Fast mobile shaders or rather Mobile! Fast!

Aras Pranckevičius
@aras_p
Renaldas Zioma
@__ReJ__
Unity Technologies

Original title: how to write fast iPhone and Android shaders in Unity; presented at Siggraph 2011.

However, this is too long and not quite true. We talked about graphics performance (not only shaders) on mobile platforms (not much Unity specific).

Slide notes were written after the talk and probably do not match what was actually said ;)

SHADOWGUN video
was here

SHADOWGUN by MADFINGER http://www.youtube.com/watch?v=7aVttB25oPo

Outline

- Desktop vs. Mobile
- GPU Architectures
- Case study: SHADOWGUN
- Tools
- Shader Performance
- Case study: Unanounced title ;)
- Case study: AngryBots
- Optimizing complex shaders

External resources can be super helpful:

* PowerVR SGX Architecture Guide http://imgtec.com/powervr/insider/powervr-sdk-docs.asp
* Tegra GLES2 feature guide http://developer.download.nvidia.com/tegra/docs/tegra_gles2_development.pdf -- ask nVIDIA for internal docs ;)
* Qualcomm Adreno GLES performance guide http://developer.qualcomm.com/file/607/adreno200performanceoptimizationopenglestipsandtricksmarch10.pdf
* Engel, Rible http://altdevblogaday.com/2011/08/04/programming-the-xperia-play-gpu-by-wolfgang-engel-and-maurice-ribble/
* ARM Mali GPU Optimization guide http://www.malideveloper.com/developer-resources/documentation/index.php

Desktop vs Mobile

Big monster on one side, tiny part of a small chip on the other side

Mobile

- Needs to use 100x less power
- Be 100x smaller
- Not burn ur pants!

- Bound to be slower

This is quite obvious. Mobile GPUs can’t be as fast as desktop/console ones because of thermal, power, size, noise, ... restrictions.

Can Haz Fast?

So, how to make rendering acceptably fast on mobile platforms?
Some generic tips follow

Old Is New

- Techniques from 2001 in full revenge
- Simple tricks
- Fakes
- Rely on artwork

Content!

- Put into artwork instead
  - Good artwork saves lives
  - Fancy tech, not so much
- Bake
- Offline/editor tools
  - Take a look at Unity :)

Last point: to make artists efficient at making these fakes/hacks/tweaks, you need really good tools to help them. Placement helpers, glow card editors, lookup texture bakers, instant visualization of how it will look like and so on.

Turns out Unity is _very_ extensible with custom editor scripts & UIs. Use them!

Workload

- Frequency of computation
- 100 objects, 50k verts, million pixels
- Pixel: short, vertex: longer, object: woohoo!

Watch out where you spend your computing power. There are way more pixels than vertices, and way more vertices than objects.

This means that you can afford only so much computation per pixel; somewhat more on vertices; and a lot more per-object.

Textures

- Use mipmaps
  - Too blurry? Sharpen mip levels and/or use aniso
- Use texture compression
  - PVRTC, DXT, ATC, ETC, oh my!

As on any other platform. Use textures with mipmaps, unless you know the texture will never be minified.

We find that using anisotropic filtering (even at 2x) and/or manually sharpening smaller mip levels makes everything much crisper.

Texture compression is a bit cumbersome on platforms like Android, where each GPU has it's own texture compression (PVRTC for PowerVR; DXT for Tegra; ATC for Adreno and so on). Android Market, however, supports app filtering by texture compression formats, so you can build separate versions of your game for these different device families. Unity can do this for you since Unity 3.4 ;)

Scene

- Opaque is the best
- Avoid alpha test/blend if you can
- Order: clear, opaque, alpha test, skybox, alpha blend
- Use post-processing with care

Rendering order is critical. In general case: 1) fully opaque objects roughly front-to-back, 2) alpha tested objects roughly front-to-back, 3) skybox, 4) alpha blended objects (back to front if needed). More details later.

Scaling with HW

- Up to 10x performance difference in popular mobile GPUs
- Your weapons:
  - Resolution
  - Post-processing
  - MSAA
  - Anisotropy
  - Shaders
  - Fx/particles density, on/off

Just like on PCs, mobile platforms like iOS and Android have devices of various levels of performance. You can easily find a phone that’s 10x more powerful for rendering than some other phone.

Quite easy way of scaling: 1) make sure it runs okay on baseline configuration, 2) use more eye-candy on higher performing configurations.

“Onload” CPU?

- More fast cores
- Skinning, batching, particle geometry
  - Easily doable & works
- Occlusion cull, post-fx, shadowmaps, particle rast
  - API limitations... someday?

Right now we very often find that games are limited by the GPU on pixel processing. So they end up having unused CPU power, especially on multicore mobile CPUs. So often it makes sense to pull some work off the GPU and put it onto CPU instead.

Some easy examples (Unity does all of them right now): mesh skinning, batching of small objects, particle geometry updates.

Possible future examples: rasterizing depth for occlusion culling on the CPU, rasterizing shadowmaps or particles on the CPU, etc. Some of those are hard to do because of OpenGL ES 2.0 limitations right now though.

Also, even if putting CPU to full use might make the game faster, it might make it drain more battery. So as always, adapt all advice to your own situation.

GPU Architectures

Details of current mobile GPU architectures

GPU Species

- ImgTec PowerVR SGX
- NVIDIA Tegra
- Qualcomm Adreno
- ARM Mali

- Different from NVIDIA+AMD+Intel in PC space

These are the popular mobile architectures right now. This is both different hardware vendors than in PC/console space, and very different GPU architectures than the “usual” GPUs.

