Optimizing Unity Games

Aleksandr Dolbilov

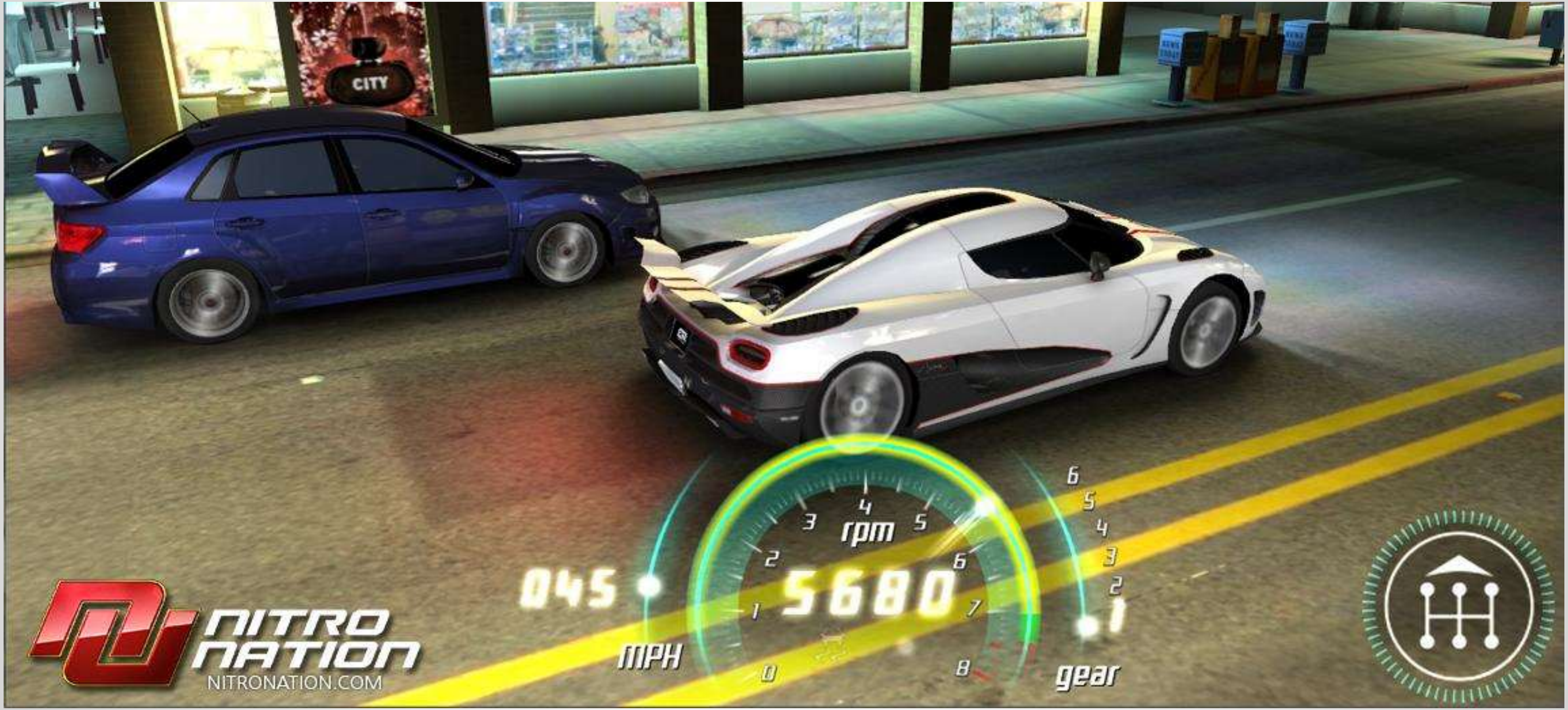
CTO, JoyCraft Games

About me

Aleksandr Dolbilov

- 10+ years in game industry
- Android, iOS, PS2, PS3, PSP, XBox, XBox 360...

Nitro Nation



Agenda

- Performance problems from real-world games
- Live profile, bottleneck detection and optimization
- Unity profiler and platform-specific tools in action
- Solutions and tips which help to improve performance of your games

- Grab your projects today and perform live performance analysis tomorrow!!

Today

- Few notes about performance in games
- CPU optimization
  - Static scene
  - Vertex constants arrays
  - Vertex constants instancing
  - Math optimization

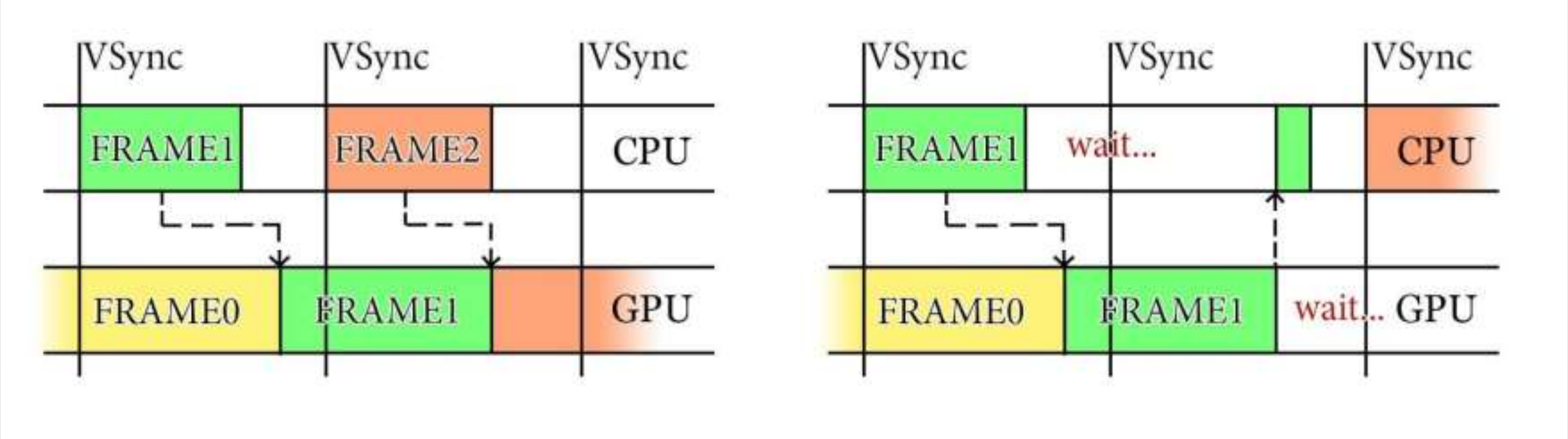
Tomorrow...

- GPU performance optimization
    - Vertex shaders optimization
    - Fragment shaders optimization

Performance in games

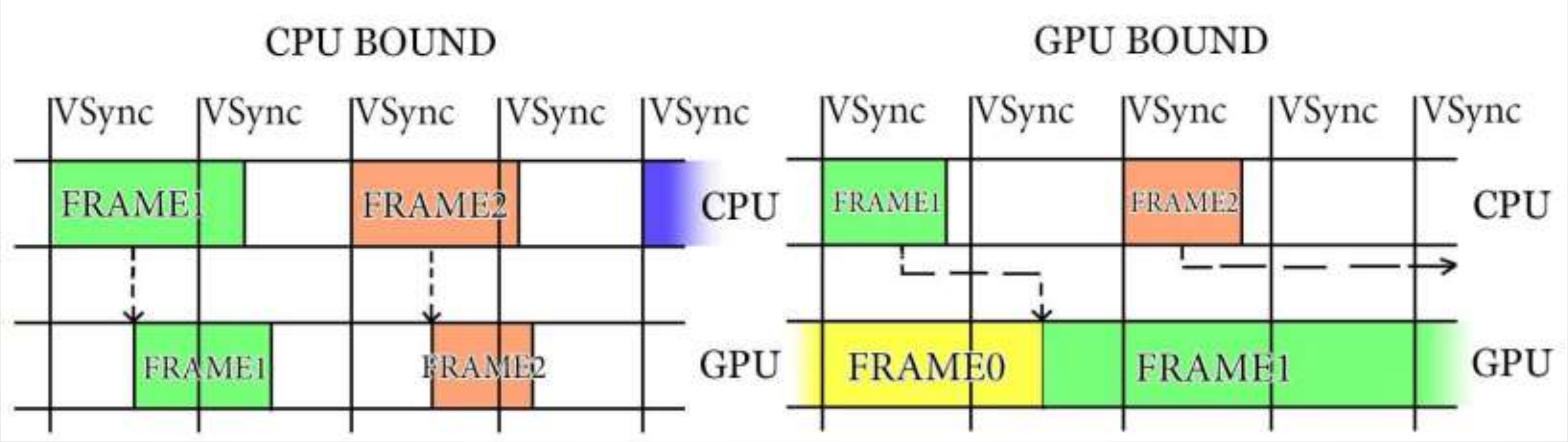
Frame time

- Frame time depends on **CPU** time and **GPU** time
- Both units work in parallel if you don’t touch **GPU** data from **CPU**



Locate your bottleneck

- Detect if game is **CPU** or **GPU** bound
    - ***Gfx.WaitForPresent*** in Unity Profiler is good indicator
    - Platform-specific tools provide units utilization info



Locate your bottleneck (CPU bound)

- Use **Unity Profiler** for further analysis
- Use ***Profiler.BeginSample*** and ***Profiler.EndSample*** to get detailed information on device
- **Deep profile** option is very helpful to find hidden pitfalls (implicit type casts, redundant constructors etc.)
- **Time.realtimeSinceStartup** can help you to monitor performance for release builds

Locate your bottleneck (GPU bound)

- Use platform-specific tools
  - *Adreno Profiler* (Qualcomm, Adreno)
  - *PerfHUD ES* (nVidia, Tegra)
  - *PVRTune, PVRTrace* (Imagination tec., PowerVR)
  - *Mali Graphics Debugger* (ARM, Mali)

- Show/hide method

Optimization tips

- Always measure optimization results
- Use milliseconds (not FPS!!)
- If it’s possible, create simple test for faster iterations

- Check if everything work properly ☺

CPU performance optimization

Render CPU time measurement

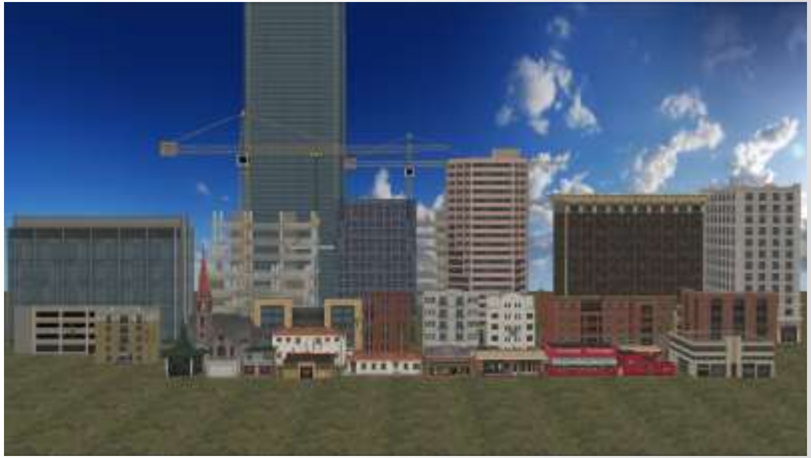
- Check ***Camera.Render*** time in Unity Profiler

- Or create custom ***MonoBehaviour***
    - ***OnPreCull*** is timestamp begin
    - ***OnPostRender*** is timestamp end
    - ***OnGUI*** for render result

Static scene

Initial scene

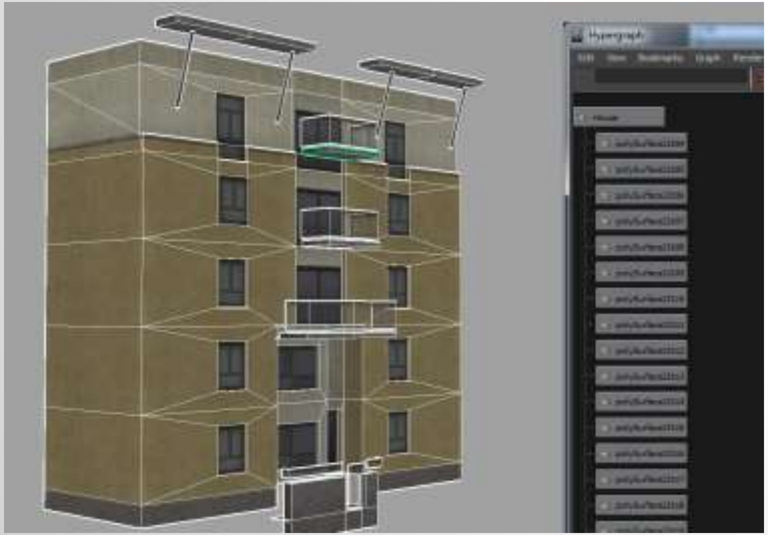
GameObject count:	1440
Material count:	32
Shader count:	2
Textures count:	32
Draw calls:	1434

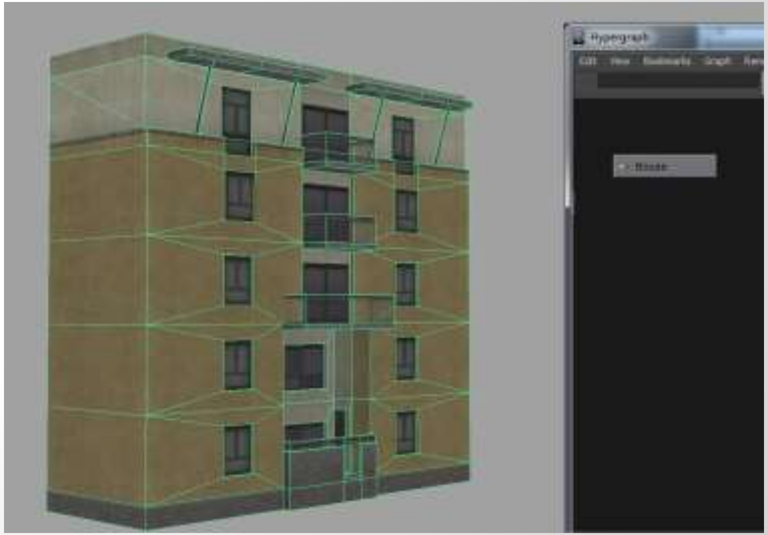


CPU time (HTC One S):**~75 ms**

Combine geometry: how to

- Usually made by artists according to ‘common sense’





Combine geometry: motivation

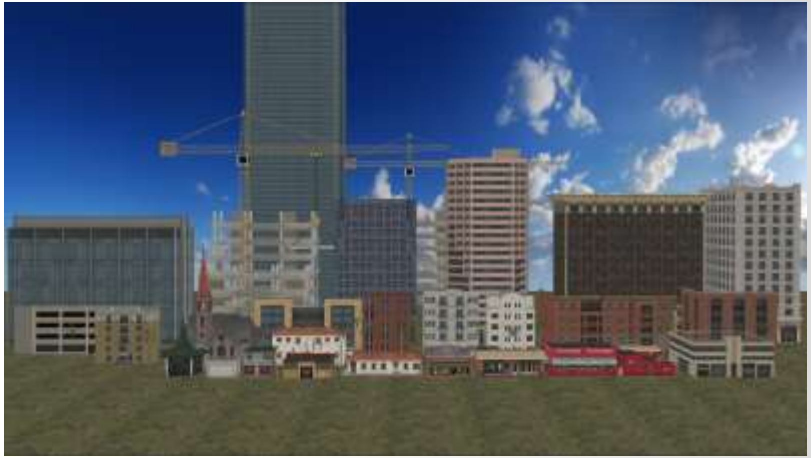
- Reduce engine overhead on **Transform** and **MeshRenderer** processing
  - Geometry transform validation
  - Visibility/Occlusion culling

  ….

