

# Optimizing Unity Games for Mobile Platforms

Angelo Theodorou

Software Engineer

*Unite 2013, 28<sup>th</sup>-30<sup>th</sup> August*



# Agenda

---

- Introduction
  - The author and ARM
  - Preliminary knowledge
  - Unity Pro, OpenGL ES 3.0
- Identify the bottleneck
  - CPU
  - Vertex processing
  - Fragment processing
  - Bandwidth
- Profiling tools
  - The Unity Profiler (Pro only)
  - ARM DS-5 Streamline Performance Analyzer
- Example application overview

---

# Introduction

# The Author

---

- Angelo Theodorou
- Software Engineer
- Media Processing Division, ARM<sup>®</sup> Ltd.
  - Mali<sup>™</sup> Ecosystem Use-Cases team
- B.Sc. In Computer Science
- Studied at Università degli Studi di Napoli “*Federico II*”, Italy
  
- Worked on the game From Cheese in the past
  - Website: <http://www.fromcheese.com>
  - Grand Prize winner at the Samsung Smart App Challenge 2012
  - Made with Unity<sup>™</sup> Pro 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

# Mali GPUs

Performance



**Mali-450 MP**

2x Mali-400 performance  
Scalable up to 8 cores  
Leading OpenGL ES 2.0 performance

**Mali-400 MP**

First OpenGL ES 2.0 multicore GPU  
Scalable from 1 to 4 cores  
Low cost solution with Mali-300

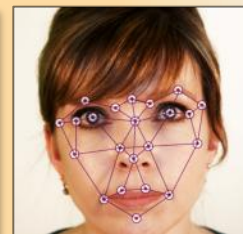
**Mali-300**

OpenGL® ES 2.0 compliant

● Date of production chips

2009 2010 2011 2012 2013

Performance



**Mali-T678**

High end solution  
Max compute capability



**Mali-T624**

**Mali-T628**

50% performance uplift  
OpenGL ES 3.0 support  
Scalable to 8 cores

**Mali-T604**

First Midgard architecture product  
OpenGL ES 3.0 support  
Scalable to 4 cores

● Date of production chips

2012 2013 2014



Bringing Visual Computing to Life



# Preliminary knowledge

---

- What is a *draw call*?
  - A call to a function of the underlying API (e.g. OpenGL ES) to draw something on screen
- What is a *fragment*?
  - A candidate pixel which 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 front to back order to make use of depth testing and minimize the overdraw, transparent ones are rendered afterwards in back to front order to preserve correctness
- What does *batching* mean?
  - To group similar draw calls in one call operating on the whole data set
- 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



# Unity Pro

---


- Pro only optimization features:
  - Profiler
  - Level of Detail
  - Occlusion Culling
  - Light Probes
  - Static Batching



# OpenGL ES 3.0

- Available with Unity 4.2, Android 4.3 and Mali-T6xx



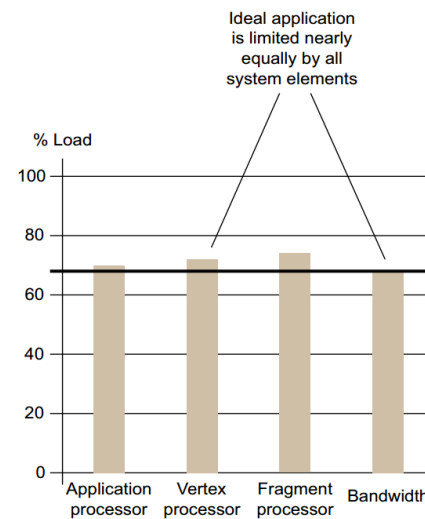
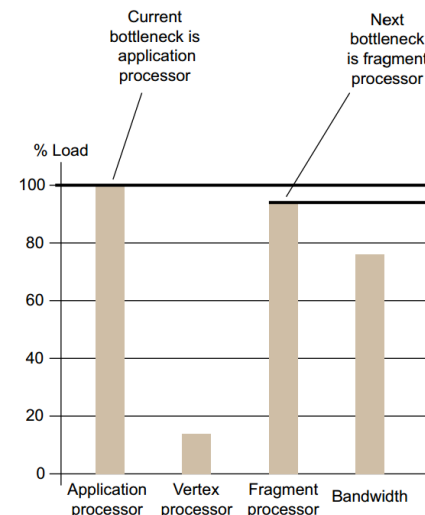
- Transform feedback 
  - Can write vertex shader output to a buffer object for data reuse
- ETC2 and EAC texture compression formats
  - Support for RGBA (ETC2) and one/two channels textures (EAC)
- Occlusion queries
  - Can query the number of samples drawn to determine visibility
- Geometry instancing
  - Plenty of instances of the same object but with unique properties
- Primitive restart, Uniform Buffer Objects, Pixel Buffer Objects, Fences, FP/3D/depth textures, Multiple Render Targets, ...

---

# Identify the bottleneck

# 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
  - Reduced Unity GUI calls
  - `OnBecomeVisible()`/`OnBecomeInvisible()`
  - Timed coroutines instead of per frame updates
- Complex physics
  - Compound colliders instead of mesh ones
  - Increased duration of “Fixed Timestep” (TimeManager)

