Vulkan Game Development in Mobile

GDC 2017

Soowan Park
Graphics Engineer
Samsung Electronics
(soft.park@samsung.com)

In the beginning…

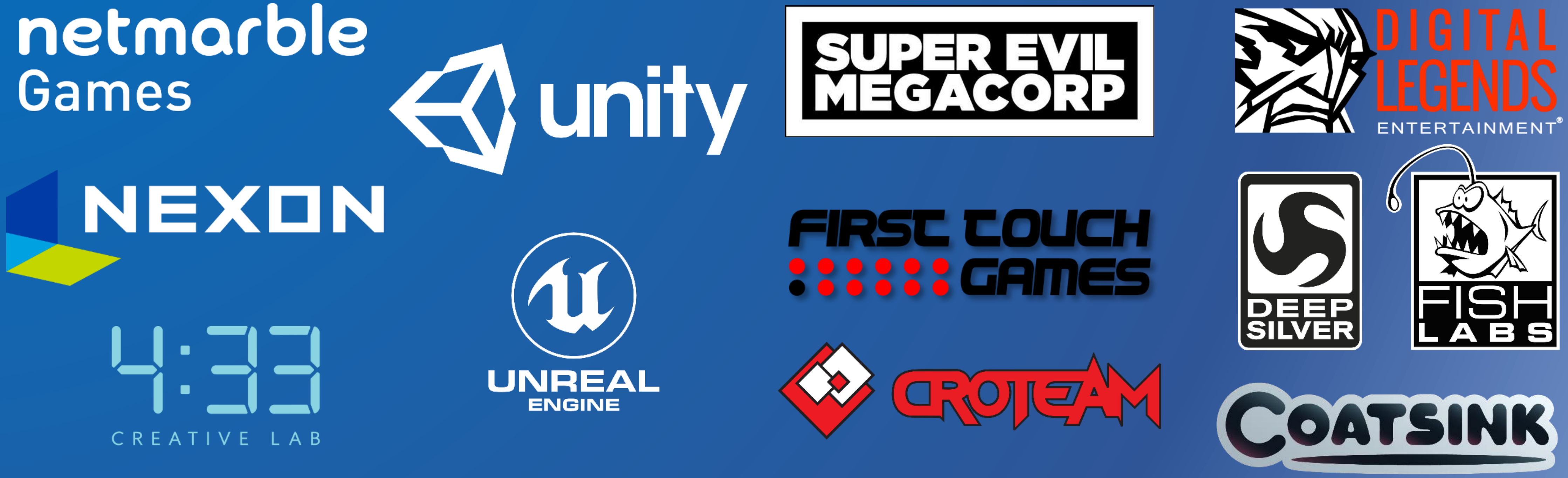
All content is based on our development experience
with Galaxy S7 spanning two chipset variants, using the ARM Mali and Qualcomm Adreno GPUs.

For whom?

- ✓ For Android Vulkan Developers.
- ✓ Developers on other platforms / markets considering to port to Android

Vulkan Partners

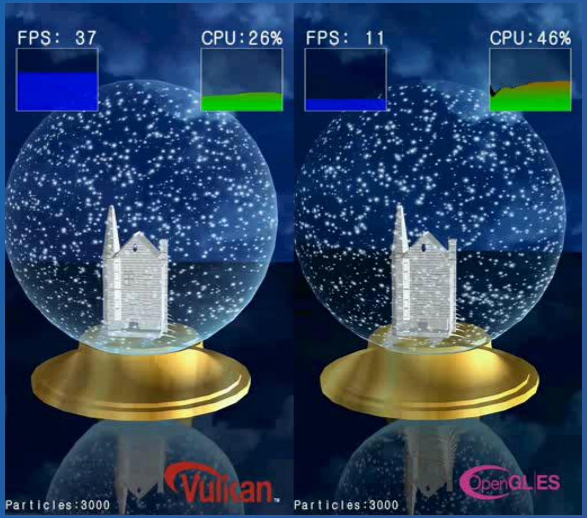
✓ We are currently working with many game studios and engine vendors to support Vulkan.

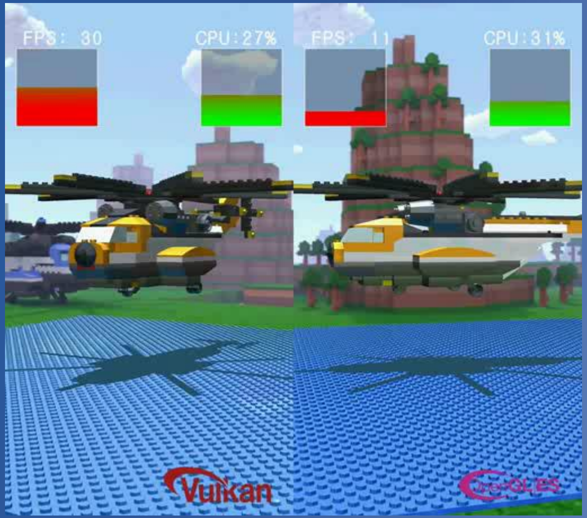


Developing Vulkan

- ✓ Our main goal is to enhance the gaming experience on mobile devices.

- ✓ OpenGL ES vs Vulkan
  - ✓ Concept demo
    - ✓ Snowball : 11 FPS -> 32 FPS
    - ✓ Lego : 11 FPS -> 26 FPS
    - ✓ Parge : 7 FPS -> 14 FPS
  - ✓ Shipping Game Titles
    - ✓ Vainglory : 51 FPS -> 59 FPS
    - ✓ HIT : 48 FPS -> 49 FPS (with more effect)
  - ✓ Upcoming Games
    - ✓ Game A: 15 FPS -> 23 FPS
    - ✓ Game B : 24 FPS -> 26 FPS
    - ✓ Game C : 21 FPS -> 24 FPS
    - ✓ …





OpenGL ES vs Vulkan

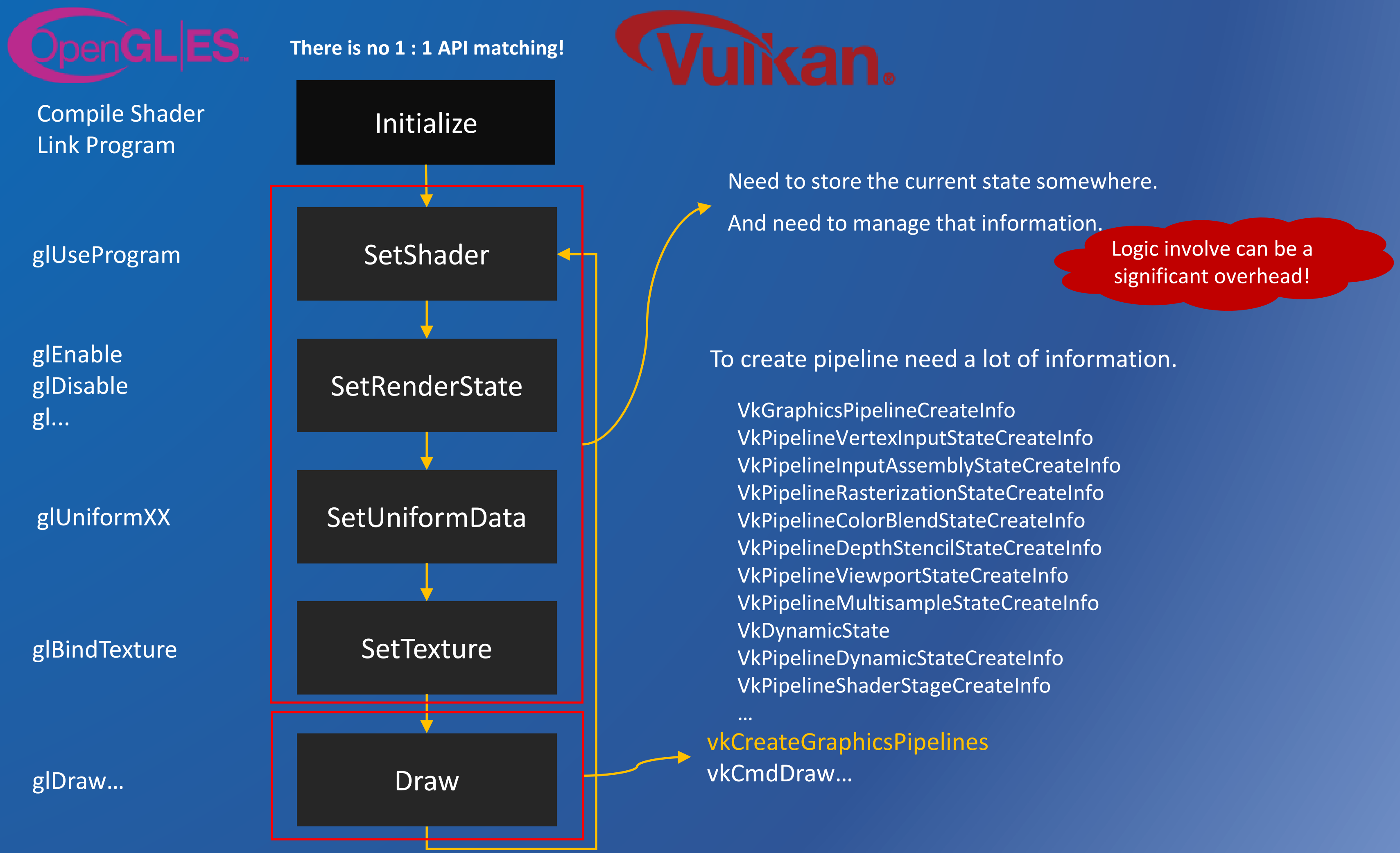
Performance improvements

Concept Demos > Real Games

Where does the performance gap between concept demos and real games come from?

The reason are as follows

- ✓ It's not easy to collect all the information needed for Vulkan in an existing game engine's "Render Interface"
- ✓ "Render interface" – the interface that is commonly found across game engines. (Just by my experience!)
- ✓ Let's think about this very simple renderer logic below.



OpenGL ES

There is no 1 : 1 API matching!

Vulkan

Compile Shader
Link Program

Initialize

glUseProgram

SetShader

glEnable
glDisable
gl…

SetRenderState

glUniformXX

SetUniformData

glBindTexture

SetTexture

glDraw…

Draw

Need to store the current state somewhere.

And need to manage that information.

Logic involve can be a significant overhead!

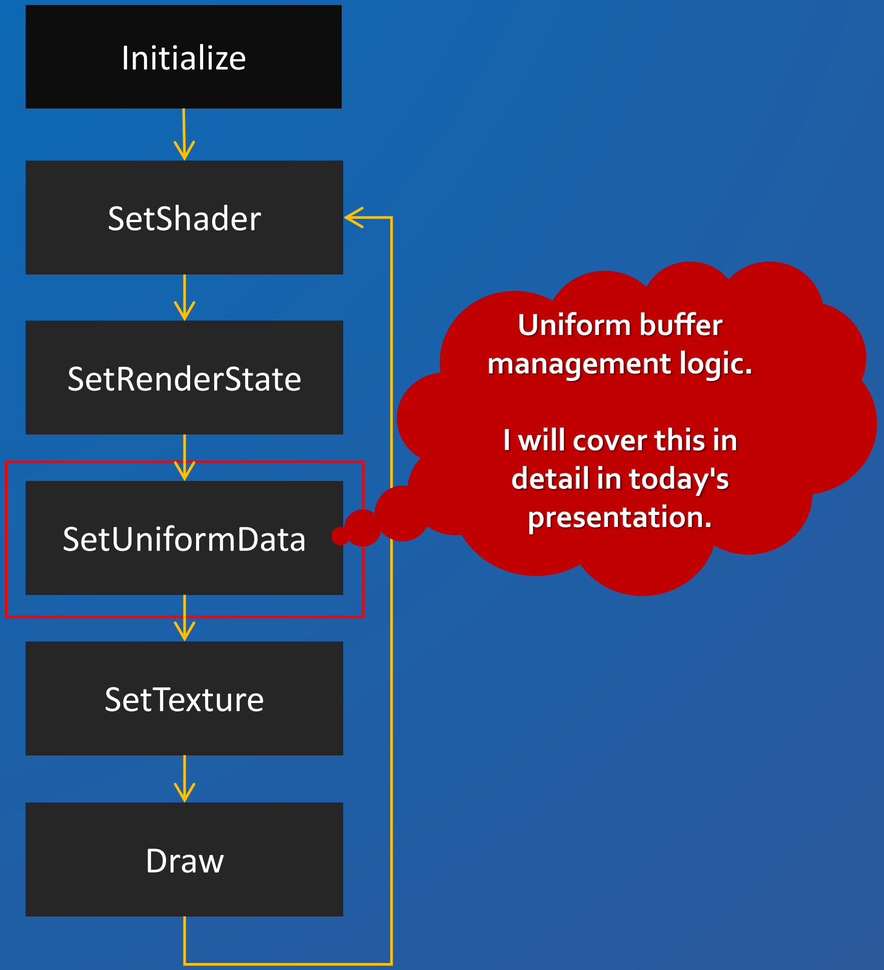
To create pipeline need a lot of information.

VkGraphicsPipelineCreateInfo
VkPipelineVertexInputStateCreateInfo
VkPipelineInputAssemblyStateCreateInfo
VkPipelineRasterizationStateCreateInfo
VkPipelineColorBlendStateCreateInfo
VkPipelineDepthStencilStateCreateInfo
VkPipelineViewportStateCreateInfo
VkPipelineMultisampleStateCreateInfo
VkDynamicState
VkPipelineDynamicStateCreateInfo
VkPipelineShaderStageCreateInfo
…

vkCreateGraphicsPipelines
vkCmdDraw…

Optimization on Android devices

- ✓ We should optimize the renderer logic for the Vulkan API within that interface !
- ✓ Below is a list of optimization points that we have experienced during porting games and creating concept demos.



Uniform buffer management logic.

I will cover this in detail in today's presentation.

Managing VkRenderPass, VkFramebuffer

Persistent PipelineCache

Reducing duplicated API calls

Clear screen cost

Managing VkPipeline

Geometry Sorting

Let's talk about the uniform buffer.

What is the best way

to implement uniform buffer logic?

For that, I tested 6 cases.

Every test is based from my experience.

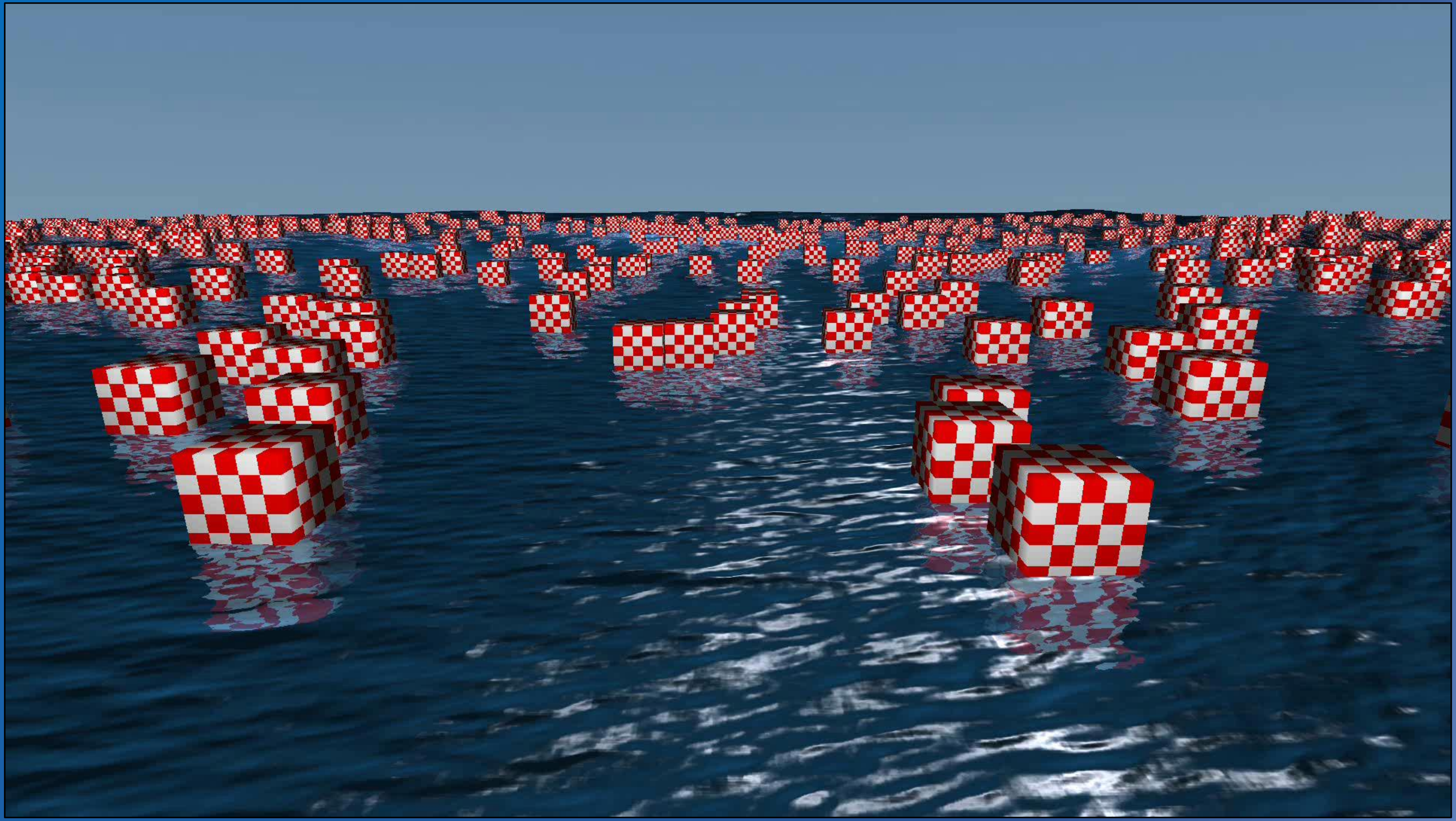
Structural Experiments

- 1st Test – Brute Force
- 2nd Test – Memory Manager
- 3rd Test – Dynamic Offsets
- 4th Test – Ideal Condition

