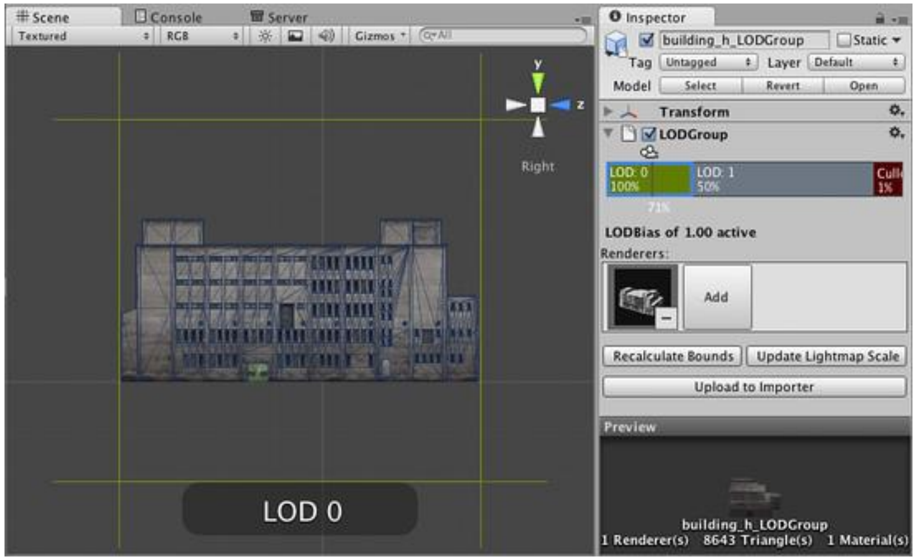
Vertex Bound

- Too many vertices in geometry
  - Remove unnecessary vertices
  - Use LOD switching through LODGroup (Unity Pro only)
  - Frustum culling (automatic)
  - Occlusion culling (Unity Pro only)

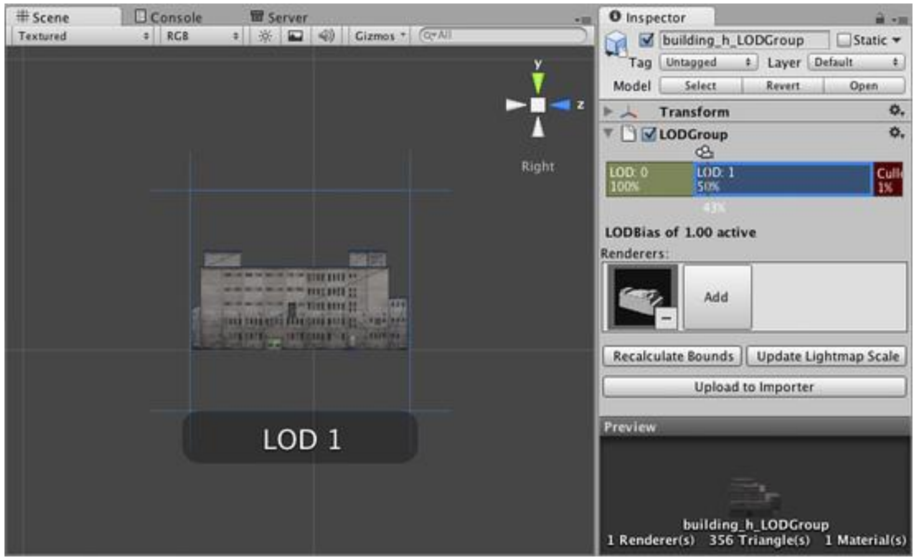
- Too much computation per vertex
  - Use the mobile version of Unity shaders whenever you can

Vertex Bound – Level of Detail

- Use LODGroup to limit vertex count when geometric detail is not strictly needed (very far or small objects)



building_h_LODGroup, Static
Tag: Untagged, Layer: Default
LOD: 0 100% | LOD: 1 50% | Cull 1%
71%
LODBias of 1.00 active
Renderers:
building_h_LODGroup
1 Renderer(s) 8643 Triangle(s) 1 Material(s)



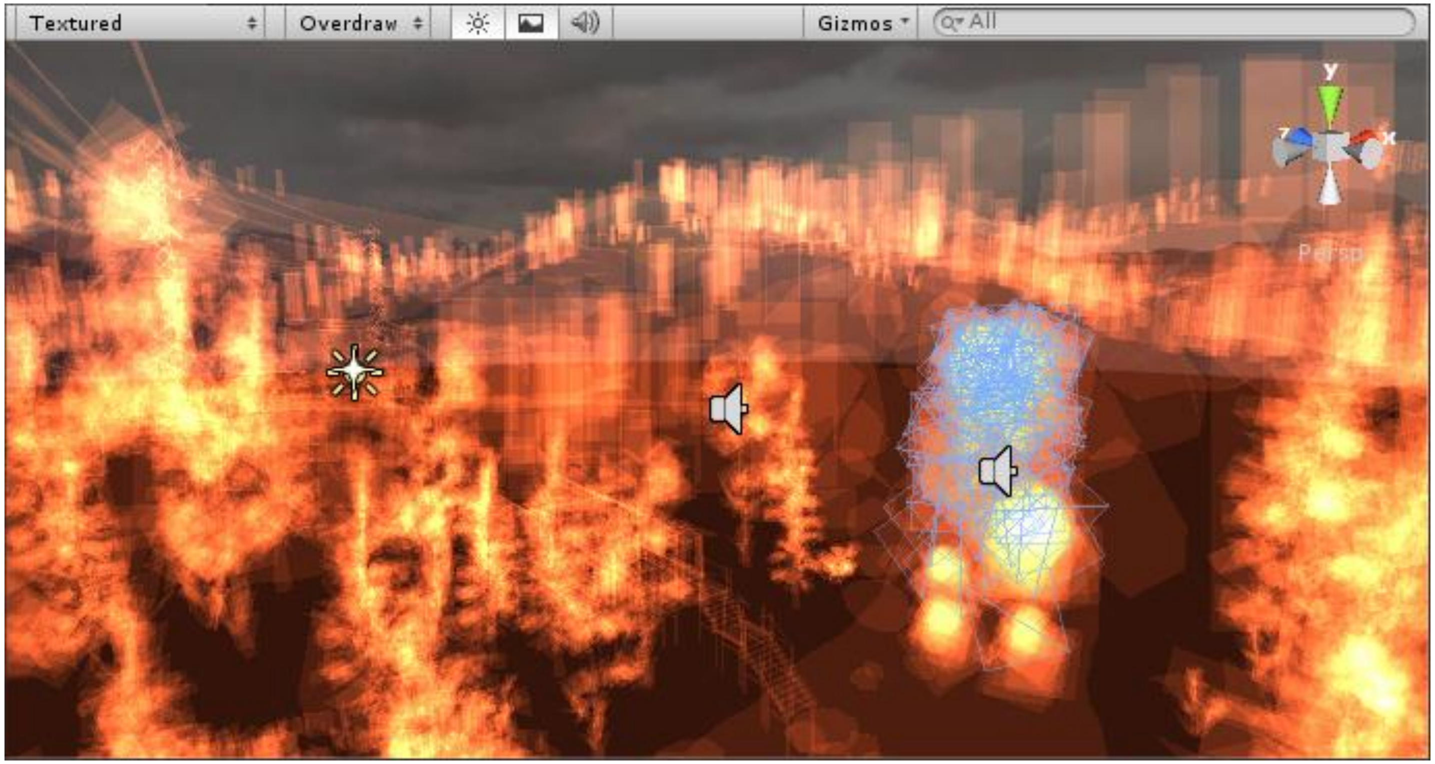
building_h_LODGroup, Static
Tag: Untagged, Layer: Default
LOD: 0 100% | LOD: 1 50% | Cull 1%
43%
LODBias of 1.00 active
Renderers:
building_h_LODGroup
1 Renderer(s) 356 Triangle(s) 1 Material(s)

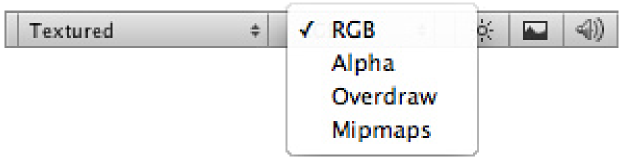
Fragment Bound

- Overdraw
  - When you are drawing to each pixel on the screen more than once
  - Drawing your objects front to back instead of back to front reduces overdraw, thanks to depth testing
  - Limit the amount of transparency in the scene (beware of particles!)
  - Unity has a *render mode* to show the amount of overdraw per pixel
- Too much computation per fragment
  - Bake as much as you can (lightmaps, light probes, etc.)
  - Contain the number of per-pixel lights
  - Limit real-time shadows (only high end mobile devices, Unity 4 Pro)
  - Try to avoid full screen post-processing
  - Use the mobile version of Unity shaders whenever you can