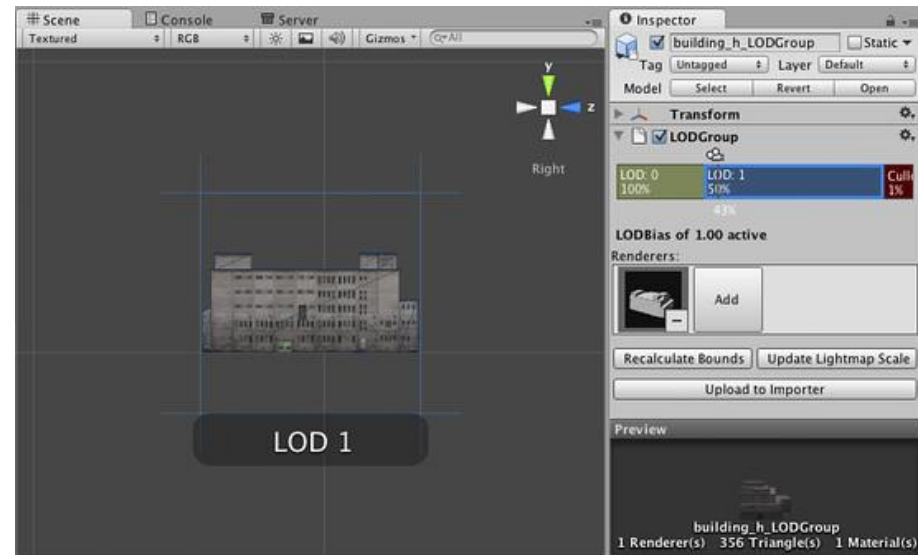
# Vertex Bound

---

- Too many vertices in geometry
  - Remove unnecessary vertices
  - Remove unnecessary hard edges and UV seams
  - 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)
  - Geometric objects can be replaced by billboards at long distance



# Fragment Bound

---

## ■ Overdraw

- When you are drawing to each pixel on 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

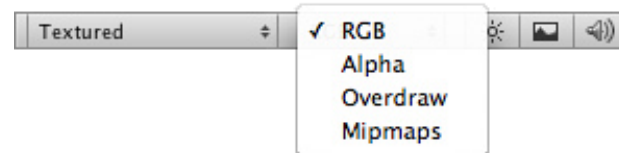
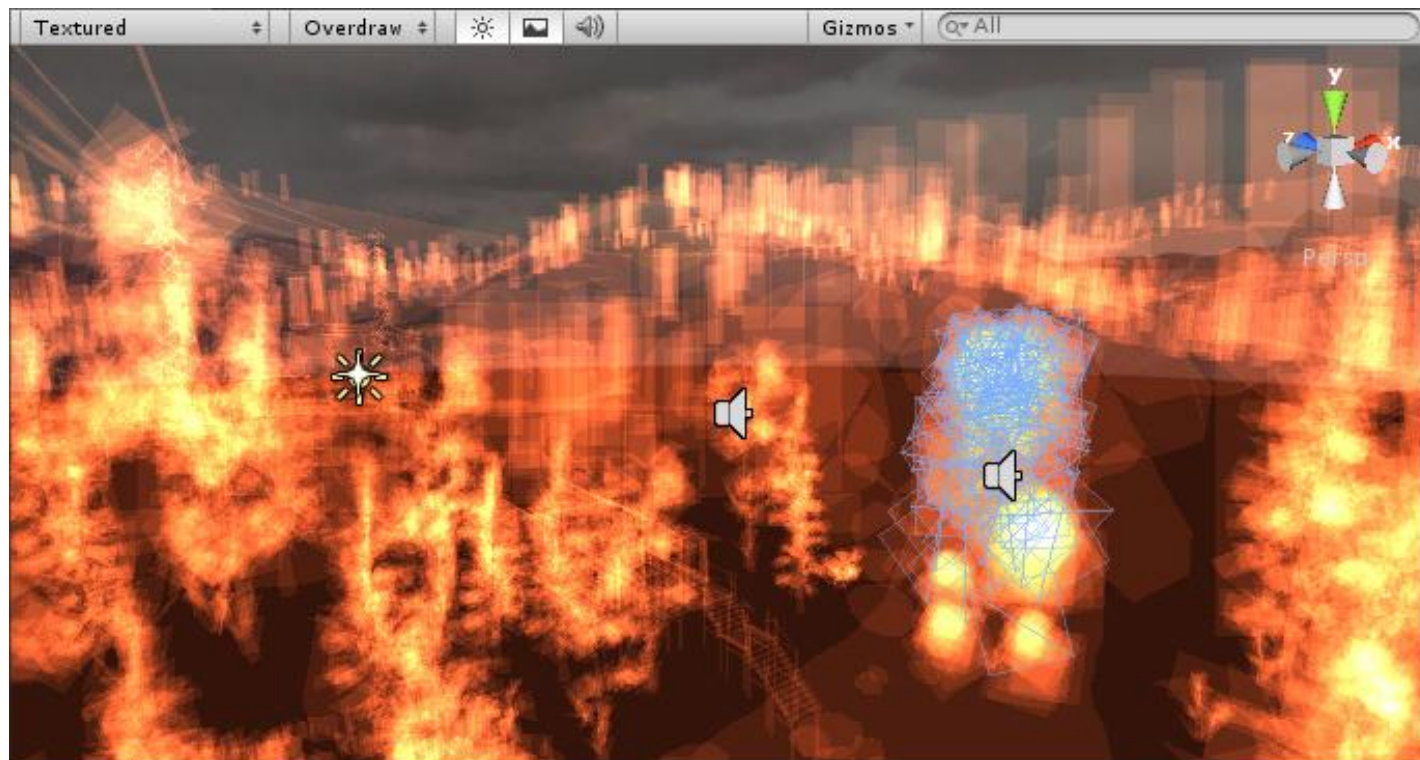
## ■ Too much computation per fragment