Additional Experiments

- 5th Test – Memory property flags on Mobile
- 6th Test – PushConstants

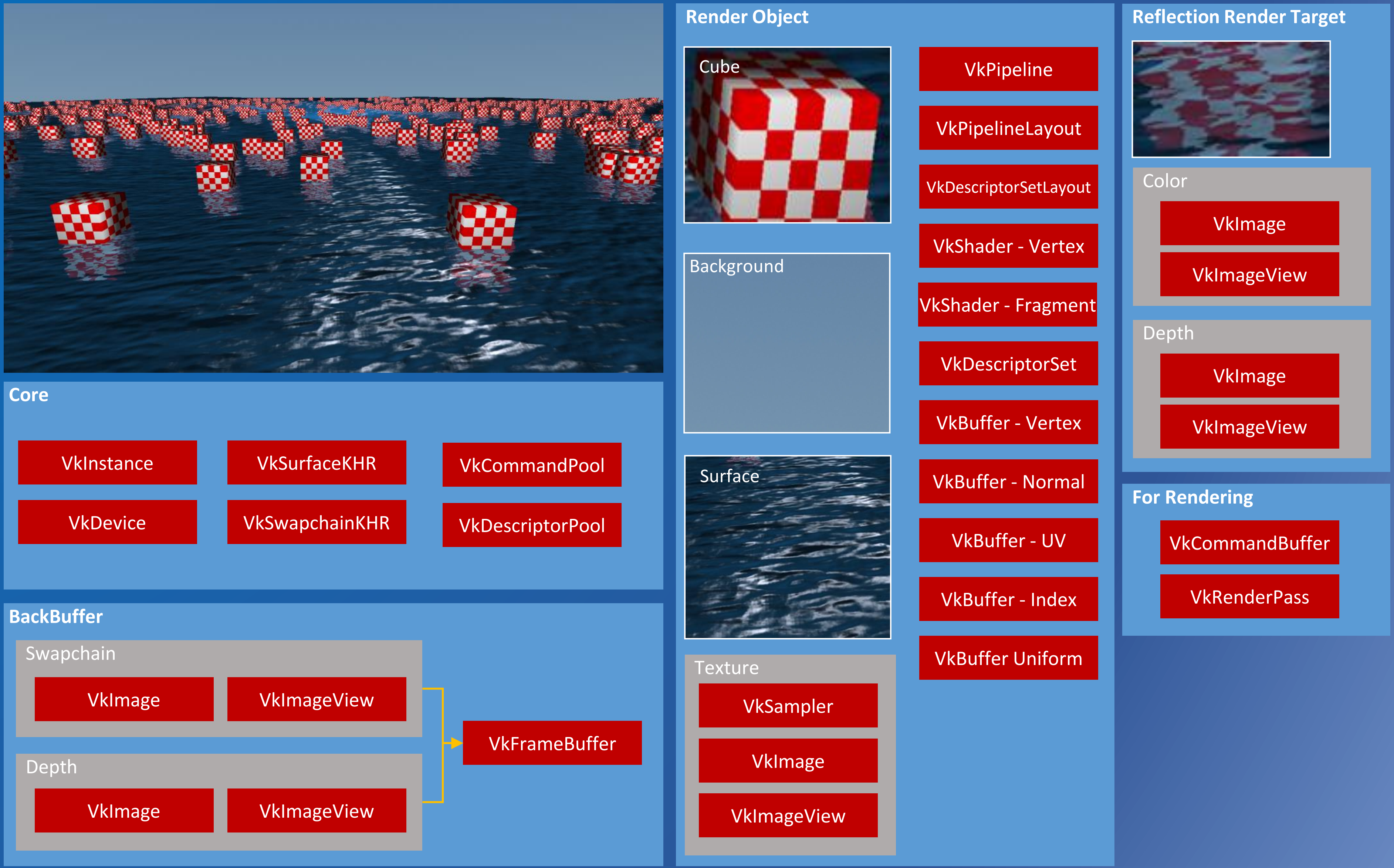
Test Project : OceanBox



Developed sample specifically to test uniform buffer performance. Planning to upload source code (subject to approval!) to: https://github.com/itrainl4/OceanBox

OceanBox overview

※ VkDeviceMemory is omitted.



Core

- VkInstance
- VkSurfaceKHR
- VkCommandPool
- VkDevice
- VkSwapchainKHR
- VkDescriptorPool

BackBuffer

Swapchain

- VkImage
- VkImageView

Depth

- VkImage
- VkImageView

VkFrameBuffer

Render Object

Cube

Background

Surface

Texture

- VkSampler
- VkImage
- VkImageView

- VkPipeline
- VkPipelineLayout
- VkDescriptorSetLayout
- VkShader - Vertex
- VkShader - Fragment
- VkDescriptorSet
- VkBuffer - Vertex
- VkBuffer - Normal
- VkBuffer - UV
- VkBuffer - Index
- VkBuffer Uniform

Reflection Render Target

Color

- VkImage
- VkImageView

Depth

- VkImage
- VkImageView

For Rendering

- VkCommandBuffer
- VkRenderPass

SAMSUNG

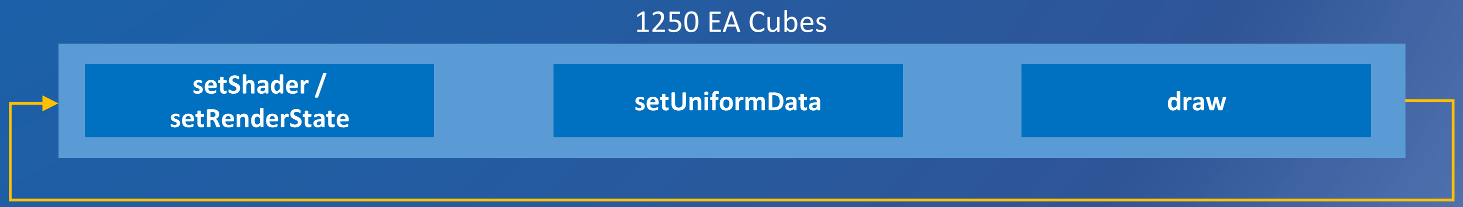
Vulkan

Test scenario

- Test Scene Information
  - 1 Background
  - 1250 Cubes, Update position
  - 1 Surface, 150x150 Grid Simulation (2 iteration per frame)

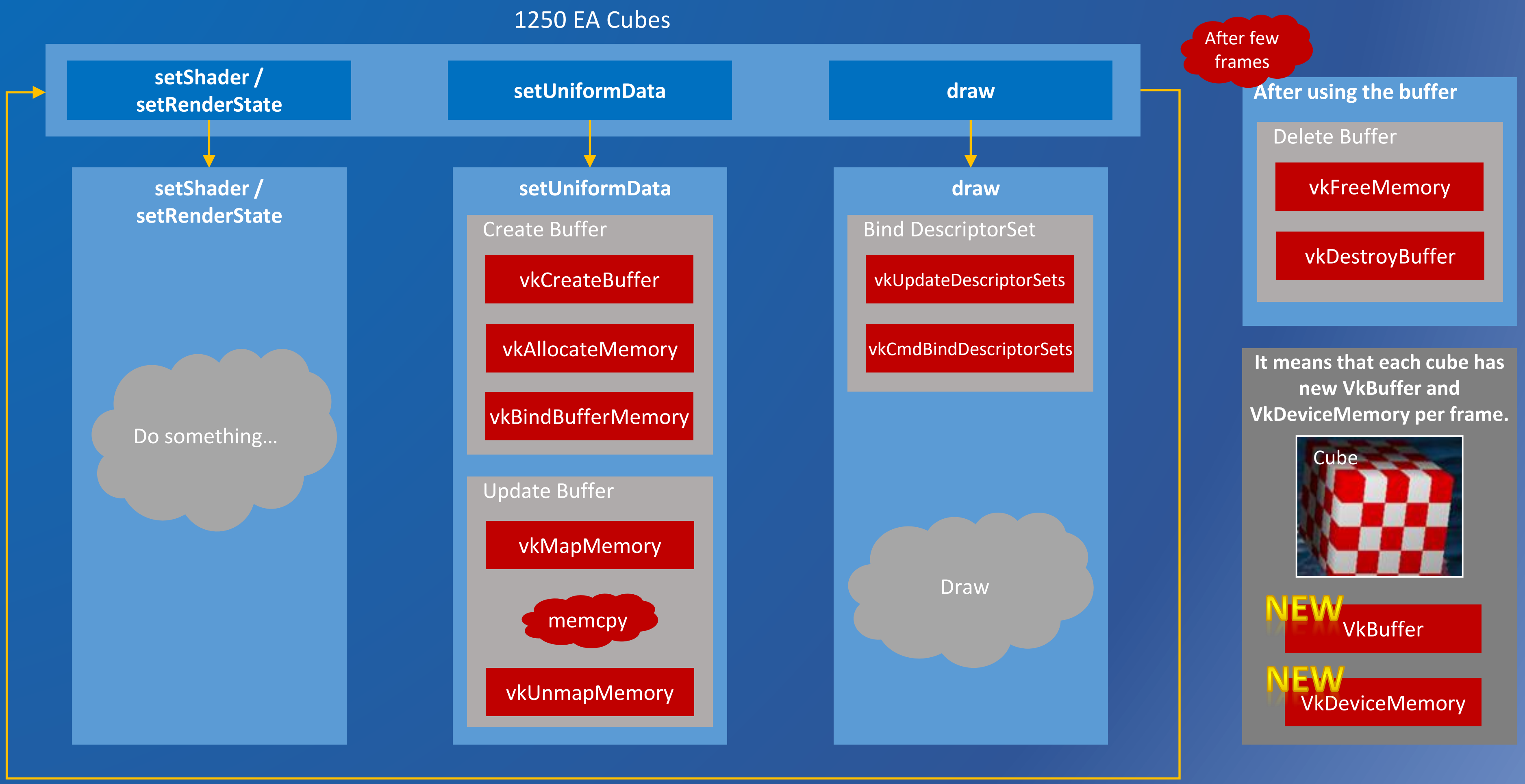
- Profiling Environment
  - Devices : G930F, G930V (MALI, Adreno)
  - Duration : 10 mins

- Assume that all of the logic (except the uniform buffer) is optimized and the texture information is unchanged for accurate testing in real time.
- The drawing function call sequence for 1250 cubes is like below.



1st Test – Brute Force

- Let’s test worst case.
- Create VkBuffer and Allocate VkDeviceMemory every draw call.



1250 EA Cubes

setShader / setRenderState

setUniformData

draw

setShader / setRenderState

Do something…

setUniformData

Create Buffer

vkCreateBuffer

vkAllocateMemory

vkBindBufferMemory

Update Buffer

vkMapMemory

memcpy

vkUnmapMemory

draw

Bind DescriptorSet

vkUpdateDescriptorSets

vkCmdBindDescriptorSets

Draw

After few frames

After using the buffer

Delete Buffer

vkFreeMemory

vkDestroyBuffer

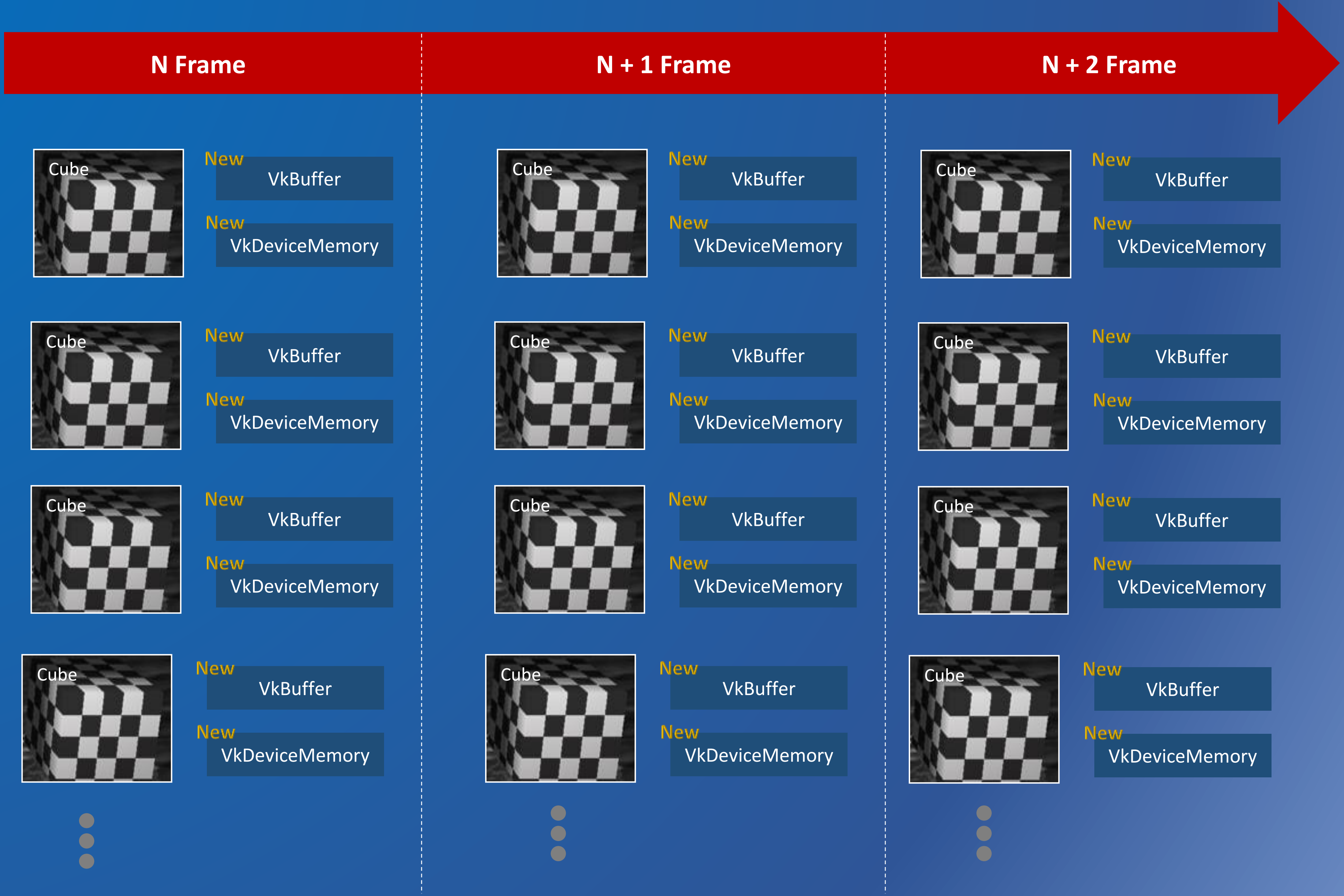
It means that each cube has new VkBuffer and VkDeviceMemory per frame.

Cube

NEW VkBuffer

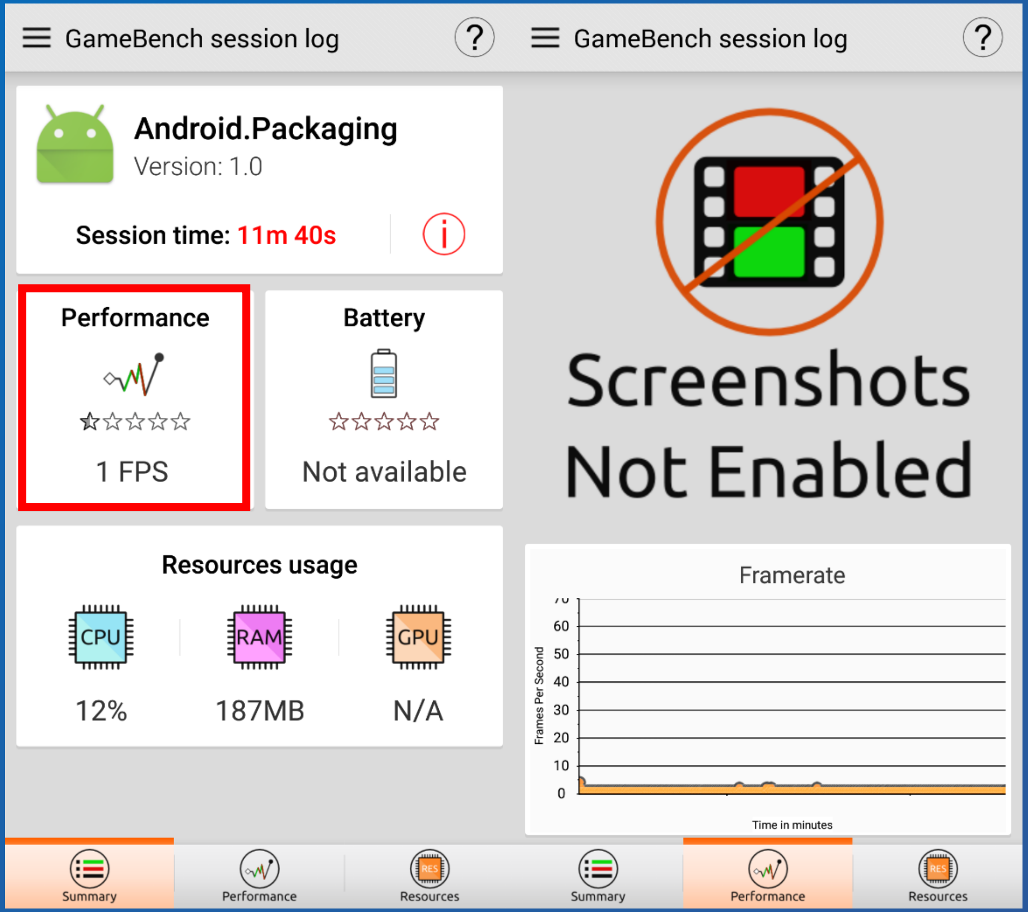
NEW VkDeviceMemory

1st Test – Brute Force



N Frame	N + 1 Frame	N + 2 Frame
Cube New VkBuffer New VkDeviceMemory	Cube New VkBuffer New VkDeviceMemory	Cube New VkBuffer New VkDeviceMemory
Cube New VkBuffer New VkDeviceMemory	Cube New VkBuffer New VkDeviceMemory	Cube New VkBuffer New VkDeviceMemory
Cube New VkBuffer New VkDeviceMemory	Cube New VkBuffer New VkDeviceMemory	Cube New VkBuffer New VkDeviceMemory
Cube New VkBuffer New VkDeviceMemory	Cube New VkBuffer New VkDeviceMemory	Cube New VkBuffer New VkDeviceMemory

1st Test – Brute Force



GameBench session log

Android.Packaging

Version: 1.0

Session time: 11m 40s

Performance

1 FPS

Battery

Not available

Resources usage

CPU	RAM	GPU
12%	187MB	N/A

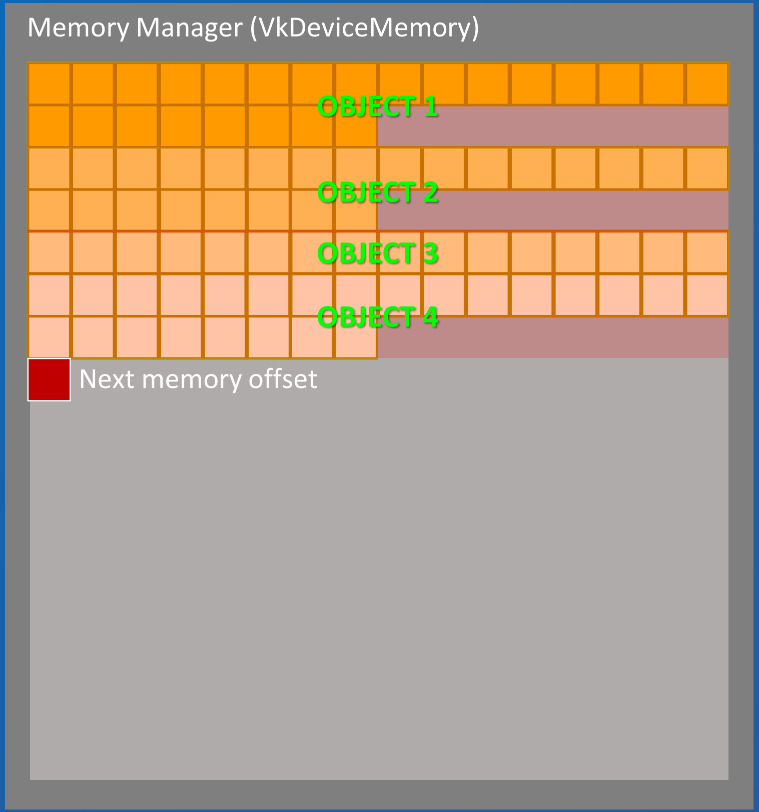
GameBench session log

Screenshots Not Enabled

1 FPS is OK because it’s worst case.

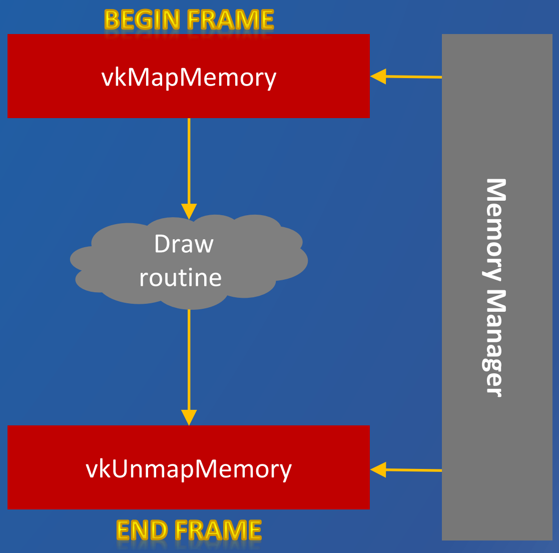
2nd Test – Memory Manager

- Let's make memory manager assign VkDeviceMemory to each object.