Fragment Bound – Overdraw

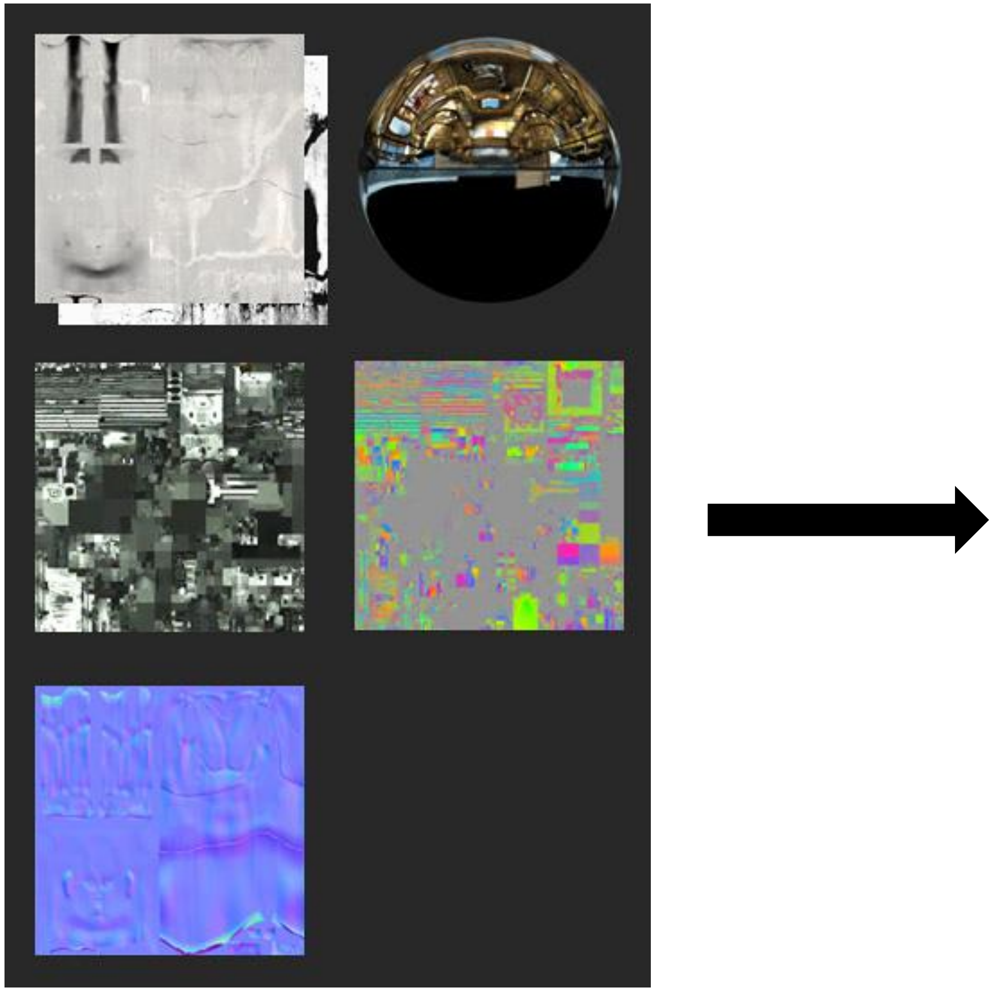
- Use the specific render mode to check the overdraw amount

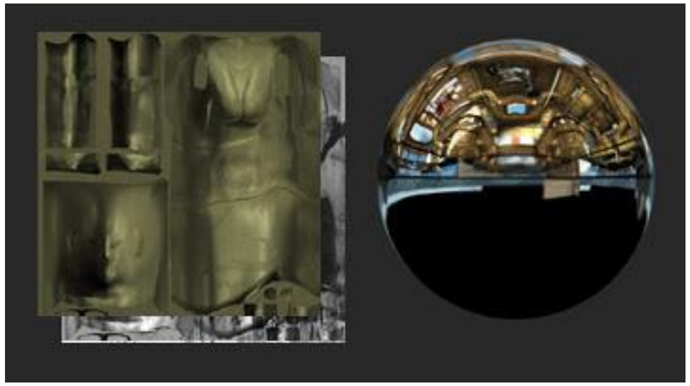




Fragment Bound – Bake to textures

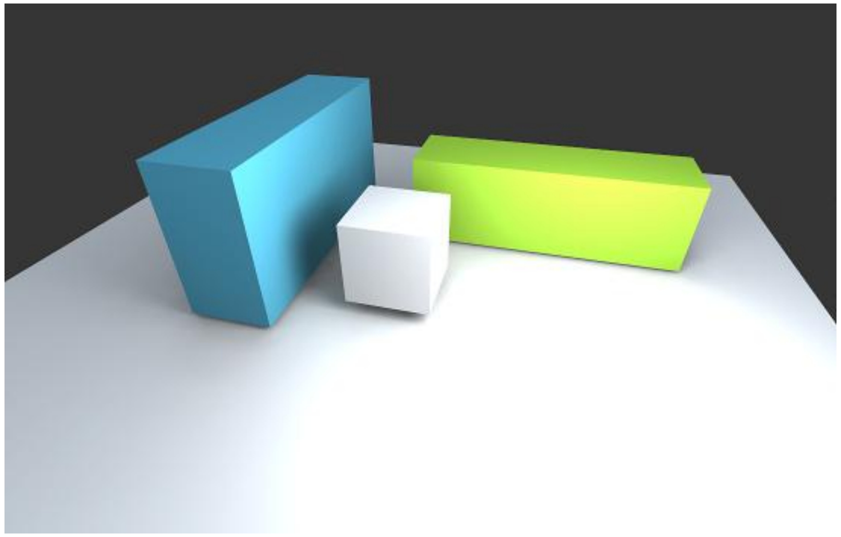
- Try to bake dynamic effects in textures

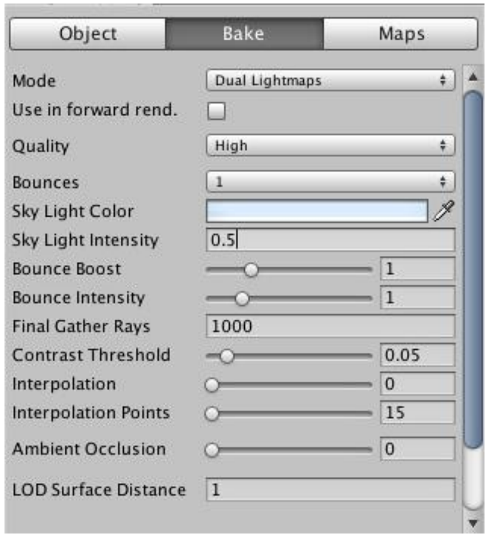




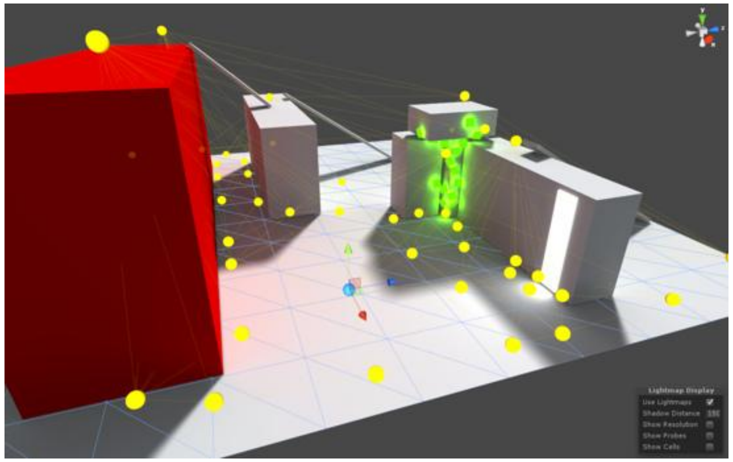
Fragment Bound – Lightmaps and probes

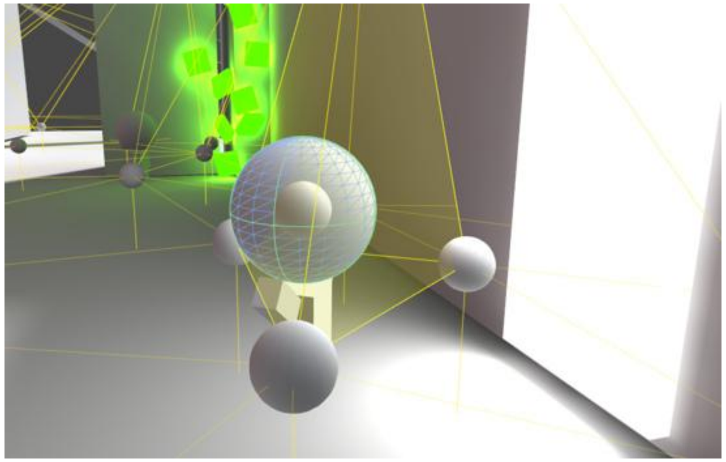
- Bake static illumination to lightmaps and exploit probes