- 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



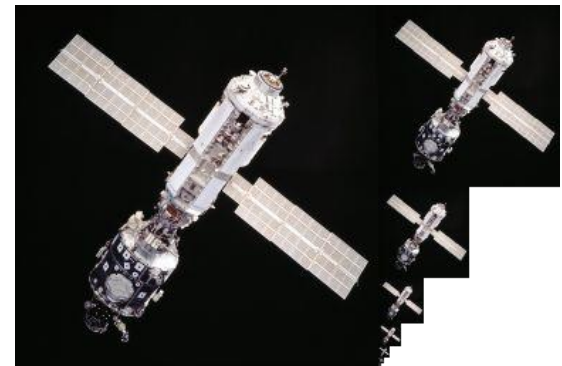
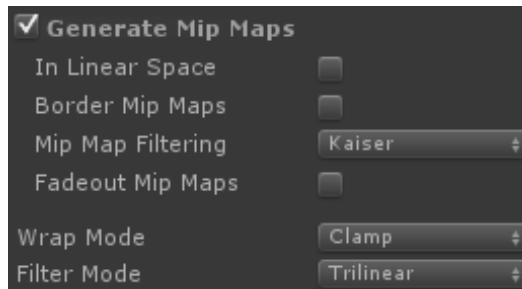
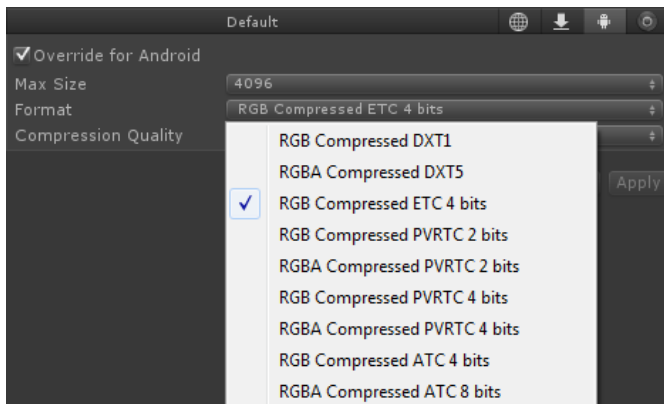
# Fragment Bound – Overdraw

- Use the specific *render mode* to check the overdraw amount



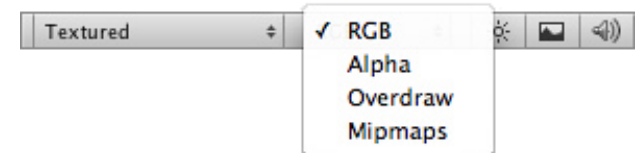
# Bandwidth Bound

- Use texture compression
  - ETC1 with OpenGL ES 2.0, ETC2 and EAC with OpenGL ES 3.0
  - 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 16bits textures where you can cope with reduced detail
- Use trilinear and anisotropic filtering in moderation



# Bandwidth Bound – Mipmaps

- Use the specific *render mode* to check mipmaps levels



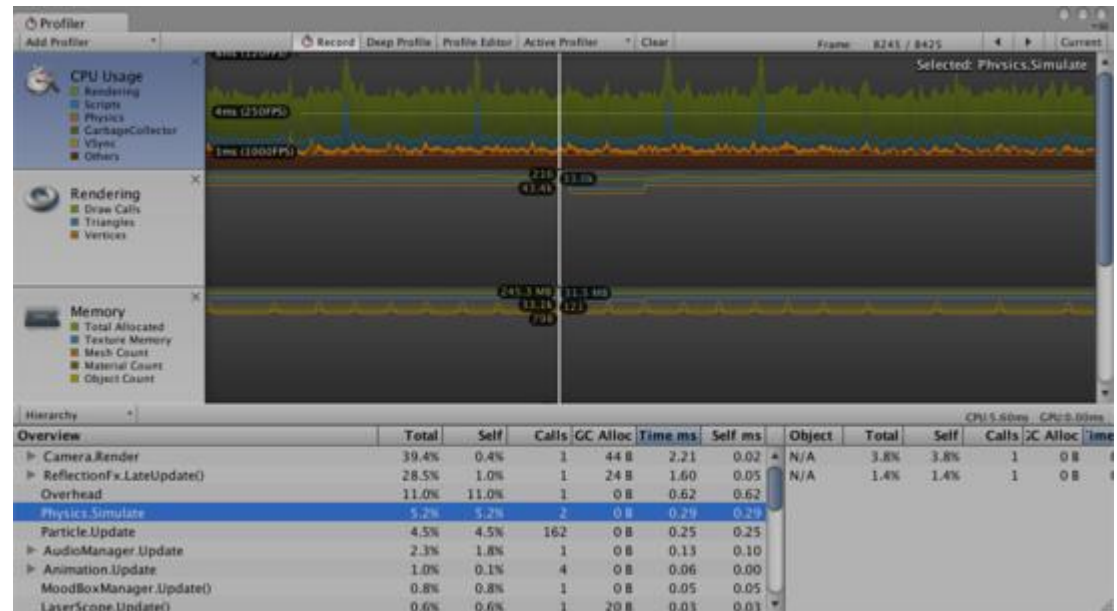
---

# Profiling tools



# Unity Profiler

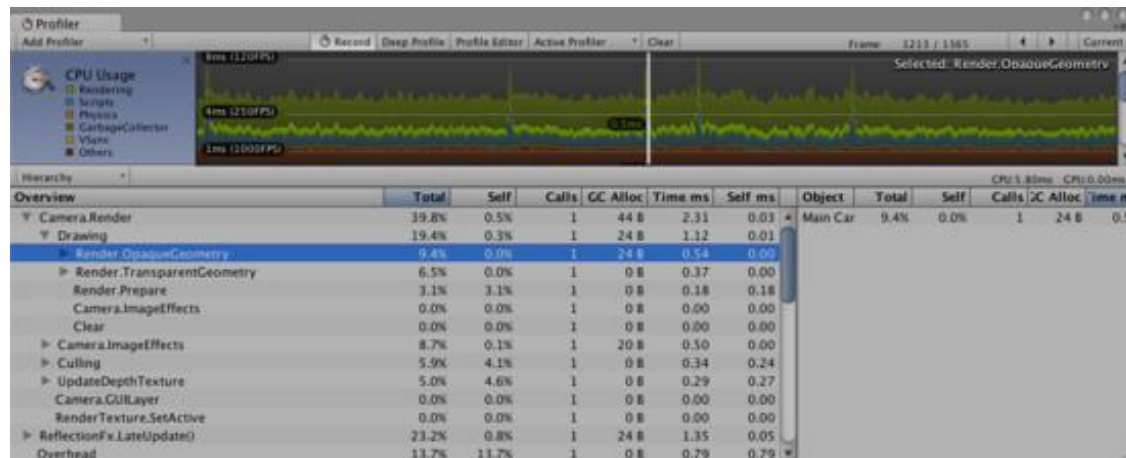
- It instruments the code to provide detailed per-frame performance data
- It provides specific data about:
  - CPU Usage
  - Rendering
  - GPU Usage
  - Memory
  - Physics
  - Audio