Mobile GPUs

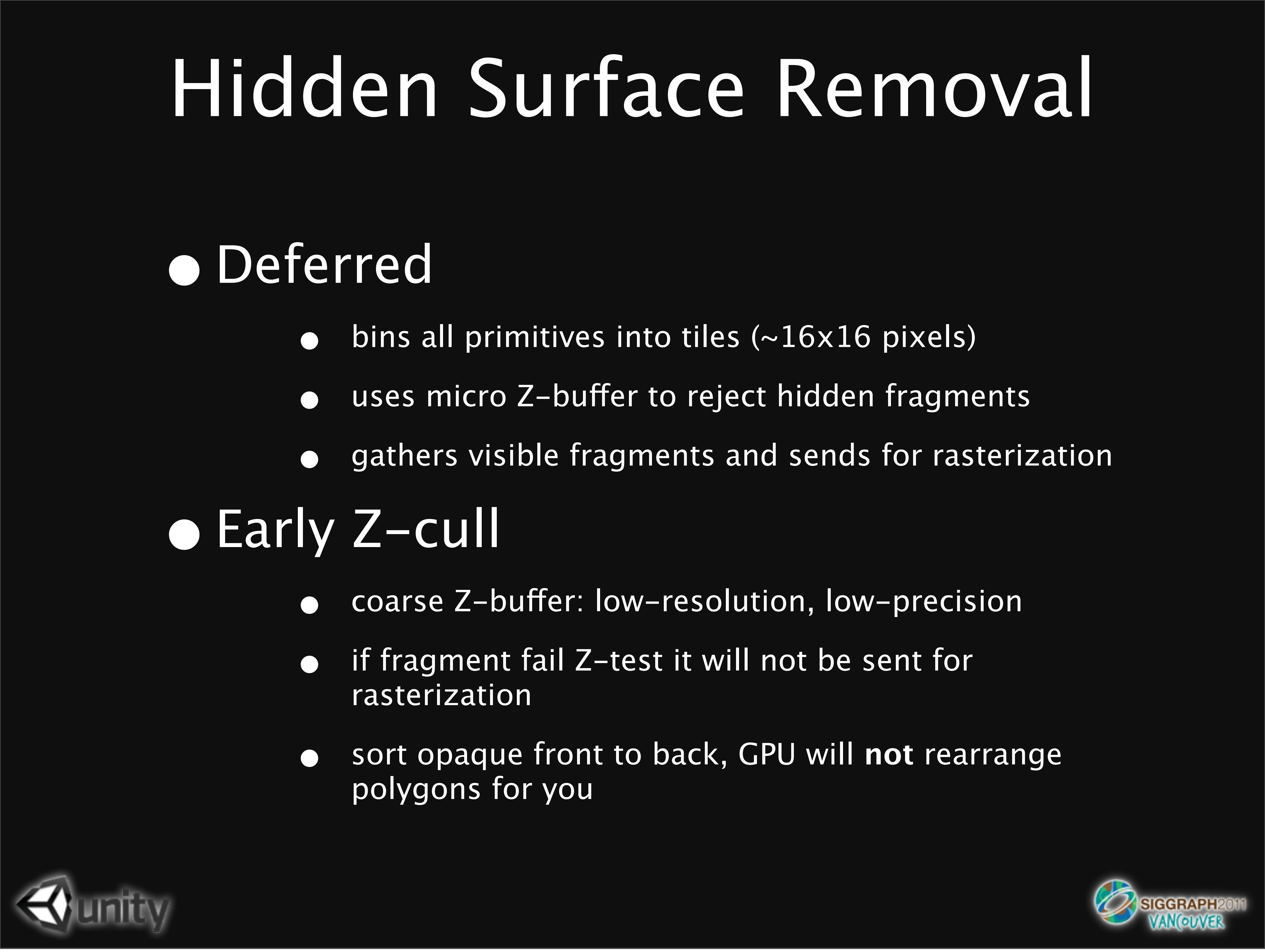
- Very smart hardware
- But: low memory BW, low ALU perf
  - Energy, thermal, size constrains
- Fullscreen effects: think more than twice
  - Lots of other ways to make it look good!

Mobile GPUs have huge constraints in how much heat they produce, how much power they use, and how large or noisy they can be. So compared to the desktop parts, mobile GPUs have way less bandwidth, math and texturing power. The architectures of the GPUs are also tuned to use as little bandwidth & power as possible, resulting in sometimes strange beasts ;)

Architectures

- Tiled Deferred
  - PowerVR
- Tiled
  - Adreno, Mali
- Classic (immediate)
  - Tegra

Big families:
1) TBDR: Render everything in tiles, shade only visible pixels
2) Tiled: Render everything in tiles
3) Classic: Render everything



Hidden Surface Removal

- Deferred
    - bins all primitives into tiles (~16x16 pixels)
    - uses micro Z-buffer to reject hidden fragments
    - gathers visible fragments and sends for rasterization
- Early Z-cull
    - coarse Z-buffer: low-resolution, low-precision
    - if fragment fail Z-test it will not be sent for rasterization
    - sort opaque front to back, GPU will **not** rearrange polygons for you

TBDR: while rendering a tile, a lot of relevant data is kept in fast on-chip memory. Additionally, visible fragments are found by the hardware and pixel shaders (and consequently all texture fetches) are ran only on them.

Others (Tiled & Classic) use “early z-cull” or similar schemes to help reject some of the hidden pixels.

Case: HSR Performance

- 0.8-3.5ms to reject 1280x600 fragments on Tegra2
    - can reject 2x2 pixels per cycle
    - real scene: rejecting skybox in SHADOWGUN
        - 0.9ms
- 0.05-0.13ms to reject 1024x768 fragments on iPad2
    - can reject 32 pixels per cycle
    - 0.05ms coplanar geometry
    - 0.11-0.13ms rejecting arbitrary geometry

If we render a screen worth of pixels that are fully occluded by things rendered before, in both TBDR and Early-Z it still takes GPU time to process them (much less time than it would take to fully shade them of course).