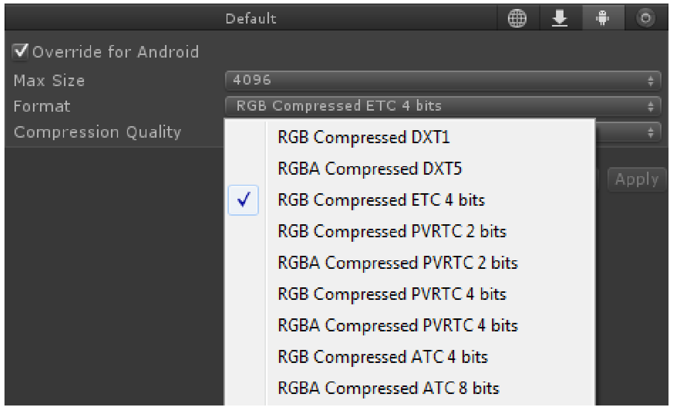
Mode	Dual Lightmaps
Use in forward rend.	☐
Quality	High
Bounces	1
Sky Light Color	
Sky Light Intensity	0.5
Bounce Boost	1
Bounce Intensity	1
Final Gather Rays	1000
Contrast Threshold	0.05
Interpolation	0
Interpolation Points	15
Ambient Occlusion	0
LOD Surface Distance	1





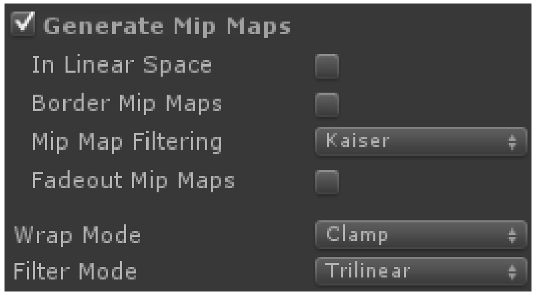
Bandwidth Bound

- Use texture compression
  - ETC is the standard compressed format that comes with OpenGL ES
  - ASTC is the new Khronos standard soon to be supported by Unity
- Use MIP maps
  - More memory but improved image quality (less aliasing artefacts)
  - Optimized memory bandwidth (when sampling from smaller maps)
- Use trilinear and anisotropic filtering in moderation
- Use Level of Detail techniques for geometry

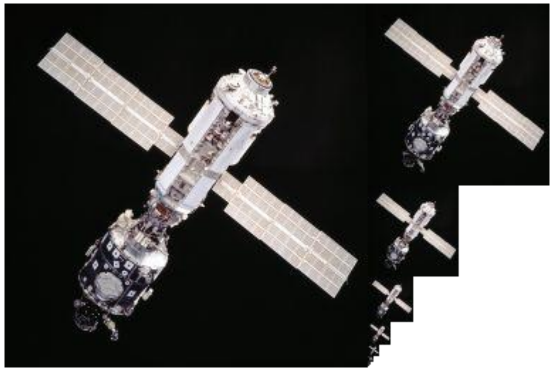


☑ Override for Android
Max Size: 4096
Format: RGB Compressed ETC 4 bits
Compression Quality

- RGB Compressed DXT1
- RGBA Compressed DXT5
- ✓ RGB Compressed ETC 4 bits
- RGB Compressed PVRTC 2 bits
- RGBA Compressed PVRTC 2 bits
- RGB Compressed PVRTC 4 bits
- RGBA Compressed PVRTC 4 bits
- RGB Compressed ATC 4 bits
- RGBA Compressed ATC 8 bits

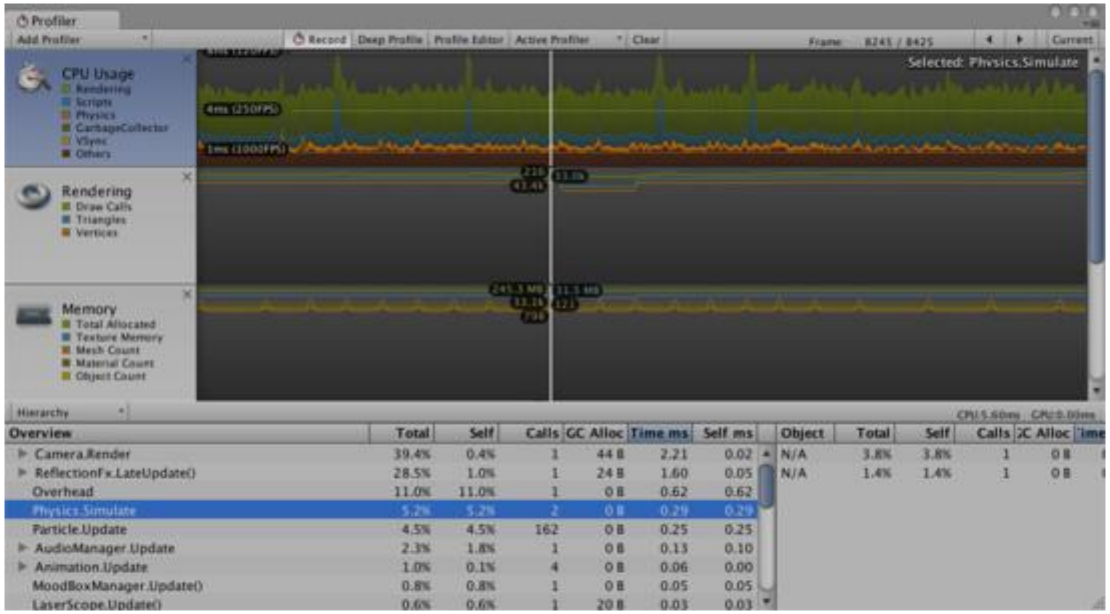


☑ Generate Mip Maps
In Linear Space ☐
Border Mip Maps ☐
Mip Map Filtering: Kaiser
Fadeout Mip Maps ☐
Wrap Mode: Clamp
Filter Mode: Trilinear



Unity Profiler

- It instruments the code to provide detailed per-frame performance data
- It provides specific data about:
  - CPU Usage
  - Rendering
  - GPU Usage
  - Memory
  - Physics
  - Audio



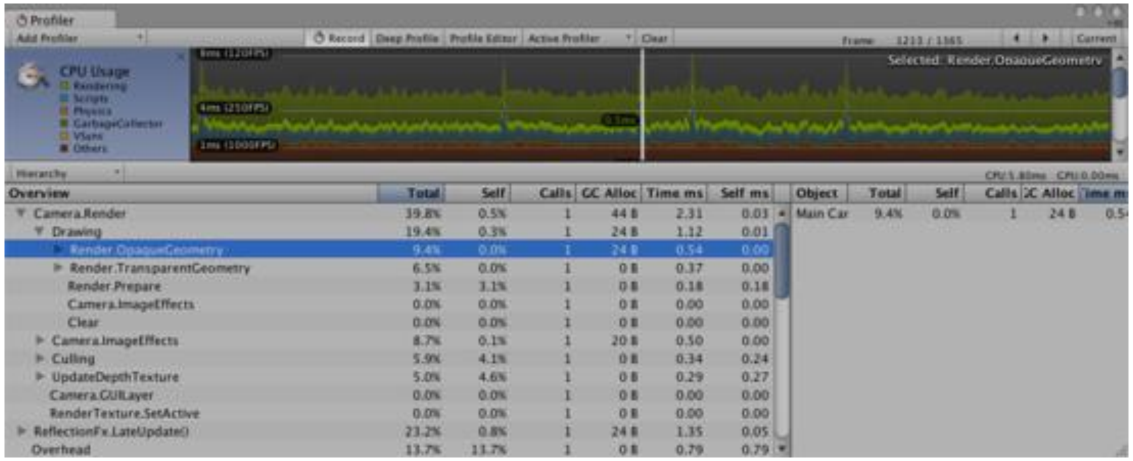
Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
▶ Camera.Render	39.4%	0.4%	1	44 B	2.21	0.02
▶ ReflectionFx.LateUpdate()	28.5%	1.0%	1	24 B	1.60	0.05
Overhead	11.0%	11.0%	1	0 B	0.62	0.62
Physics.Simulate	5.2%	5.2%	2	0 B	0.29	0.29
Particle.Update	4.5%	4.5%	162	0 B	0.25	0.25
▶ AudioManager.Update	2.3%	1.8%	1	0 B	0.13	0.10
▶ Animation.Update	1.0%	0.1%	4	0 B	0.06	0.00
MoodBoxManager.Update()	0.8%	0.8%	1	0 B	0.05	0.05
LaserScope.Update()	0.6%	0.6%	1	20 B	0.03	0.03

Object	Total	Self	Calls	GC Alloc
N/A	3.8%	3.8%	1	0 B
N/A	1.4%	1.4%	1	0 B

- Can profile content running on mobile devices
- Only available in Unity Pro

Unity Profiler – CPU Usage

- It shows CPU utilization for rendering, scripts, physics, garbage collection, etc.
- It shows detailed info about how the time was spent
- You can enable *Deep Profile* for additional data about all the function calls occurring in your scripts
- You can manually instrument specific blocks of code with `Profiler.BeginSample()` and `Profiler.EndSample()`