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

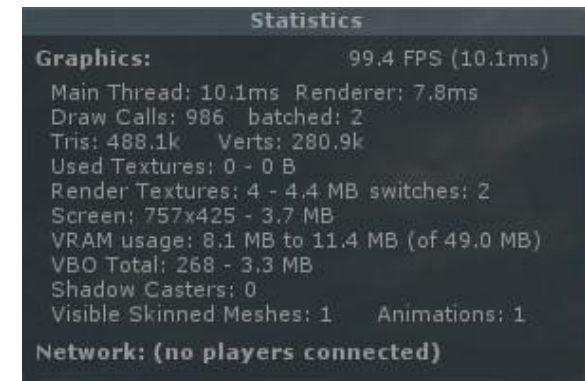
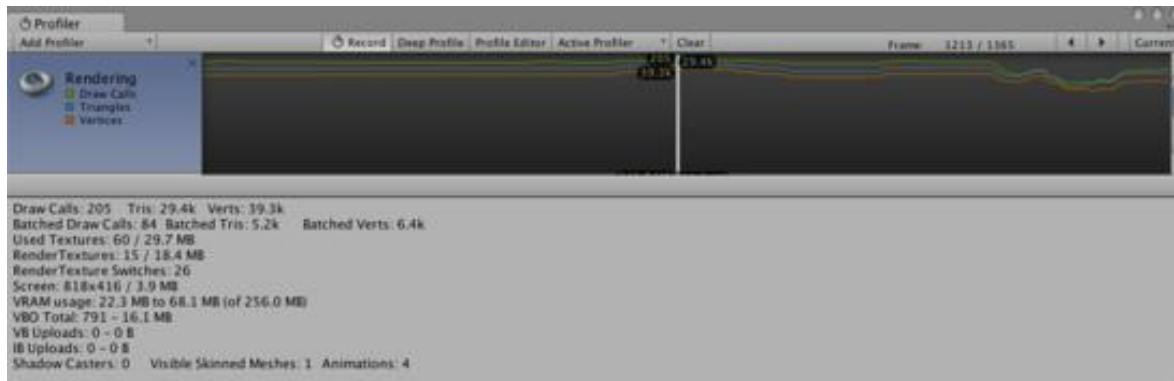
# Unity Profiler – CPU Usage

- 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()`



# Unity Profiler – Rendering

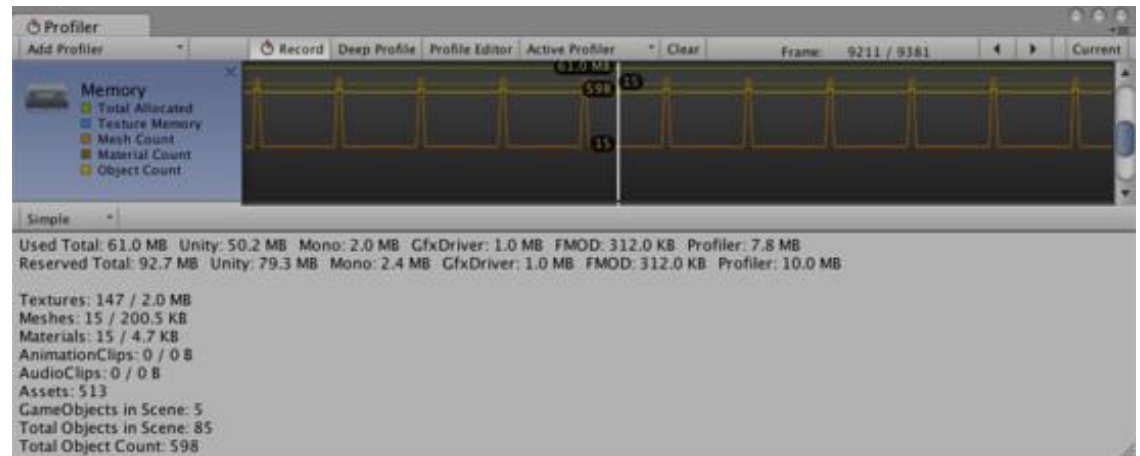
- Statistics about the rendering subsystem
  - Draw calls
  - Triangles
  - Vertices
- The lower pane shows data similar to the rendering statistics window of the editor