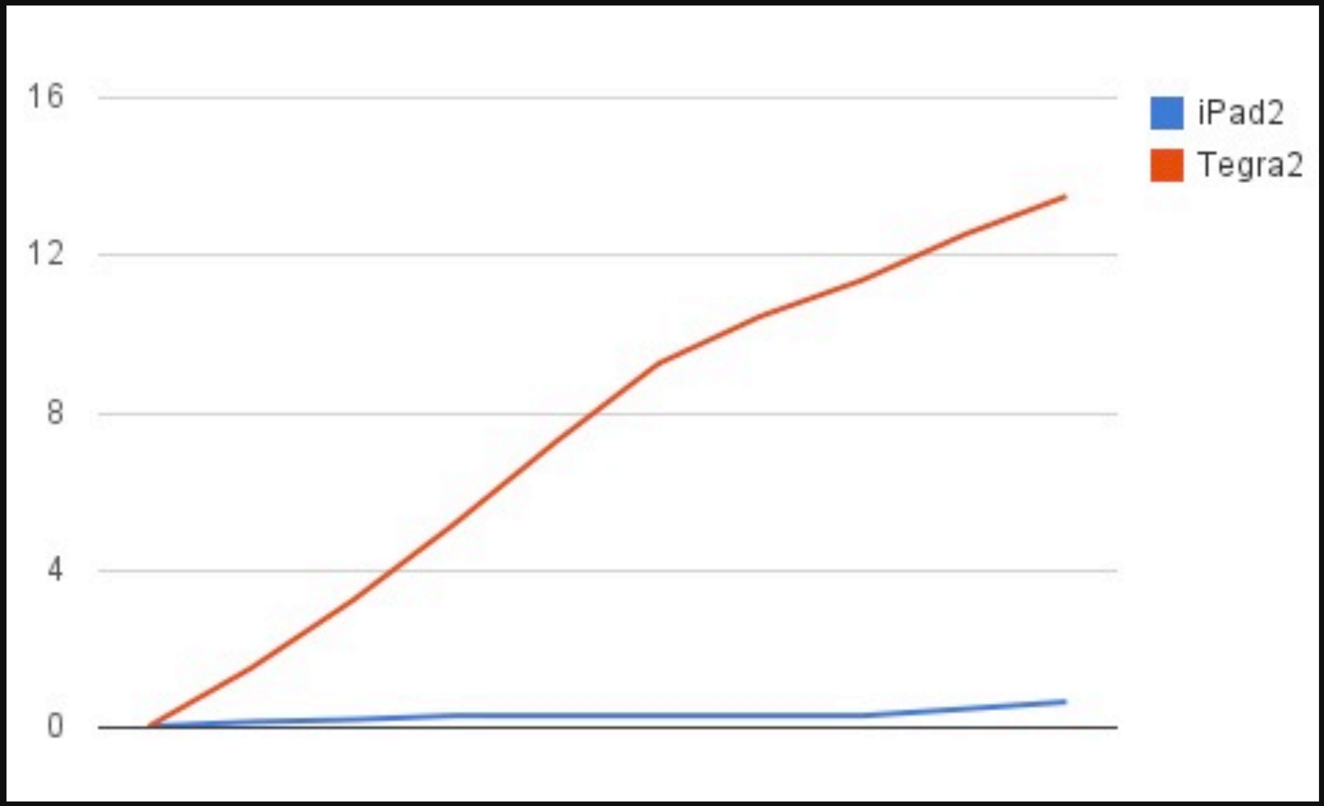
In iPad2 vs. Tegra2 case (NOTE: these GPUs are not equivalent in performance in general), we can see that iPad2 (PowerVR SGX, TBDR architecture) takes way less time to reject hidden pixels than Tegra2 (Classic architecture).

Case: HSR Performance

- Insight: it makes sense to spend lots of CPU cycles on Tegra to do occlusion culling and better sorting (to take advantage of Early Z-cull)
  - 0.8-3.5ms to reject 1280x600 fragments on Tegra2 GPU
  - 0.05-0.13ms to reject 1024x768 fragments on iPad2 GPU