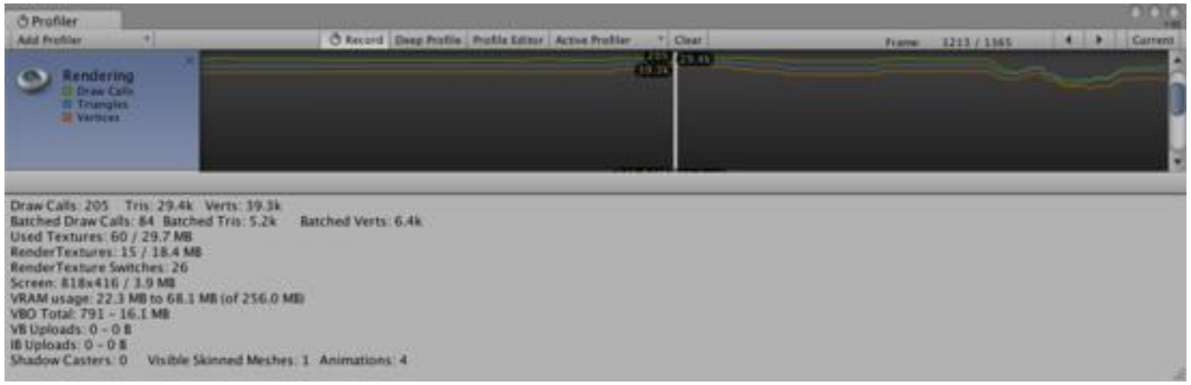


Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
Camera.Render	39.8%	0.5%	1	44 B	2.31	0.03
Drawing	19.4%	0.3%	1	24 B	1.12	0.01
Render.OpaqueGeometry	9.4%	0.0%	1	24 B	0.54	0.00
Render.TransparentGeometry	6.5%	0.0%	1	0 B	0.37	0.00
Render.Prepare	3.1%	3.1%	1	0 B	0.18	0.18
Camera.ImageEffects	0.0%	0.0%	1	0 B	0.00	0.00
Clear	0.0%	0.0%	1	0 B	0.00	0.00
Camera.ImageEffects	8.7%	0.1%	1	20 B	0.50	0.00
Culling	5.9%	4.1%	1	0 B	0.34	0.24
UpdateDepthTexture	5.0%	4.6%	1	0 B	0.29	0.27
Camera.GUILayer	0.0%	0.0%	1	0 B	0.00	0.00
RenderTexture.SetActive	0.0%	0.0%	1	0 B	0.00	0.00
ReflectionFx.LateUpdate()	23.2%	0.8%	1	24 B	1.35	0.05
Overhead	13.7%	13.7%	1	0 B	0.79	0.79

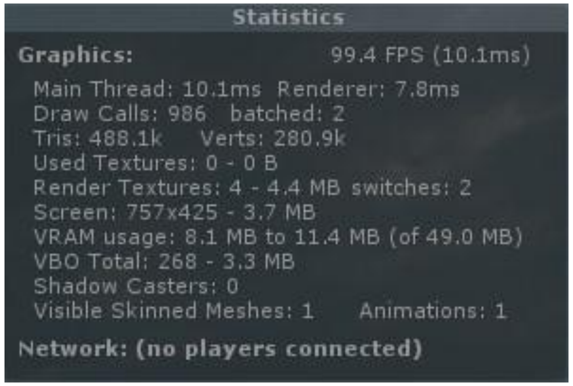
Object	Total	Self	Calls	GC Alloc	Time ms
Main Car	9.4%	0.0%	1	24 B	0.54

Unity Profiler – Rendering

- Statistics about the rendering subsystem
  - Draw calls
  - Triangles
  - Vertices
- The lower pane shows data similar to the rendering statistics window of the editor



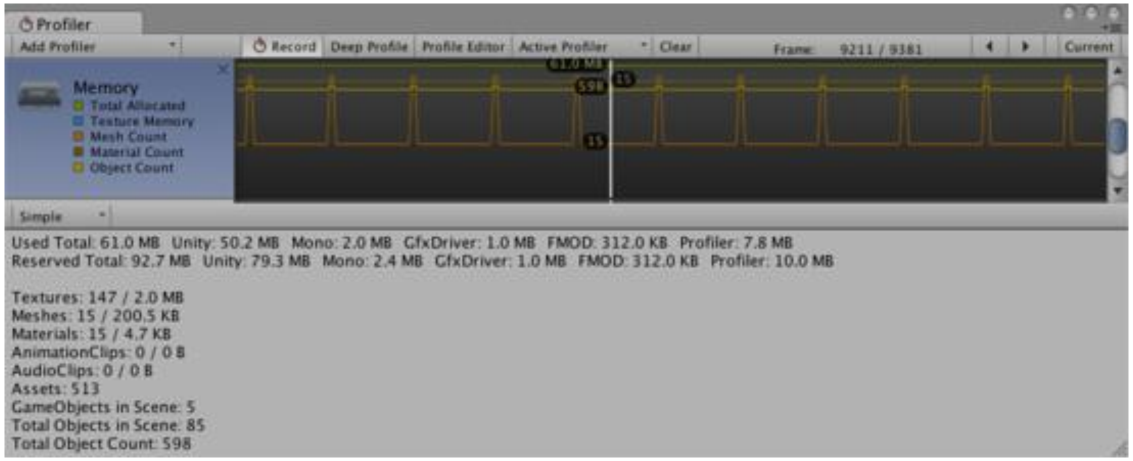
```
Draw Calls: 205    Tris: 29.4k   Verts: 39.3k
Batched Draw Calls: 84  Batched Tris: 5.2k     Batched Verts: 6.4k
Used Textures: 60 / 29.7 MB
RenderTextures: 15 / 18.4 MB
RenderTexture Switches: 26
Screen: 818x416 / 3.9 MB
VRAM usage: 22.3 MB to 68.1 MB (of 256.0 MB)
VBO Total: 791 - 16.1 MB
VB Uploads: 0 - 0 B
IB Uploads: 0 - 0 B
Shadow Casters: 0    Visible Skinned Meshes: 1  Animations: 4
```



```
Statistics
Graphics:                        99.4 FPS (10.1ms)
  Main Thread: 10.1ms  Renderer: 7.8ms
  Draw Calls: 986   batched: 2
  Tris: 488.1k     Verts: 280.9k
  Used Textures: 0 - 0 B
  Render Textures: 4 - 4.4 MB switches: 2
  Screen: 757x425 - 3.7 MB
  VRAM usage: 8.1 MB to 11.4 MB (of 49.0 MB)
  VBO Total: 268 - 3.3 MB
  Shadow Casters: 0
  Visible Skinned Meshes: 1      Animations: 1
Network: (no players connected)
```

Unity Profiler – Memory

- It shows memory used/reserved on a higher level
  - Unity native allocations
  - Garbage collected Mono managed allocations
  - Graphics and audio total used memory estimation
  - Memory used by the profiler data itself
- It also shows memory used by assets/objects
  - Textures
  - Meshes
  - Materials
  - Animation clips
  - Audio clips

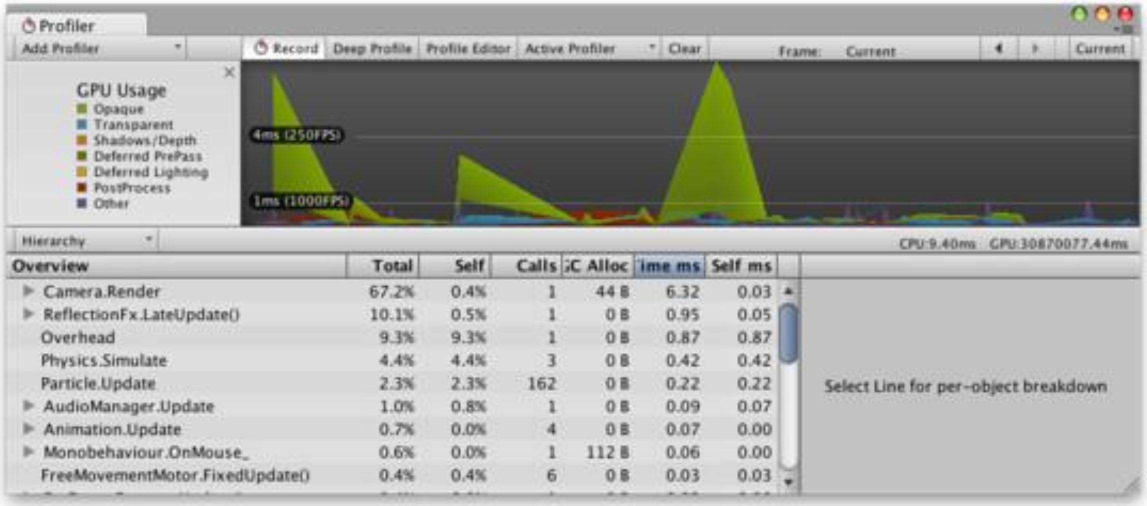


```
Used Total: 61.0 MB   Unity: 50.2 MB   Mono: 2.0 MB   GfxDriver: 1.0 MB   FMOD: 312.0 KB   Profiler: 7.8 MB
Reserved Total: 92.7 MB   Unity: 79.3 MB   Mono: 2.4 MB   GfxDriver: 1.0 MB   FMOD: 312.0 KB   Profiler: 10.0 MB

Textures: 147 / 2.0 MB
Meshes: 15 / 200.5 KB
Materials: 15 / 4.7 KB
AnimationClips: 0 / 0 B
AudioClips: 0 / 0 B
Assets: 513
GameObjects in Scene: 5
Total Objects in Scene: 85
Total Object Count: 598
```