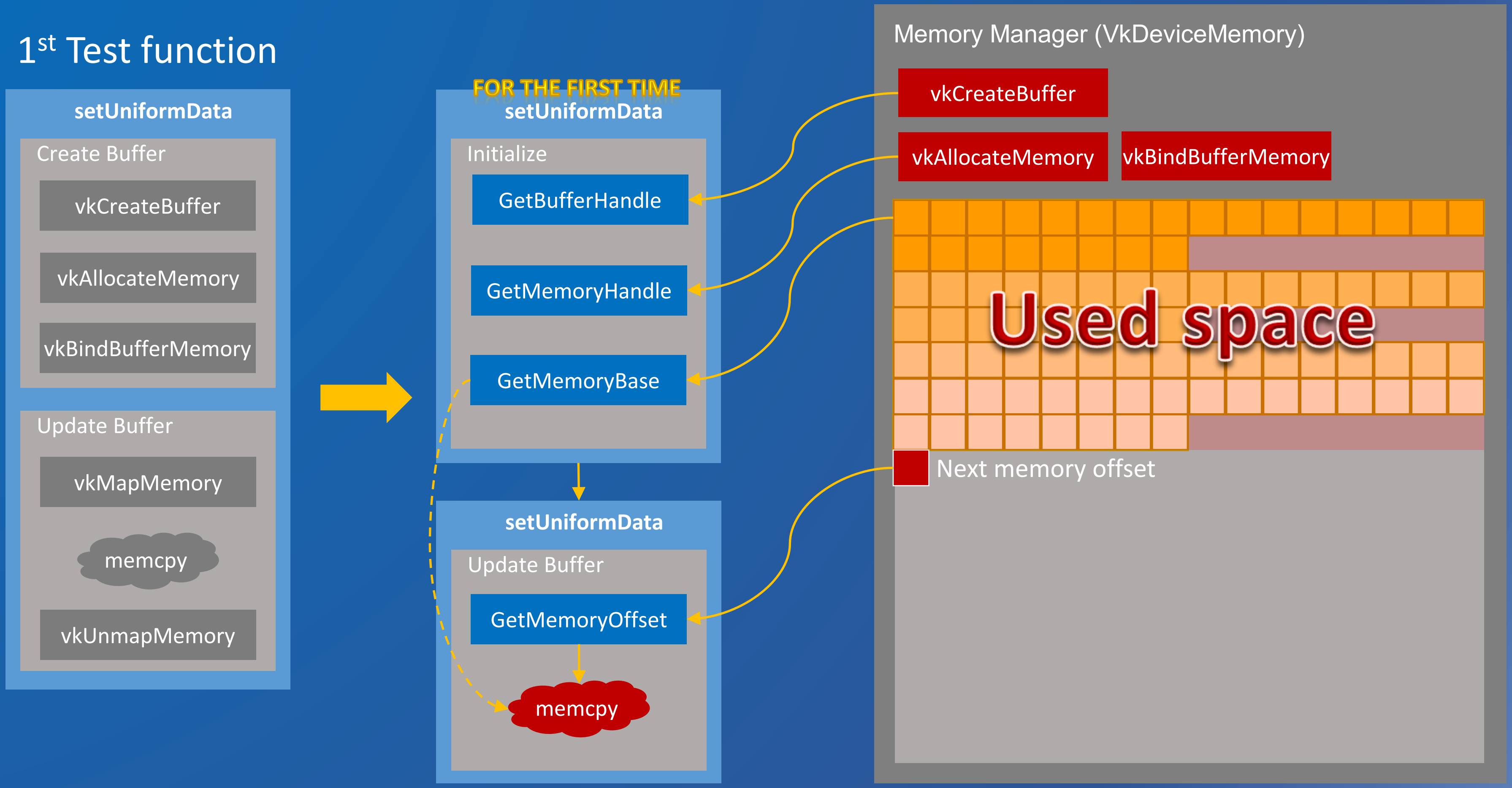
※ Should be take care with given alignment from physical device limits.
Please refer to "Vulkan Case Study" at 2016 Khronos DevU in Seoul.

With the memory manager,
you do not have to call vkMapMemory every time.



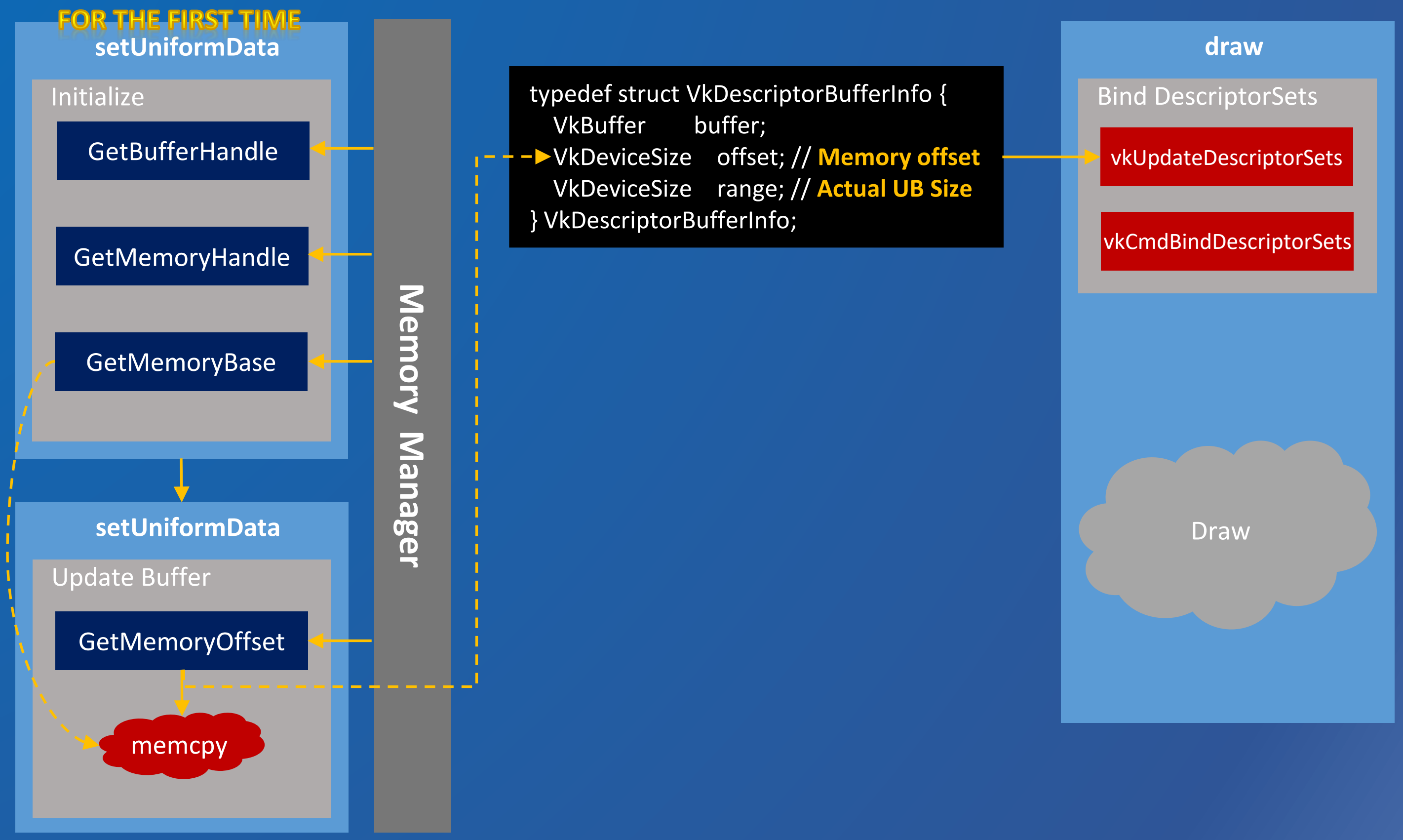
2nd Test – Memory Manager

- So functionality should be changed like this.



2nd Test – Memory Manager

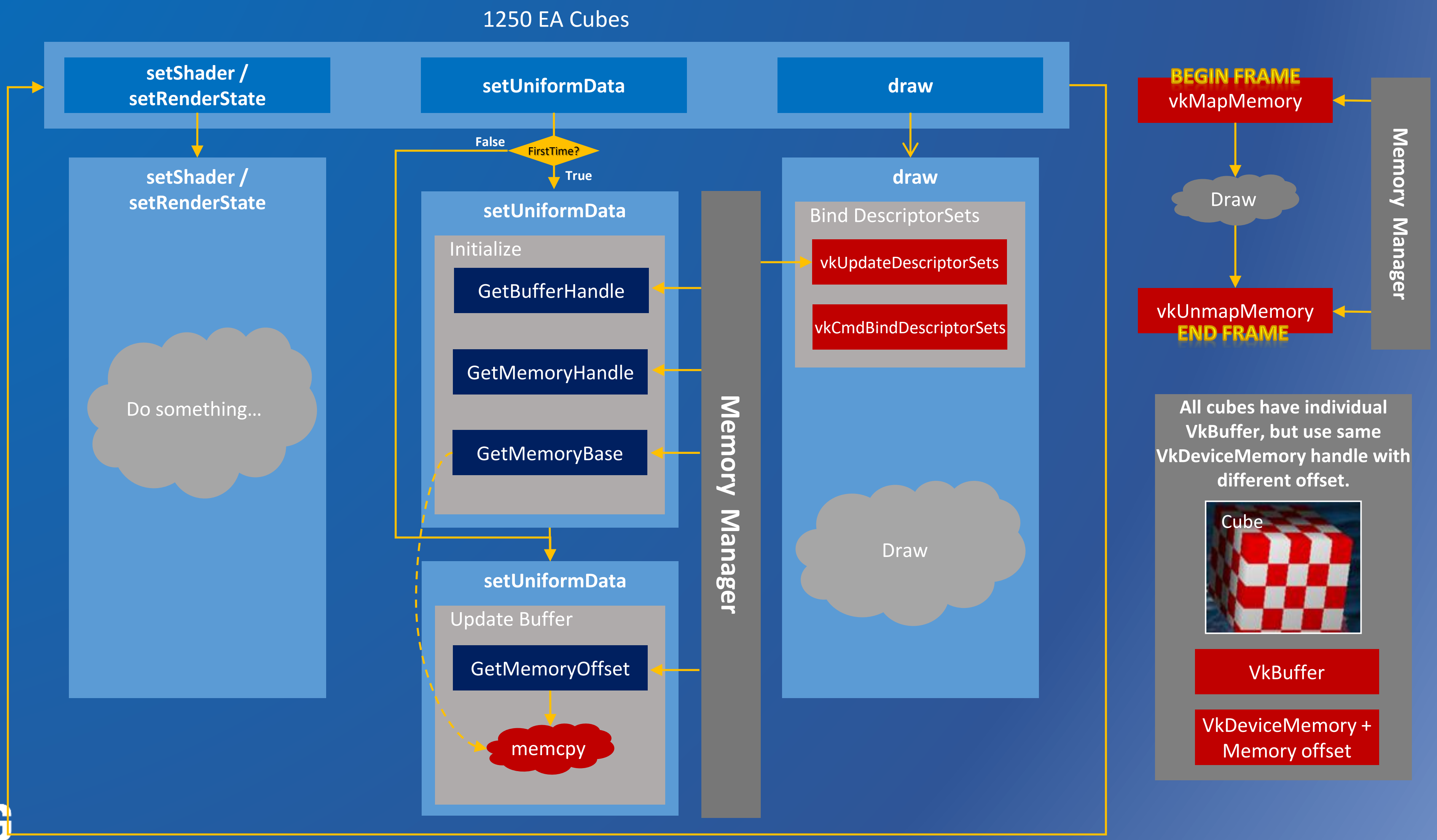
- And you should update VkDescriptorSet using appropriate offsets.



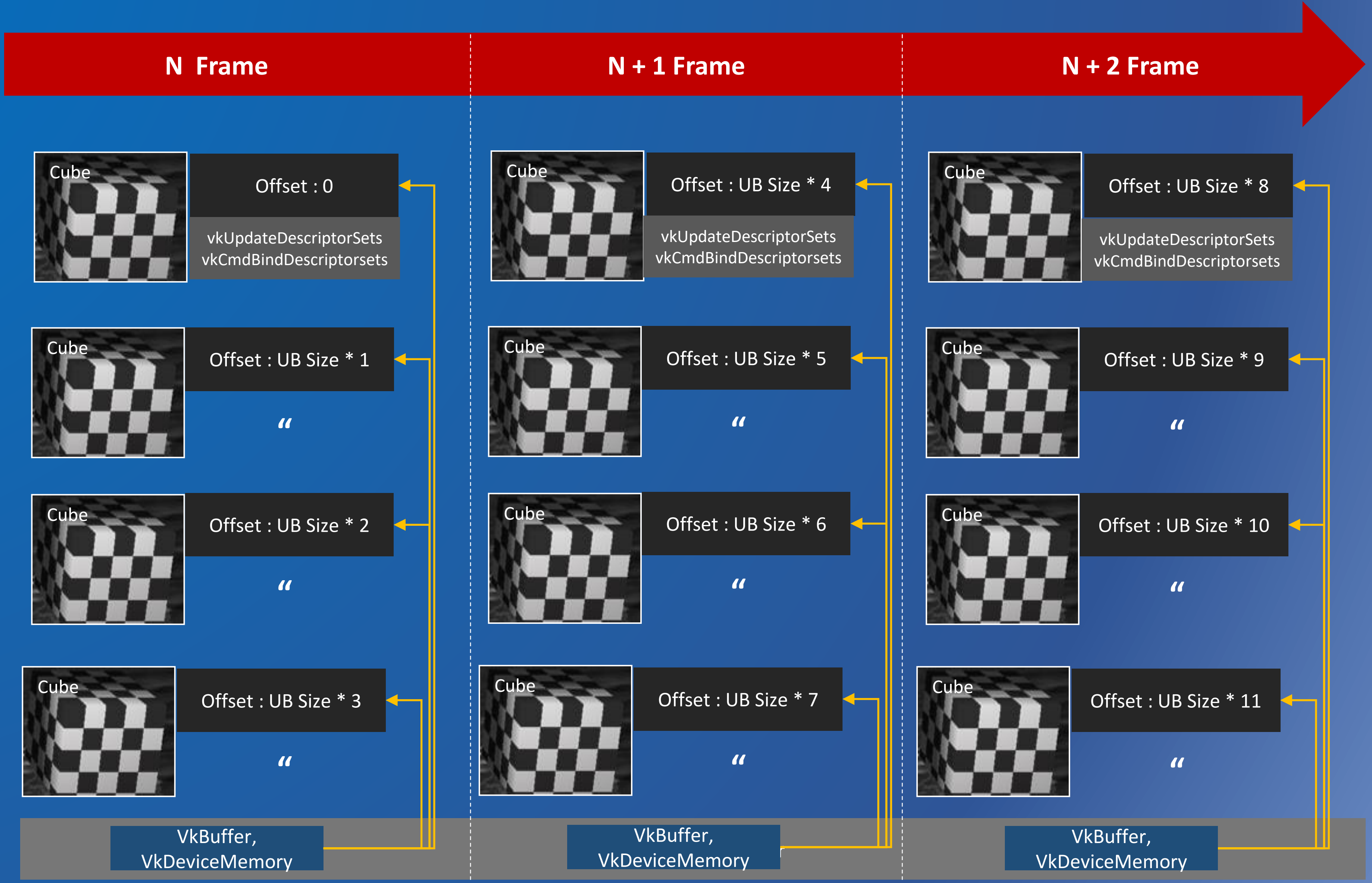
```
typedef struct VkDescriptorBufferInfo {
   VkBuffer       buffer;
   VkDeviceSize   offset; // Memory offset
   VkDeviceSize   range; // Actual UB Size
} VkDescriptorBufferInfo;
```

2nd Test – Memory Manager

- The overall logic is as follows.