- Reduce draw calls count

Combine geometry: result

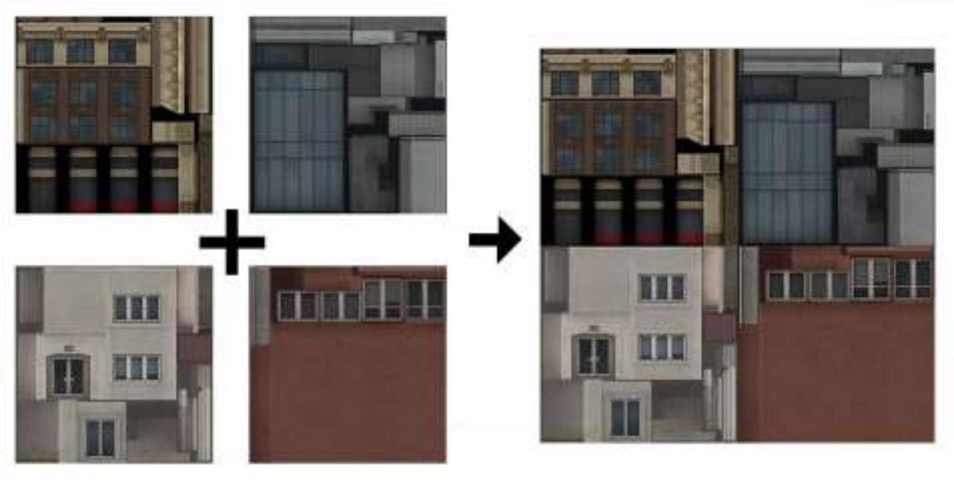
GameObject count:	**51**/1440
Material count:	32
Shader count:	2
Textures count:	32
Draw calls:	**90**/1434



CPU time (HTC One S):**~5.2 ms** /~75 ms

Bake textures to atlases: how to

- Can be made manually by artists but...
- … better to automate the process with script
(use ***Texture2D.PackTextures***)



Bake textures to atlases: how to

- Our approach is to create separate atlases for opaque and transparent objects
- Tiled textures are excluded from atlas
- Huge textures are excluded from atlas
- 2048x2048 texture resolution is usually enough to fit all static scene textures

Bake textures to atlases: motivation

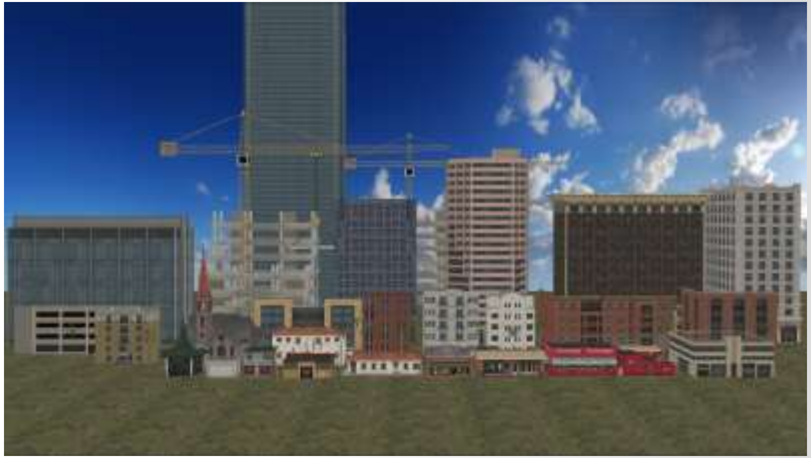
- Reduce **Material** and **Texture** count

- Reduce engine overhead on **Material** setup

- Reduce engine overhead on **Texture** switch

Bake textures to atlases: result

GameObject count:	51
Material count:	4/32
Shader count:	2
Textures count:	4/32
Draw calls:	90



CPU time (HTC One S):**~3.2 ms/**~5.2 ms

Move geometry to world space

- Pre-transform geometry to world space
- Use **Shader.SetGlobalVector** for ViewProjection matrix:
    - Reduce engine overhead on World*ViewProjection
    - Reduce engine overhead on shader constants setup, because Unity has setup cache for Vector properties

- We have ~25% performance boost on some scenes
- CPU time (HTC One S):**~2.8 ms**/~3.2 ms

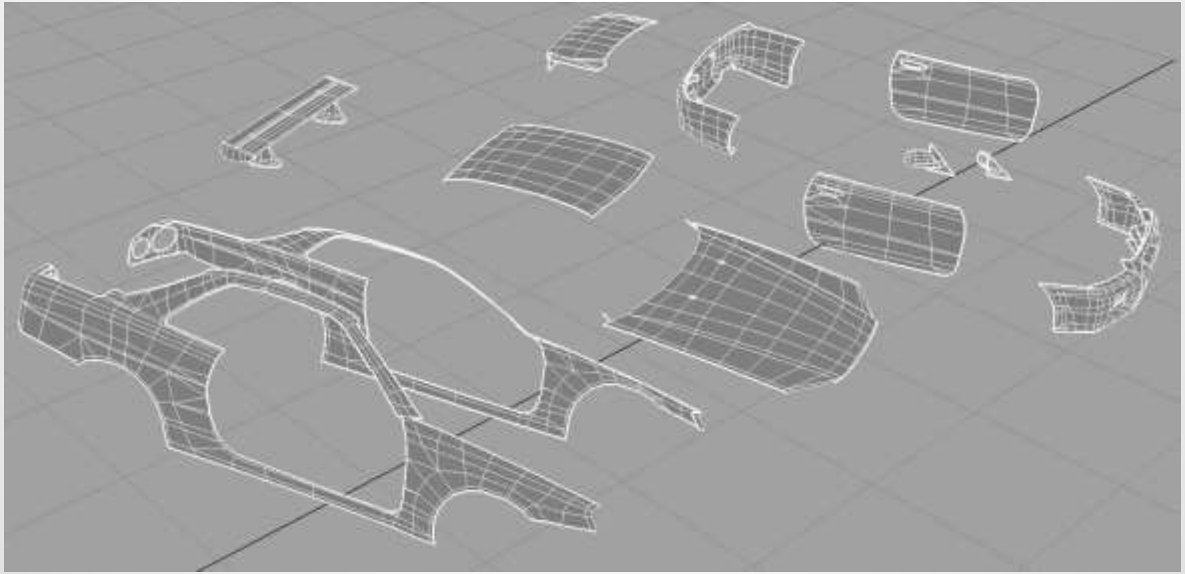
Static batching

- Reduce draw calls count but…
- … this is not for free
    - Performance depends on triangle count
    - Some devices doesn’t like dynamic geometry
    - We have severe performance degrade on **Mali** devices
    - Doesn’t get any speed improvement for **HTC One S**
    - Your mileage may vary

Vertex shader arrays

Motivation

- Reduce draw call count for objects with slightly different materials



General algorithm

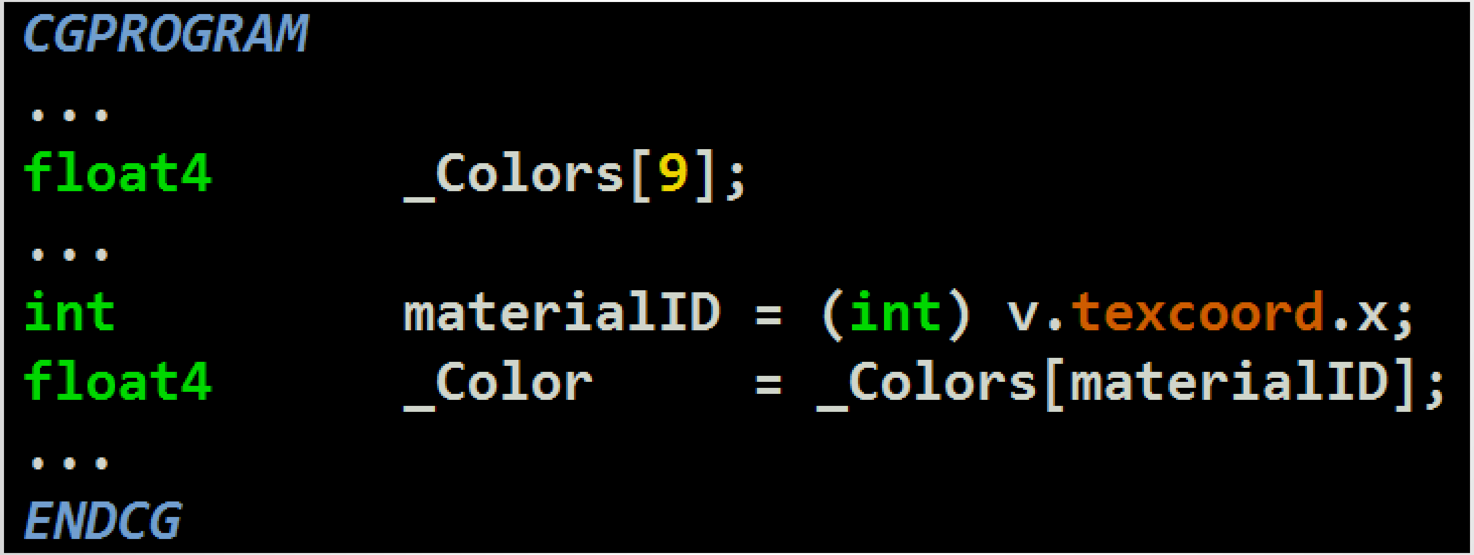
- Combine all meshes and store **MaterialID** for each vertex (index in array)
- Add array of uniforms to vertex shader
- Add value extraction code to vertex shader
- Map array elements as material properties

MaterialID pack/unpack

- Using texcoord (uv/uv2)
  - pack: ***mesh.uv[i].x = mesh.uv[i].x + MaterialID;***
  - unpack: ***int MaterialID = (int) v.texcoord.x;***

- Using color
  - pack: ***mesh.colors32[i].a = MaterialID;***
  - unpack: **int MaterialID = (int) (v.color.a*255.0f + 0.5f);**

Shader code

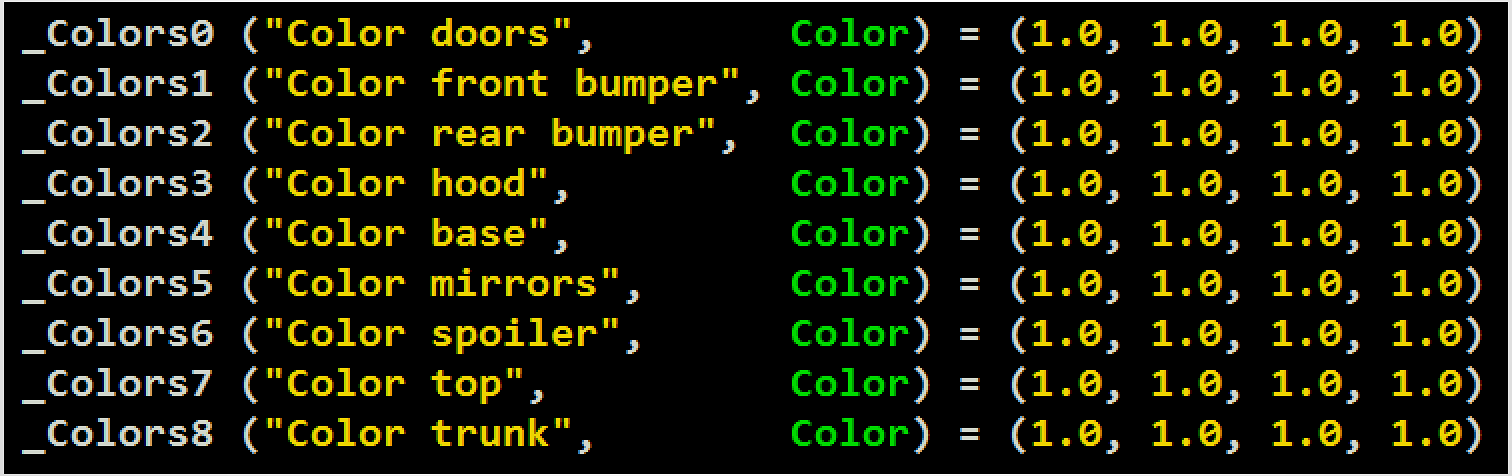


```
CGPROGRAM

...
float4      _Colors[9];
...
int         materialID = (int) v.texcoord.x;
float4      _Color     = _Colors[materialID];
...
ENDCG
```

Material properties

- Each array element can be mapped as individual property

PropertyName = ArrayName+ElementIndex



```
_Colors0 ("Color doors",        Color) = (1.0, 1.0, 1.0, 1.0)
_Colors1 ("Color front bumper", Color) = (1.0, 1.0, 1.0, 1.0)
_Colors2 ("Color rear bumper",  Color) = (1.0, 1.0, 1.0, 1.0)
_Colors3 ("Color hood",         Color) = (1.0, 1.0, 1.0, 1.0)
_Colors4 ("Color base",         Color) = (1.0, 1.0, 1.0, 1.0)
_Colors5 ("Color mirrors",      Color) = (1.0, 1.0, 1.0, 1.0)
_Colors6 ("Color spoiler",      Color) = (1.0, 1.0, 1.0, 1.0)
_Colors7 ("Color top",          Color) = (1.0, 1.0, 1.0, 1.0)
_Colors8 ("Color trunk",        Color) = (1.0, 1.0, 1.0, 1.0)
```

Usage tips

- Don’t make vertex array too large

    - Every array element require ***glUniform…*** call (in Unity3D)

    - It can significantly degrade vertex shader performance on some platforms

- You can use custom property names to display in inspector

    - hood color, body color, etc..