Unity Profiler – GPU Usage

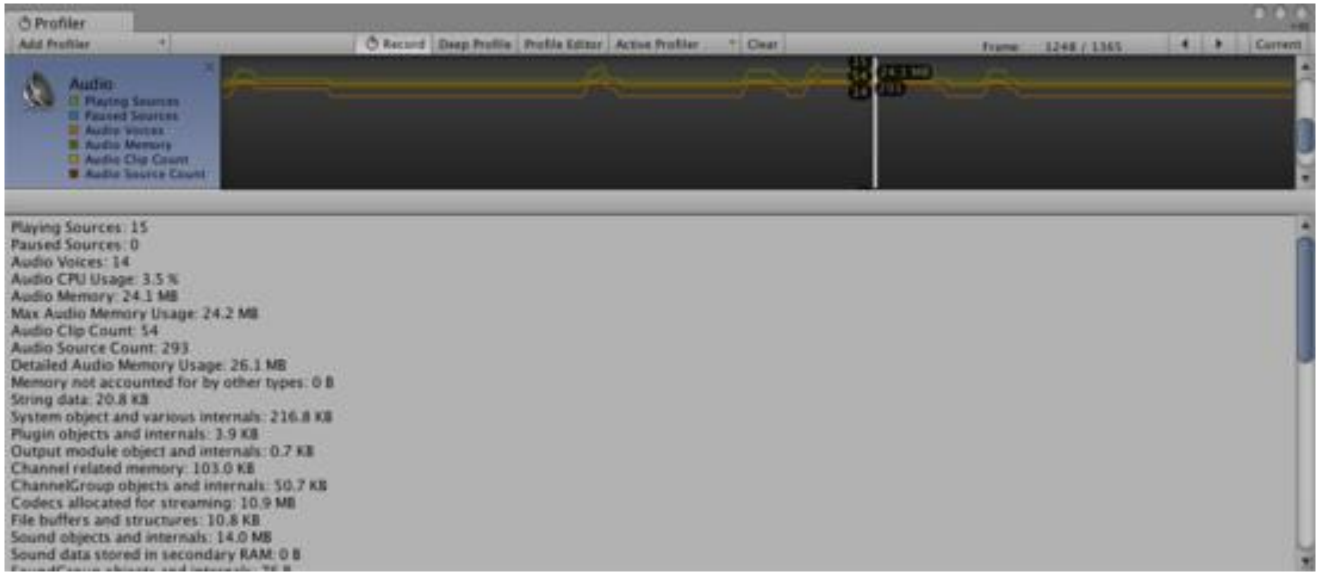
- It shows a contribution breakdown similar to the CPU profiler
  - Rendering of opaque objects
  - Rendering of transparent objects
  - Shadows
  - Deferred shading passes
  - Post processing
- Not yet available on mobile platforms ☹



Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
▶ Camera.Render	67.2%	0.4%	1	44 B	6.32	0.03
▶ ReflectionFx.LateUpdate()	10.1%	0.5%	1	0 B	0.95	0.05
Overhead	9.3%	9.3%	1	0 B	0.87	0.87
Physics.Simulate	4.4%	4.4%	3	0 B	0.42	0.42
Particle.Update	2.3%	2.3%	162	0 B	0.22	0.22
▶ AudioManager.Update	1.0%	0.8%	1	0 B	0.09	0.07
▶ Animation.Update	0.7%	0.0%	4	0 B	0.07	0.00
▶ Monobehaviour.OnMouse_	0.6%	0.0%	1	112 B	0.06	0.00
FreeMovementMotor.FixedUpdate()	0.4%	0.4%	6	0 B	0.03	0.03

Unity Profiler – Audio

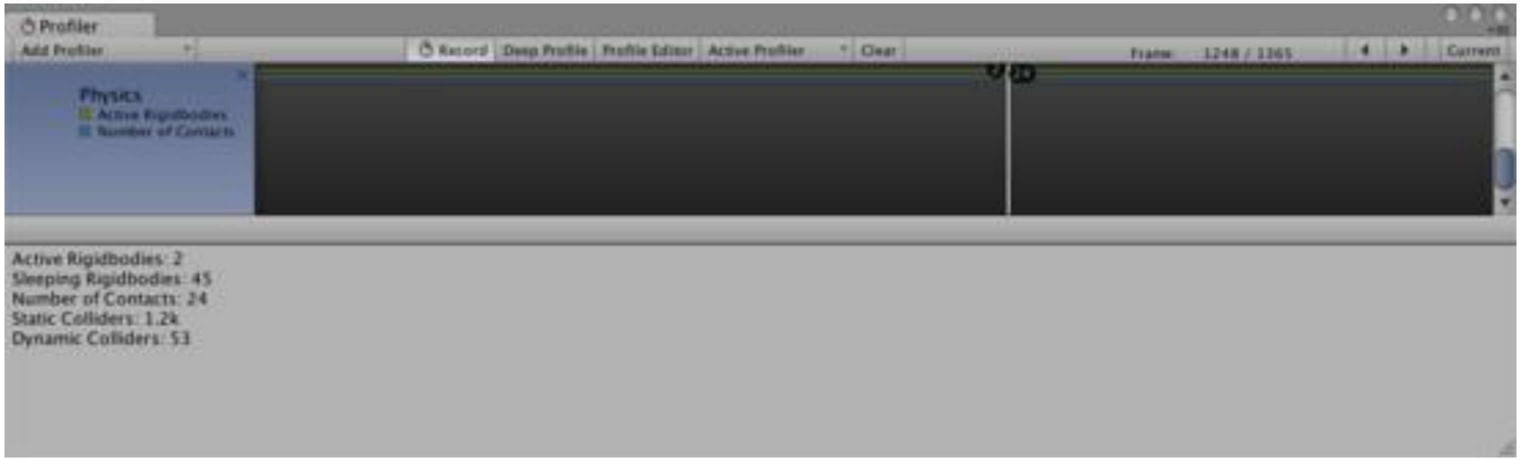
- Statistics about the audio subsystem
  - Total number of audio clips and sources
  - Number of playing/paused audio sources
  - Audio channels in use
  - Total amount of memory used by the audio engine



```
Playing Sources: 15
Paused Sources: 0
Audio Voices: 14
Audio CPU Usage: 3.5 %
Audio Memory: 24.1 MB
Max Audio Memory Usage: 24.2 MB
Audio Clip Count: 54
Audio Source Count: 293
Detailed Audio Memory Usage: 26.1 MB
Memory not accounted for by other types: 0 B
String data: 20.8 KB
System object and various internals: 216.8 KB
Plugin objects and internals: 3.9 KB
Output module object and internals: 0.7 KB
Channel related memory: 103.0 KB
ChannelGroup objects and internals: 50.7 KB
Codecs allocated for streaming: 10.9 MB
File buffers and structures: 10.8 KB
Sound objects and internals: 14.0 MB
Sound data stored in secondary RAM: 0 B
```

Unity Profiler – Physics

- Statistics about the physics simulation
  - Active rigid bodies (moving)
  - Sleeping rigid bodies (not processed by the physics engine)
  - Points of contact between all colliders
  - Number of static colliders
  - Number of dynamic colliders (attached to rigid bodies)



Active Rigidbodies: 2
Sleeping Rigidbodies: 45
Number of Contacts: 24
Static Colliders: 1.2k
Dynamic Colliders: 53

Unity Profiler on Android

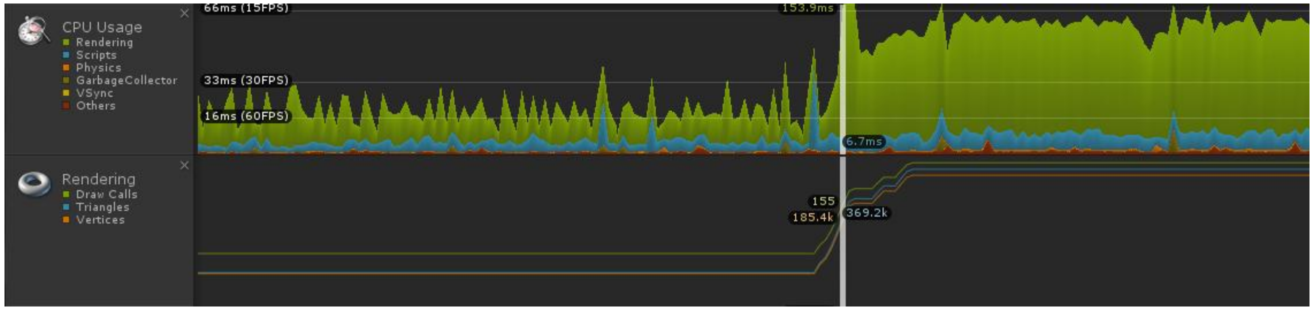
- WiFi profiling
  1. Make sure the Android device and the host are on the same subnet
  2. Check the *Autoconnect Profiler* checkbox in build settings dialog
  3. Attach your device to host via cable and hit *Build & Run*
  4. When the app starts open the profiler window and select the appropriate device from the *Active Profiler* drop down menu
- ADB profiling
  1. Attach your device via cable and make sure ADB recognizes it
  2. `adb forward tcp:54999 localabstract:Unity-<insert bundle identifier here>`
  3. Check the *Development Build* checkbox and hit *Build & Run*
  4. When the app starts open the profiler window and select `AndroidProfiler(ADB@127.0.0.1:54999)`
  5. Ports from 54998 to 55511 should be open in your firewall rules