# 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

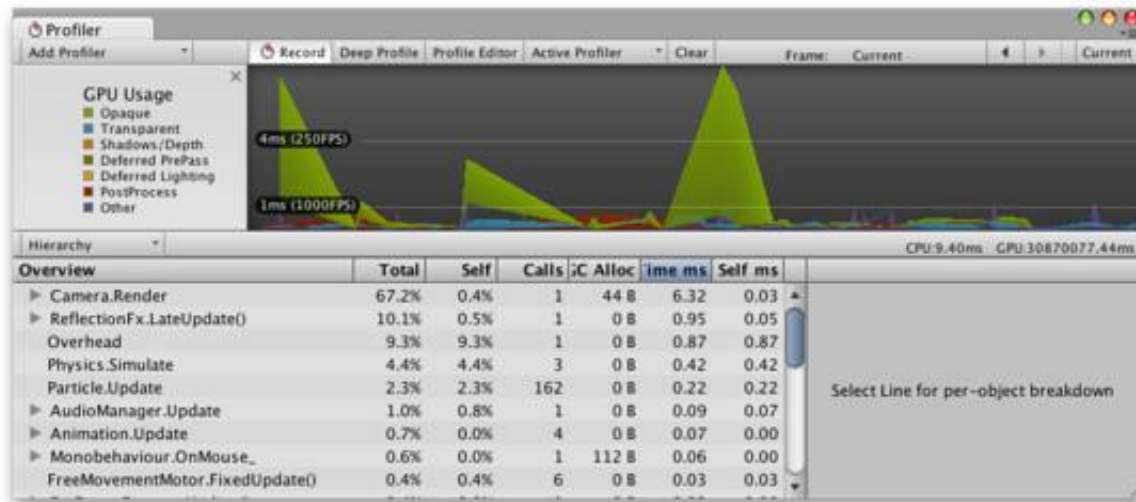


- New detailed breakdown since Unity 4.1

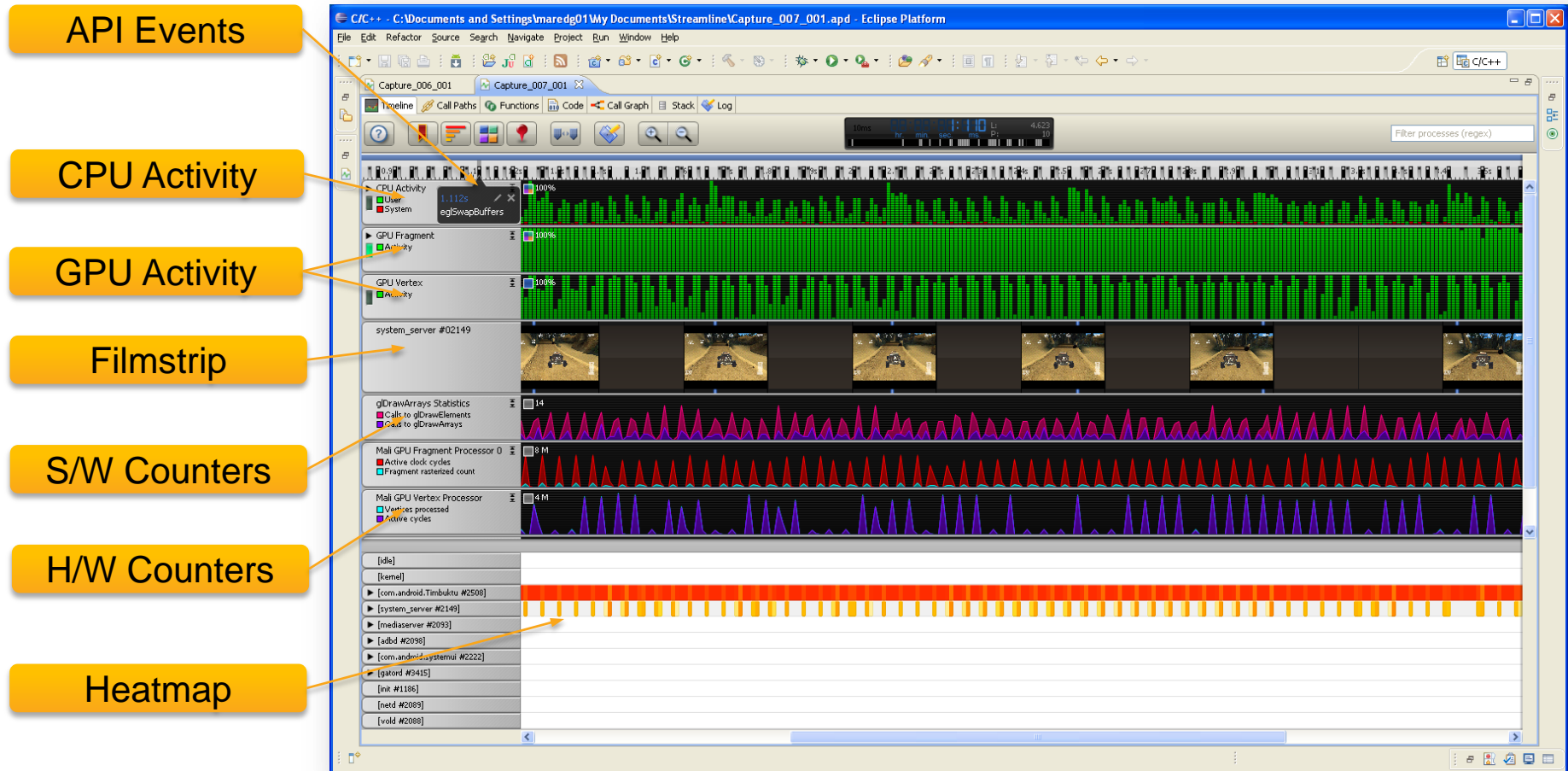


# Unity Profiler – GPU Usage

- 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 ☹️



# ARM Streamline Performance Analyzer



---

# Example application overview

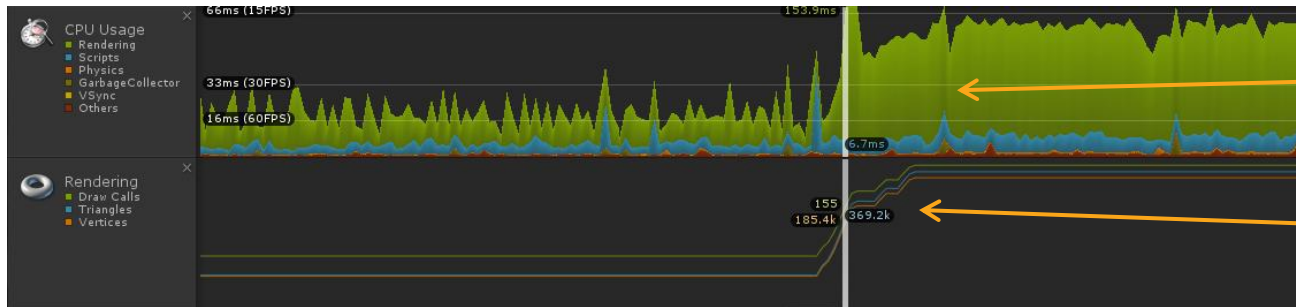
# Example application overview

---

- Composed of seven 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)
  - Atlas (texture atlas)
- 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
  - Positioning is made in screen space and based on camera aspect

# Geometry Scene

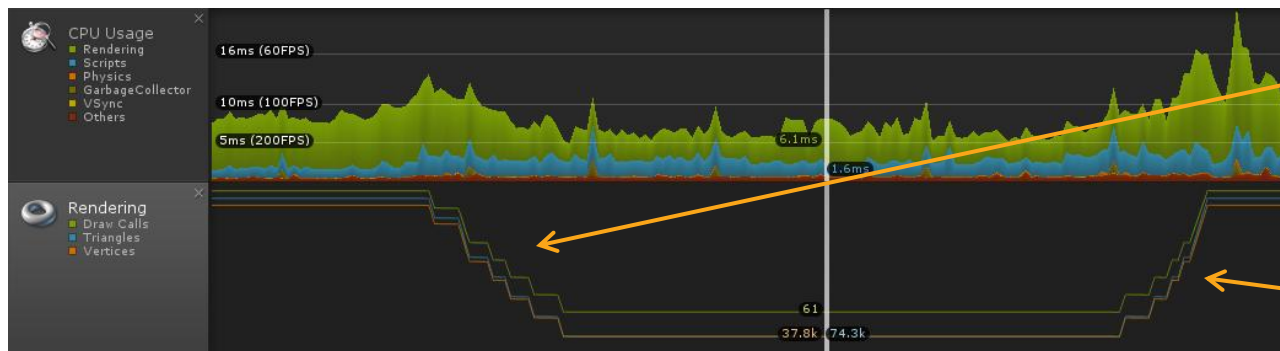
- Stressing the vertex units with lots of high poly objects
  - You can set the amount of visible objects or a target FPS value



More time spent on rendering (green)