Results

	Before	After
Draw calls	37	**29**
Frame time (HTC One S), ms	3.7	**2.7**

Vertex constants instancing

Motivation

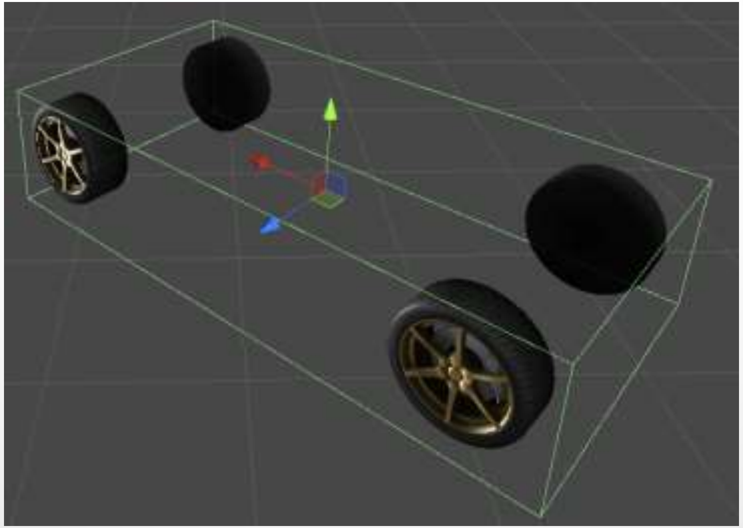
- Reduce draw call count for objects with similar materials even if they have different transform

General algorithm

- Create custom Mesh which contains N copies of original Mesh with InstanceID stored in vertex
- Set custom bounding box
- Add array of uniform matrices to vertex shader (**InstanceData**)
- Add value extraction code to vertex shader
- Write custom script to update **InstanceData**

Visibility problem

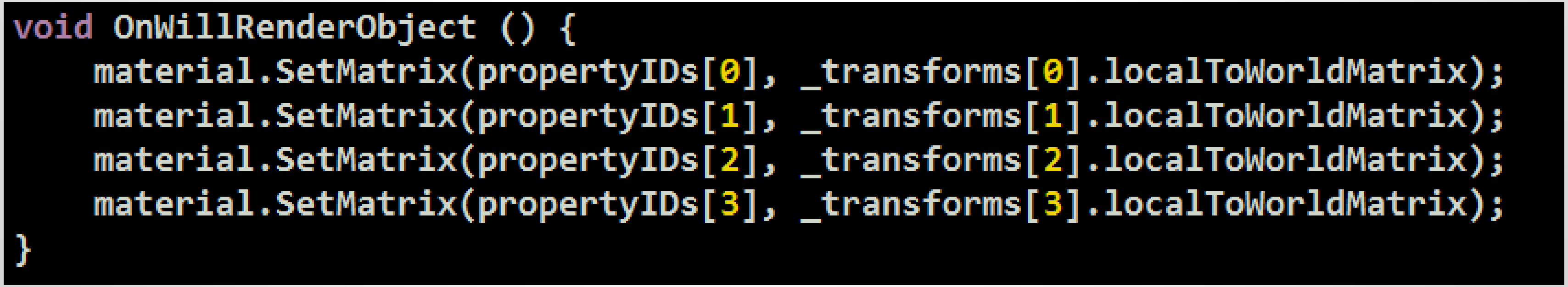
- Create custom bounding box which preserve geometry to become hidden when it should be visible



Custom setup code

- Use **OnWillRenderObject** to setup your data
  - called only if object is visible

- Use **propertyID** for material setup
  - works much faster than usual ‘string’ version



```
void OnWillRenderObject () {
    material.SetMatrix(propertyIDs[0], _transforms[0].localToWorldMatrix);
    material.SetMatrix(propertyIDs[1], _transforms[1].localToWorldMatrix);
    material.SetMatrix(propertyIDs[2], _transforms[2].localToWorldMatrix);
    material.SetMatrix(propertyIDs[3], _transforms[3].localToWorldMatrix);
}
```

Results

	Before	After
Draw calls	29	**17**
Frame time (HTC One S), ms	2.7	**2.0**

Efficiency depending on batch size

Efficiency = timeBefore/timeAfter

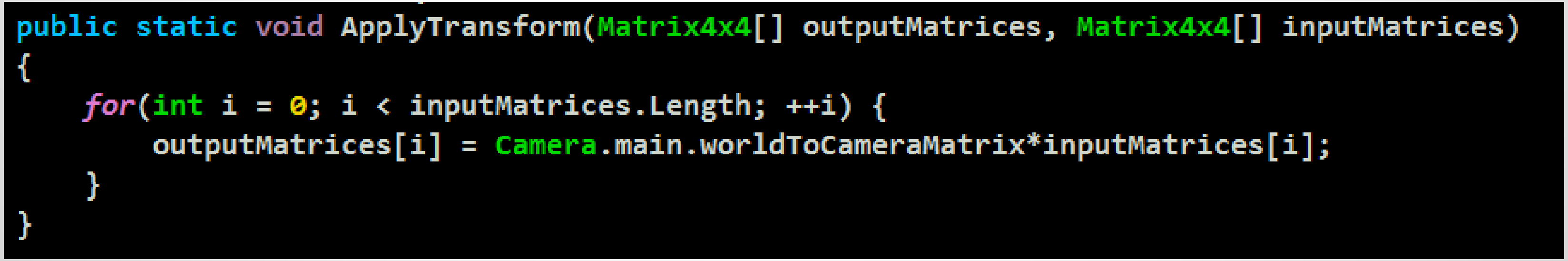
	4 elements batch	8 elements batch	16 elements batch
Mali-400 MP	x2.8	x4.0	x6.0
Adreno 220	x2.9	x5.0	x6.5
PowerVR SGX 540	x3.1	x5.0	x6.5
nVidia Tegra 3	x2.5	x4.0	x5.0

Math optimization

Case study

- Getting 20 000 matrices which transform object from local space to camera space

- Naive implementation: **125 ms** (HTC One S)

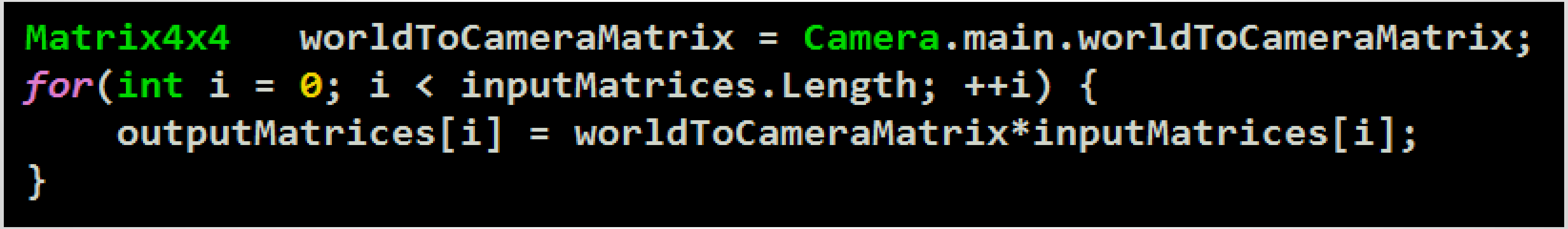


```
public static void ApplyTransform(Matrix4x4[] outputMatrices, Matrix4x4[] inputMatrices)
{
    for(int i = 0; i < inputMatrices.Length; ++i) {
        outputMatrices[i] = Camera.main.worldToCameraMatrix*inputMatrices[i];
    }
}
```

Cache complex expressions

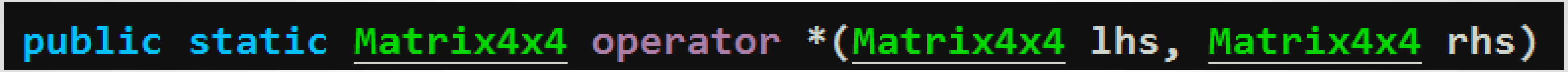
- Most of the properties evaluate complex expression every time they invoked!! (**Transform.forward**, **Vector3.zero** ...)
- Sometimes they result to safe/unsafe context switch

- Optimized implementation: **33.5 ms**/125 ms (HTC One S)



```
Matrix4x4   worldToCameraMatrix = Camera.main.worldToCameraMatrix;
for(int i = 0; i < inputMatrices.Length; ++i) {
    outputMatrices[i] = worldToCameraMatrix*inputMatrices[i];
}
```

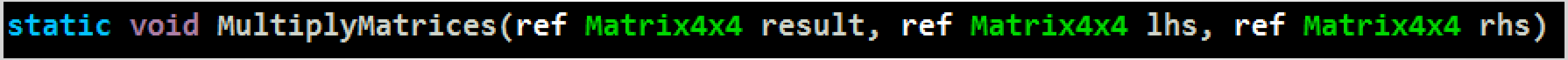
Remove redundant copies



```
public static Matrix4x4 operator *(Matrix4x4 lhs, Matrix4x4 rhs)
```

- Matrix4x4 is value-type
    - We have 2 redundant copies for input arguments
    - We have 1 redundant copy for output result
- Create our own method using reference semantic

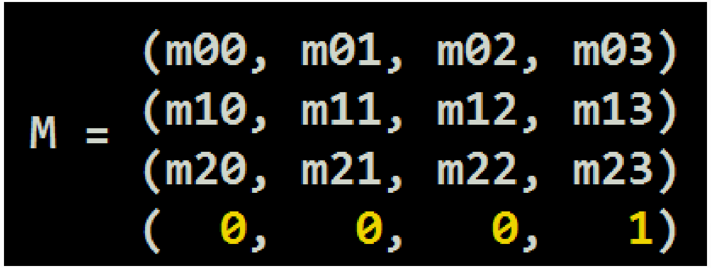
- Optimized implementation: **21.5 ms**/33.5 ms (HTC One S)



```
static void MultiplyMatrices(ref Matrix4x4 result, ref Matrix4x4 lhs, ref Matrix4x4 rhs)
```

Remove redundant calculations

- All our matrices has **Matrix43** semantic



```
    (m00, m01, m02, m03)
M = (m10, m11, m12, m13)
    (m20, m21, m22, m23)
    (  0,   0,   0,   1)
```

- We can remove a lot of calculations using this fact

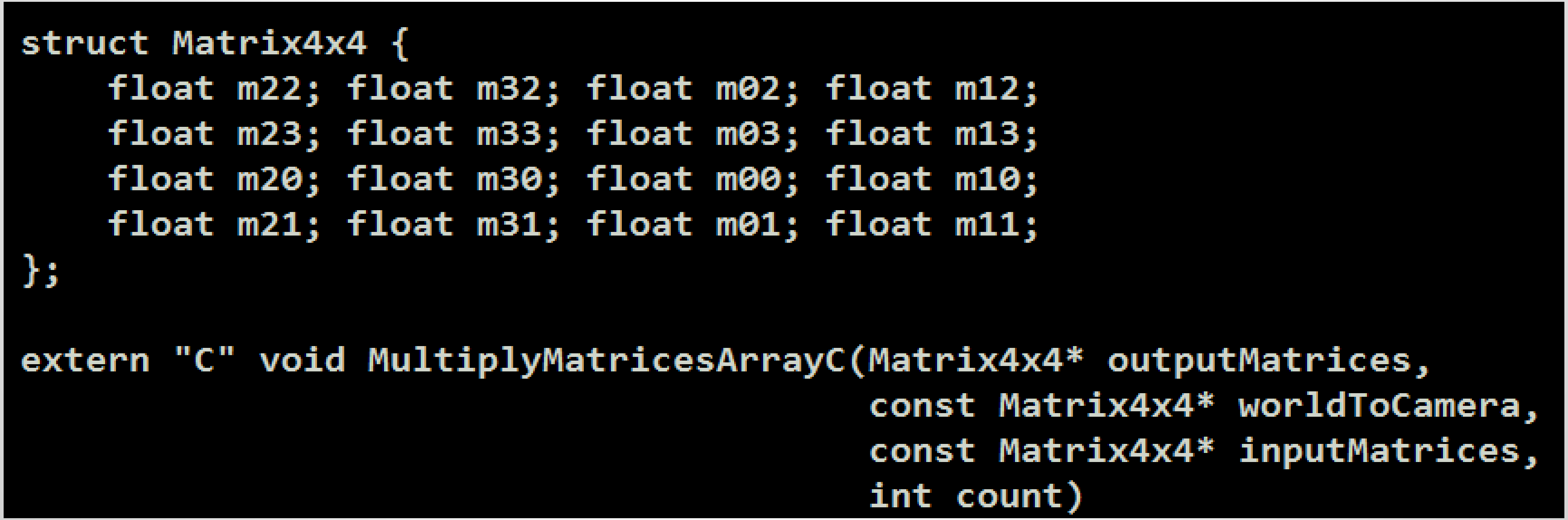
- Optimized implementation: **15.2 ms**/21.5 ms (HTC One S)

Write native plug-in (C code)

- You can use functions from android shared libraries using Platform Invoke Service
- Create your own library using Android-NDK
- Create functions with C-linkage (***extern "C"*** for *.cpp)
- Define your own ***struct Matrix4x4*** (be careful with elements order !!)
- Pass and return elements using pointers
- Now you can use all C++ optimizations !!

Write native plug-in (C code)

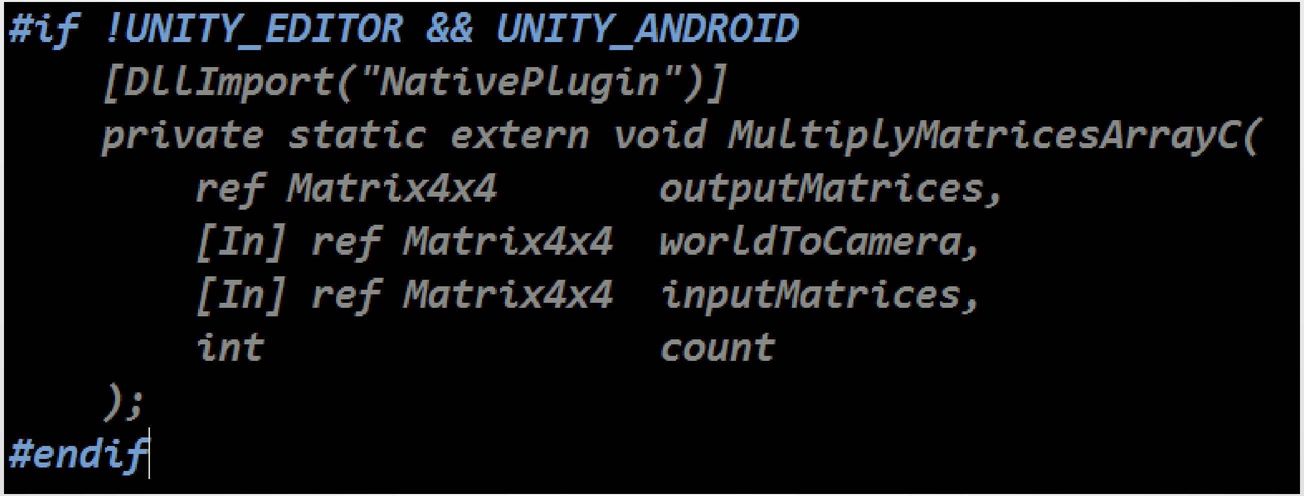
- Simply port our optimized operator to C



```
struct Matrix4x4 {
    float m22; float m32; float m02; float m12;
    float m23; float m33; float m03; float m13;
    float m20; float m30; float m00; float m10;
    float m21; float m31; float m01; float m11;
};

extern "C" void MultiplyMatricesArrayC(Matrix4x4* outputMatrices,
                                       const Matrix4x4* worldToCamera,
                                       const Matrix4x4* inputMatrices,
                                       int count)
```

Write native plug-in (C# code)

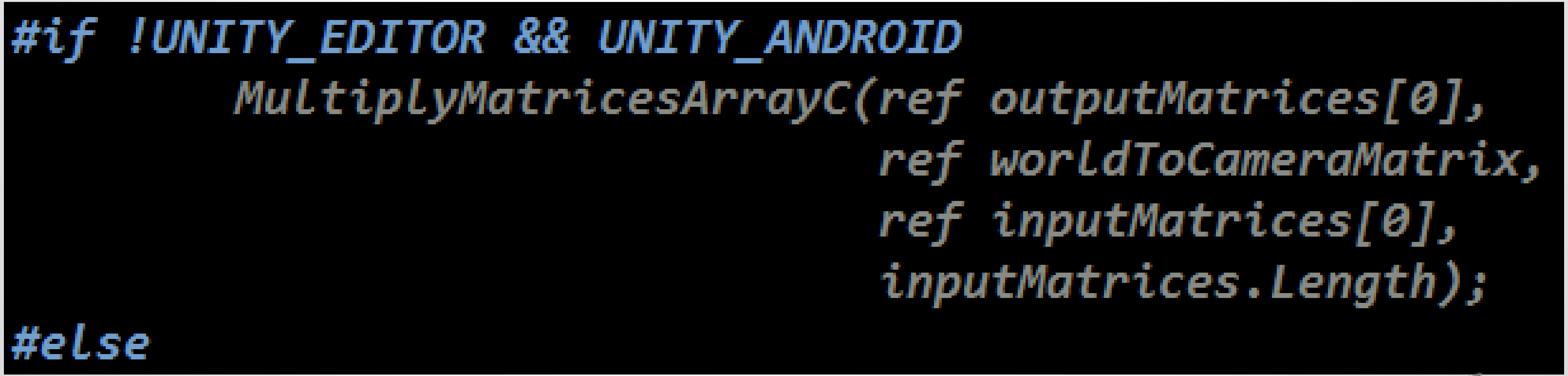
- Insert ‘***using System.Runtime.InteropServices;***’
- Declare P/Invoke method



```
#if !UNITY_EDITOR && UNITY_ANDROID
    [DllImport("NativePlugin")]
    private static extern void MultiplyMatricesArrayC(
        ref Matrix4x4       outputMatrices,
        [In] ref Matrix4x4  worldToCamera,
        [In] ref Matrix4x4  inputMatrices,
        int                 count
    );
#endif
```