... and going forward you will have more CPU cores to get busy

Hidden Surface Removal

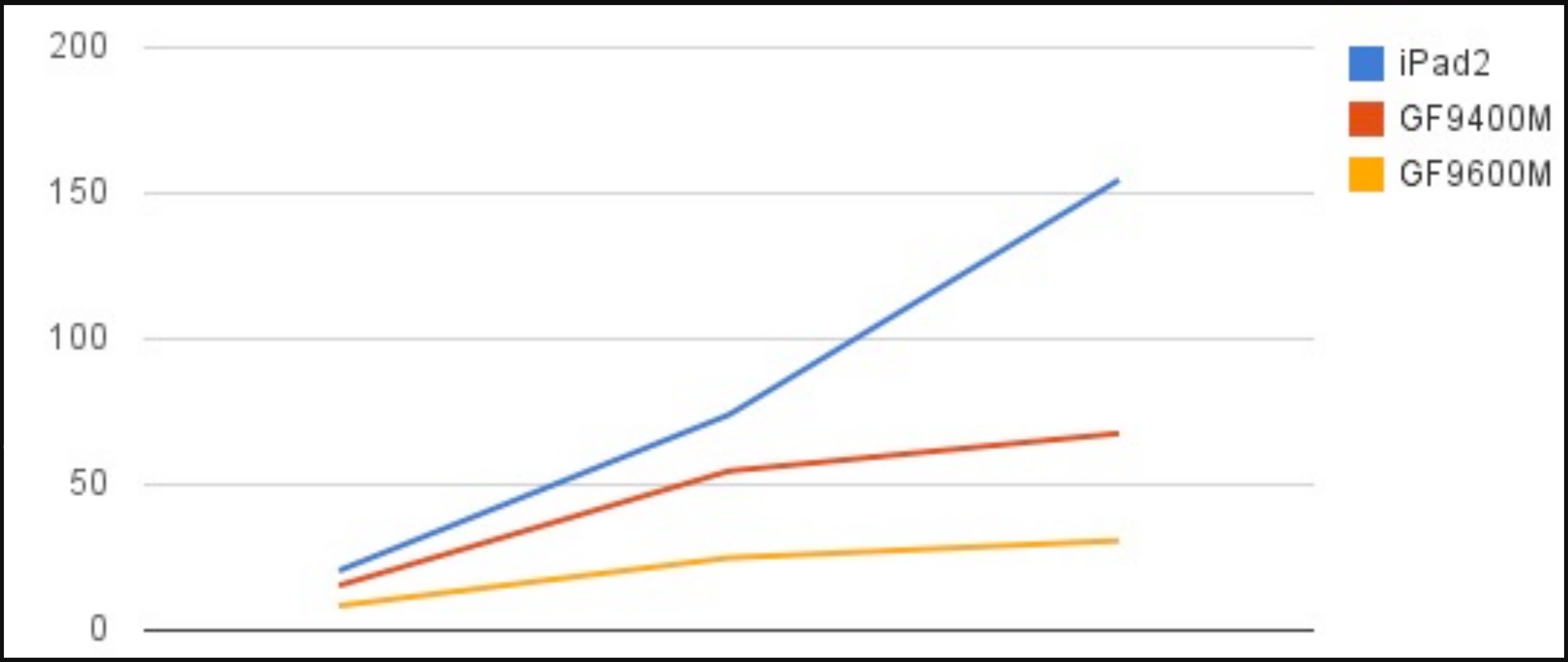


- Simple geometry
  - iPad2: 0.07ms, Tegra2: 2.0ms
- Complex geometry
  - iPad2: 0.13ms, Tegra2: 3.8ms
- Scales linearly with overdraw

Graph: overdraw on horizontal axis (1 to 10), time on vertical axis.

Hidden Surface Removal

- iPad2 vs. GeForce in MacBookPro ;)
- iPad2 only 2-4x slower at rejection!



PowerVR SGX in particular seems to be really fast at rejecting occluded pixels. Here we compare it to GeForce 9600M GT and 9400M found in laptops; the mobile part is only 2-4x slower at rejection (even if in many other characteristics something like 10x slower).

Graph: overdraw on horizontal axis (1 to 1000), time on vertical axis.

Saving BW & ALU

- Framebuffer writes
  - Tiled & Tiled Deferred: great!
  - Classic: not so much
- ALU & Texturing per fragment
  - Deferred will save on all hidden fragments
  - Early Z-Cull saves on coarse blocks
    - Depends on submission order

TBDR and Tiled architectures process the screen in tiles. While a tile is processed, the framebuffer and depth buffer data is kept in an on-chip fast memory. This saves a lot of bandwidth (and consequently energy), however it has downsides as well (tiling process is not free, etc.).

Saving Texture Cache

- Deferred
  - can prefetch texture **before** pixel fragment starts
  - mipmaps not **that** critical
- Small tiles
  - help hit the cache
  - good for anisotropic
- Non-Tiled
  - internal swizzling and 2x2 blocks help
- Anyway
  - Mipmaps good, aniso costs, texture compression good!

PowerVR SGX can prefetch texture for visible fragments before submitting them to rasterizer module (TSP), if texcoords are coming from XY(Z) components of the interpolant (varying) and no mipbias is specified.

Tiled Framebuffers

- Higher Z precision possible
  - SGX, 32 bit internally
- Better quality 16 bit image
  - 32 bit on tile, resolves to 16 bit
- MSAA much cheaper
  - Tiles become smaller
  - No extra memory cost

SHADOWGUN Ingame screenshot

ShadowGun combines tactical combat with 3rd person harrowing action, ushering in a new standard for handheld action-gaming.

SHADOWGUN Ingame screenshot

ShadowGun combines tactical combat with 3rd person harrowing action, ushering in a new standard for handheld action-gaming.

Case: MSAA & Aniso

unity

SIGGRAPH2011 VANCOUVER

Case study, how much MSAA and Anisotropic filtering costs for real relatively complex scenes on mobile GPUs.

The scenes are from SHADOWGUN by MADFINGER games, and unannounced title by Luma Arcade.

Case: MSAA

- 4xMSAA, iPad2
  - 6ms in <game2>
  - ~2ms in SHADOWGUN
- Still **not** free!
  - More tiles to process
  - More per-sample work on polygon edges

4xMSAA = 6ms in <game2> and around 2ms in SHADOWGUN. The cost is more in <game2> because it uses post-processing and there are some not-so-fast paths involved in that case. Still, usable in practice!

Even if there’s no extra memory cost for MSAA in TBDR architecture, it does not come for free. The tiles become smaller, so more of them have to be processed. Additional work has to be done by the GPU on polygon edges, etc.

Case: Aniso

- Everything aniso, real scene
  - Costs 3ms on iPad2
    - Max. aniso = 2
  - Tegra supports higher aniso, but more expensive - be careful!
- Both SHADOWGUN and <game2>

Case: Mipmaps

- Turn off mipmaps...
  - "The horror! The horror!"
  - Meanwhile...
- Only 2-3ms cost on iPad2
- But completely destroys Tegra

A peculiar thing; turning off mipmaps completely (on a decent drawing distance and high resolution textures) did not destroy performance on iPad2!

Tiled Deferred Caveats

- Scene splits
- More transistors for smarts
  - Less “raw power”
- No GPU time per draw call
  - Easier to profile on non-TBDR: Tegra or Adreno

So it seems that TBDR architecture is all awesome, but of course it has some drawbacks as well.

If vertex data per frame (number of vertices * storage required after vertex shader) exceeds the internal buffers allocated by the driver, the scene has to be “split” which costs performance. The driver might allocate a larger buffer after this point, or you might need to reduce your vertex count. We ran into this on iPad2 (iOS 4.3) at around 100 thousand vertices with quite complex shaders.

TBDR needs more transistors allocated for the tiling & deferred parts, leaving conceptually less transistors for “raw performance”.

It’s very hard (i.e. practically impossible) to get GPU timing for a draw call on TBDR, making profiling hard. Often easier to profile on Tegra or Adreno.