2nd Test – Memory Manager



N Frame

Cube — Offset : 0 — vkUpdateDescriptorSets vkCmdBindDescriptorsets

Cube — Offset : UB Size * 1 — “

Cube — Offset : UB Size * 2 — “

Cube — Offset : UB Size * 3 — “

VkBuffer, VkDeviceMemory

N + 1 Frame

Cube — Offset : UB Size * 4 — vkUpdateDescriptorSets vkCmdBindDescriptorsets

Cube — Offset : UB Size * 5 — “

Cube — Offset : UB Size * 6 — “

Cube — Offset : UB Size * 7 — “

VkBuffer, VkDeviceMemory

N + 2 Frame

Cube — Offset : UB Size * 8 — vkUpdateDescriptorSets vkCmdBindDescriptorsets

Cube — Offset : UB Size * 9 — “

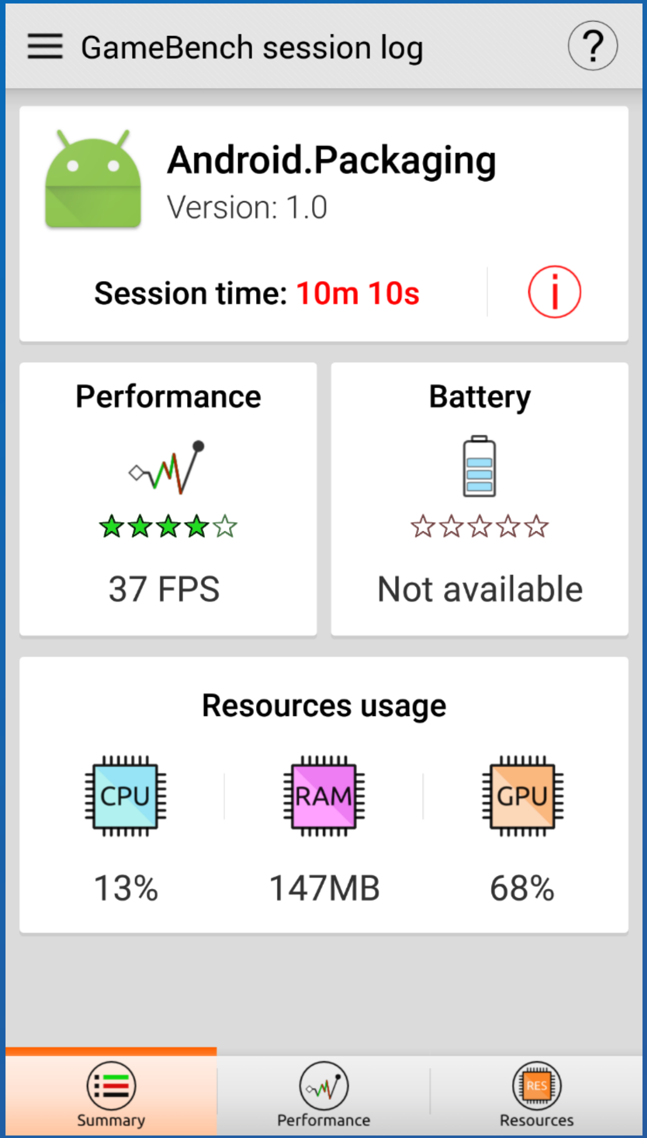
Cube — Offset : UB Size * 10 — “

Cube — Offset : UB Size * 11 — “

VkBuffer, VkDeviceMemory

※ SwapChain count related logic should be considered.

2nd Test – Memory Manager



GameBench session log

Android.Packaging

Version: 1.0

Session time: 10m 10s

Performance

★★★★☆

37 FPS

Battery

☆☆☆☆☆

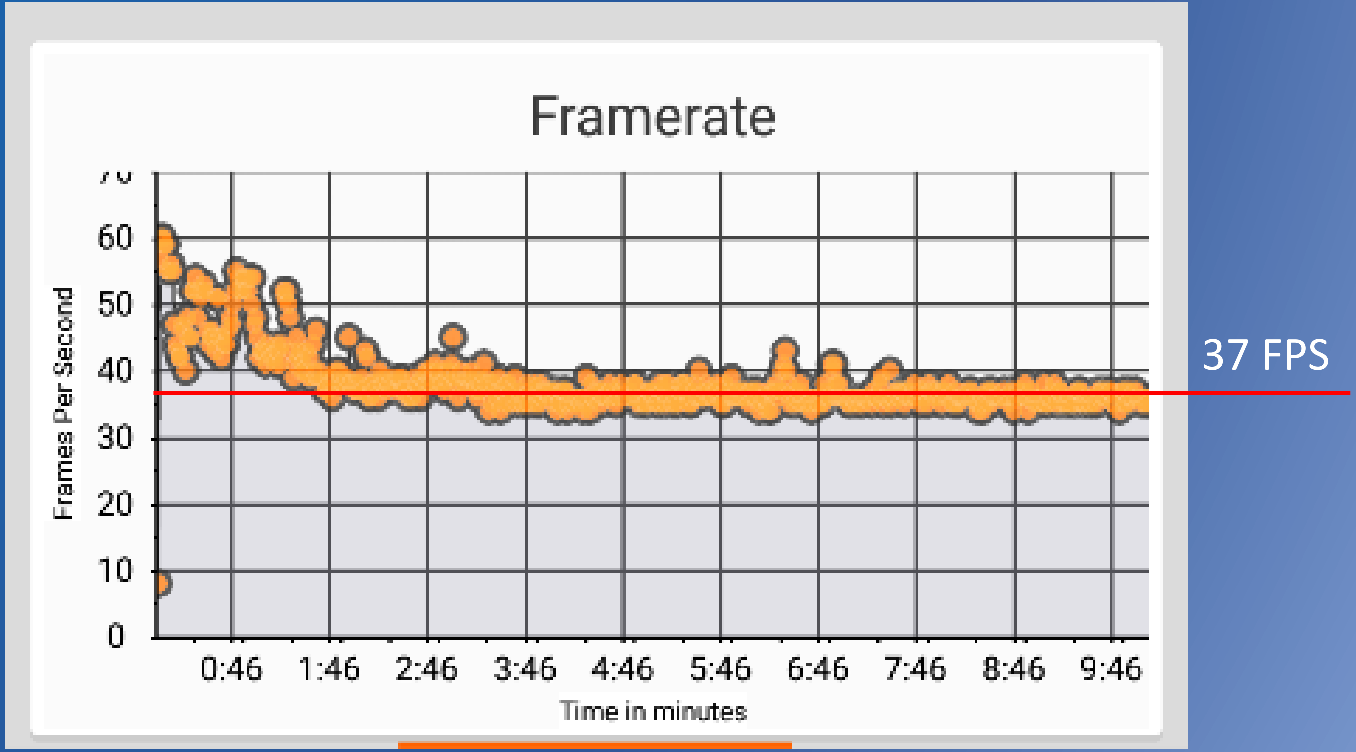
Not available

Resources usage

CPU 13%

RAM 147MB

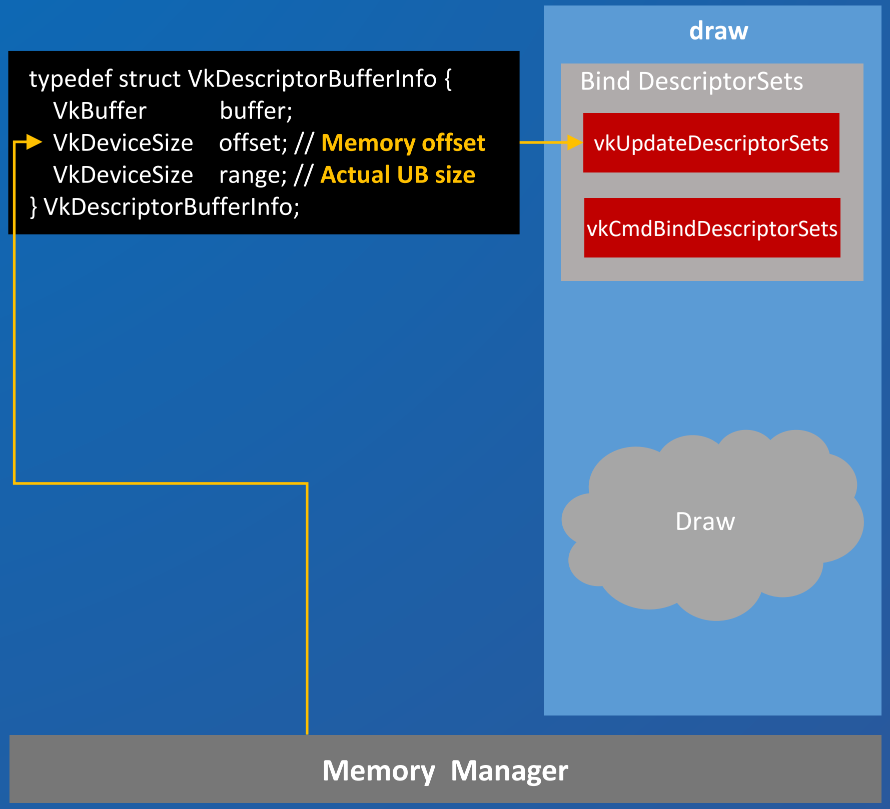
GPU 68%



3rd Test – Dynamic Offsets

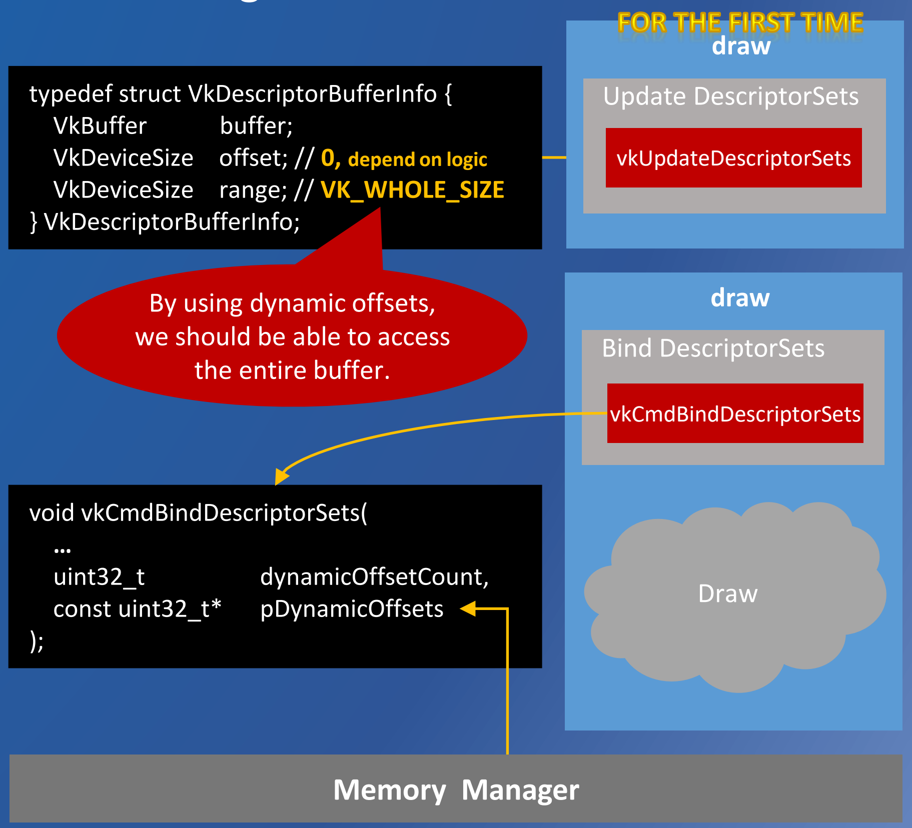
- Let's skip vkUpdateDescriptorSets API using dynamic offsets.

2nd Test logic



```
typedef struct VkDescriptorBufferInfo {
    VkBuffer        buffer;
    VkDeviceSize    offset; // Memory offset
    VkDeviceSize    range; // Actual UB size
} VkDescriptorBufferInfo;
```

3rd Test logic



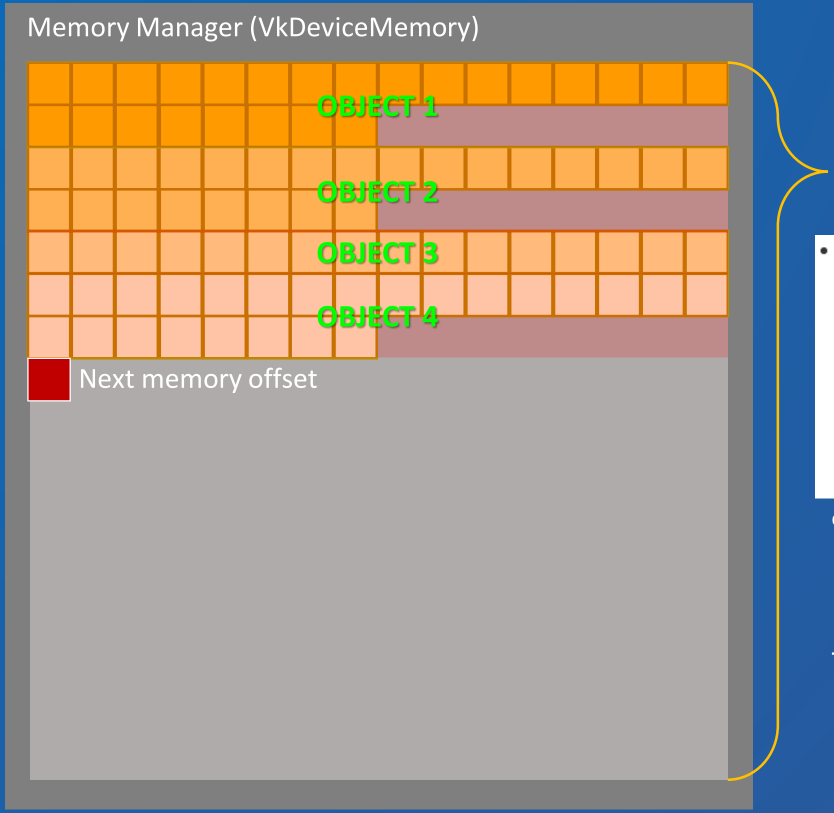
```
typedef struct VkDescriptorBufferInfo {
    VkBuffer        buffer;
    VkDeviceSize    offset; // 0, depend on logic
    VkDeviceSize    range; // VK_WHOLE_SIZE
} VkDescriptorBufferInfo;
```

By using dynamic offsets, we should be able to access the entire buffer.

```
void vkCmdBindDescriptorSets(
    …
    uint32_t            dynamicOffsetCount,
    const uint32_t*     pDynamicOffsets
);
```

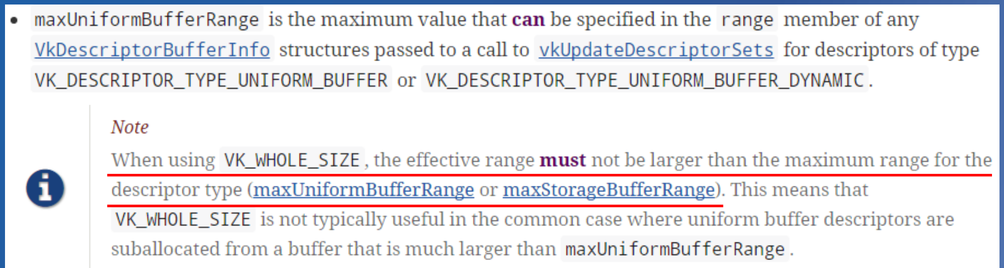
3rd Test – Dynamic Offsets

- Memory manager is almost the same, but there is a limitation on the VkDeviceMemory size.



```
typedef struct VkDescriptorBufferInfo {
   VkBuffer          buffer;
   VkDeviceSize    offset; // 0, depend on logic
   VkDeviceSize    range; // VK_WHOLE_SIZE
} VkDescriptorBufferInfo;
```

Size <= VkPhysicalDeviceLimits::maxUniformBufferRange



- `maxUniformBufferRange` is the maximum value that **can** be specified in the `range` member of any `VkDescriptorBufferInfo` structures passed to a call to `vkUpdateDescriptorSets` for descriptors of type `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` or `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC`.

Note

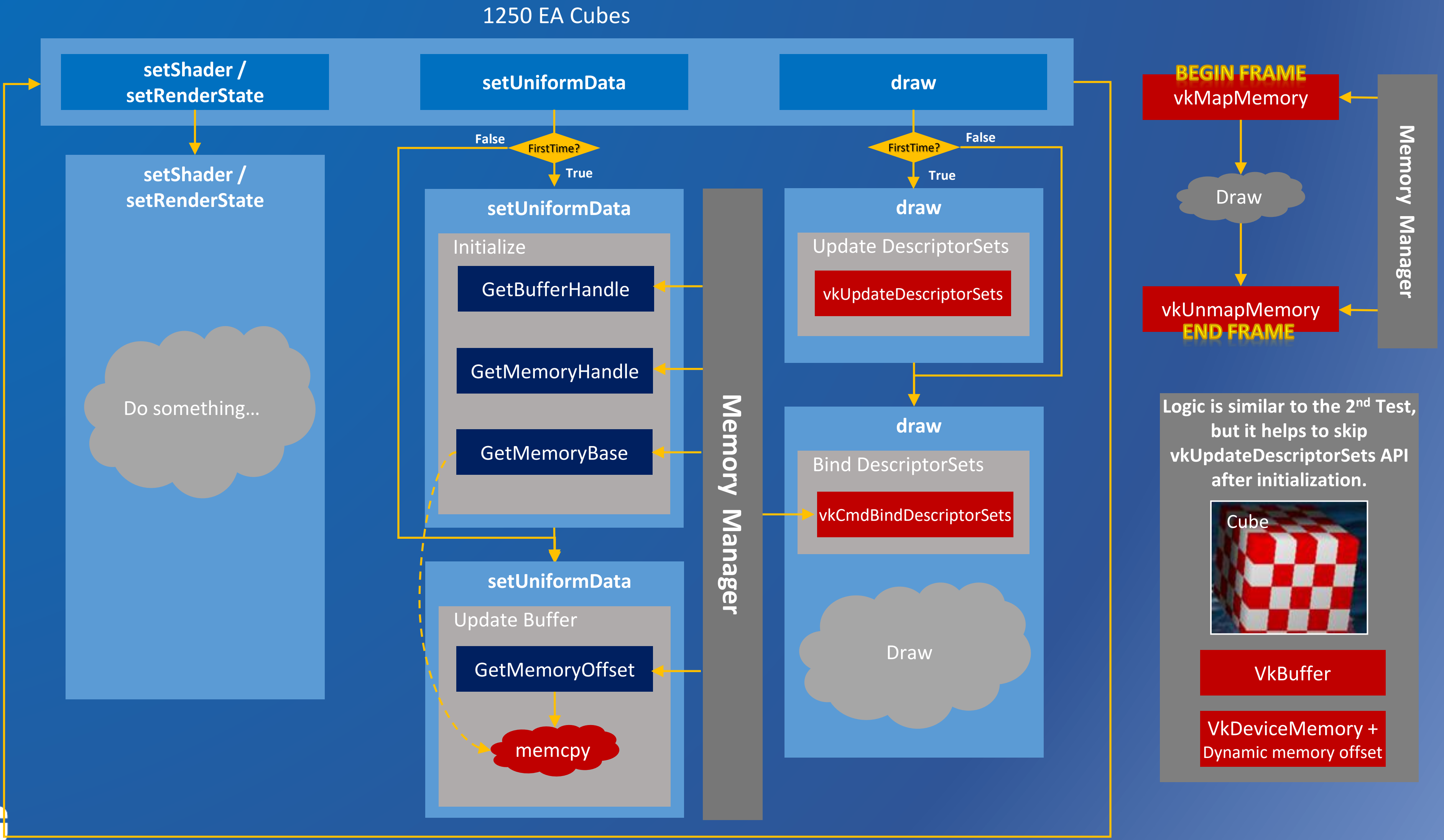
When using `VK_WHOLE_SIZE`, the effective range **must** not be larger than the maximum range for the descriptor type (maxUniformBufferRange or maxStorageBufferRange). This means that `VK_WHOLE_SIZE` is not typically useful in the common case where uniform buffer descriptors are suballocated from a buffer that is much larger than `maxUniformBufferRange`.

q.v : https://www.khronos.org/registry/vulkan/specs/1.0/html/vkspec.html#VkDescriptorBufferInfo

The limitation of VkDeviceMemory size depends on the memory manager's logic.
But for this test, I will use the maximum size.