Write native plug-in (C# code)

- Pass pointers to array as reference to first element



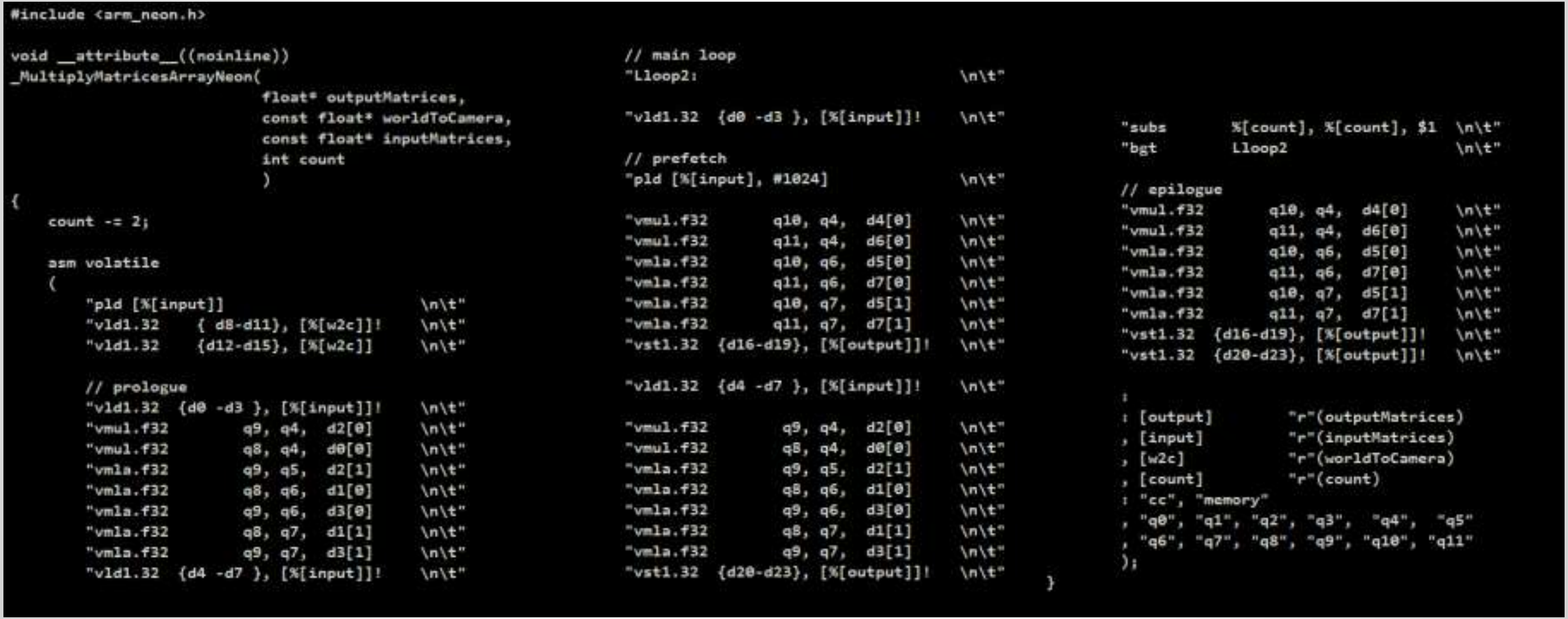
```
#if !UNITY_EDITOR && UNITY_ANDROID
        MultiplyMatricesArrayC(ref outputMatrices[0],
                               ref worldToCameraMatrix,
                               ref inputMatrices[0],
                               inputMatrices.Length);
#else
```

Write native plug-in

- Native plug-ins has constant overhead due to safe/unsafe context switch, so try to use them for large batches of data
- Make sure you enable all compiler/linker optimizations to get maximum performance

- Optimized implementation: **6.8 ms**/15.2 ms (HTC One S)

Use ASM code



```
#include <arm_neon.h>

void __attribute__((noinline))
_MultiplyMatricesArrayNeon(
                           float* outputMatrices,
                           const float* worldToCamera,
                           const float* inputMatrices,
                           int count
                           )
{
    count -= 2;

    asm volatile
    (
        "pld [%[input]]                     \n\t"
        "vld1.32    { d8-d11}, [%[w2c]]!    \n\t"
        "vld1.32    {d12-d15}, [%[w2c]]     \n\t"

        // prologue
        "vld1.32  {d0 -d3 }, [%[input]]!    \n\t"
        "vmul.f32        q9, q4,  d2[0]     \n\t"
        "vmul.f32        q8, q4,  d0[0]     \n\t"
        "vmla.f32        q9, q5,  d2[1]     \n\t"
        "vmla.f32        q8, q6,  d1[0]     \n\t"
        "vmla.f32        q9, q6,  d3[0]     \n\t"
        "vmla.f32        q8, q7,  d1[1]     \n\t"
        "vmla.f32        q9, q7,  d3[1]     \n\t"
        "vld1.32  {d4 -d7 }, [%[input]]!    \n\t"

        // main loop
        "Lloop2:                            \n\t"

        "vld1.32  {d0 -d3 }, [%[input]]!    \n\t"

        // prefetch
        "pld [%[input], #1024]              \n\t"

        "vmul.f32       q10, q4,  d4[0]     \n\t"
        "vmul.f32       q11, q4,  d6[0]     \n\t"
        "vmla.f32       q10, q6,  d5[0]     \n\t"
        "vmla.f32       q11, q6,  d7[0]     \n\t"
        "vmla.f32       q10, q7,  d5[1]     \n\t"
        "vmla.f32       q11, q7,  d7[1]     \n\t"
        "vst1.32  {d16-d19}, [%[output]]!   \n\t"

        "vld1.32  {d4 -d7 }, [%[input]]!    \n\t"

        "vmul.f32        q9, q4,  d2[0]     \n\t"
        "vmul.f32        q8, q4,  d0[0]     \n\t"
        "vmla.f32        q9, q5,  d2[1]     \n\t"
        "vmla.f32        q8, q6,  d1[0]     \n\t"
        "vmla.f32        q9, q6,  d3[0]     \n\t"
        "vmla.f32        q8, q7,  d1[1]     \n\t"
        "vmla.f32        q9, q7,  d3[1]     \n\t"
        "vst1.32  {d20-d23}, [%[output]]!   \n\t"

        "subs       %[count], %[count], $1  \n\t"
        "bgt        Lloop2                  \n\t"

        // epilogue
        "vmul.f32       q10, q4,  d4[0]     \n\t"
        "vmul.f32       q11, q4,  d6[0]     \n\t"
        "vmla.f32       q10, q6,  d5[0]     \n\t"
        "vmla.f32       q11, q6,  d7[0]     \n\t"
        "vmla.f32       q10, q7,  d5[1]     \n\t"
        "vmla.f32       q11, q7,  d7[1]     \n\t"
        "vst1.32  {d16-d19}, [%[output]]!   \n\t"
        "vst1.32  {d20-d23}, [%[output]]!   \n\t"

        :
        : [output]        "r"(outputMatrices)
        , [input]         "r"(inputMatrices)
        , [w2c]           "r"(worldToCamera)
        , [count]         "r"(count)
        : "cc", "memory"
        , "q0", "q1", "q2", "q3",  "q4",  "q5"
        , "q6", "q7", "q8", "q9", "q10", "q11"
        );
}
```

Use ASM code

- Lowest possible level
    - cache prefetch
    - parallel load/store
    - hardware registers management
    - …
- NEON extension is really helpful for vector/matrix math

- Optimized implementation: **3.3 ms**/6.8 ms (HTC One S)

Don’t give up ☺

- Final result **~40 times faster** than naive approach !!

- For most situations it’s enough to perform smart C# optimizations (**~8 times faster**)

- But keep in mind native plug-ins for computationally intensive tasks (additional **~5 times faster**)

GPU performance optimization

Mobile GPU vendors

- Imagination Technologies (PowerVR)
  - Series 5, 5XT, 6, 6XT, Wizard
- ARM (Mali)
  - Utgard (Mali 400), Midgard (Mali T624)
- Qualcomm (Adreno)
  - Adreno 2xx, 3xx, 4xx
- nVidia (Tegra)
  - Tegra 2, 3, 4, K1

Rendering approach PowerVR

- Tile-Based Deferred Rendering (TBDR)
    - pixel-perfect HSR (Hidden Surface Removal)
    - small tiles 32x32 (on-chip registers)
- Scalar ALU architecture (from 6 series)
- Different fragment shader precision supported
- Unified ALU
- High-quality fast MSAA
- 32-bit internal color precision

Rendering approach Mali

- Tile-Based Rendering (TBR)
    - early-Z used for HSR
    - small tiles 16x16 (on-chip registers)
- Separate ALU (Utgard), unified ALU (Midgard)
- Different fragment shader precision supported
- High-quality fast MSAA
- 32-bit internal color precision

Rendering approach Adreno

- Tile-Based Rendering (TBR)
  - early-Z used for HSR
  - large tiles (about 256 Kb, on-chip GMem)
  - binning algorithm to classify polygons
  - vertex shader can be called multiple times
- Unified ALU
- Different fragment shader precision supported
- Scalar ALU (from 3xx)
- MSAA can significantly increase vertex processing cost
- Color precision can be used to reduce tile count

Rendering approach Tegra

- Immediate-Mode Rendering
  - early-Z used for HSR (Hi-Z for Tegra K1)
  - render directly to framebuffer
  - framebuffer compression (from Tegra 4)
- Separate ALU (Tegra 2, 3, 4), unified (Tegra K1)
- CSAA (Tegra 2, 3)
- MSAA (Tegra 4, K1)
- Color precision can reduce bandwidth

Vertex shaders optimization

Motivation

- High-poly meshes
- Some **GPUs** have separate vertex **ALU**
- Adreno **GPU** can process vertices multiple times due to binning algorithm (especially when **MSAA** is enabled)

Optimize before vertex ALU

- Always enable ‘**Optimize Mesh**’ option in Mesh Import Settings
    - Post-transform vertex cache optimization
    - Pre-transform vertex cache optimization

- Always enable ‘**Optimize Mesh Data**’ option in ‘**Player Settings->Other Settings**’
    - Remove redundant vertex attributes (tangents, normals, color etc.)

Case study: Car paint vertex shader

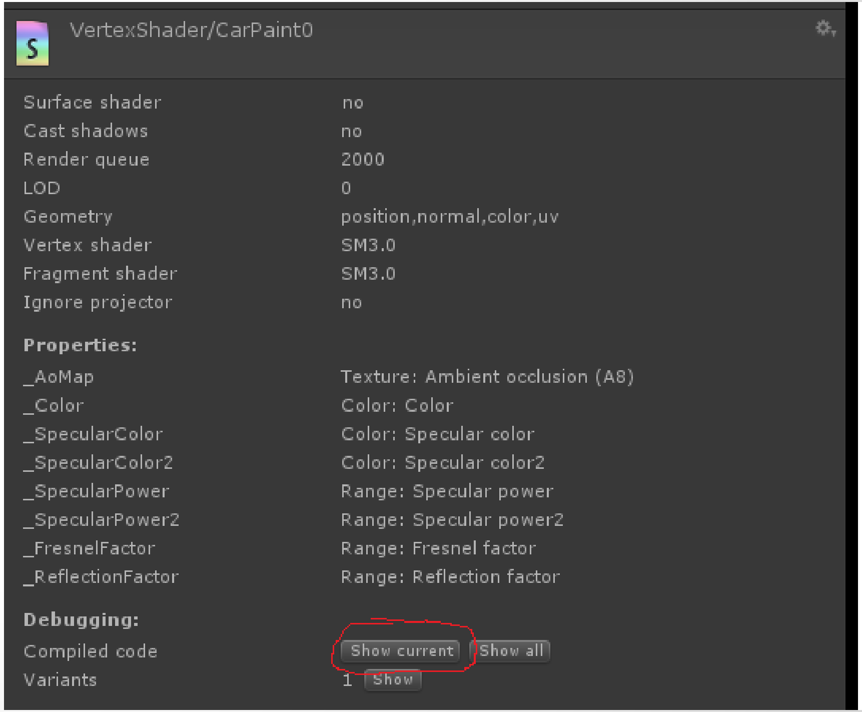
- Use a lot of features
  - diffuse light from 3 light sources
  - fresnel
  - tunnel mapping

  ….

- Optimize for **PowerVR SGX 540**

Optimization pipeline

- Grab **GLSL ES** code



VertexShader/CarPaint0

Surface shader	no
Cast shadows	no
Render queue	2000
LOD	0
Geometry	position,normal,color,uv
Vertex shader	SM3.0
Fragment shader	SM3.0
Ignore projector	no

Properties:

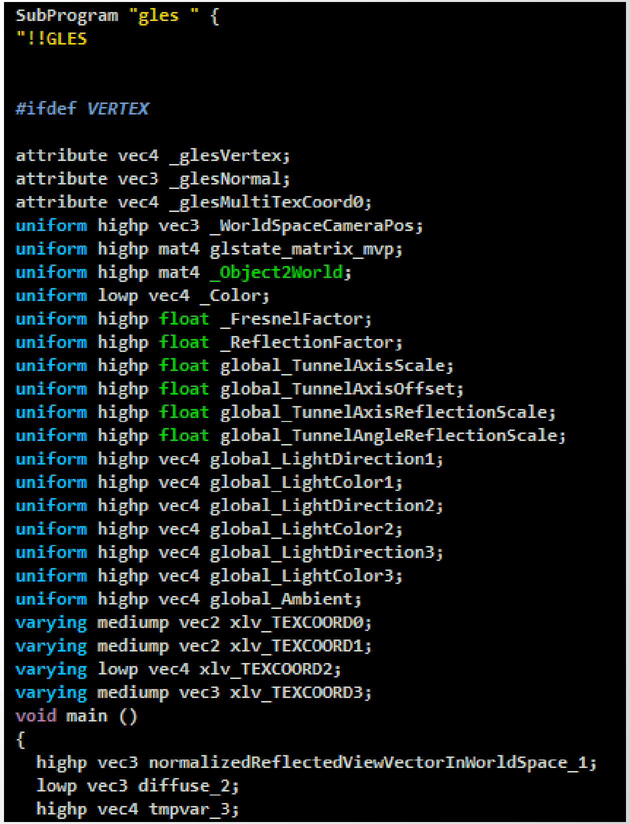
_AoMap	Texture: Ambient occlusion (A8)
_Color	Color: Color
_SpecularColor	Color: Specular color
_SpecularColor2	Color: Specular color2
_SpecularPower	Range: Specular power
_SpecularPower2	Range: Specular power2
_FresnelFactor	Range: Fresnel factor
_ReflectionFactor	Range: Reflection factor

Debugging:

Compiled code	
Variants	1

Optimization pipeline

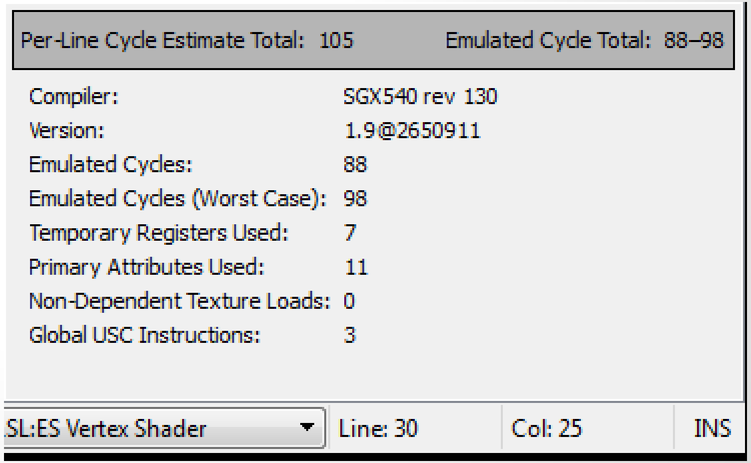
- Copy code from ‘**#ifdef VERTEX**’ section



```
SubProgram "gles " {
"!!GLES


#ifdef VERTEX

attribute vec4 _glesVertex;
attribute vec3 _glesNormal;
attribute vec4 _glesMultiTexCoord0;
uniform highp vec3 _WorldSpaceCameraPos;
uniform highp mat4 glstate_matrix_mvp;
uniform highp mat4 _Object2World;
uniform lowp vec4 _Color;
uniform highp float _FresnelFactor;
uniform highp float _ReflectionFactor;
uniform highp float global_TunnelAxisScale;
uniform highp float global_TunnelAxisOffset;
uniform highp float global_TunnelAxisReflectionScale;
uniform highp float global_TunnelAngleReflectionScale;
uniform highp vec4 global_LightDirection1;
uniform highp vec4 global_LightColor1;
uniform highp vec4 global_LightDirection2;
uniform highp vec4 global_LightColor2;
uniform highp vec4 global_LightDirection3;
uniform highp vec4 global_LightColor3;
uniform highp vec4 global_Ambient;
varying mediump vec2 xlv_TEXCOORD0;
varying mediump vec2 xlv_TEXCOORD1;
varying lowp vec4 xlv_TEXCOORD2;
varying mediump vec3 xlv_TEXCOORD3;
void main ()
{
  highp vec3 normalizedReflectedViewVectorInWorldSpace_1;
  lowp vec3 diffuse_2;
  highp vec4 tmpvar_3;
```

Optimization pipeline

- Paste code into **PVRShaderEditor**
- Select appropriate compiler ‘**Preferences->Compiler Settings->Compiler**’
- Analyze the result



```
Per-Line Cycle Estimate Total:  105          Emulated Cycle Total:  88–98

Compiler:                         SGX540 rev 130
Version:                          1.9@2650911
Emulated Cycles:                  88
Emulated Cycles (Worst Case):     98
Temporary Registers Used:         7
Primary Attributes Used:          11
Non-Dependent Texture Loads:      0
Global USC Instructions:          3

SL:ES Vertex Shader       Line: 30       Col: 25       INS
```

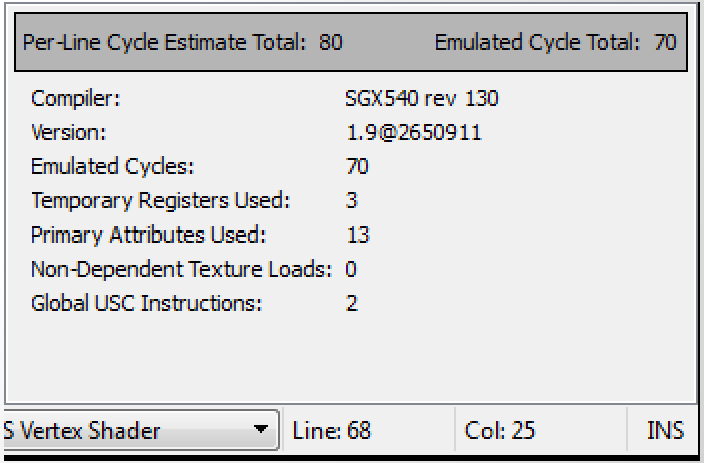
Analyze the result

- Emulated Cycles (Worst Case)
    - maximum number of cycles spent by **ALU** to process single vertex

- Temporary Registers Used
    - **ALU** has limited amount of temporary registers
    - the number of shader threads that work simultaneously depends on this parameter
    - more threads help you to avoid stalls when vertex attribute fetch occurs

Bake complex calculations

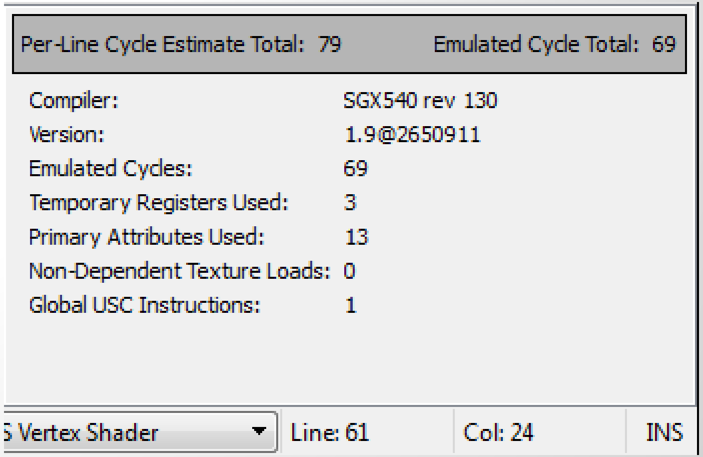
- ***atan2*** function consume too many cycles
- We can pre-compute it using ***AssetPostprocessor.OnPostprocessModel***
- Store result in **mesh.uv2.x**



```
Per-Line Cycle Estimate Total:  80                Emulated Cycle Total:  70

Compiler:                       SGX540 rev 130
Version:                        1.9@2650911
Emulated Cycles:                70
Temporary Registers Used:       3
Primary Attributes Used:        13
Non-Dependent Texture Loads:    0
Global USC Instructions:        2
```

Skip redundant normalization

- Use ***#pragma glsl_no_auto_normalization*** if your normals and tangents don’t need normalization



```
Per-Line Cycle Estimate Total:  79            Emulated Cycle Total:  69

Compiler:                        SGX540 rev 130
Version:                         1.9@2650911
Emulated Cycles:                 69
Temporary Registers Used:        3
Primary Attributes Used:         13
Non-Dependent Texture Loads:     0
Global USC Instructions:         1
```

Simplify math

- Original expression:

```
fresnel = _ReflectionFactor*((1.0f - _FresnelFactor) + _FresnelFactor*pow(1.0f - RdotN, 5.0f));
```

- Simplified expression:

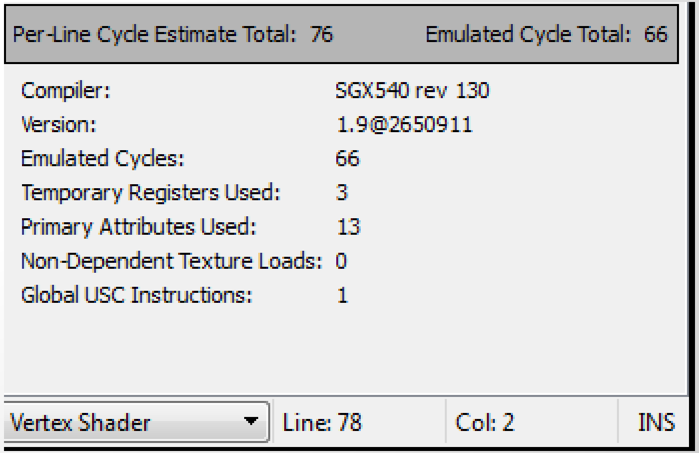
```
fresnel = _FresnelAdd + _FresnelMul*pow(1.0f - RdotN, 5.0f));
```

Simplify math

- Swap ***pow(x, 5.0f)*** with ***x*x*** (visually effect is similar):

fresnelTemp = 1.0f - RdotN;

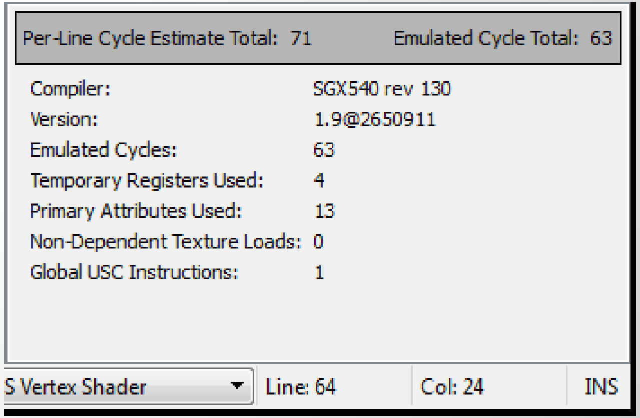
fresnel = _FresnelAdd + _FresnelMul*fresnelTemp*fresnelTemp;



```
Per-Line Cycle Estimate Total:  76               Emulated Cycle Total:  66

Compiler:                        SGX540 rev 130
Version:                         1.9@2650911
Emulated Cycles:                 66
Temporary Registers Used:        3
Primary Attributes Used:         13
Non-Dependent Texture Loads:     0
Global USC Instructions:         1
```

saturate(x) vs. max(x, 0)

- Usually GPU apply ***saturate*** for free…
  - … but this is not correct for **PowerVR**
  - ***saturate(x) = max(min(x, 1), 0)***
  - so it’s more beneficial to use ***max(x, 0)*** for **PowerVR**



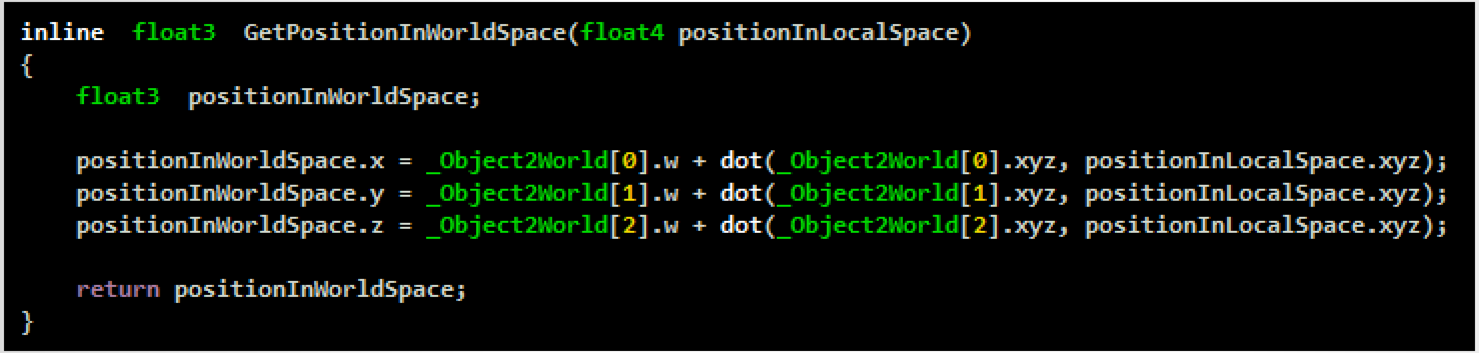
Per-Line Cycle Estimate Total: 71 Emulated Cycle Total: 63

```
Compiler:                      SGX540 rev 130
Version:                       1.9@2650911
Emulated Cycles:               63
Temporary Registers Used:      4
Primary Attributes Used:       13
Non-Dependent Texture Loads:   0
Global USC Instructions:       1
```

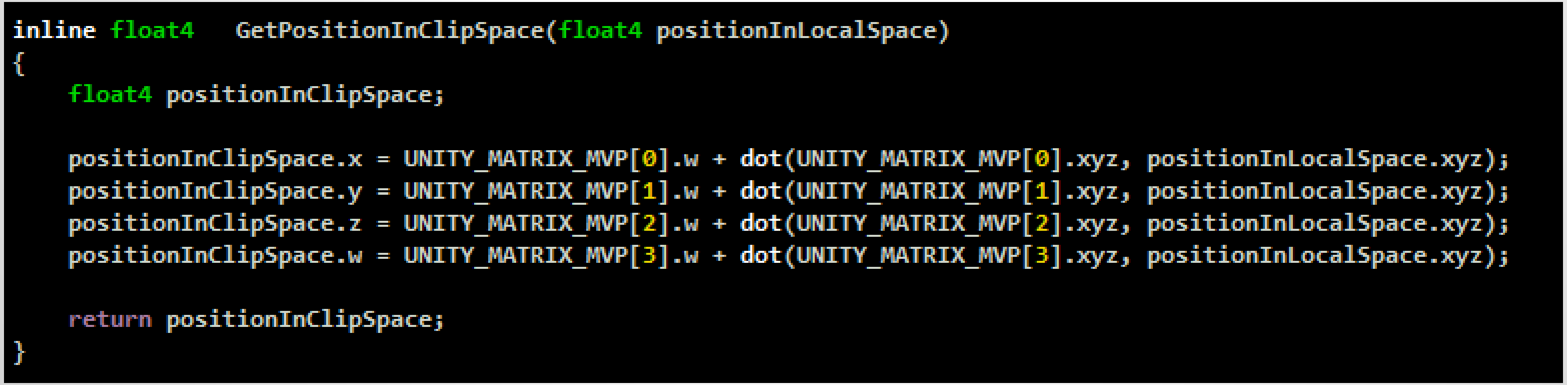
Optimize Point-Matrix multiplication

- ***mul(float4x4, float4)*** doesn’t know anything about actual values
- **PowerVR SGX 540** consume 1 cycle for each scalar ***madd*** in vertex shader
- So it’s beneficial to know that **point = (x, y, z, 1)**
- And some matrices has **(0, 0, 0, 1)** row

Optimize Point-Matrix multiplication



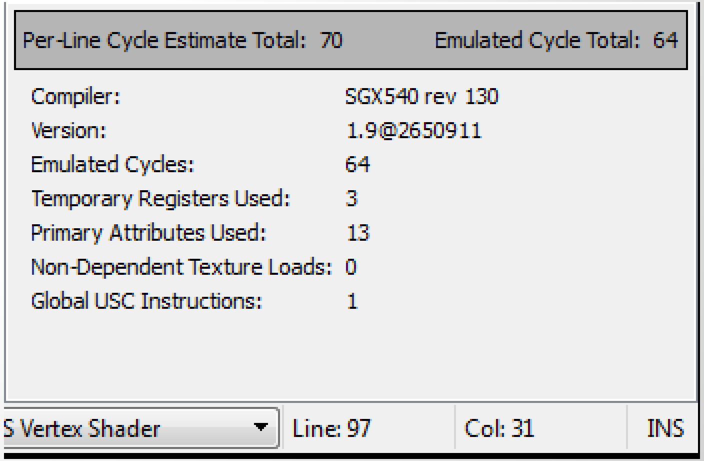
```
inline  float3  GetPositionInWorldSpace(float4 positionInLocalSpace)
{
    float3  positionInWorldSpace;

    positionInWorldSpace.x = _Object2World[0].w + dot(_Object2World[0].xyz, positionInLocalSpace.xyz);
    positionInWorldSpace.y = _Object2World[1].w + dot(_Object2World[1].xyz, positionInLocalSpace.xyz);
    positionInWorldSpace.z = _Object2World[2].w + dot(_Object2World[2].xyz, positionInLocalSpace.xyz);

    return positionInWorldSpace;
}
```



```
inline float4   GetPositionInClipSpace(float4 positionInLocalSpace)
{
    float4 positionInClipSpace;

    positionInClipSpace.x = UNITY_MATRIX_MVP[0].w + dot(UNITY_MATRIX_MVP[0].xyz, positionInLocalSpace.xyz);
    positionInClipSpace.y = UNITY_MATRIX_MVP[1].w + dot(UNITY_MATRIX_MVP[1].xyz, positionInLocalSpace.xyz);
    positionInClipSpace.z = UNITY_MATRIX_MVP[2].w + dot(UNITY_MATRIX_MVP[2].xyz, positionInLocalSpace.xyz);
    positionInClipSpace.w = UNITY_MATRIX_MVP[3].w + dot(UNITY_MATRIX_MVP[3].xyz, positionInLocalSpace.xyz);

    return positionInClipSpace;
}
```