Example application overview

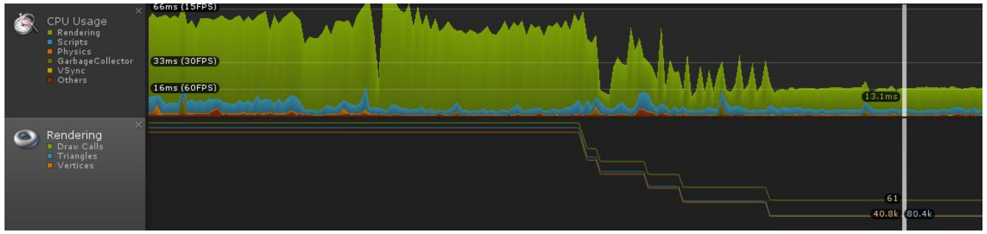
- Made up of six different scenes
  - Geometry (high vertex count, frustum culling)
  - LOD (using level of detail through LODGroup)
  - Particles (transparent rendering)
  - Texture (MIP mapping and compression)
  - Overdraw (alpha blending and alpha test)
  - Lights (vertex and pixel lighting)
- Most of them are based on a Prefab and `Instantiate()`
  - All instances are created at once in the beginning and deactivated
  - Number of active instances can adapt to maintain a target FPS value
  - Instantiation is made in screen space and based on camera aspect

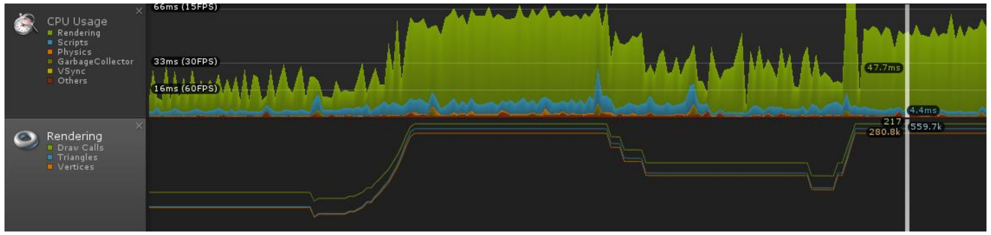
Geometry Scene

- Stressing the vertex units with lots of high poly objects
  - You can set the amount of visible objects or a target FPS value



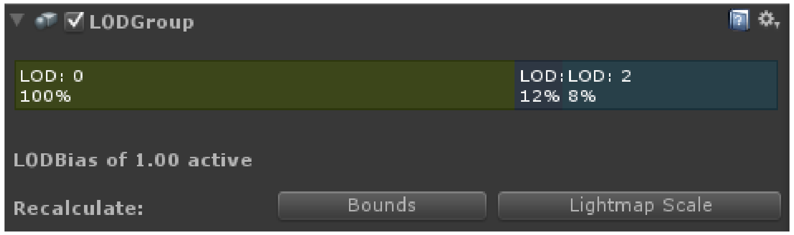
- It is possible to rotate and translate the camera to show frustum culling in action (and lack of occlusion culling)





LOD Scene

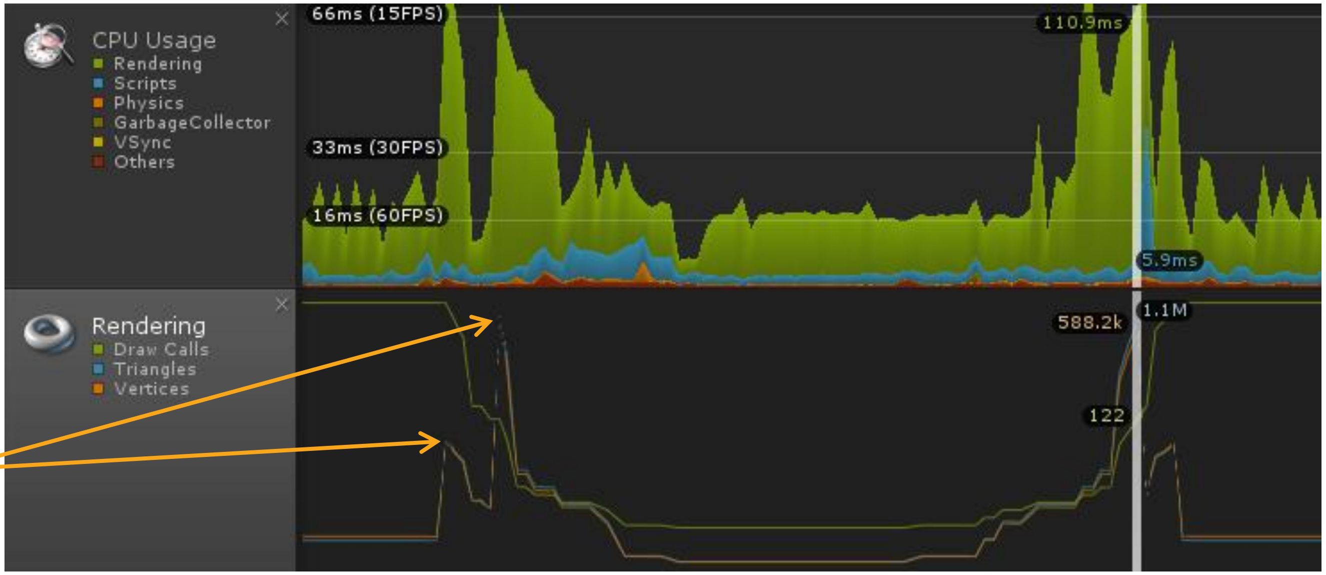
- LODGroup with three different levels



LODBias of 1.00 active

Recalculate:

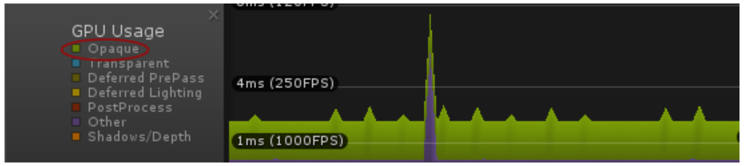
- Triangles and vertices spikes match LOD level switch, draw calls are dependent on frustum culling



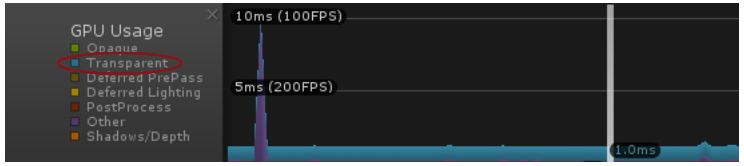
Switching LOD

Particles Scene

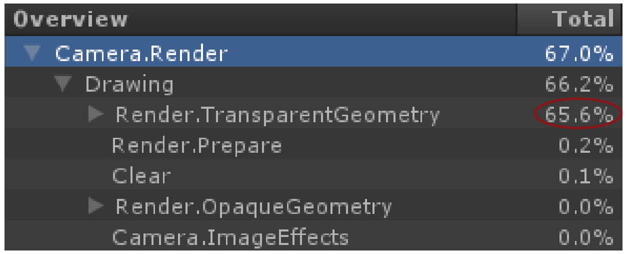
- All the rendering is done in the “Transparent Queue”



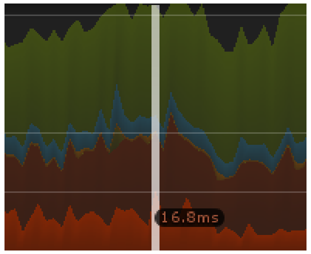
Geometry Scene

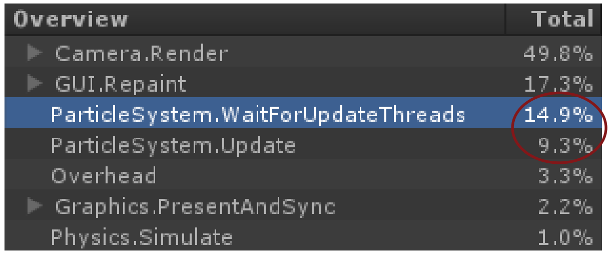


Particles Scene



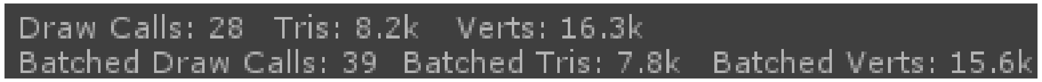
Overview	Total
▼ Camera.Render	67.0%
▼ Drawing	66.2%
▶ Render.TransparentGeometry	65.6%
Render.Prepare	0.2%
Clear	0.1%
▶ Render.OpaqueGeometry	0.0%
Camera.ImageEffects	0.0%