3rd Test – Dynamic Offsets

- The overall logic is as follows.

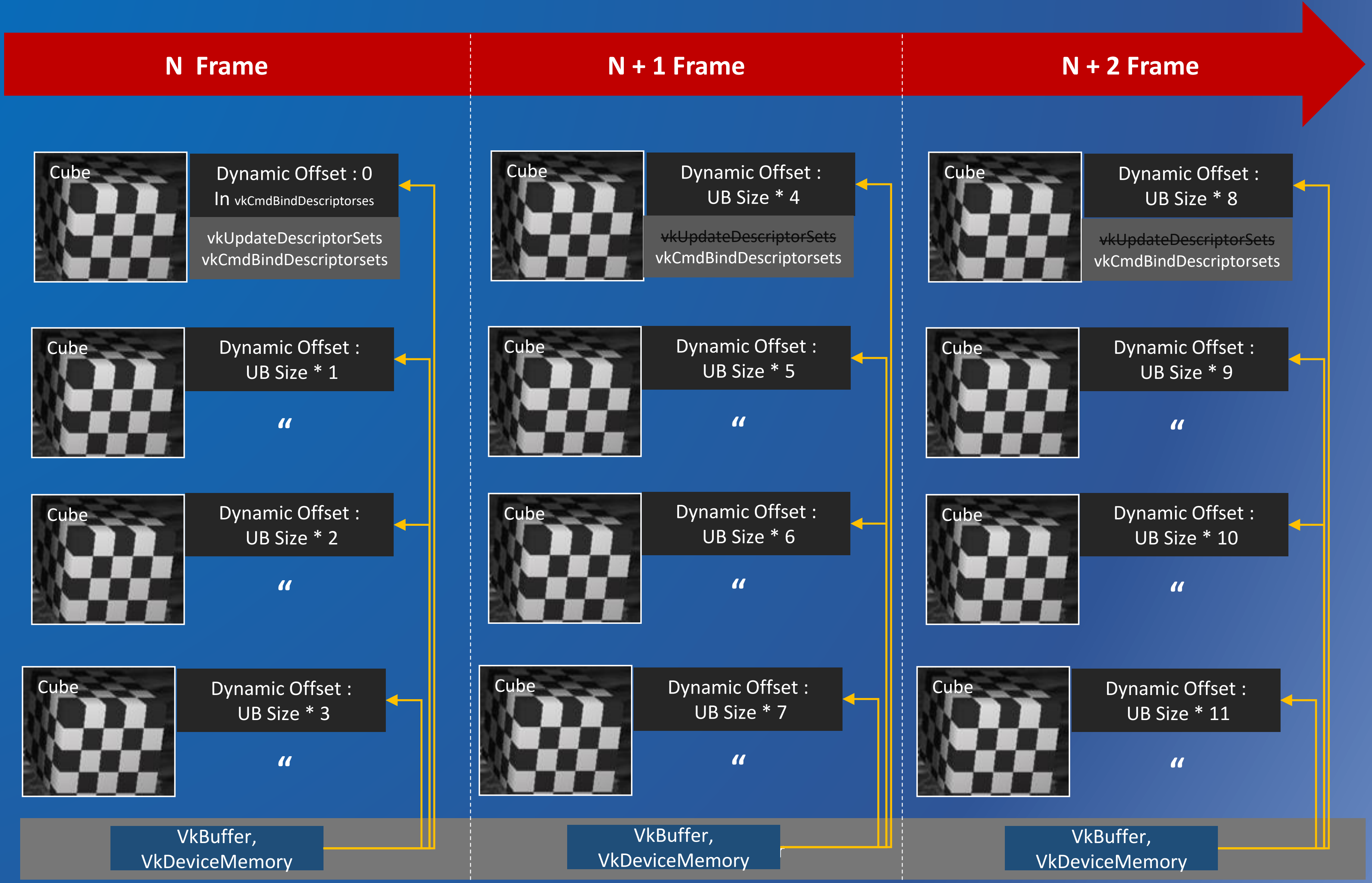


Logic is similar to the 2nd Test, but it helps to skip vkUpdateDescriptorSets API after initialization.

SAMSUNG

Vulkan.

3rd Test – Dynamic Offsets



N Frame

Cube | Dynamic Offset : 0
In vkCmdBindDescriptorses | vkUpdateDescriptorSets
vkCmdBindDescriptorsets

Cube | Dynamic Offset :
UB Size * 1 | “

Cube | Dynamic Offset :
UB Size * 2 | “

Cube | Dynamic Offset :
UB Size * 3 | “

VkBuffer,
VkDeviceMemory

N + 1 Frame

Cube | Dynamic Offset :
UB Size * 4 | ~~vkUpdateDescriptorSets~~
vkCmdBindDescriptorsets

Cube | Dynamic Offset :
UB Size * 5 | “

Cube | Dynamic Offset :
UB Size * 6 | “

Cube | Dynamic Offset :
UB Size * 7 | “

VkBuffer,
VkDeviceMemory

N + 2 Frame

Cube | Dynamic Offset :
UB Size * 8 | ~~vkUpdateDescriptorSets~~
vkCmdBindDescriptorsets

Cube | Dynamic Offset :
UB Size * 9 | “

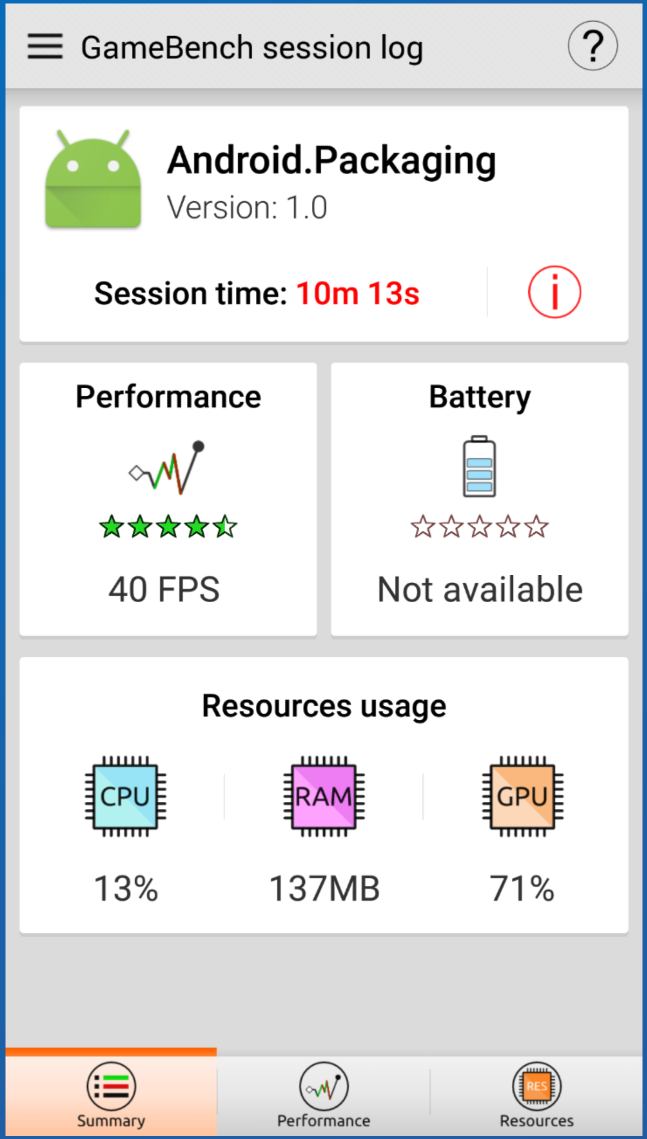
Cube | Dynamic Offset :
UB Size * 10 | “

Cube | Dynamic Offset :
UB Size * 11 | “

VkBuffer,
VkDeviceMemory

※ Swapchain count related logic should be considered.

3rd Test – Dynamic Offsets



GameBench session log

Android.Packaging

Version: 1.0

Session time: 10m 13s

Performance

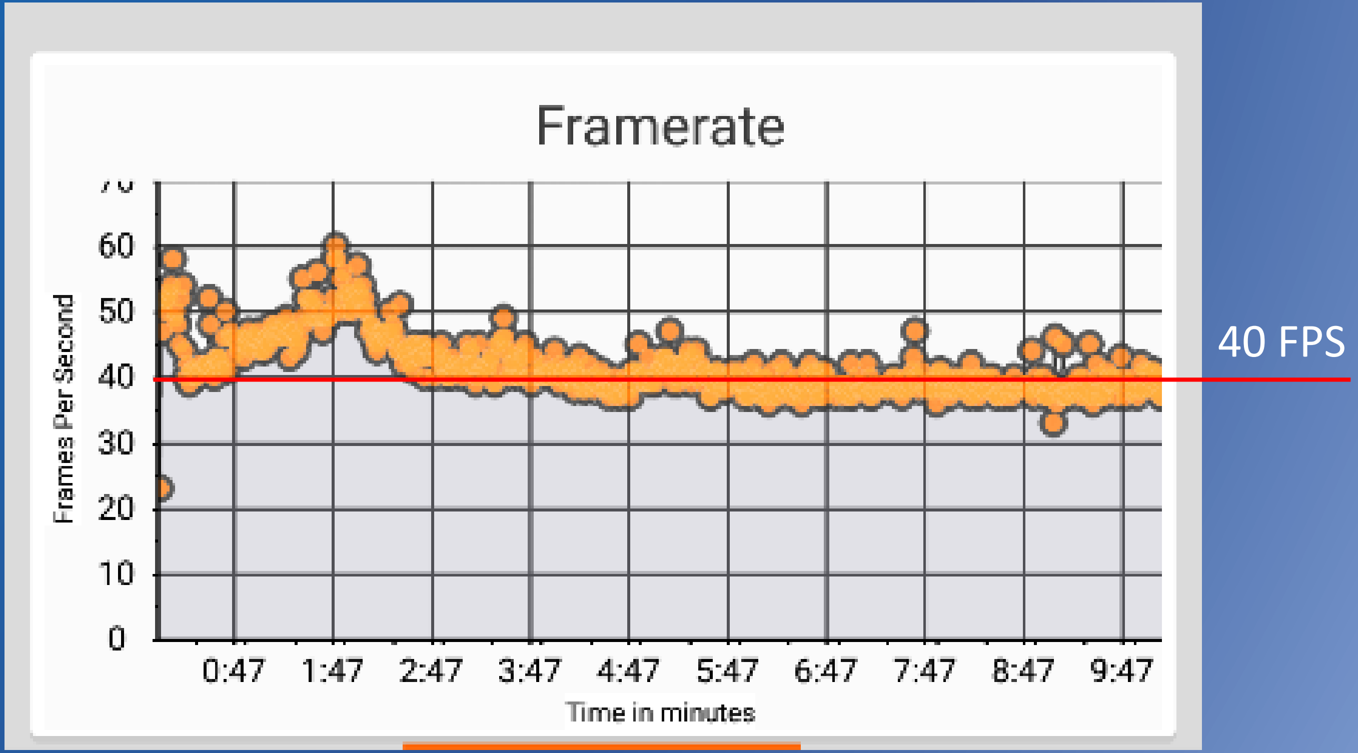
40 FPS

Battery

Not available

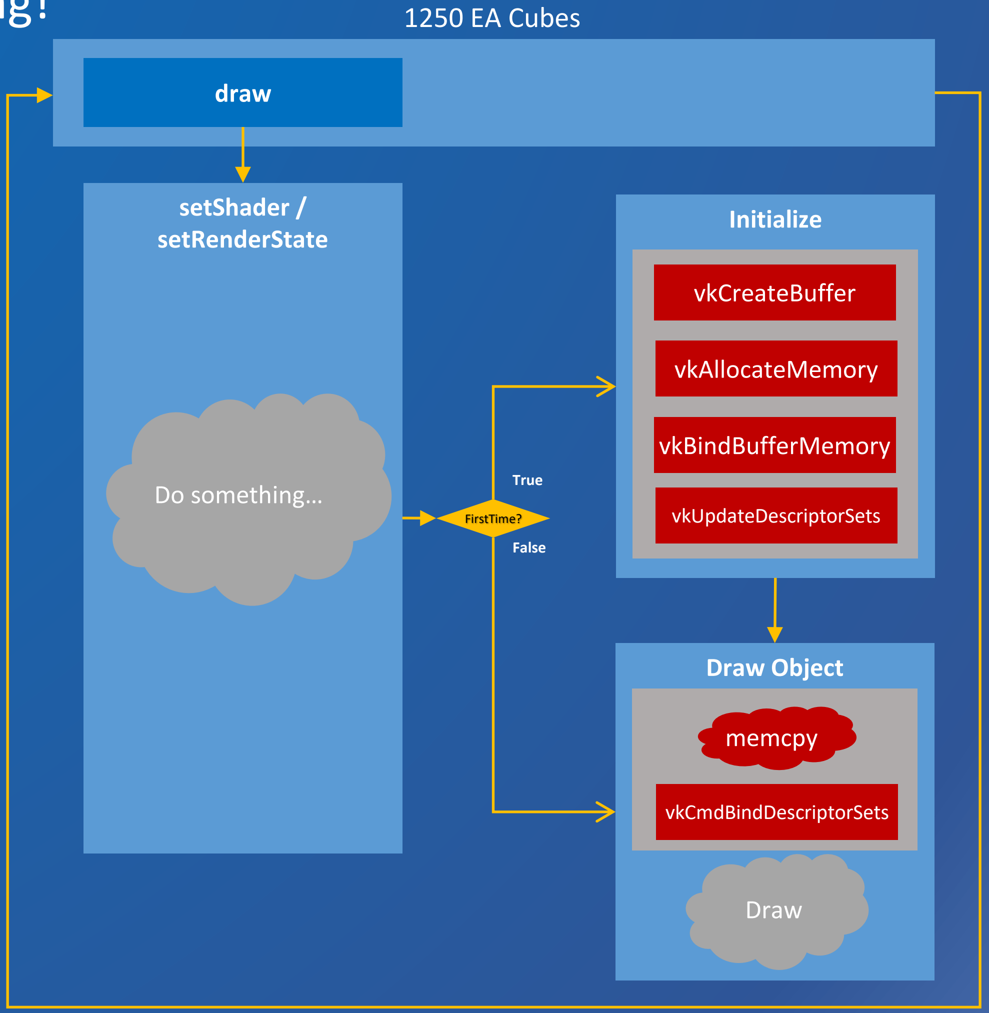
Resources usage

13% 137MB 71%



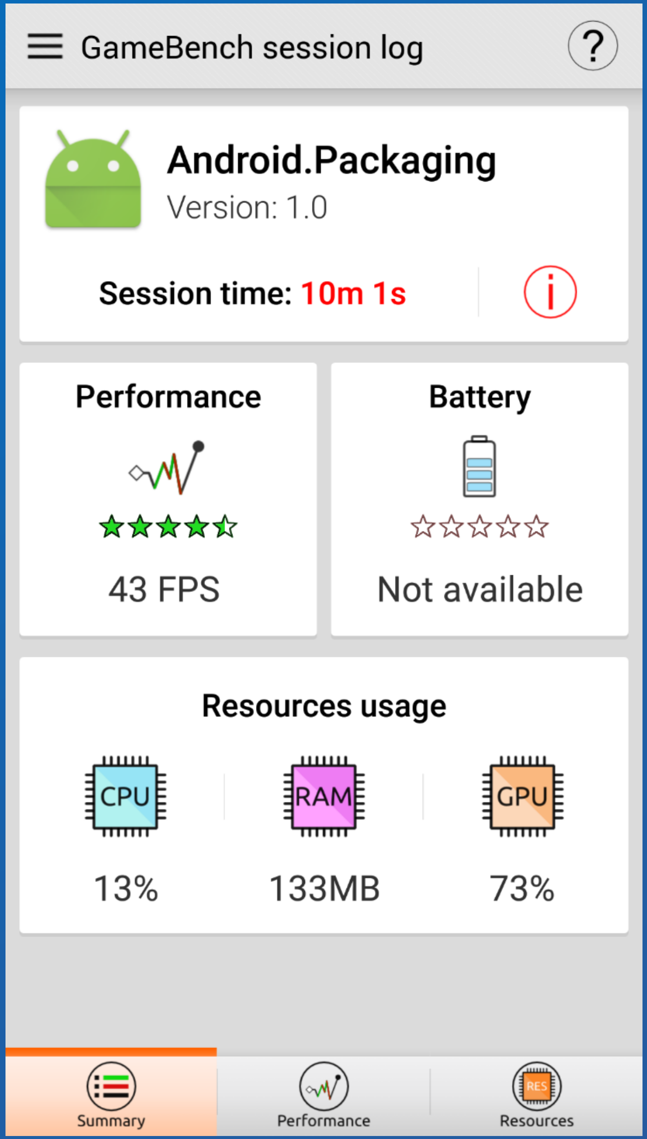
4th Test - Ideal condition

- If everything is in a predictable situation.
- It is similar to the concept demo. In fact, it’s difficult to apply to real engines.
- But just for testing!



SAMSUNG

4th Test - Ideal condition



GameBench session log

Android.Packaging

Version: 1.0

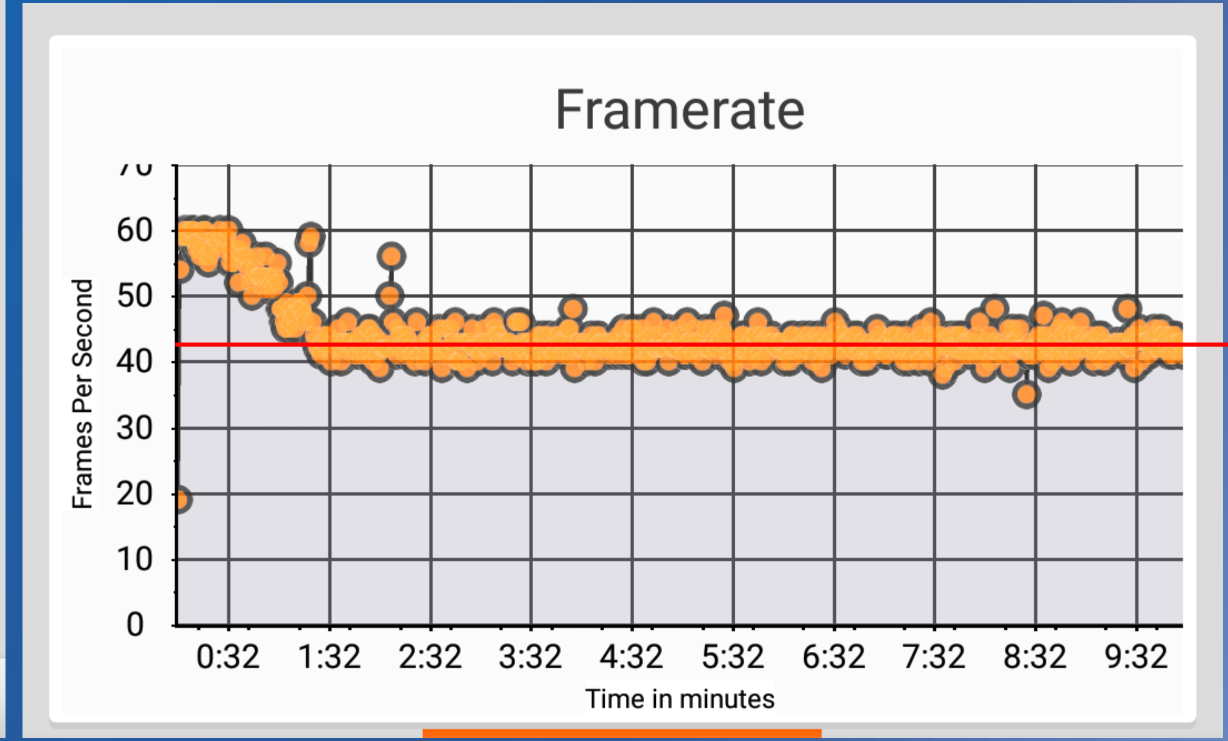
Session time: 10m 1s

Performance	**Battery**
★★★★☆	☆☆☆☆☆
43 FPS	Not available

Resources usage

CPU	RAM	GPU
13%	133MB	73%

Summary · Performance · Resources



43 FPS

5th Test – Memory property flags on Mobile

- Many people curious about impact of different memory flags on performance on mobile.
- This test is based on 3rd test.

VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT

VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_CACHED_BIT

VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_CACHED_BIT |VK_MEMORY_PROPERTY_HOST_COHERENT_BIT

VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT

VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT | VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |VK_MEMORY_PROPERTY_HOST_COHERENT_BIT

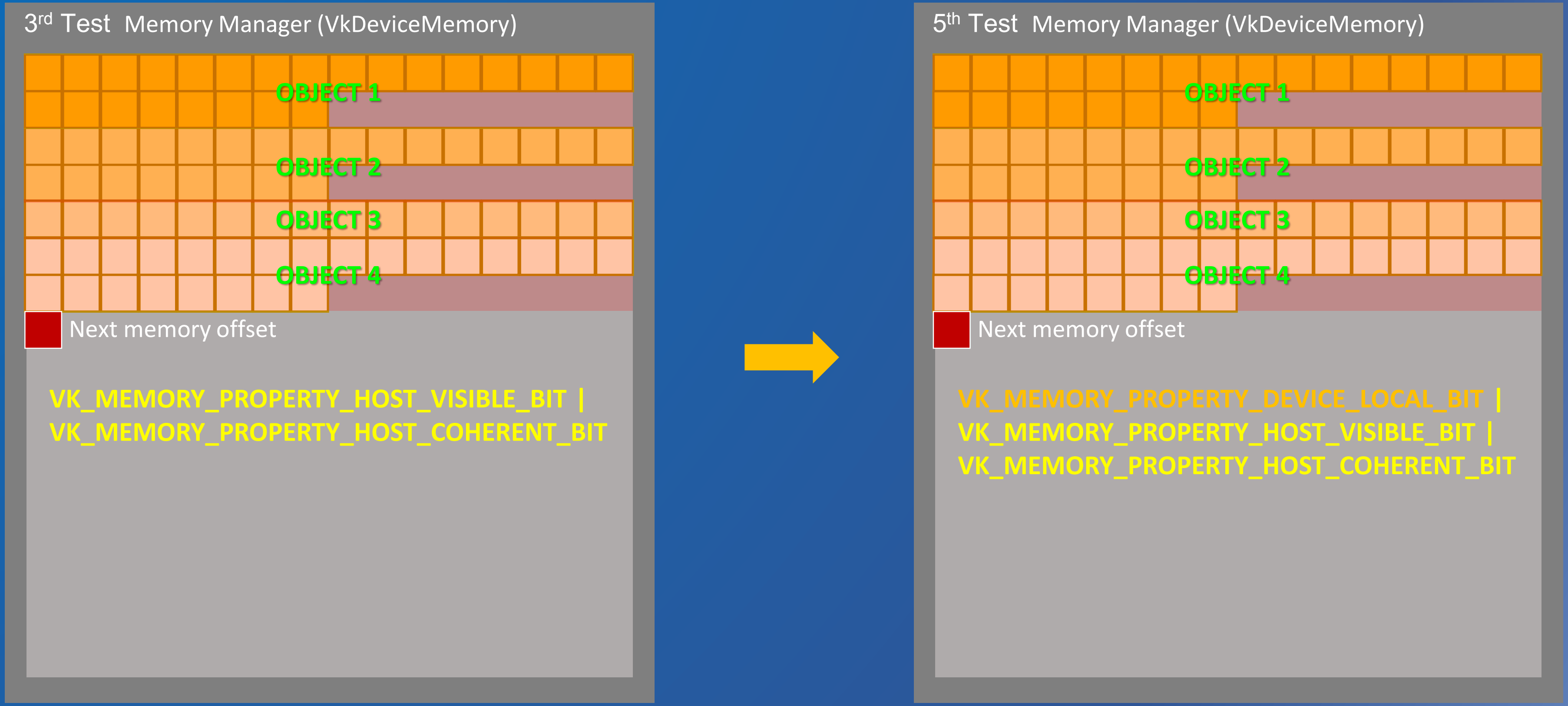
VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT | VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |VK_MEMORY_PROPERTY_HOST_CACHED_BIT

VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT | VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |VK_MEMORY_PROPERTY_HOST_CACHED_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT

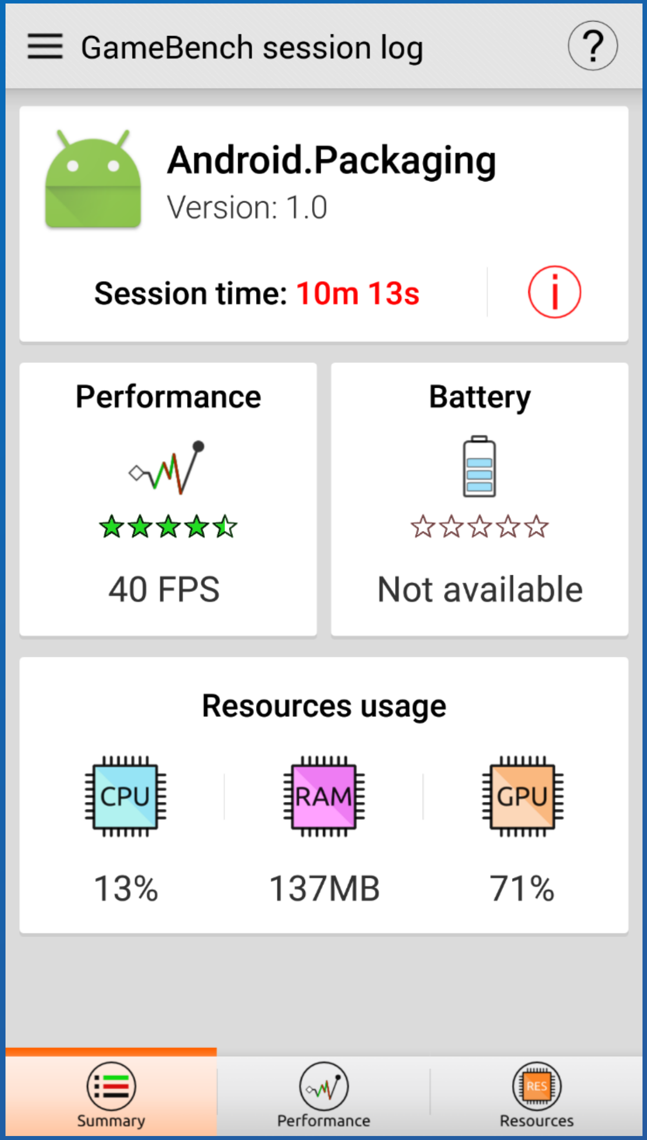
VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT | VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT

5th Test – Memory property flags on Mobile

- All logics are the same except memory flag. VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT is added.


3rd Test Memory Manager (VkDeviceMemory)
OBJECT 1
OBJECT 2
OBJECT 3
OBJECT 4
Next memory offset
VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
VK_MEMORY_PROPERTY_HOST_COHERENT_BIT
5th Test Memory Manager (VkDeviceMemory)
OBJECT 1
OBJECT 2
OBJECT 3
OBJECT 4
Next memory offset
VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT |
VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
VK_MEMORY_PROPERTY_HOST_COHERENT_BIT

5th Test – Memory property flags on Mobile



GameBench session log

Android.Packaging

Version: 1.0

Session time: 10m 13s

Performance

★★★★☆

40 FPS

Battery

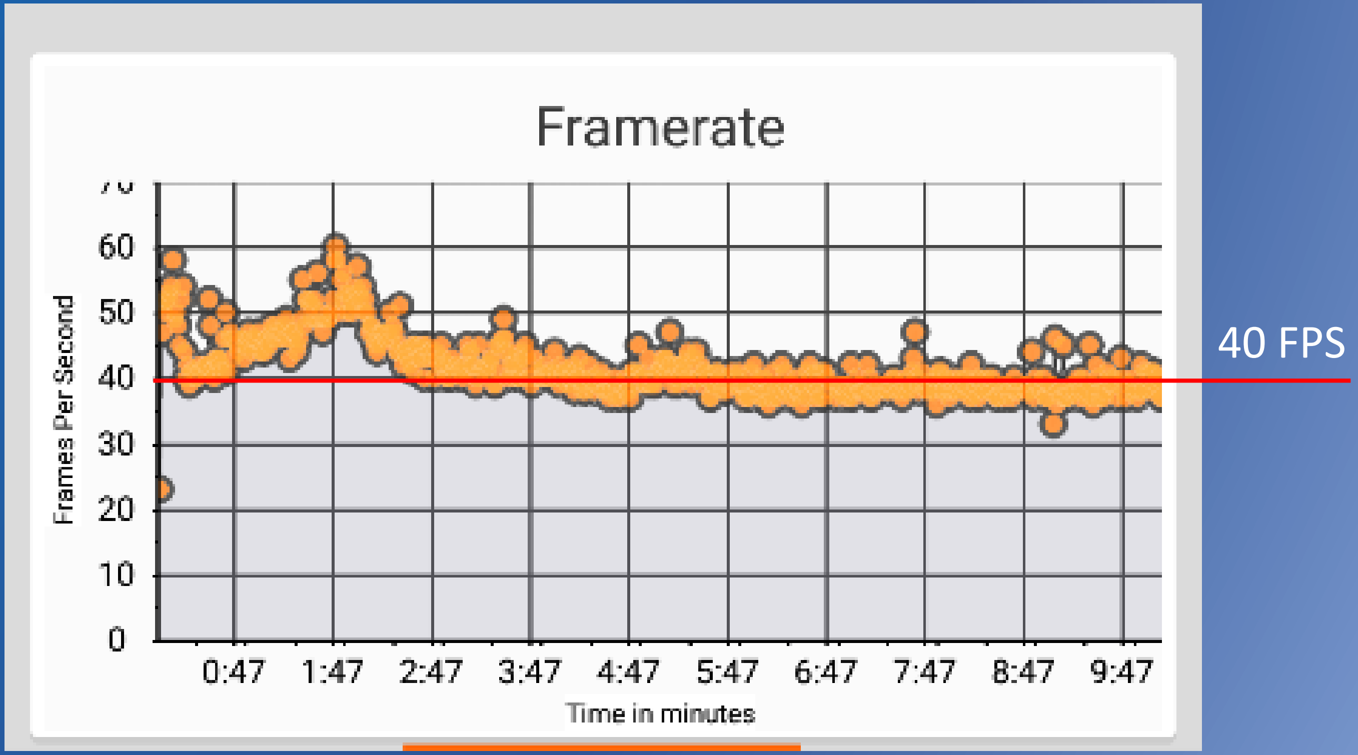
☆☆☆☆☆

Not available

Resources usage

CPU	RAM	GPU
13%	137MB	71%

Summary · Performance · Resources

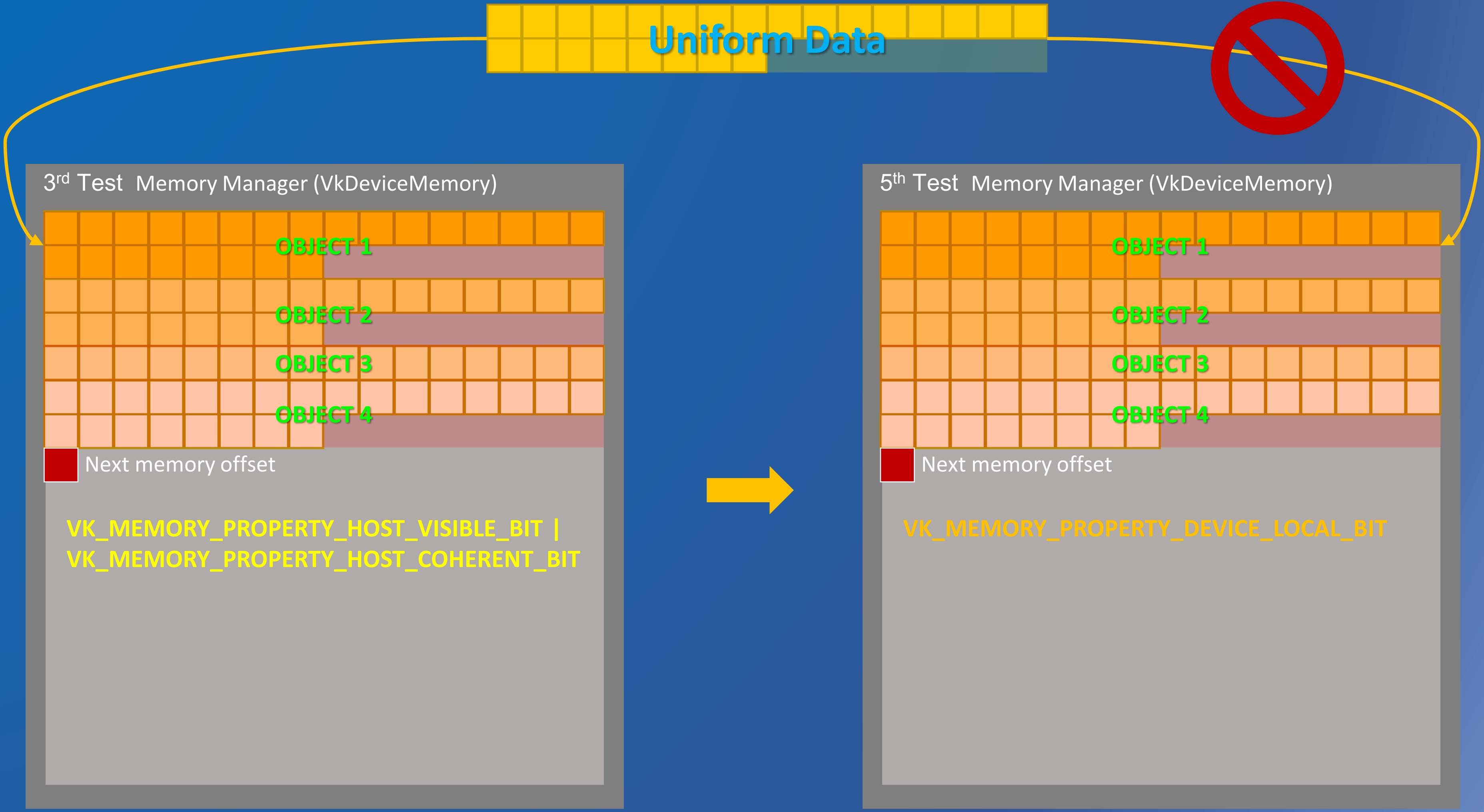


5th Test – Memory property flags on Mobile

Not recommended

- VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT

Without VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT
We cannot directly copy data to memory.

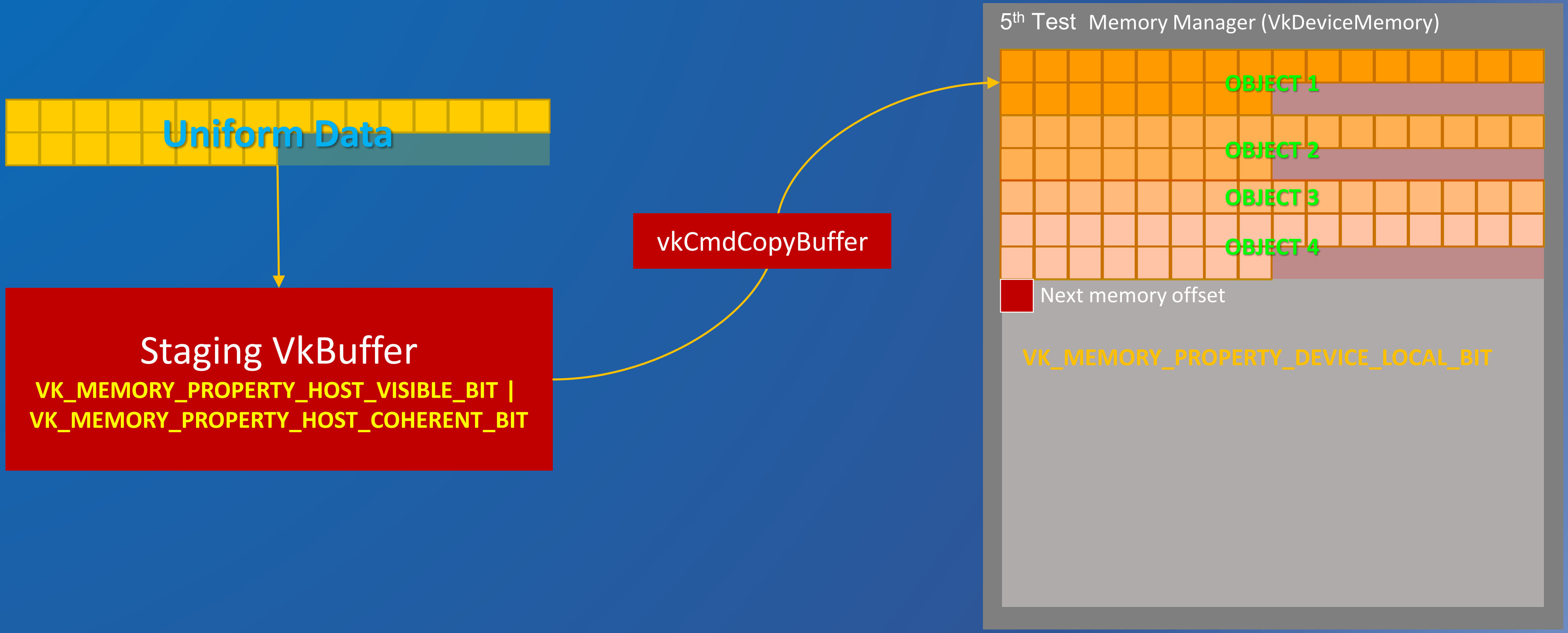

Uniform Data
3rd Test Memory Manager (VkDeviceMemory)
OBJECT 1
OBJECT 2
OBJECT 3
OBJECT 4
Next memory offset
VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
VK_MEMORY_PROPERTY_HOST_COHERENT_BIT
5th Test Memory Manager (VkDeviceMemory)
OBJECT 1
OBJECT 2
OBJECT 3
OBJECT 4
Next memory offset
VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT

SAMSUNG

Vulkan.