Increasing the number of visible objects

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

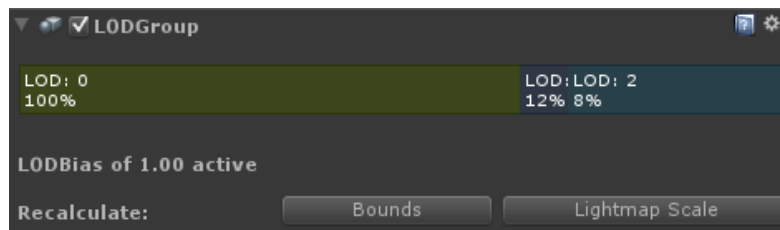


Most of the objects are out of camera view, frustum culling is working

Most of the objects are hidden behind front ones, but occlusion culling is not enabled

# LOD Scene

- LODGroup with three different levels

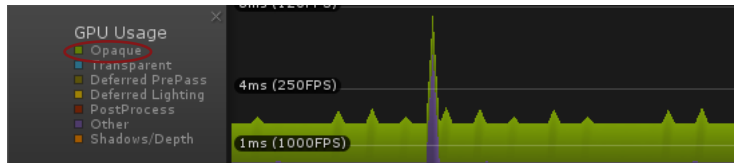


- Triangles and Vertices spikes match LOD level switch, draw calls are dependent on frustum culling



# Particles Scene

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

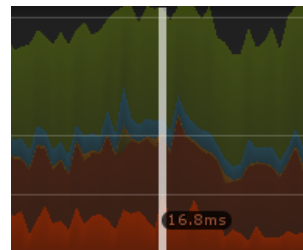


Geometry Scene



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



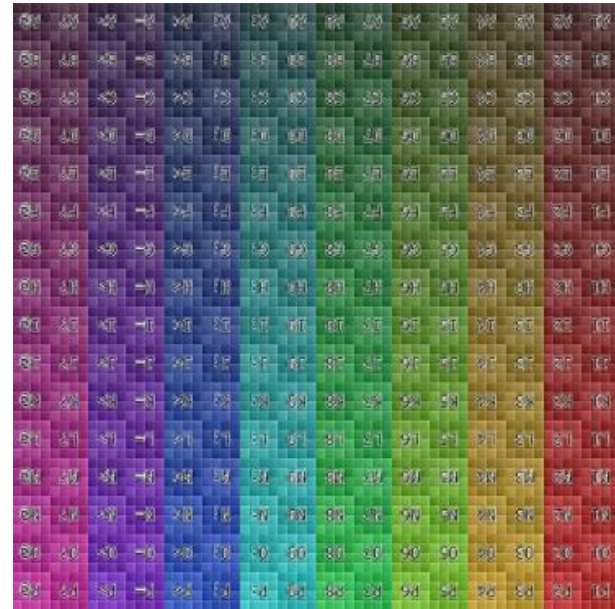
# Texture Scene

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



Uncompressed, bilinear filtering

```
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
```