Case: Optimizing SHADOWGUN for Tegra 2

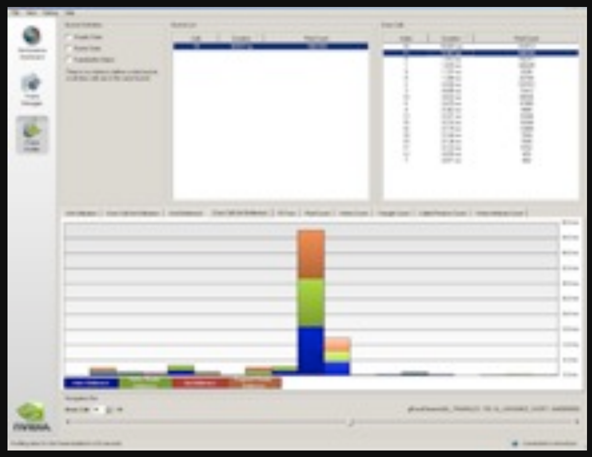
Case study: optimizing SHADOWGUN

SHADOWGUN Ingame screenshot
ShadowGun combines tactical combat with 3rd person harrowing action, ushering in a new standard for handheld action-gaming.
MADFINGER GAMES.COM

SHADOWGUN

SHADOWGUN+Tegra2

- Target 30 FPS
- Use artwork!
  - to avoid expensive full-screen effects
  - to avoid expensive shaders
  - to reduce overdraw on alpha blended
- Use PerfHUD ES for profiling



Even if the game looks complex, it actually is not. A lot of “smarts” are put into artwork instead.

NVIDIA’s PerfHUD ES was used for profiling.

Sort opaque geometry!

- Ideal front-to-back not possible
  - Would need per-poly sort
  - Different materials
- Big | Close: front to back
  - Rest by material
- Character shader expensive
  - Player big, occludes a lot: always first
  - Enemies often occluded by cover: always last
- Skybox always last
  - Turn skybox off on “no way it’s visible” trigger zones

Big or close (~10 meters from camera) objects were sorted front to back. Further objects were sorted by material to improve batching and reduce state changes.

Although main character shader is more expensive than the scene, _but_ it always occludes a large part of screen: always draw the character first (set smaller “render queue” in Unity)

Enemies also use expensive shader, but very often are occluded by walls or cover: draw them after regular opaque objects (set larger “render queue” in Unity).

Skybox always drawn after all opaque geometry (this is always done in Unity). Additionally, since rejecting fully occluded geometry still costs ~1ms on Tegra2, special trigger zones were placed that would turn skybox off when it’s surely not visible.

Sort opaque geometry!

- Saved ~15ms per frame

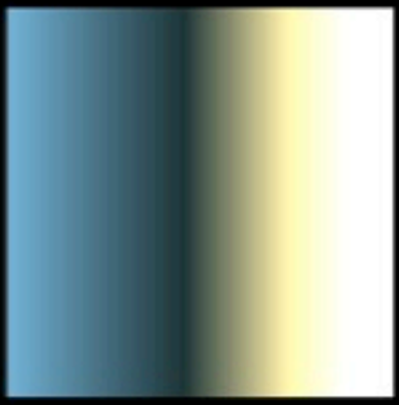
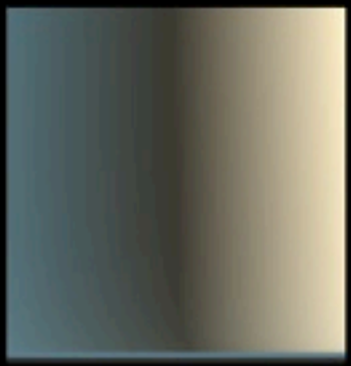
All this sorting saved ~15 milliseconds per frame!

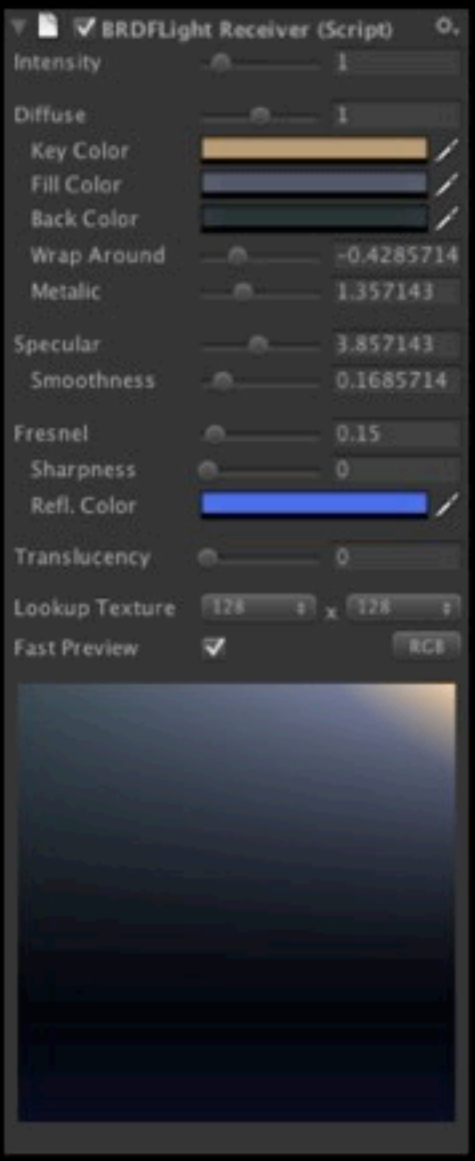
Optimizing shaders

- Character shader, 10 cycles/pixel
  - Normal mapped
  - Specular, wrap around diffuse
  - Per-vertex from light probes
- But how?
  - Lighting into LUT texture
  - LUT size tradeoff performance/quality

Character shader takes 10 cycles/pixel on Tegra2. Per pixel normal mapping, specular, wrap around diffuse; per vertex lighting from light probes.