Overview	Total
▶ Camera.Render	49.8%
▶ GUI.Repaint	17.3%
ParticleSystem.WaitForUpdateThreads	14.9%
ParticleSystem.Update	9.3%
Overhead	3.3%
▶ Graphics.PresentAndSync	2.2%
Physics.Simulate	1.0%

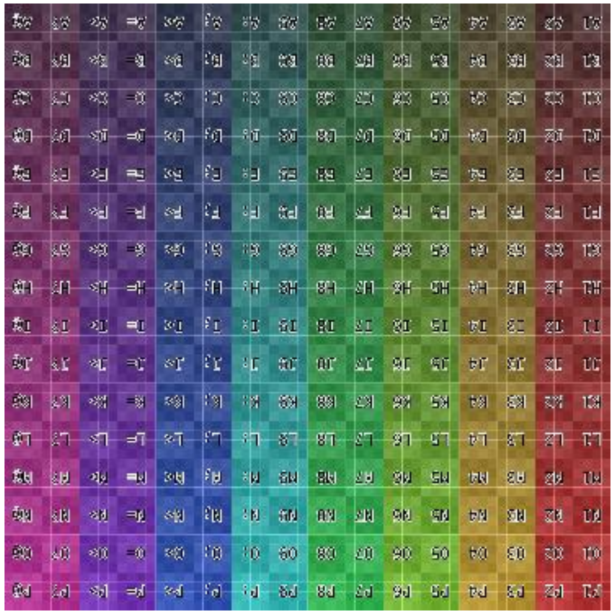
- Perfect case of geometry batching (one draw call per emitter)

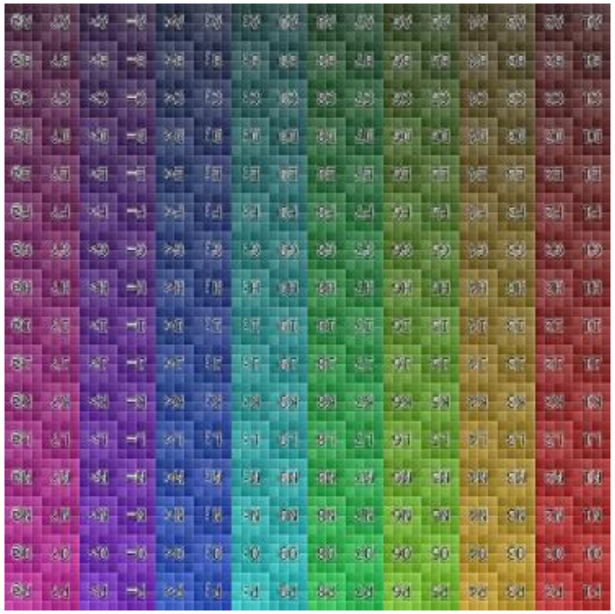


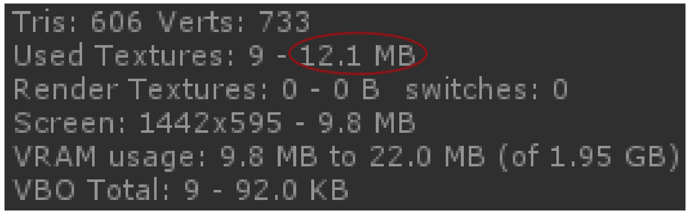
```
Draw Calls: 28   Tris: 8.2k   Verts: 16.3k
Batched Draw Calls: 39   Batched Tris: 7.8k   Batched Verts: 15.6k
```

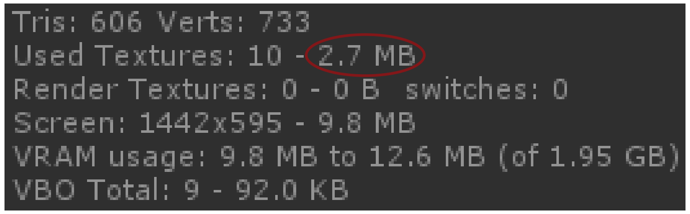
Texture Scene

- Compressed texture with MIP maps (right) takes less memory and looks better than uncompressed one (left)


Uncompressed, bilinear filtering

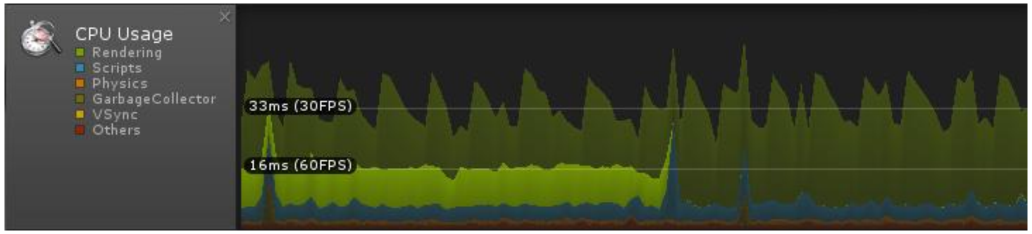

ETC compressed, trilinear filtering, MIP mapped

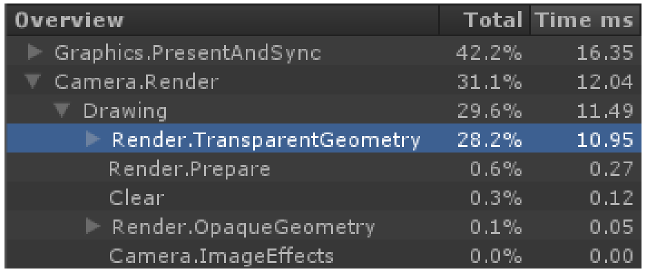

```
Tris: 606 Verts: 733
Used Textures: 9 - 12.1 MB
Render Textures: 0 - 0 B  switches: 0
Screen: 1442x595 - 9.8 MB
VRAM usage: 9.8 MB to 22.0 MB (of 1.95 GB)
VBO Total: 9 - 92.0 KB
```


```
Tris: 606 Verts: 733
Used Textures: 10 - 2.7 MB
Render Textures: 0 - 0 B  switches: 0
Screen: 1442x595 - 9.8 MB
VRAM usage: 9.8 MB to 12.6 MB (of 1.95 GB)
VBO Total: 9 - 92.0 KB
```

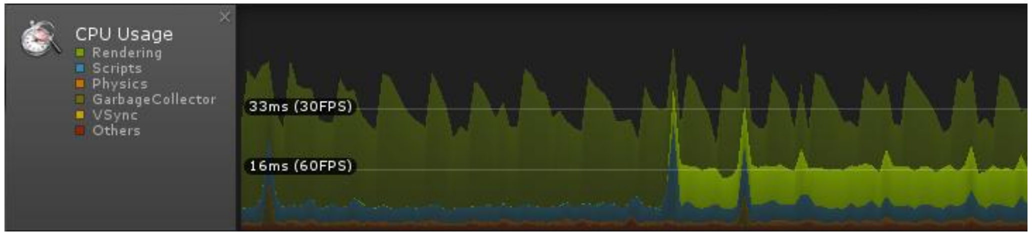
Overdraw Scene

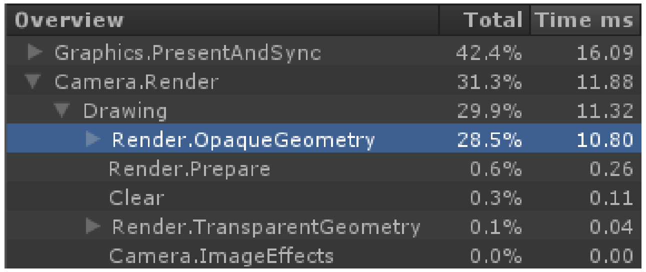
- Alpha blended quads are rendered in the transparent queue, alpha tested ones in the opaque queue, same performances





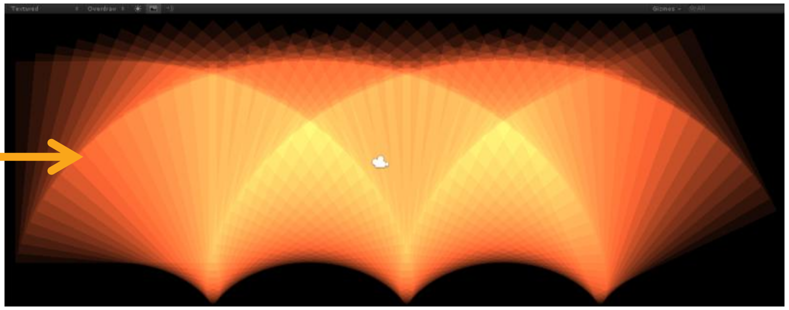
Overview	Total	Time ms
▶ Graphics.PresentAndSync	42.2%	16.35
▼ Camera.Render	31.1%	12.04
▼ Drawing	29.6%	11.49
▶ Render.TransparentGeometry	28.2%	10.95
Render.Prepare	0.6%	0.27
Clear	0.3%	0.12
▶ Render.OpaqueGeometry	0.1%	0.05
Camera.ImageEffects	0.0%	0.00