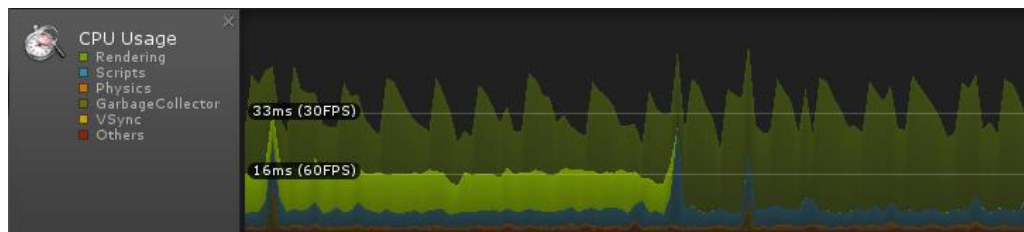
ETC compressed, trilinear filtering, MIP mapped

```
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
```

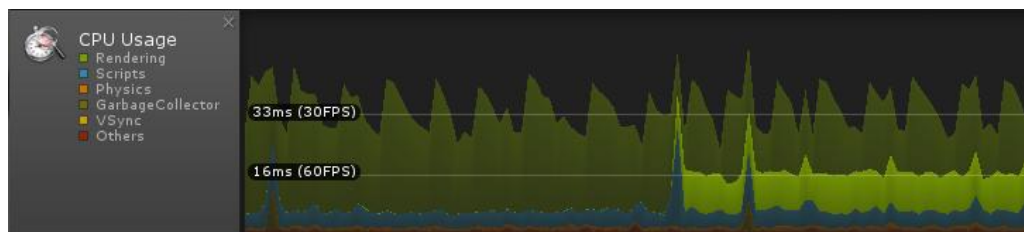


# Overdraw Scene

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

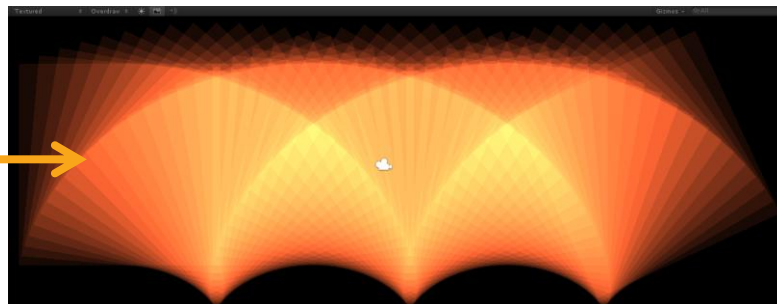


Overview	Total	Time ms
▶ Graphics.PresentAndSync	42.2%	16.35
▼ Camera.Render	31.1%	12.04
▼ Drawing	29.6%	11.49
▶ Render.TransparentGeometry	28.2%	10.95
Render.Prepare	0.6%	0.27
Clear	0.3%	0.12
▶ Render.OpaqueGeometry	0.1%	0.05
Camera.ImageEffects	0.0%	0.00



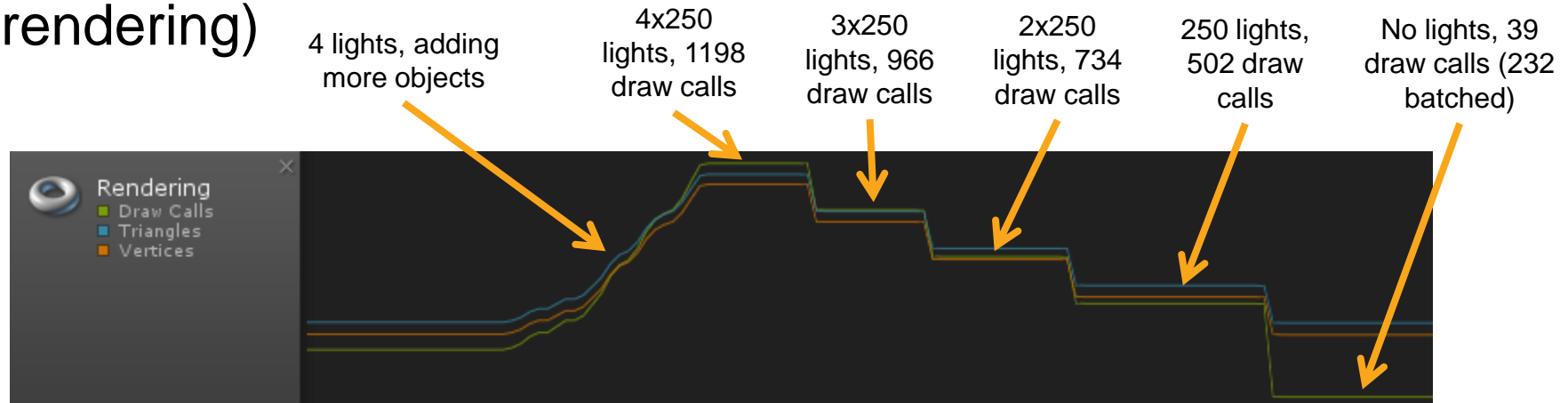
Overview	Total	Time ms
▶ Graphics.PresentAndSync	42.4%	16.09
▼ Camera.Render	31.3%	11.88
▼ Drawing	29.9%	11.32
▶ Render.OpaqueGeometry	28.5%	10.80
Render.Prepare	0.6%	0.26
Clear	0.3%	0.11
▶ Render.TransparentGeometry	0.1%	0.04
Camera.ImageEffects	0.0%	0.00