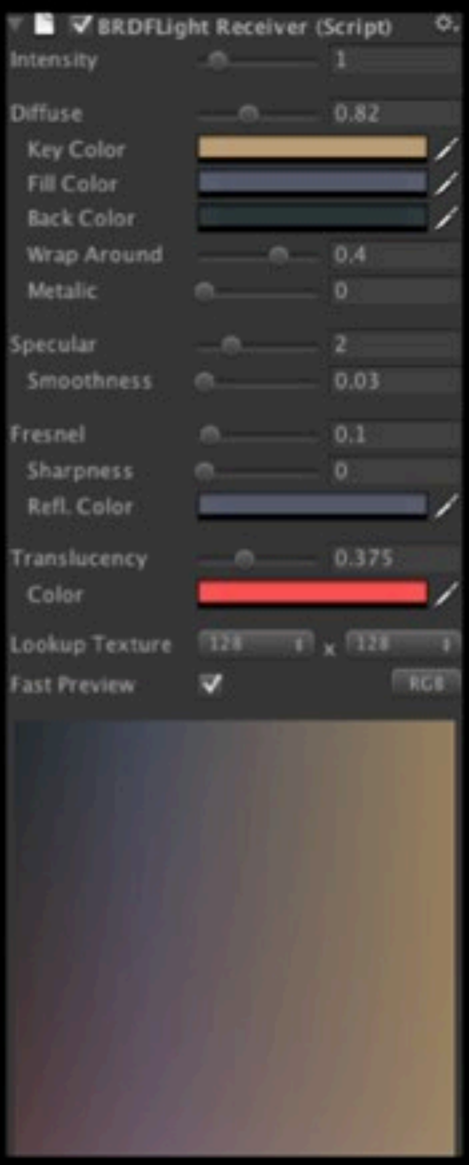
This is achieved by putting all lighting calculations into a lookup texture. Nice bonus: changing LUT size can trade performance for quality (e.g. use larger LUT on main character to get better specular; smaller LUT on enemies for better performance).



BRDFLight Receiver (Script)

Intensity	1
Diffuse	1
Key Color	🟫
Fill Color	🟦
Back Color	⬛
Wrap Around	-0.4285714
Metalic	1.357143
Specular	3.857143
Smoothness	0.1685714
Fresnel	0.15
Sharpness	0
Refl. Color	🟦
Translucency	0
Lookup Texture	128 x 128
Fast Preview	✓



BRDFLight Receiver (Script)

Intensity	1
Diffuse	0.82
Key Color	🟫
Fill Color	🟦
Back Color	⬛
Wrap Around	0.4
Metalic	0
Specular	2
Smoothness	0.03
Fresnel	0.1
Sharpness	0
Refl. Color	🟦
Translucency	0.375
Color	🟥
Lookup Texture	128 x 128
Fast Preview	✓

LUT Baking Tools

Another bonus: since all lighting is in a lookup texture, we can have much better lighting model than standard Blinn-Phong for the same cost.

Give artists tools to tweak this lighting model! Here, it's a tri-light with adjustable wrap around, energy conserving specular & hacky-translucency.

LUT lighting

- I = diffuse * N.L + spec * N.H ^ n
  - ^n is expensive!
- I = BRDF(N, L, V)
- I = tex2d (N.L, N.H)
- Energy conserving Blinn-Phong
- Can approximate unusual BRDFs

This is an old approach actually; lookup lighting by <N.L, N.H> in the texture. It does not allow fully generic BRDFs to be put into the texture, but Blinn-Phong and friends can.

We do lookup based on <N.L*0.5+0.5, N.H> which allows wrap-around lighting and essentially emulating two opposite light colors, with arbitrary equatorial light (aka tri-light model).

For specular, we can do energy conservation as well. Specular power needs to be fixed however.

Texture Compression

- Tegra has DXT5, use it!
- 2x smaller than RGBA16
- 4x smaller than RGBA32

Optimizing for Tegra means: use DXT texture compression!