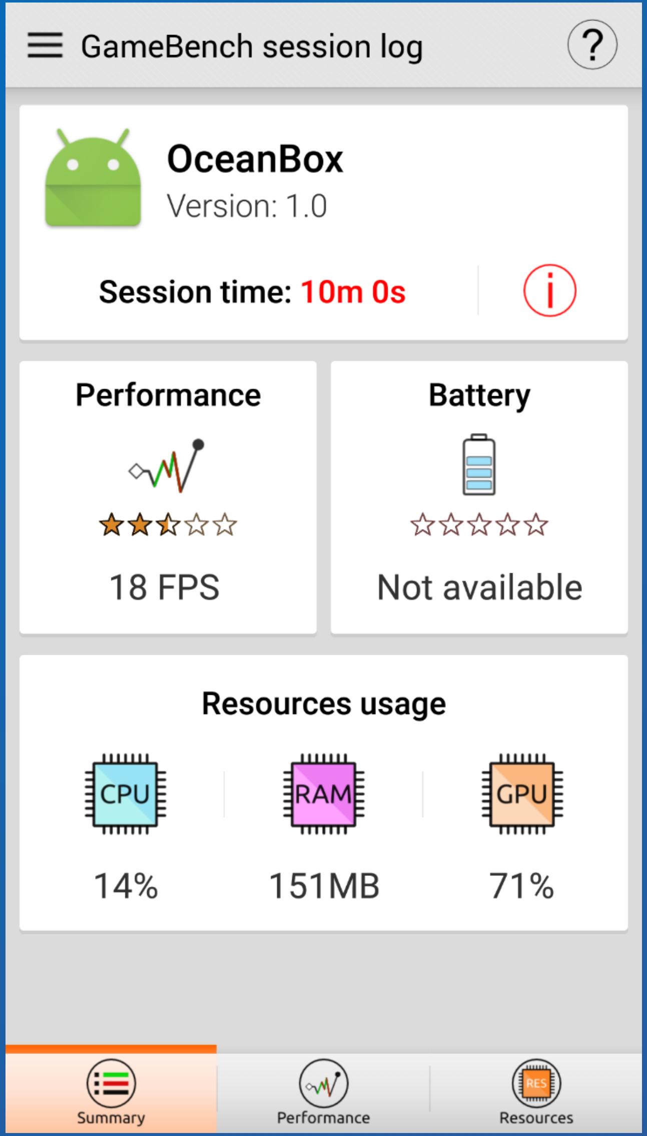
5th Test – Memory property flags on Mobile

- VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT



Uniform Data
Staging VkBuffer
VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
VK_MEMORY_PROPERTY_HOST_COHERENT_BIT
vkCmdCopyBuffer
5th Test Memory Manager (VkDeviceMemory)
OBJECT 1
OBJECT 2
OBJECT 3
OBJECT 4
Next memory offset
VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT

5th Test – Memory property flags on Mobile



GameBench session log

OceanBox

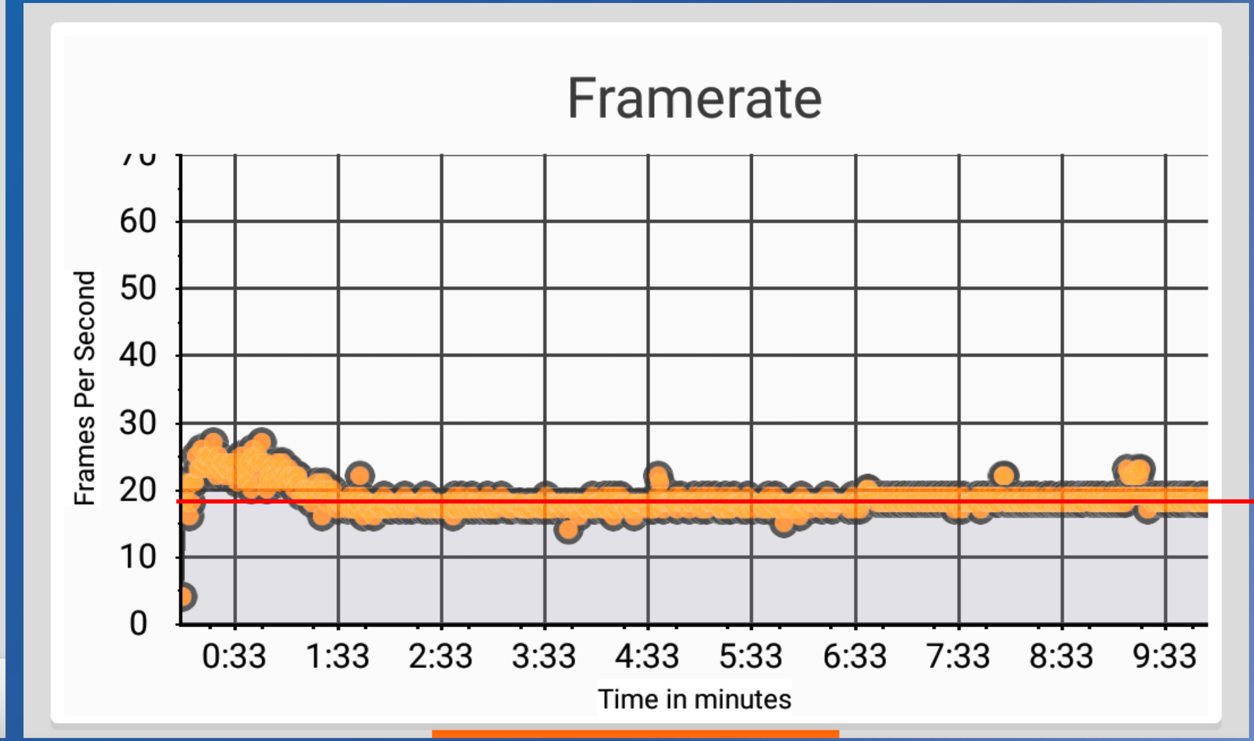
Version: 1.0

Session time: 10m 0s

Performance	Battery
★★⯪☆☆	☆☆☆☆☆
18 FPS	Not available

Resources usage

CPU	RAM	GPU
14%	151MB	71%



18 FPS

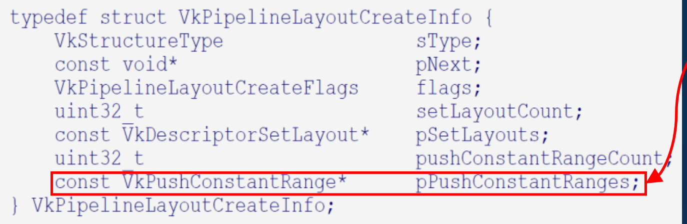
6th Test - PushConstants

✓ "Push constants" are helpful to improve performance. (the effect is GPU dependent.)

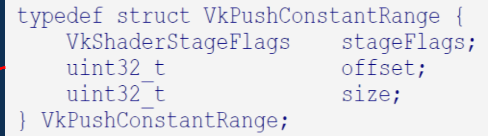
✓ They are very easy to use.

✓ However, VkPhysicalDeviceLimits::maxPushConstantsSize should be checked.

VkPipelineLayout

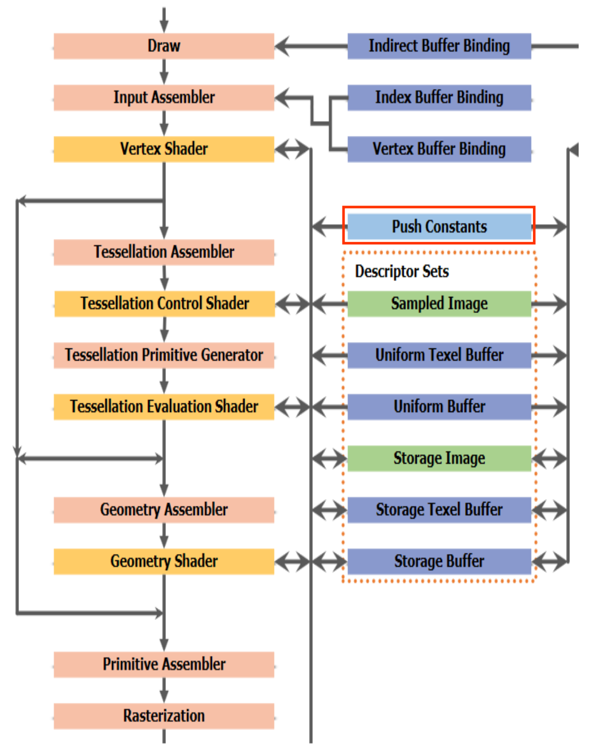


```
typedef struct VkPipelineLayoutCreateInfo {
    VkStructureType                 sType;
    const void*                     pNext;
    VkPipelineLayoutCreateFlags     flags;
    uint32_t                        setLayoutCount;
    const VkDescriptorSetLayout*    pSetLayouts;
    uint32_t                        pushConstantRangeCount;
    const VkPushConstantRange*      pPushConstantRanges;
} VkPipelineLayoutCreateInfo;
```



```
typedef struct VkPushConstantRange {
    VkShaderStageFlags    stageFlags;
    uint32_t              offset;
    uint32_t              size;
} VkPushConstantRange;
```

```
// VertexShader
 …
layout(push_constant) uniform buf1{
  mat4 _unif00;
} pc; // you cannot skip instancing, if uniform is push_constant.
void main()
{
  gl_position = pc._unif00 * _in_vertex;
}
```



```
vkCmdPushConstants(commandBuffer, layout, stageFlags, offset, MVPMatrix.size(), MVPMatrix.data());
```

6th Test - PushConstants

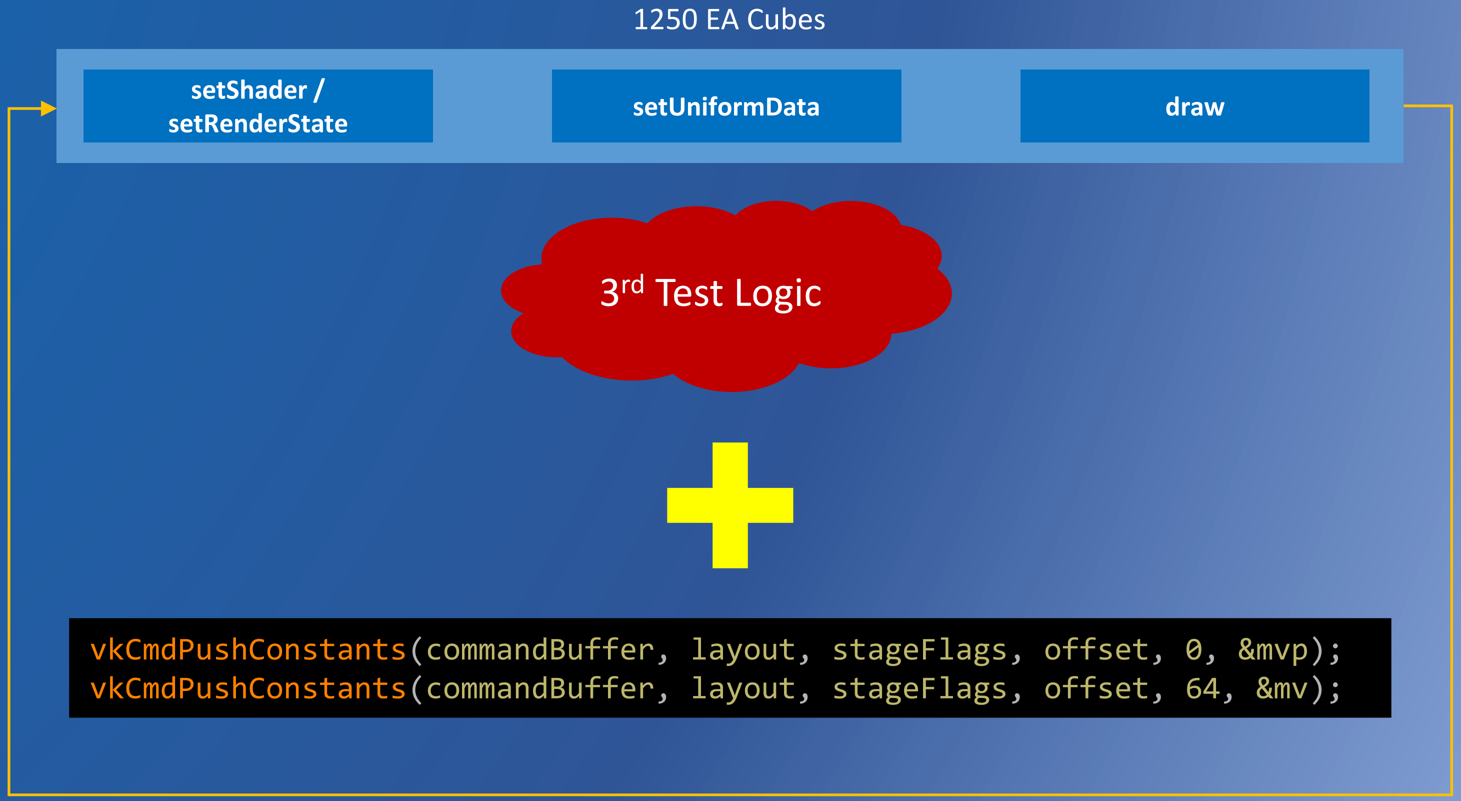
✓ By the way, if PushConstants data is changed in every draw call, is it helpful for performance?

Part of cube vertex shader

```
layout(set = 0, binding = 0, std140) uniform buf1 {
           mat4 mvp;
           mat4 mv;
           mat3 normalMatrix;
           vec3 lightPosition;
           float timeStep;
} ubuf1;
```

↓

```
layout(set = 0, binding = 0, std140) uniform buf1 {
           mat3 normalMatrix;
           vec3 lightPosition;
           float timeStep;
} ubuf1;

layout(push_constant) uniform buf2 {
           mat4 mvp;
           mat4 mv;
} pc;
```

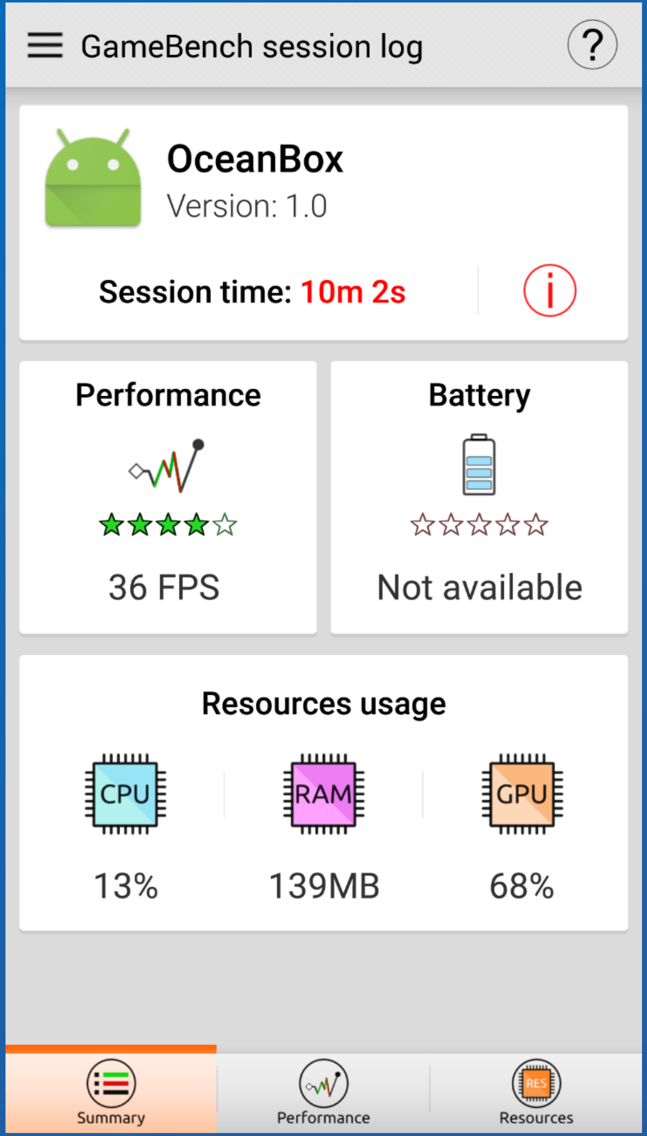


```
vkCmdPushConstants(commandBuffer, layout, stageFlags, offset, 0, &mvp);
vkCmdPushConstants(commandBuffer, layout, stageFlags, offset, 64, &mv);
```

SAMSUNG

Vulkan.

6th Test - PushConstants



GameBench session log

OceanBox

Version: 1.0

Session time: 10m 2s

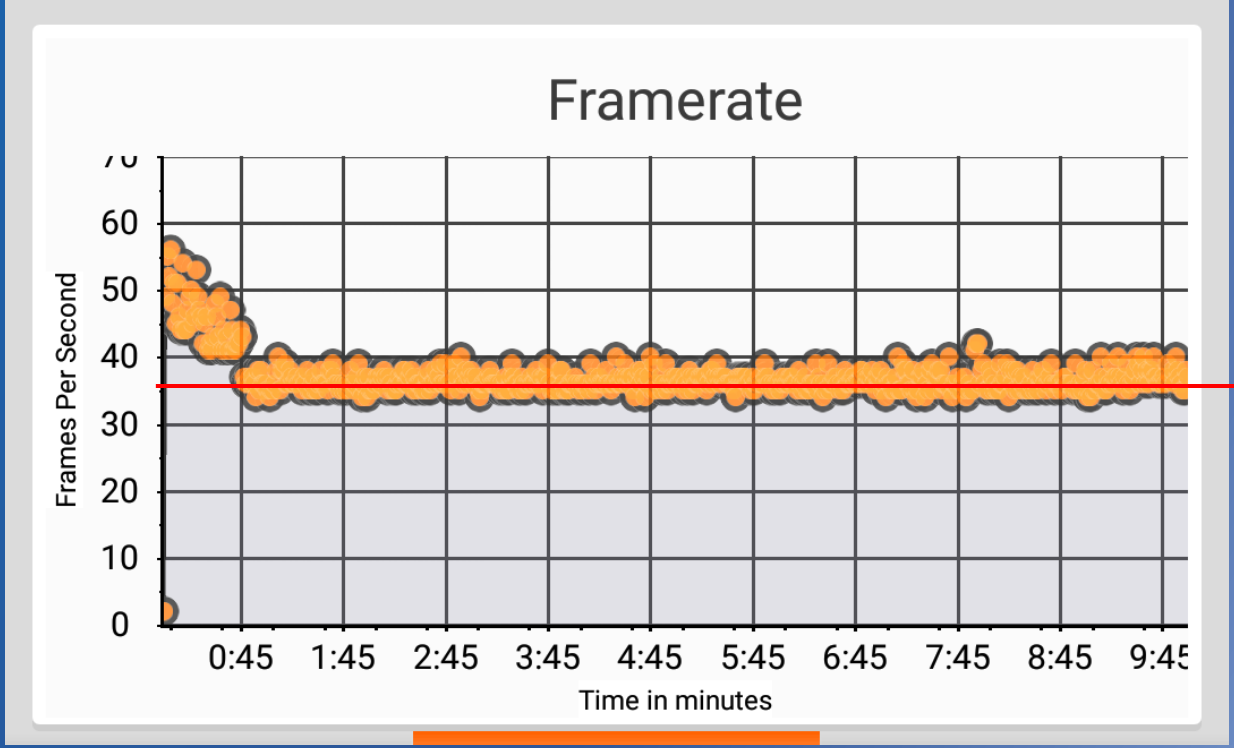
Performance	Battery
★★★★☆	☆☆☆☆☆
36 FPS	Not available

Resources usage

CPU	RAM	GPU
13%	139MB	68%

1250 * 2 * vkCmdPushConstants() = 2500 vkCmdPushConstants per frame

Misuse can be poisonous.