Overdraw  
render mode

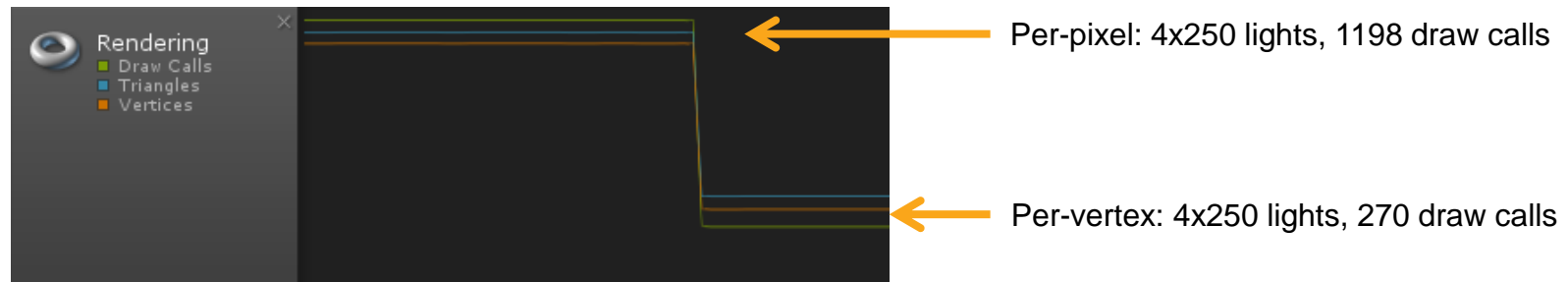


# Lights Scene (1/2)

- 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.OpacityGeometry	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.OpacityGeometry	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

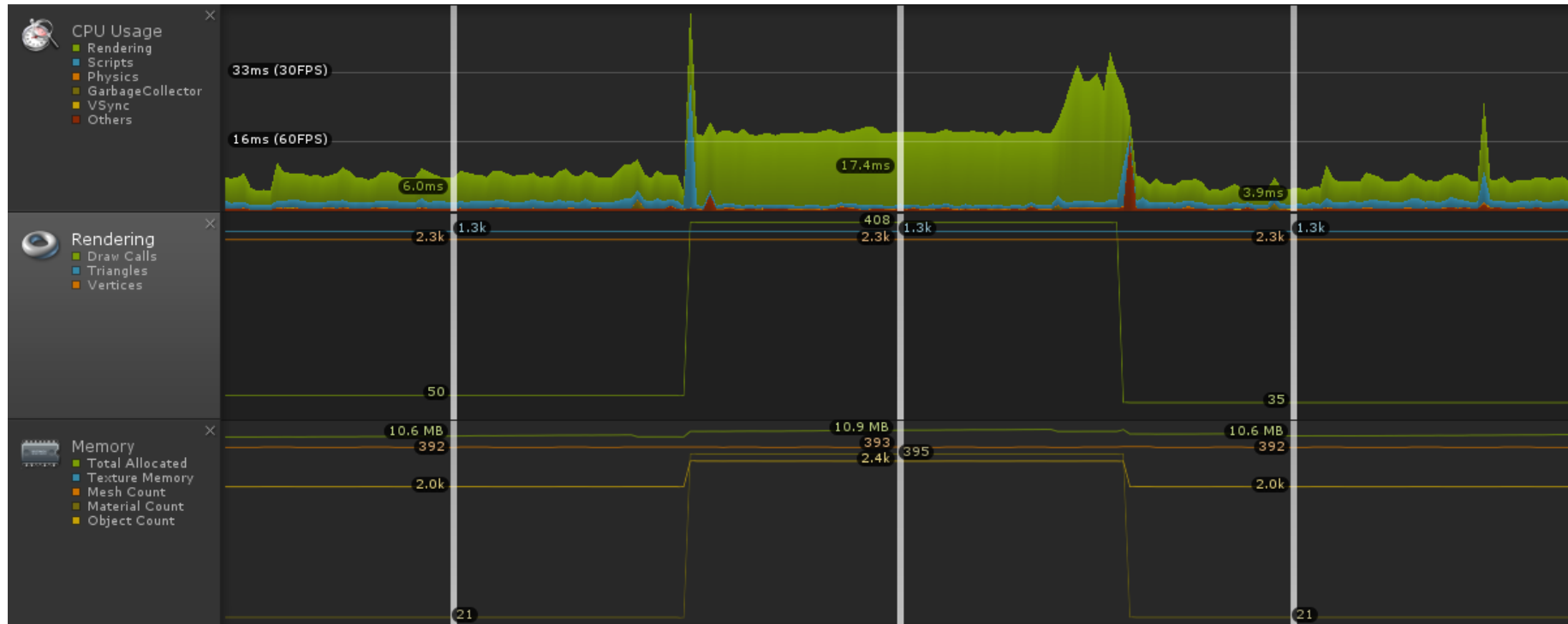
← Per-vertex lighting

# Atlas Scene (1/2)

---

- No Atlas rendering
  - Each instance has its own texture
  - Same material if multiple instances share the same texture
  - One draw call per material (multiple instances)
- Atlas rendering changing material UV
  - Each instance share the same texture atlas
  - UV coordinates are changed using `mainTextureScale` and `mainTextureOffset`
  - Leads to a new unique material per instance and breaks instancing
  - One draw call per instance
- Atlas rendering changing mesh UV
  - Each instance share the same texture atlas
  - UV coordinates are changed accessing the instance mesh
  - One draw call for all the instances (single material)

# Atlas Scene (2/2)



## No Atlas texture

- 60 FPS
- 50 draw calls, 374 batched
- 21 materials

## Atlas with material UV

- 40 FPS
- 408 draw calls, 0 batched
- 395 materials

No batching, lots of materials!

## Atlas with mesh UV

- 60 FPS
- 35 draw calls, 374 batched
- 21 materials

Batching works, less draw calls

# 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)
- [Introducing the new Memory Profiler](#) (Unity Blog)
- [Unity 4.2 has arrived](#) (Unity Blog)
  
- <http://malideveloper.arm.com/>
- [OpenGL ES 3.0 Developer Resources](#) (Mali Developer Center)
- [ASTC Texture Compression: ARM Pushes the Envelope in Graphics Technology](#) (ARM Blogs)

# The End

---