SHADOWGUN

- All that runs at 60FPS on iPad2 BTW

Tools

Profiling tools

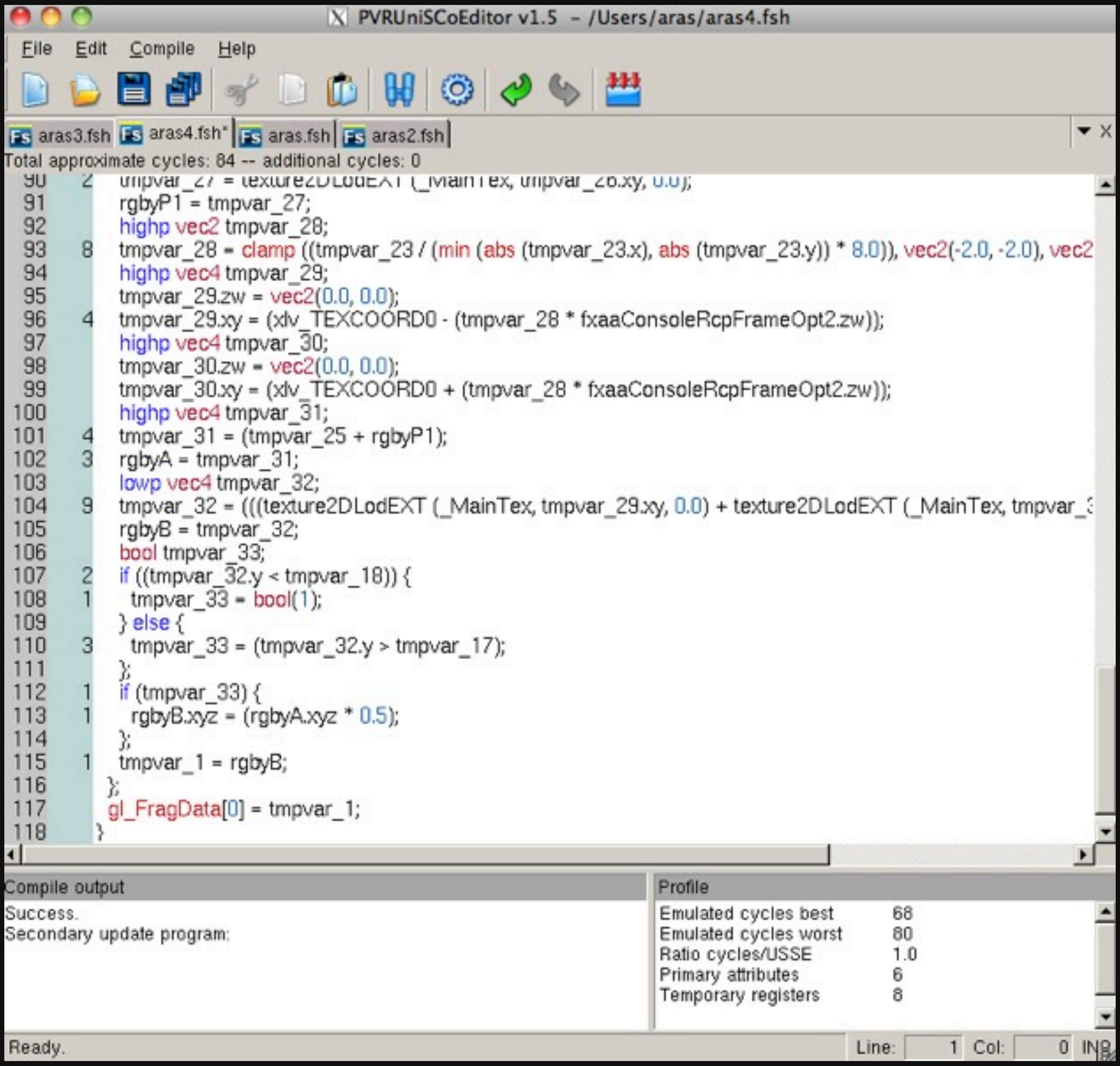
Tools: iOS + PowerVR

- iOS: Unity's internal profiler
  - GPU time for the whole scene
- Apple's Instruments / Detective
  - How busy is GPU in %
- PowerVR's PVRUniSCo shader analyzer
  - Take with a grain of salt
  - No SGX543 profile

PowerVR is tile based deferred renderer, so it's impossible to get GPU timings per draw call. However you can get GPU times for the whole scene using Unity's built-in profiler (the one that prints results to Xcode output).

Apple's tools currently can only tell you how busy GPU and it's parts are, but do not give times in milliseconds.

For shader profiling, PVRUniSCo can be used, but it does not have GPU profile for SGX543 (iPad2), and it won't match what Apple's drivers are doing exactly anyway. Still, a good ballpark measure.



```
Total approximate cycles: 84 -- additional cycles: 0
  2   tmpvar_27 = texture2DLodEXT (_MainTex, tmpvar_26.xy, 0.0);
      rgbyP1 = tmpvar_27;
      highp vec2 tmpvar_28;
  8   tmpvar_28 = clamp ((tmpvar_23 / (min (abs (tmpvar_23.x), abs (tmpvar_23.y)) * 8.0)), vec2(-2.0, -2.0), vec2
      highp vec4 tmpvar_29;
      tmpvar_29.zw = vec2(0.0, 0.0);
  4   tmpvar_29.xy = (xlv_TEXCOORD0 - (tmpvar_28 * fxaaConsoleRcpFrameOpt2.zw));
      highp vec4 tmpvar_30;
      tmpvar_30.zw = vec2(0.0, 0.0);
      tmpvar_30.xy = (xlv_TEXCOORD0 + (tmpvar_28 * fxaaConsoleRcpFrameOpt2.zw));
      highp vec4 tmpvar_31;
  4   tmpvar_31 = (tmpvar_25 + rgbyP1);
  3   rgbyA = tmpvar_31;
      lowp vec4 tmpvar_32;
  9   tmpvar_32 = (((texture2DLodEXT (_MainTex, tmpvar_29.xy, 0.0) + texture2DLodEXT (_MainTex, tmpvar_
      rgbyB = tmpvar_32;
      bool tmpvar_33;
  2   if ((tmpvar_32.y < tmpvar_18)) {
  1     tmpvar_33 = bool(1);
      } else {
  3     tmpvar_33 = (tmpvar_32.y > tmpvar_17);
      };
  1   if (tmpvar_33) {
  1     rgbyB.xyz = (rgbyA.xyz * 0.5);
      };
  1   tmpvar_1 = rgbyB;
    };
    gl_FragData[0] = tmpvar_1;
  }
```

```
Compile output
Success.
Secondary update program:
```

```
Profile
Emulated cycles best       68
Emulated cycles worst      80
Ratio cycles/USSE          1.0
Primary attributes         6
Temporary registers        8
```

```
Ready.                                         Line:    1  Col:    0  INS
```