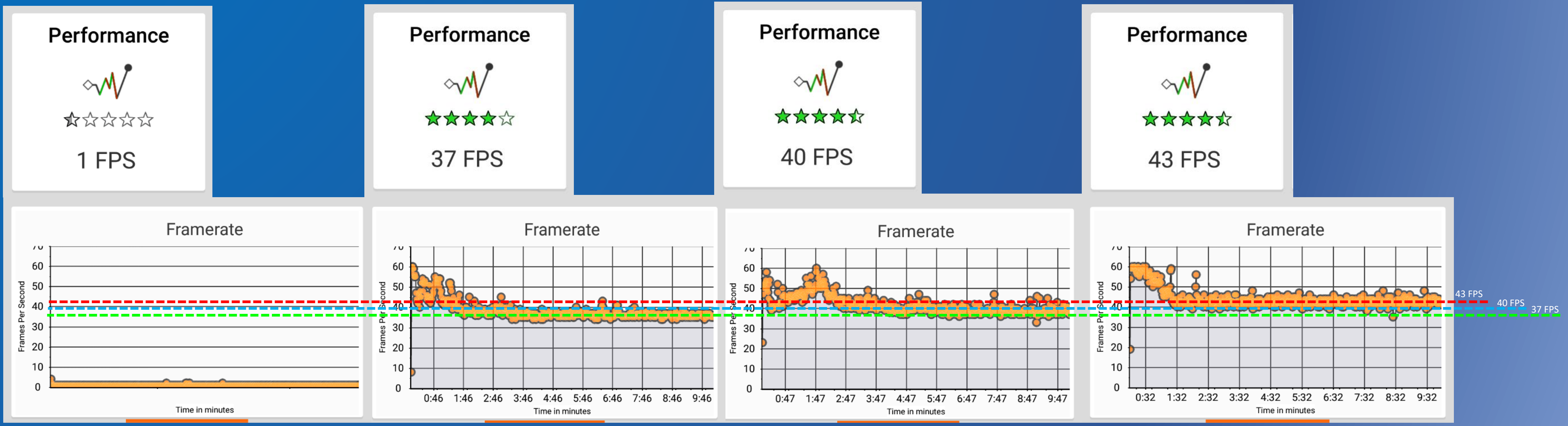


36 FPS

Summary - Uniform Buffer Test

Structural Experiments

1st Brute Force　　2nd Memory Manager　　3rd Dynamic Offsets　　4th Ideal condition



Remember : Structural selection depends on your renderer interface.
Please use these result for reference only.

Additional Experiments

5th Test – Memory property flags on Mobile : There is no significant difference in the test results.

6th Test – PushConstants : Misuse can be poisonous.

Other topics

Persistent PipelineCache

✓ Calling vkCreateGraphicsPipelines without VkPipelineCache will be very costly.

It is recommended to use it as a storage saved persistent cache.

Loading cost comparison (createGraphicPipeline 300 EA + @)

Without VkPipelineCache	With VkPipelineCache (Persistent)
13.260 seconds	4.187 seconds

onResume

```
std::vector<unsigned char*>& pipelineCacheData = getPipelineCacheFromSDcard();
VkPipelineCacheCreateInfo pipelineCacheCreateInfo = {};
pipelineCacheCreateInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_CACHE_CREATE_INFO;
pipelineCacheCreateInfo.initialDataSize = pipelineCacheData.size();
pipelineCacheCreateInfo.pInitialData = pipelineCacheData.data();
VkPipelineCache pipelineCache = VK_NULL_HANDLE;
vkCreatePipelineCache(device, &pipelineCacheCreateInfo, VK_NULL_HANDLE, &pipelineCache);
```

createGraphicPipeline

```
vkCreateGraphicsPipelines(device, pipelineCache, 1, &createInfo, VK_NULL_HANDLE, &pipline);
```

onPause

```
size_t pDataSize = 0;
vkGetPipelineCacheData(device, pipelineCache, &pDataSize, VK_NULL_HANDLE);
// if is valid
vkGetPipelineCacheData(device, pipelineCache, &pDataSize, pipelineCacheData.data());
savePipelineCacheToSDcard(pipelineCacheData);
```

Clear framebuffer cost

✓ There are 3 ways to clear framebuffer. (color, depth, stencil)

- *Renderpass Load Operation*
- *vkCmdClearAttachments*
- *vkCmdClearColorImage/vkCmdClearDepthStencilImage*

✓ It's important to use proper and clear approach to not waste additional clear cost

(e.g. clear all, color only, depth only)

- 1 clear color & 30 clear depth

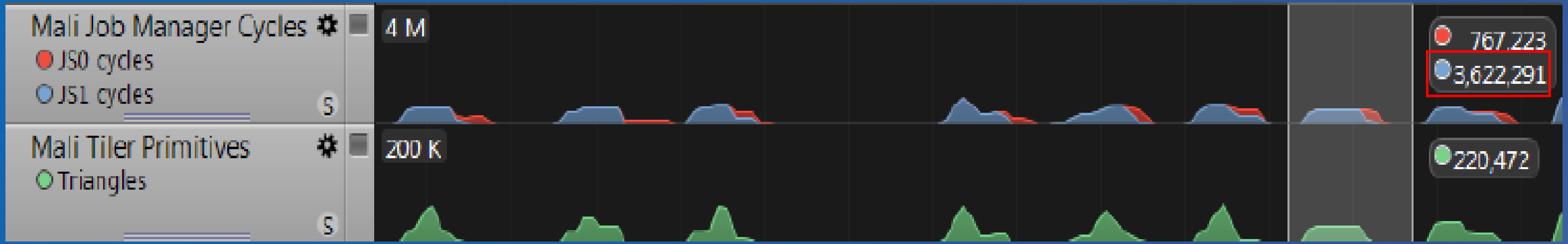
Renderpass begin/end using LoadOpClear	vkCmdClearAttachments
24 FPS	57 FPS

✓ It's not recommended to clear framebuffer by loading empty Renderpass begin()/end() without actual draw calls, etc.

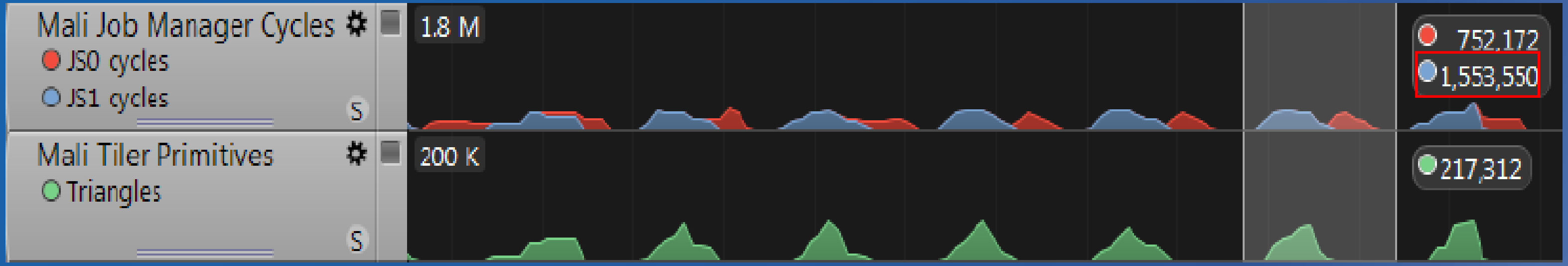
OpenGL ES vs. Vulkan: Geometry sorting

- Geometry sorting (vertex & index buffers)
  - Improves cache read/write efficiency
    - Can affect how work is submitted to the GPU
  - Some OpenGL ES drivers do this automatically

Without Geometry Sorting



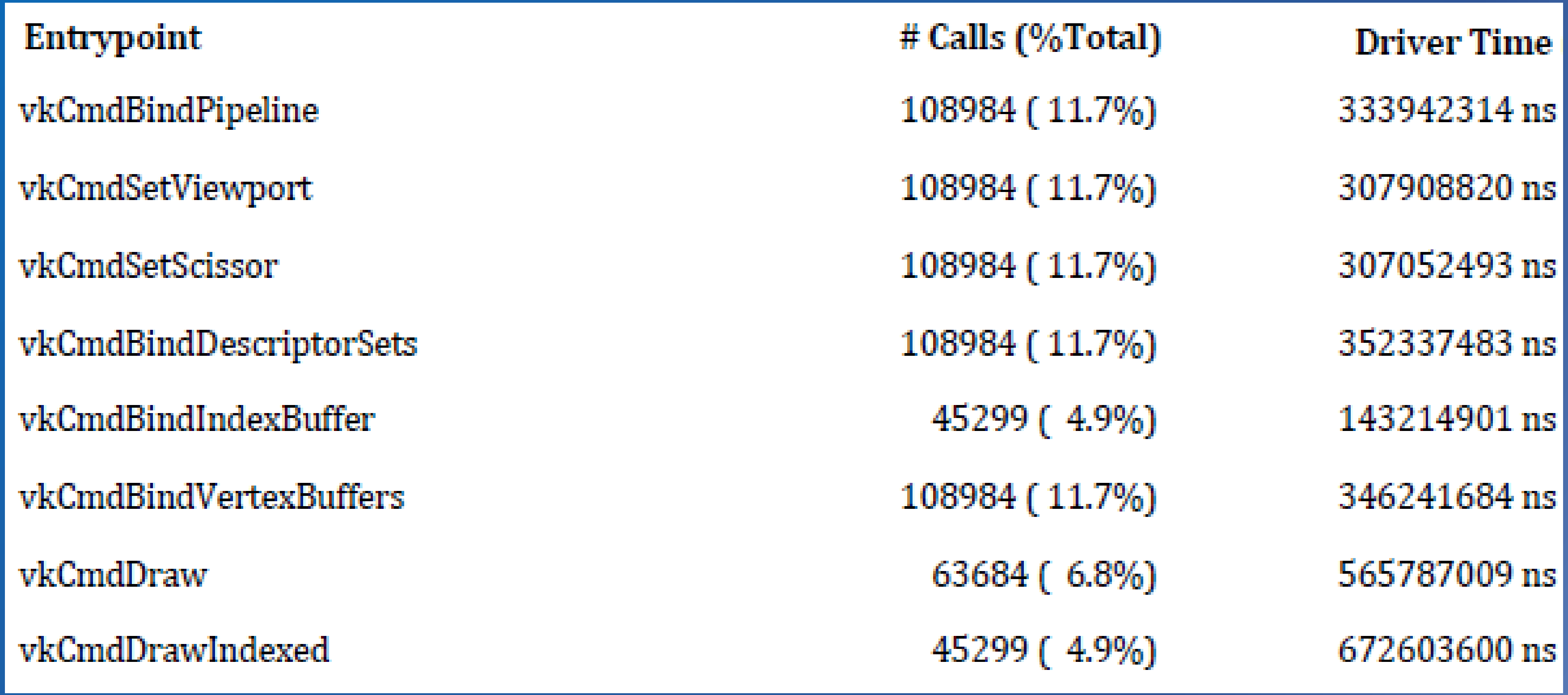
With Geometry Sorting



Reducing duplicated API calls

- ✓ It is important to call bind/set function once in a VkCommandBuffer to prevent duplication of vkCmdSetXXX and vkCmdBindXXX call with same value / parameter.

Worst case



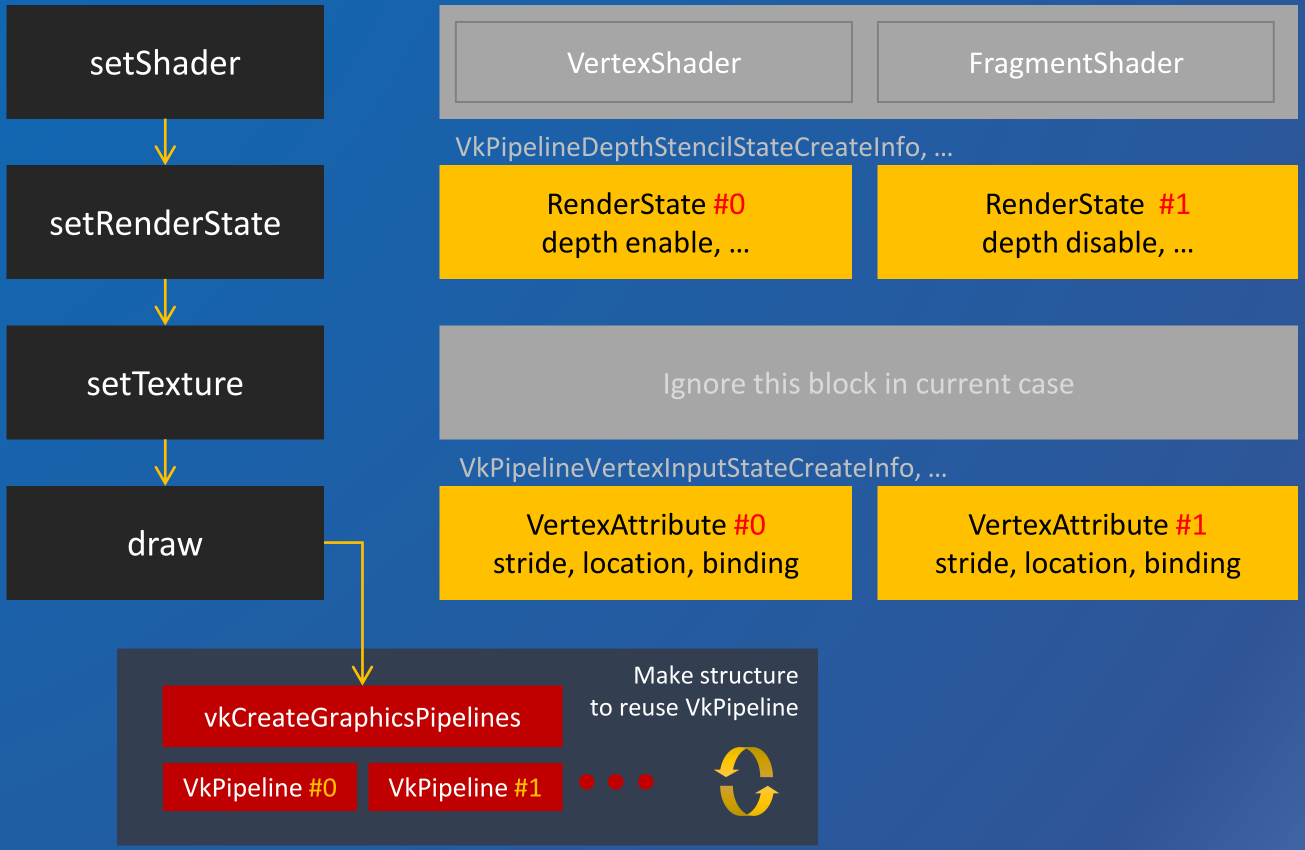
Entrypoint	# Calls (%Total)	Driver Time
vkCmdBindPipeline	108984 (11.7%)	333942314 ns
vkCmdSetViewport	108984 (11.7%)	307908820 ns
vkCmdSetScissor	108984 (11.7%)	307052493 ns
vkCmdBindDescriptorSets	108984 (11.7%)	352337483 ns
vkCmdBindIndexBuffer	45299 (4.9%)	143214901 ns
vkCmdBindVertexBuffers	108984 (11.7%)	346241684 ns
vkCmdDraw	63684 (6.8%)	565787009 ns
vkCmdDrawIndexed	45299 (4.9%)	672603600 ns

※ In our test case, 500 Calls vkCmdSetViewPort and vkCmdSetScissor take 1.412 ms.

Managing VkPipeline

Worst case, Given RenderState & Attributes can be changed every single draw call.

Therefore, having efficiently designed pipeline management structure will be essential for your performance optimization.

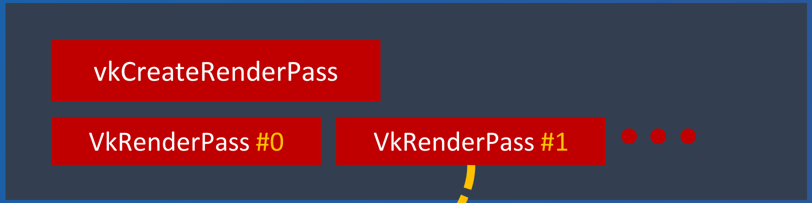


Managing VkRenderpass, VkFramebuffer

VkRenderpass

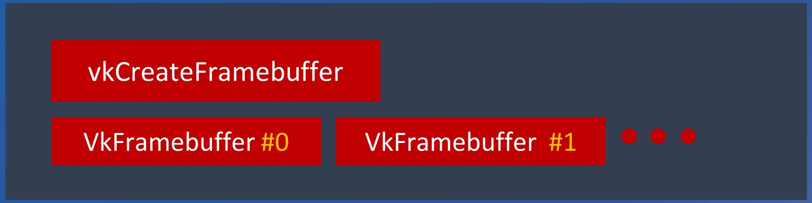
```
VkRenderPassCreateInfo {
  …
  uint32_t                            attachmentCount;
  const VkAttachmentDescription*      pAttachments;
  …
}

VkAttachmentDescription {
  …
   VkAttachmentLoadOp          loadOp;       VK_ATTACHMENT_LOAD_OP_LOAD
   VkAttachmentStoreOp         storeOp;  →   VK_ATTACHMENT_LOAD_OP_CLEAR
                                             VK_ATTACHMENT_LOAD_OP_DONT_CARE
  …
} VkAttachmentDescription;
```

Reusing VkRenderpass & VkFramebuffer are also essential.



VkFreambuffer

```
VkFramebufferCreateInfo {
  …
  VkRenderPass            renderPass;
  …
}
```



Wrap-Up

- Vulkan gives CPU off-load, predictable behavior by explicit control and various ways to optimize games.
- No more driver magic, so you need to manage things by yourself.

Samsung will keep go on supporting game developers and players!

If you have any questions, offers or suggestions, please contact

gamedev@samsung.com or soft.park@samsung.com

Thank you! ☺