Optimize Point-Matrix multiplication

- We see that actual cycle count has grown slightly…
- … but temporary registers usage reduced



Per-Line Cycle Estimate Total: 70   Emulated Cycle Total: 64

Compiler:	SGX540 rev 130
Version:	1.9@2650911
Emulated Cycles:	64
Temporary Registers Used:	3
Primary Attributes Used:	13
Non-Dependent Texture Loads:	0
Global USC Instructions:	1

S Vertex Shader   Line: 97   Col: 31   INS

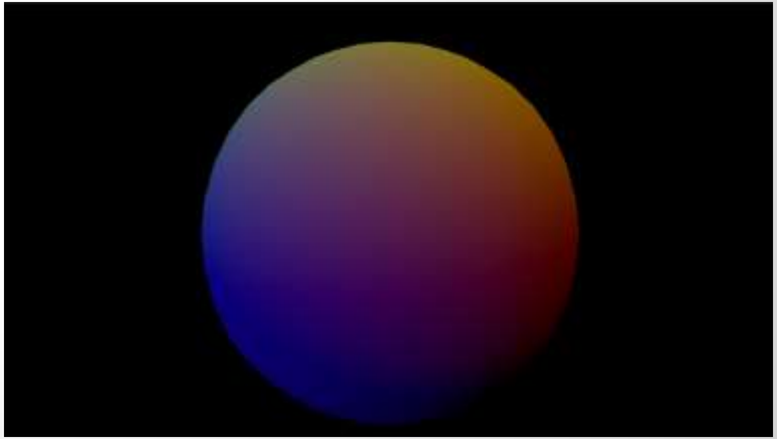
Fragment shaders optimization

Motivation

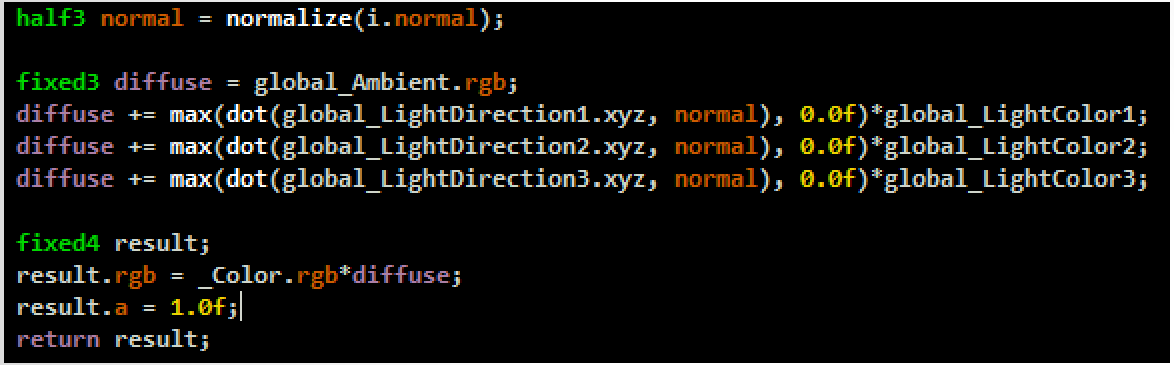
- Modern android devices has lots of megapixels ☺
- So every cycle can make you GPU bound

Case study: diffuse lighting

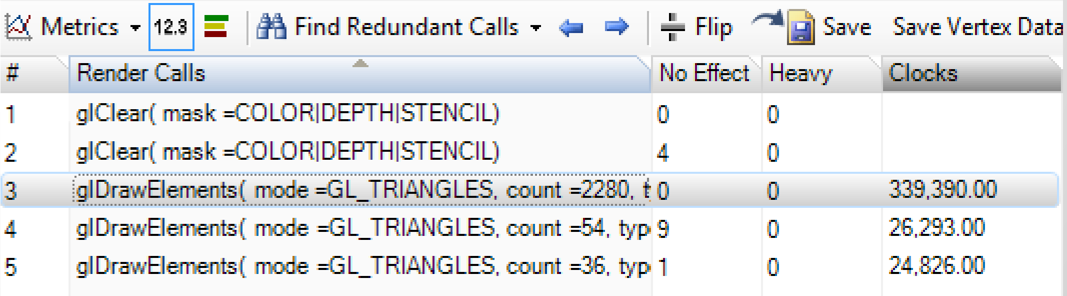
- Per-pixel diffuse lighting from 3 directional light sources
- Target device **HTC One S** (Adreno 220)
- Use **Adreno Profiler** to measure actual cycles spent by GPU



Naive implementation

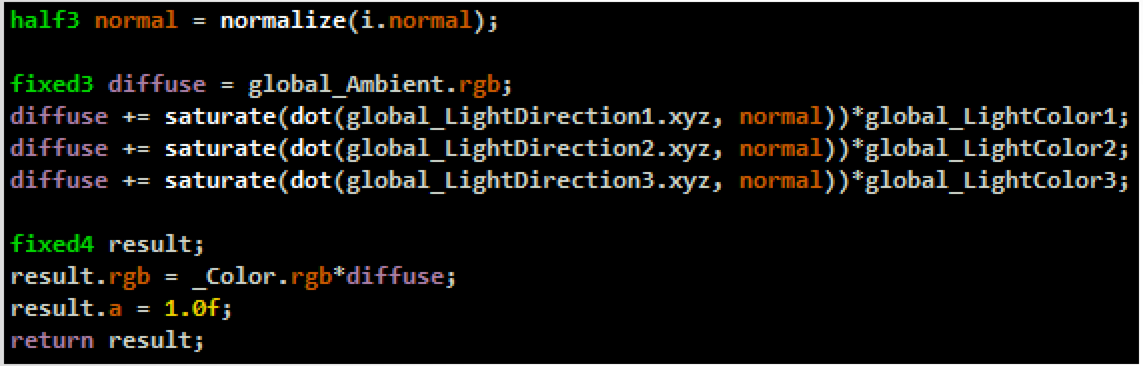


```
half3 normal = normalize(i.normal);

fixed3 diffuse = global_Ambient.rgb;
diffuse += max(dot(global_LightDirection1.xyz, normal), 0.0f)*global_LightColor1;
diffuse += max(dot(global_LightDirection2.xyz, normal), 0.0f)*global_LightColor2;
diffuse += max(dot(global_LightDirection3.xyz, normal), 0.0f)*global_LightColor3;

fixed4 result;
result.rgb = _Color.rgb*diffuse;
result.a = 1.0f;
return result;
```

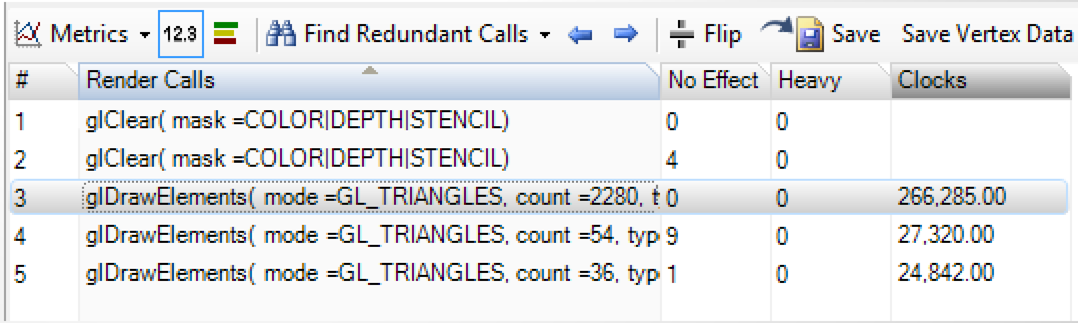


#	Render Calls	No Effect	Heavy	Clocks
1	glClear(mask =COLOR|DEPTH|STENCIL)	0	0	
2	glClear(mask =COLOR|DEPTH|STENCIL)	4	0	
3	glDrawElements(mode =GL_TRIANGLES, count =2280, t	0	0	339,390.00
4	glDrawElements(mode =GL_TRIANGLES, count =54, typ	9	0	26,293.00
5	glDrawElements(mode =GL_TRIANGLES, count =36, typ	1	0	24,826.00

saturate(x) vs. max(x, 0)



```
half3 normal = normalize(i.normal);

fixed3 diffuse = global_Ambient.rgb;
diffuse += saturate(dot(global_LightDirection1.xyz, normal))*global_LightColor1;
diffuse += saturate(dot(global_LightDirection2.xyz, normal))*global_LightColor2;
diffuse += saturate(dot(global_LightDirection3.xyz, normal))*global_LightColor3;

fixed4 result;
result.rgb = _Color.rgb*diffuse;
result.a = 1.0f;
return result;
```

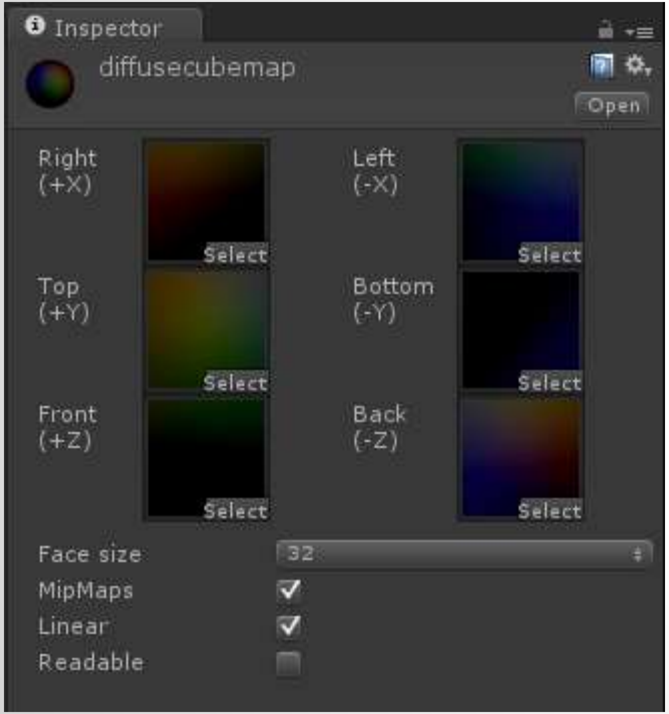


#	Render Calls	No Effect	Heavy	Clocks
1	glClear(mask =COLOR|DEPTH|STENCIL)	0	0	
2	glClear(mask =COLOR|DEPTH|STENCIL)	4	0	
3	glDrawElements(mode =GL_TRIANGLES, count =2280, t	0	0	266,285.00
4	glDrawElements(mode =GL_TRIANGLES, count =54, typ	9	0	27,320.00
5	glDrawElements(mode =GL_TRIANGLES, count =36, typ	1	0	24,842.00

Bake lighting to diffuse cubemap

- Contain lighting environment information about all directional light sources and ambient
- **Normal** vector used to sample diffuse lighting with single ***texCUBE*** instruction
- Generated on **CPU** with relatively simple code
- Pre-multiplied by 0.5 to increase intensity range
- Size of 32 texels is usually enough

Bake lighting to diffuse cubemap

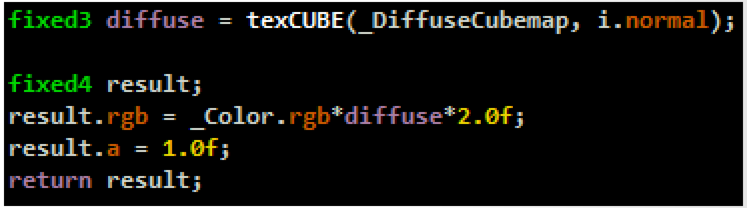


Face size 32

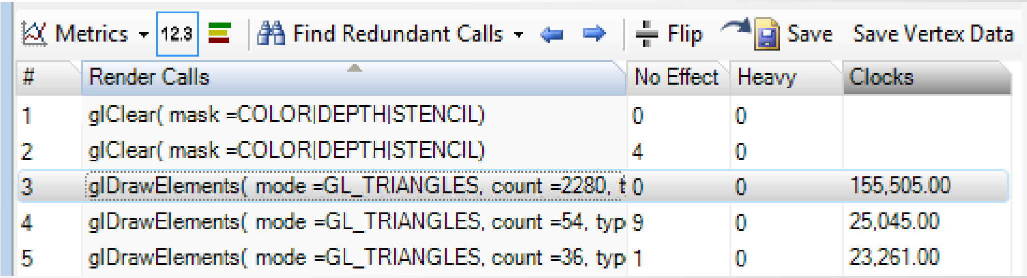
MipMaps ☑

Linear ☑

Readable □



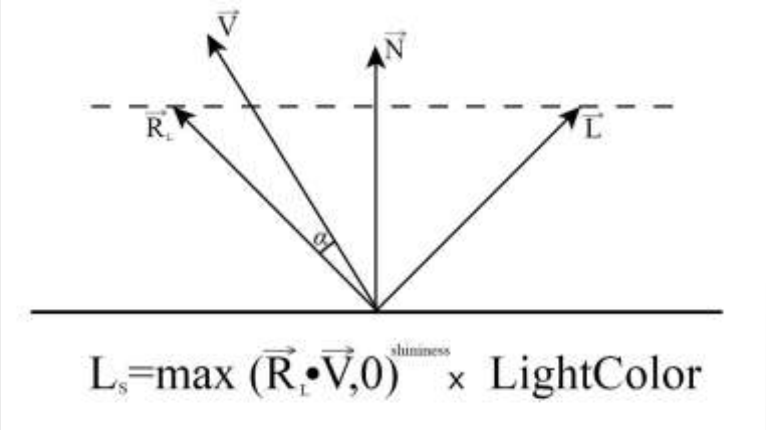
```
fixed3 diffuse = texCUBE(_DiffuseCubemap, i.normal);

fixed4 result;
result.rgb = _Color.rgb*diffuse*2.0f;
result.a = 1.0f;
return result;
```



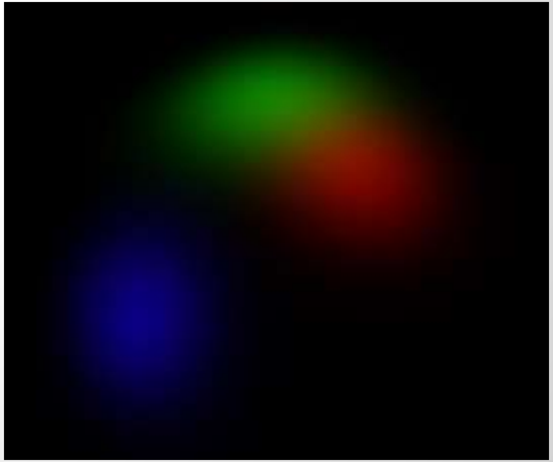
#	Render Calls	No Effect	Heavy	Clocks
1	glClear(mask =COLOR|DEPTH|STENCIL)	0	0	
2	glClear(mask =COLOR|DEPTH|STENCIL)	4	0	
3	glDrawElements(mode =GL_TRIANGLES, count =2280, t	0	0	155,505.00
4	glDrawElements(mode =GL_TRIANGLES, count =54, typ	9	0	25,045.00
5	glDrawElements(mode =GL_TRIANGLES, count =36, typ	1	0	23,261.00