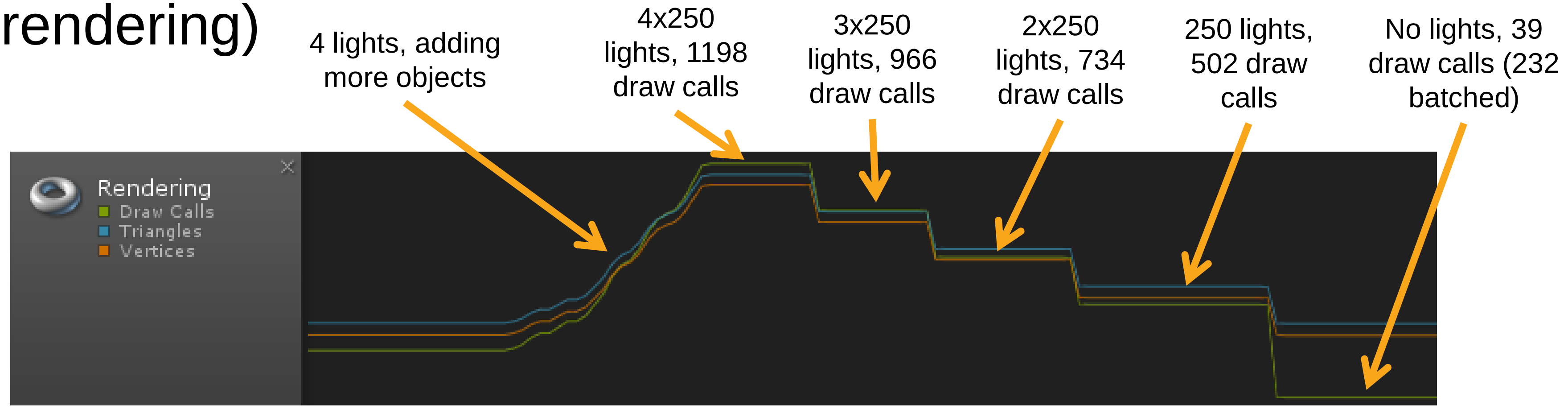
Overview	Total	Time ms
▶ Graphics.PresentAndSync	42.4%	16.09
▼ Camera.Render	31.3%	11.88
▼ Drawing	29.9%	11.32
▶ Render.OpaqueGeometry	28.5%	10.80
Render.Prepare	0.6%	0.26
Clear	0.3%	0.11
▶ Render.TransparentGeometry	0.1%	0.04
Camera.ImageEffects	0.0%	0.00

Overdraw render mode

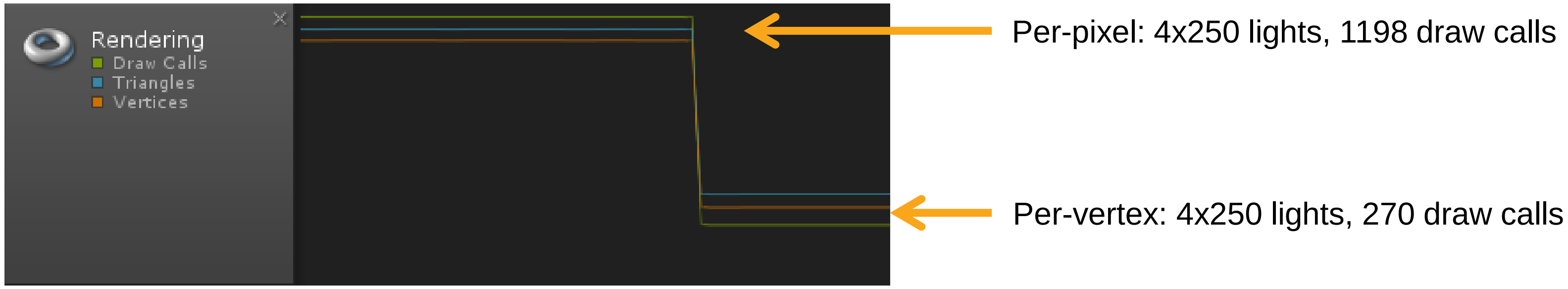


Lights Scene (1/2)

- Per-pixel lighting uses a draw call per light (forward rendering)

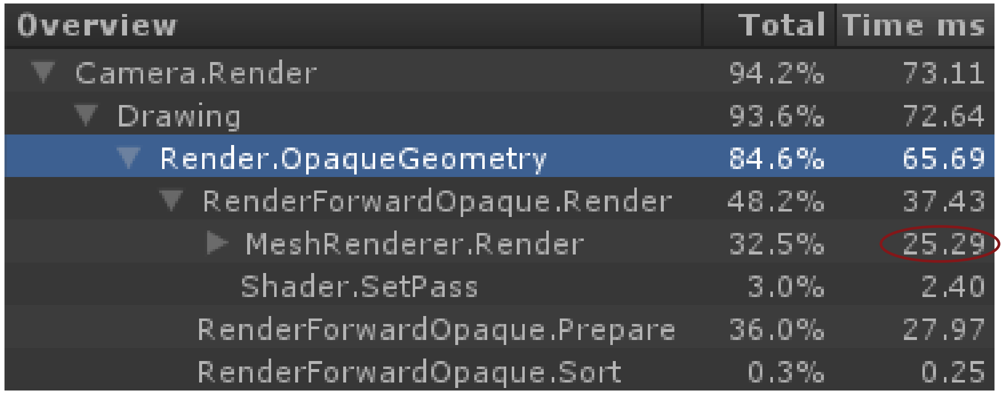


- Per-vertex lighting uses one geometric pass per object, lighting information is a vertex attribute



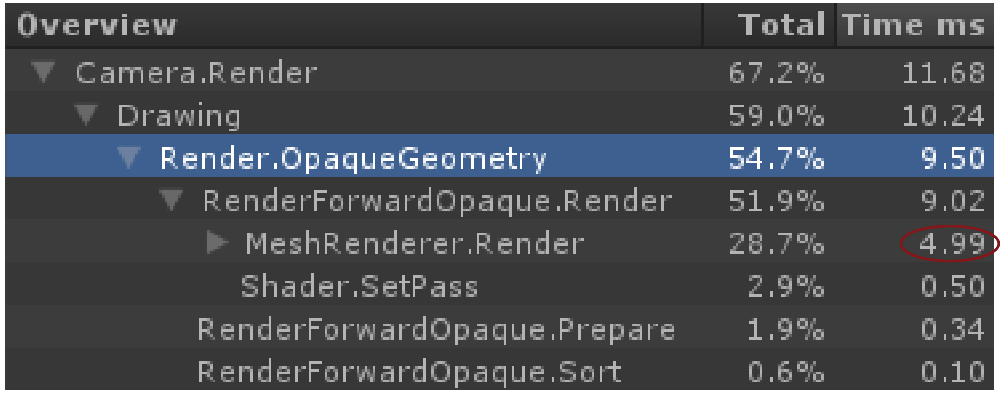
Lights Scene (2/2)

- `MeshRenderer.Render` takes 5 times the amount of *ms* when performing per-pixel lighting (one pass per light)



Overview	Total	Time ms
Camera.Render	94.2%	73.11
Drawing	93.6%	72.64
Render.OpaqueGeometry	84.6%	65.69
RenderForwardOpaque.Render	48.2%	37.43
MeshRenderer.Render	32.5%	25.29
Shader.SetPass	3.0%	2.40
RenderForwardOpaque.Prepare	36.0%	27.97
RenderForwardOpaque.Sort	0.3%	0.25

Per-pixel lighting



Overview	Total	Time ms
Camera.Render	67.2%	11.68
Drawing	59.0%	10.24
Render.OpaqueGeometry	54.7%	9.50
RenderForwardOpaque.Render	51.9%	9.02
MeshRenderer.Render	28.7%	4.99
Shader.SetPass	2.9%	0.50
RenderForwardOpaque.Prepare	1.9%	0.34
RenderForwardOpaque.Sort	0.6%	0.10

Per-vertex lighting

Links

- Practical Guide to Optimization for Mobiles (Unity Manual)
- Optimizing Graphics Performance (Unity Manual)
- Profiler (Unity Manual)

- ShadowGun: Optimizing for Mobile Sample Level (Unity Blog)
- "Fast Mobile Shaders" talk at SIGGRAPH 2011 (Unity Blog)

- ASTC Texture Compression: ARM Pushes the Envelope in Graphics Technology (ARM Blogs)

The End

Any questions?