Case study: specular lighting

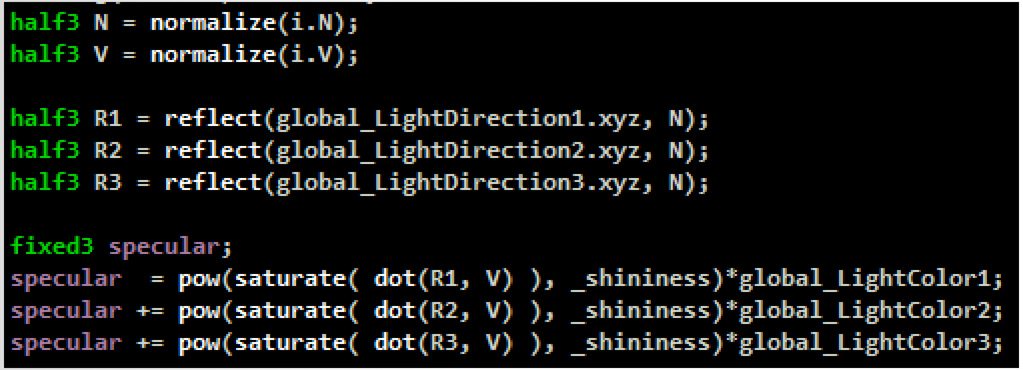
- Per-pixel phong specular lighting from 3 directional light sources
- Target device **HTC One S** (Adreno 220)



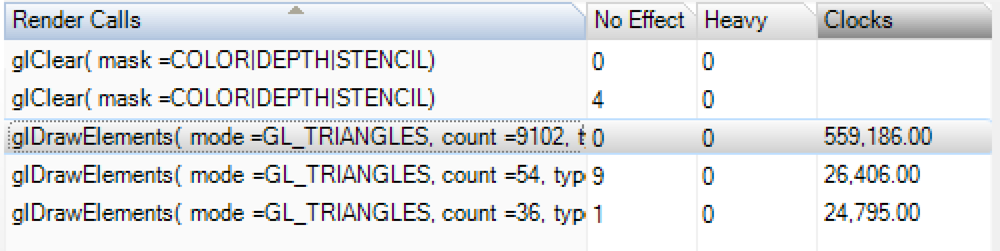
$$L_s = \max(\vec{R}_L \bullet \vec{V}, 0)^{\text{shininess}} \times \text{LightColor}$$



Naive implementation

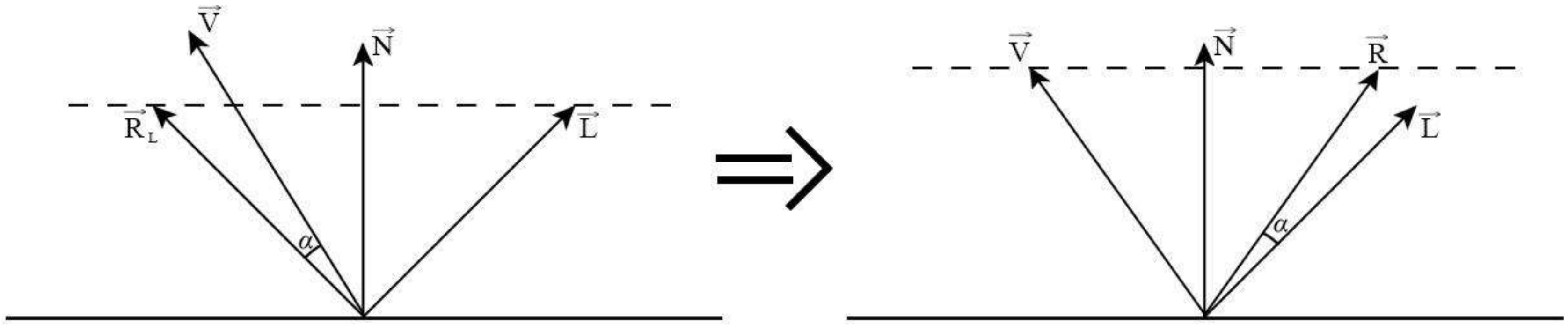


```
half3 N = normalize(i.N);
half3 V = normalize(i.V);

half3 R1 = reflect(global_LightDirection1.xyz, N);
half3 R2 = reflect(global_LightDirection2.xyz, N);
half3 R3 = reflect(global_LightDirection3.xyz, N);

fixed3 specular;
specular  = pow(saturate( dot(R1, V) ), _shininess)*global_LightColor1;
specular += pow(saturate( dot(R2, V) ), _shininess)*global_LightColor2;
specular += pow(saturate( dot(R3, V) ), _shininess)*global_LightColor3;
```



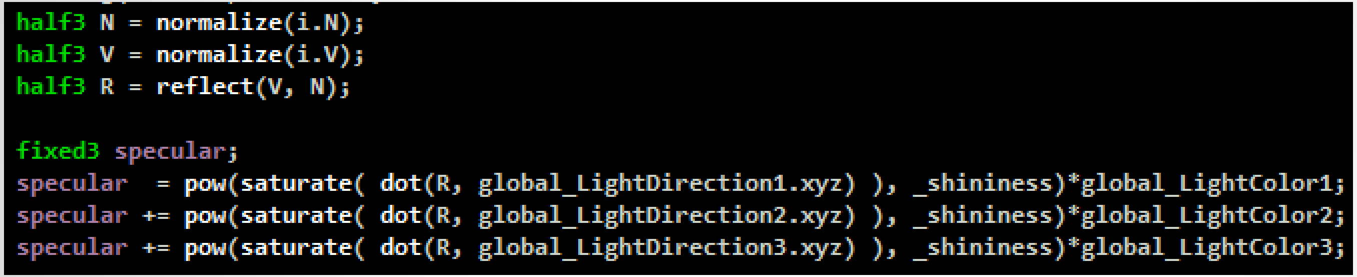
Render Calls	No Effect	Heavy	Clocks
glClear(mask =COLOR|DEPTH|STENCIL)	0	0	
glClear(mask =COLOR|DEPTH|STENCIL)	4	0	
glDrawElements(mode =GL_TRIANGLES, count =9102, t	0	0	559,186.00
glDrawElements(mode =GL_TRIANGLES, count =54, typ	9	0	26,406.00
glDrawElements(mode =GL_TRIANGLES, count =36, typ	1	0	24,795.00

Optimize computation scheme

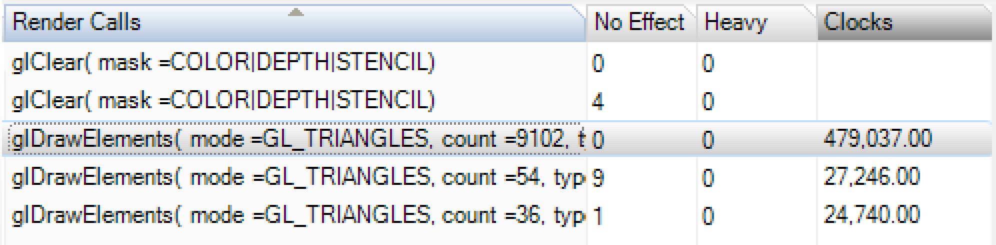


$$L_S = \max(\vec{R} \bullet \vec{L}, 0)^{\text{shininess}} \times \text{LightColor}$$

Optimize computation scheme



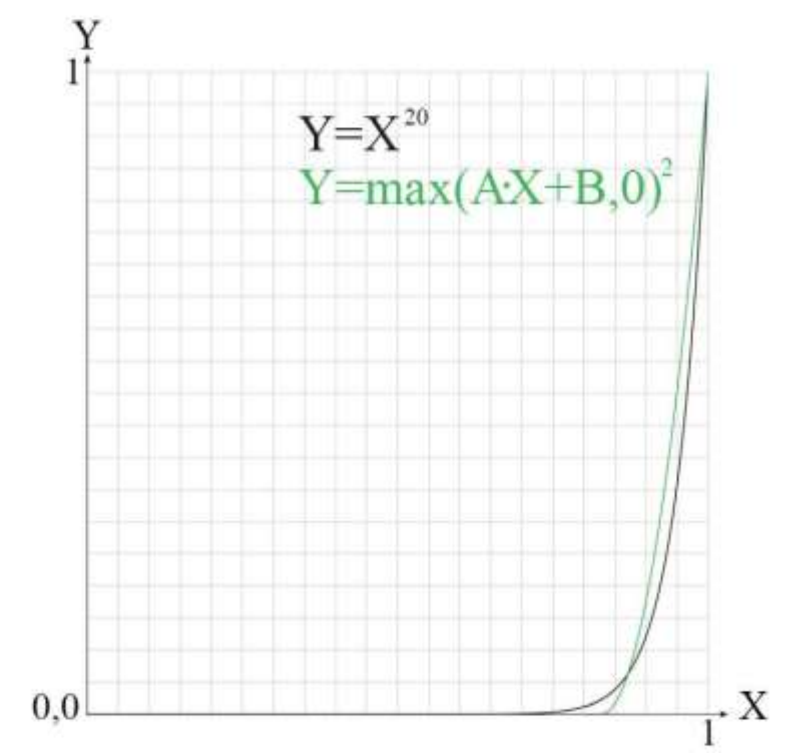
```
half3 N = normalize(i.N);
half3 V = normalize(i.V);
half3 R = reflect(V, N);

fixed3 specular;
specular  = pow(saturate( dot(R, global_LightDirection1.xyz) ), _shininess)*global_LightColor1;
specular += pow(saturate( dot(R, global_LightDirection2.xyz) ), _shininess)*global_LightColor2;
specular += pow(saturate( dot(R, global_LightDirection3.xyz) ), _shininess)*global_LightColor3;
```



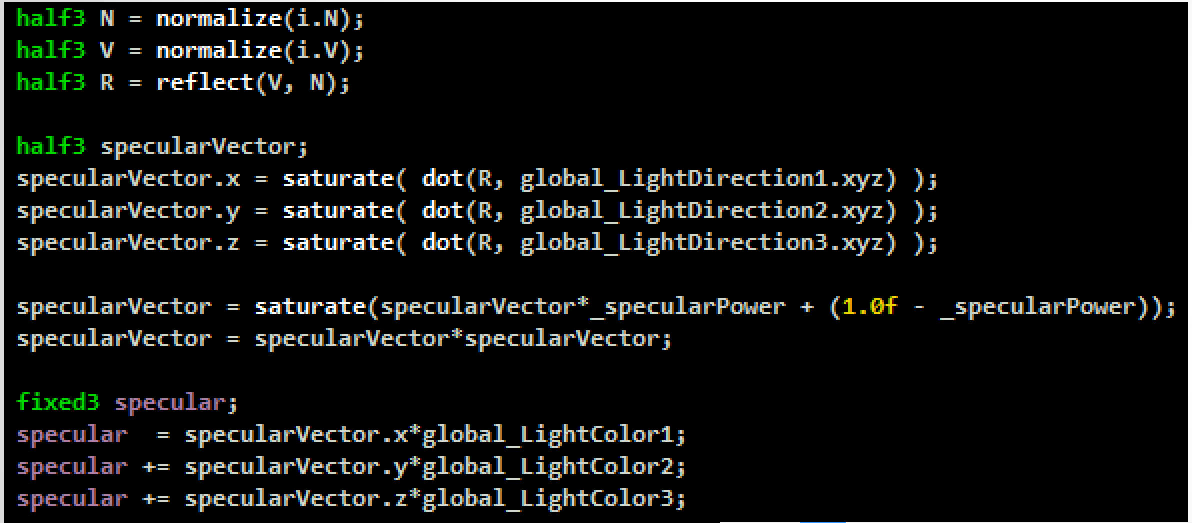
Render Calls	No Effect	Heavy	Clocks
glClear(mask =COLOR|DEPTH|STENCIL)	0	0	
glClear(mask =COLOR|DEPTH|STENCIL)	4	0	
glDrawElements(mode =GL_TRIANGLES, count =9102, t	0	0	479,037.00
glDrawElements(mode =GL_TRIANGLES, count =54, typ	9	0	27,246.00
glDrawElements(mode =GL_TRIANGLES, count =36, typ	1	0	24,740.00

Use fast pow approximation

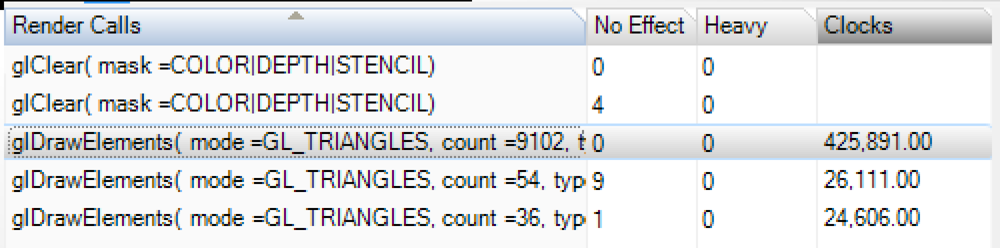
- ***pow*** is usually scalar and consume a lot of cycles
- approximation can make up to 4 pows in just 2 cycles!!



Use fast pow approximation

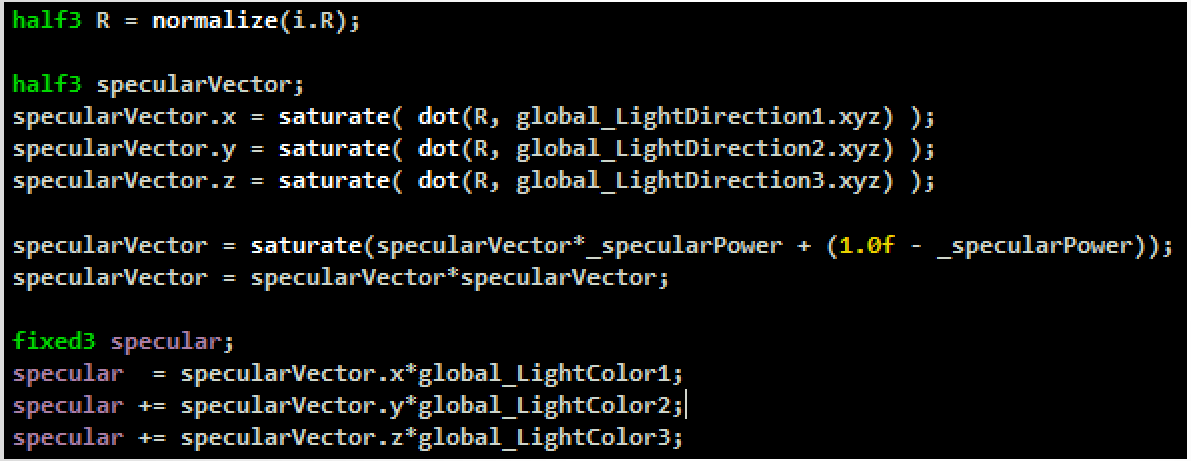


```
half3 N = normalize(i.N);
half3 V = normalize(i.V);
half3 R = reflect(V, N);

half3 specularVector;
specularVector.x = saturate( dot(R, global_LightDirection1.xyz) );
specularVector.y = saturate( dot(R, global_LightDirection2.xyz) );
specularVector.z = saturate( dot(R, global_LightDirection3.xyz) );

specularVector = saturate(specularVector*_specularPower + (1.0f - _specularPower));
specularVector = specularVector*specularVector;

fixed3 specular;
specular  = specularVector.x*global_LightColor1;
specular += specularVector.y*global_LightColor2;
specular += specularVector.z*global_LightColor3;
```

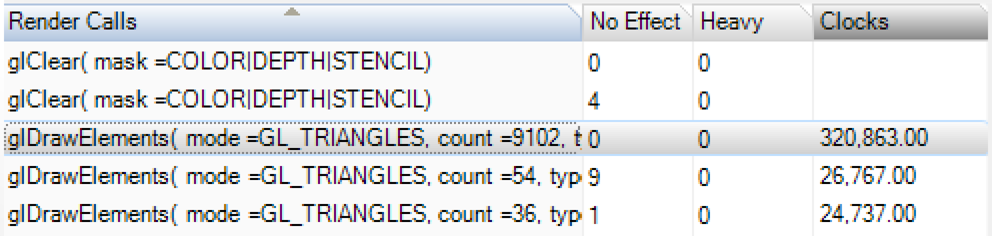


Render Calls	No Effect	Heavy	Clocks
glClear(mask =COLOR|DEPTH|STENCIL)	0	0	
glClear(mask =COLOR|DEPTH|STENCIL)	4	0	
glDrawElements(mode =GL_TRIANGLES, count =9102, t	0	0	425,891.00
glDrawElements(mode =GL_TRIANGLES, count =54, typ	9	0	26,111.00
glDrawElements(mode =GL_TRIANGLES, count =36, typ	1	0	24,606.00

Move calculations to vertex shader



```
half3 R = normalize(i.R);

half3 specularVector;
specularVector.x = saturate( dot(R, global_LightDirection1.xyz) );
specularVector.y = saturate( dot(R, global_LightDirection2.xyz) );
specularVector.z = saturate( dot(R, global_LightDirection3.xyz) );

specularVector = saturate(specularVector*_specularPower + (1.0f - _specularPower));
specularVector = specularVector*specularVector;

fixed3 specular;
specular  = specularVector.x*global_LightColor1;
specular += specularVector.y*global_LightColor2;
specular += specularVector.z*global_LightColor3;
```



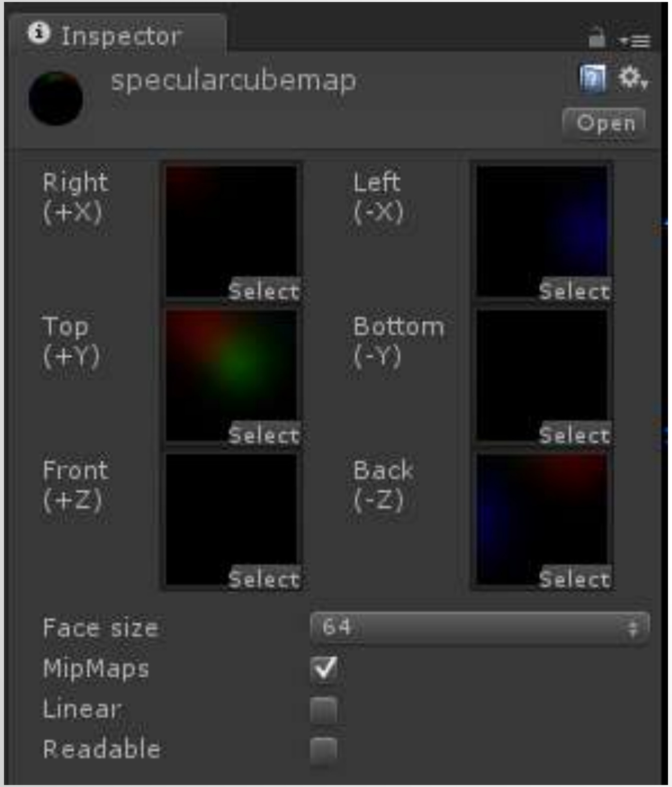
Render Calls	No Effect	Heavy	Clocks
glClear(mask =COLOR|DEPTH|STENCIL)	0	0	
glClear(mask =COLOR|DEPTH|STENCIL)	4	0	
glDrawElements(mode =GL_TRIANGLES, count =9102, t	0	0	320,863.00
glDrawElements(mode =GL_TRIANGLES, count =54, typ	9	0	26,767.00
glDrawElements(mode =GL_TRIANGLES, count =36, typ	1	0	24,737.00

Bake lighting to specular cubemap

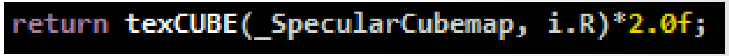
- **Reflected view** vector used to sample specular lighting with single ***texCUBE*** instruction
- Size of 64 texels is usually enough
- Pre-multiplied by 0.5 to increase intensity range

- Really interesting BRDF can be baked using this approach

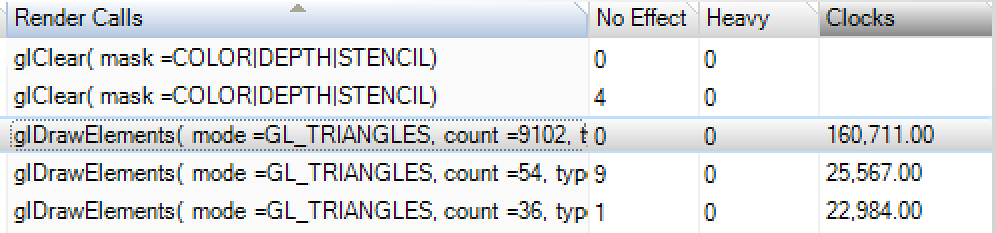
Bake lighting to specular cubemap



Face size 64
MipMaps ☑
Linear □
Readable □



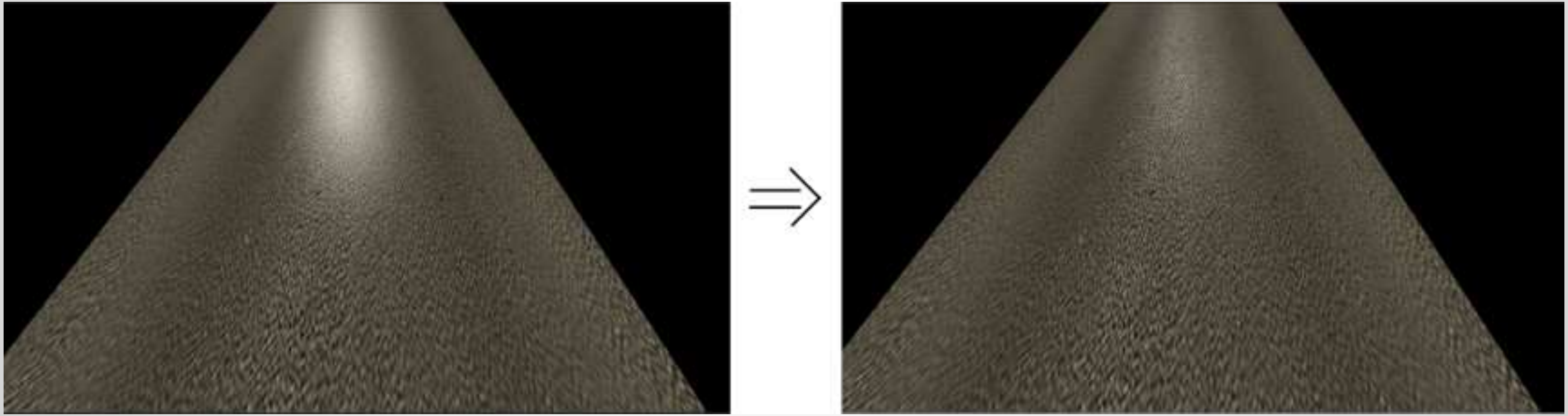
```
return texCUBE(_SpecularCubemap, i.R)*2.0f;
```



Render Calls	No Effect	Heavy	Clocks
glClear(mask =COLOR|DEPTH|STENCIL)	0	0	
glClear(mask =COLOR|DEPTH|STENCIL)	4	0	
glDrawElements(mode =GL_TRIANGLES, count =9102, t	0	0	160,711.00
glDrawElements(mode =GL_TRIANGLES, count =54, typ	9	0	25,567.00
glDrawElements(mode =GL_TRIANGLES, count =36, typ	1	0	22,984.00

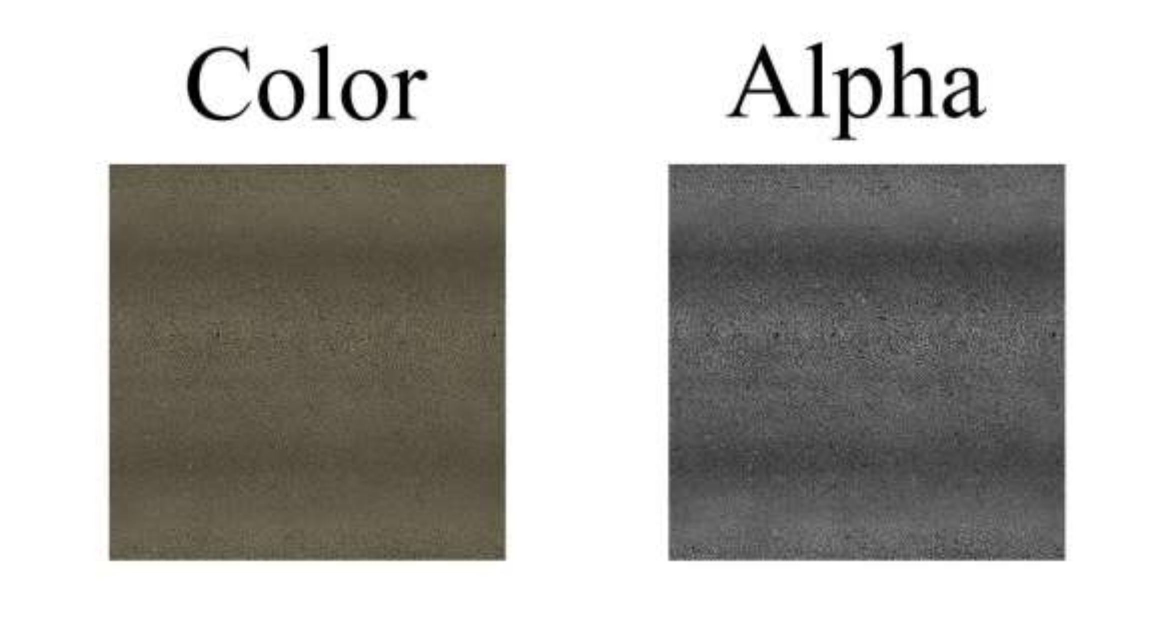
Case study: specular map

- Specular intensity map greatly improve realism
- Optimize for **HTC One S**

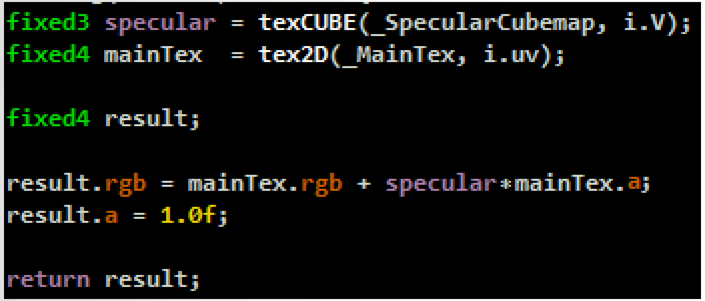


Naive approach

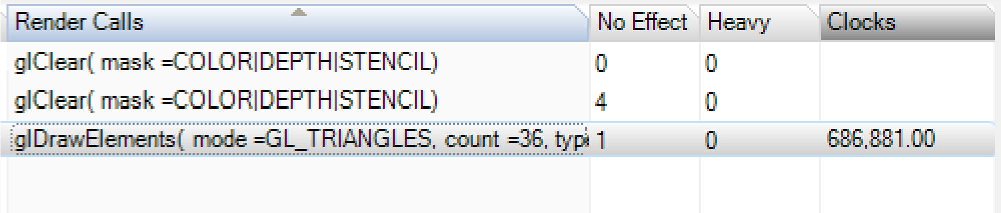
- Store in alpha channel of diffuse texture
- Use ARGB32 texture format



Naive approach



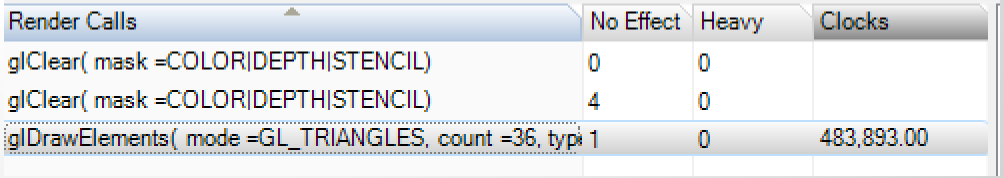
```
fixed3 specular = texCUBE(_SpecularCubemap, i.V);
fixed4 mainTex  = tex2D(_MainTex, i.uv);

fixed4 result;

result.rgb = mainTex.rgb + specular*mainTex.a;
result.a = 1.0f;

return result;
```



Render Calls	No Effect	Heavy	Clocks
glClear(mask =COLOR|DEPTH|STENCIL)	0	0	
glClear(mask =COLOR|DEPTH|STENCIL)	4	0	
glDrawElements(mode =GL_TRIANGLES, count =36, typ	1	0	686,881.00

Use better texture format

- Adreno GPUs support compressed texture format with 8 bits per pixel which contains alpha channel
- The problem is you should select different format for all platforms
- **Mali Utgard** doesn't support compressed textures with alpha channel ☹



Render Calls	No Effect	Heavy	Clocks
glClear(mask =COLOR|DEPTH|STENCIL)	0	0	
glClear(mask =COLOR|DEPTH|STENCIL)	4	0	
glDrawElements(mode =GL_TRIANGLES, count =36, typ	1	0	483,893.00

Recover specular map in shader

- Use ETC1 to store diffuse map
- Compute diffuse luminance for each pixel

luminance = 0.3*R + 0.58*G + 0.12*B

- Scale and bias this luminance to get specular intensity

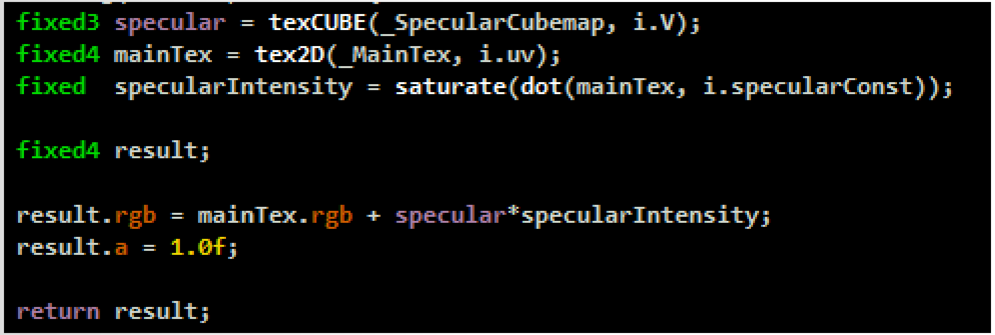
specIntensity = saturate(luminance*scale + bias)

- Combine both formulas

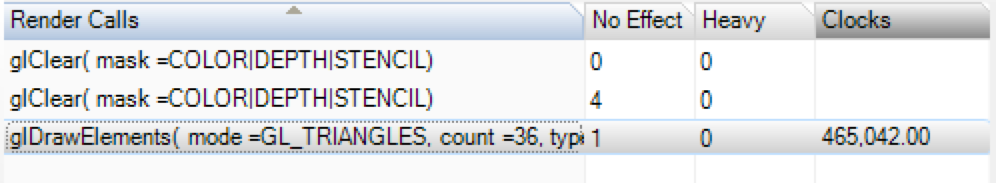
specIntensity = saturate(dot(RGB1, specConst))

specConst = (0.3*scale, 0.58*scale, 0.12*scale, bias)

Recover specular map in shader



```
fixed3 specular = texCUBE(_SpecularCubemap, i.V);
fixed4 mainTex = tex2D(_MainTex, i.uv);
fixed  specularIntensity = saturate(dot(mainTex, i.specularConst));

fixed4 result;

result.rgb = mainTex.rgb + specular*specularIntensity;
result.a = 1.0f;

return result;
```



Render Calls	No Effect	Heavy	Clocks
glClear(mask =COLOR|DEPTH|STENCIL)	0	0	
glClear(mask =COLOR|DEPTH|STENCIL)	4	0	
glDrawElements(mode =GL_TRIANGLES, count =36, typ	1	0	465,042.00

Questions ?