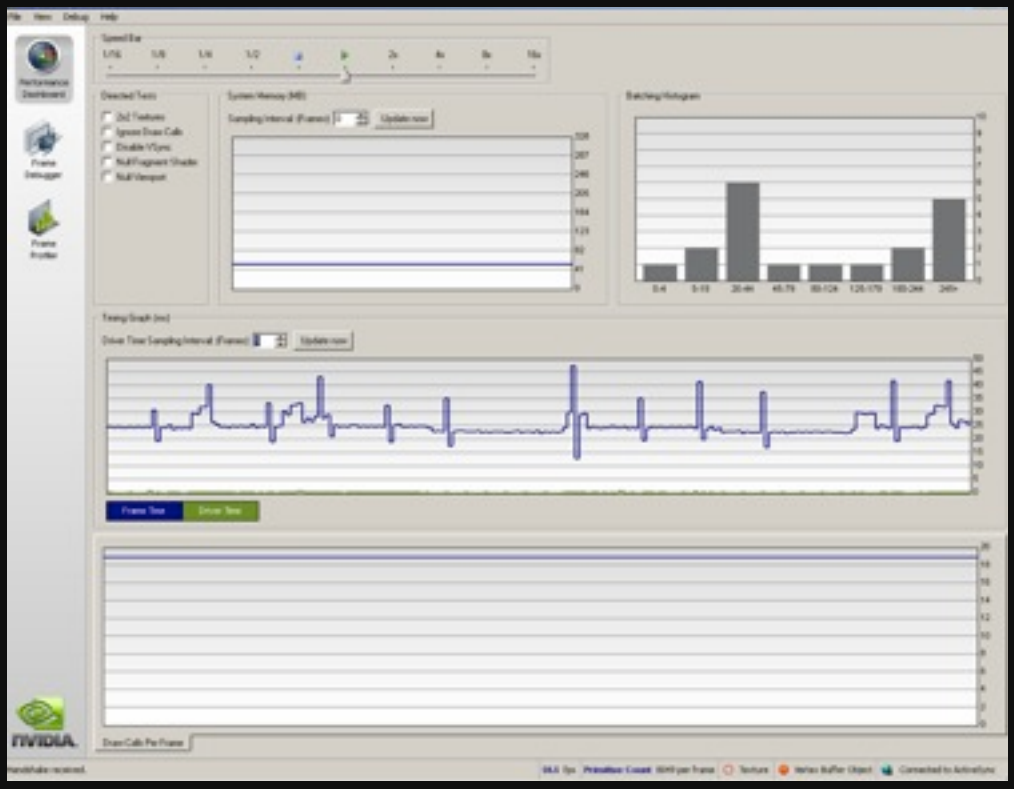
PVRUniSCo

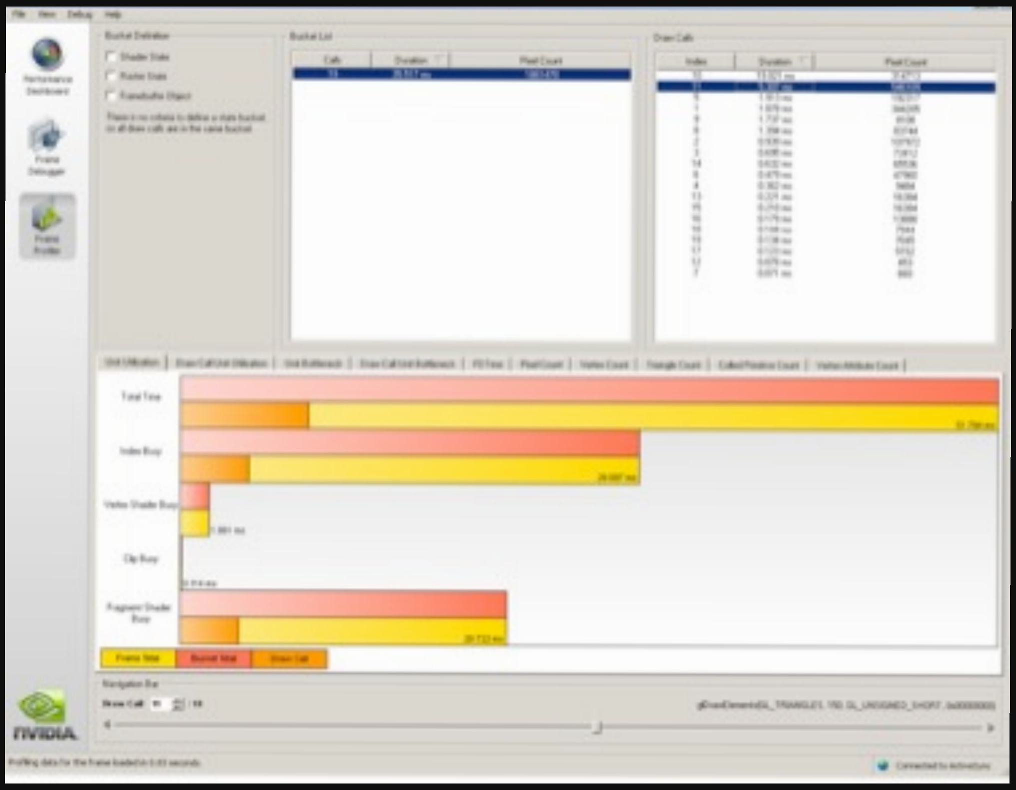
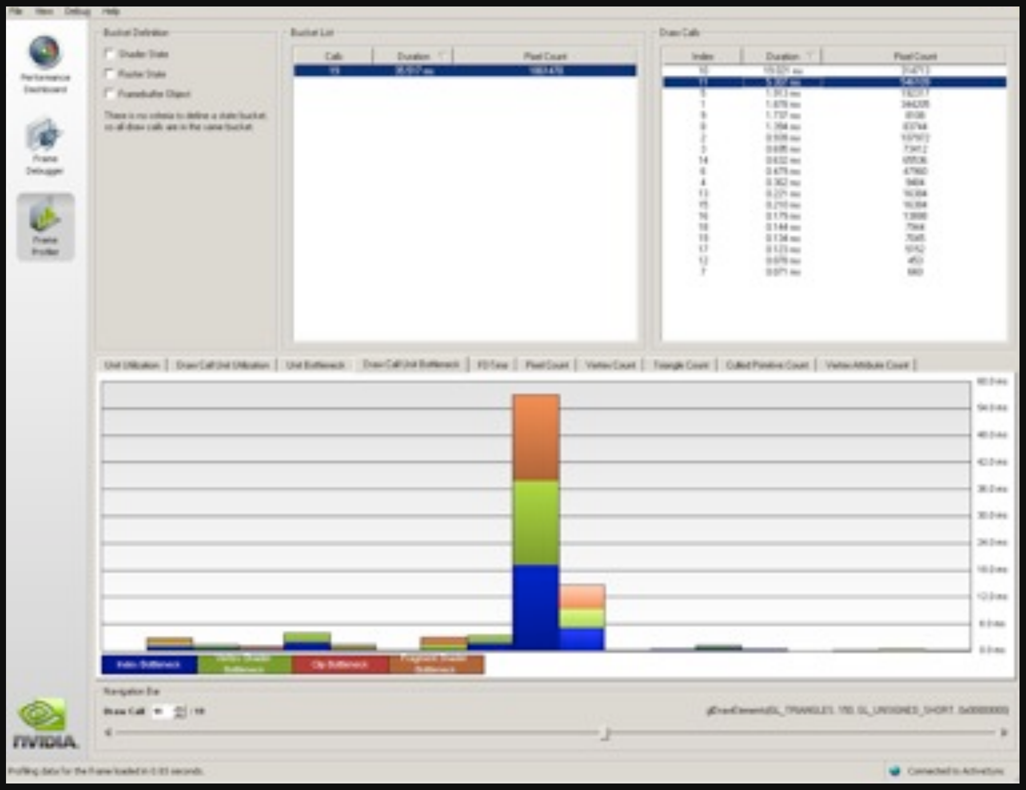
PVRUniSCo gives cycles for the whole shader, and approximate cycles for each line in the shader code. Windows & Mac!

Tools: Android + Tegra

- NVIDIA PerfHUD ES
  - GPU time per draw call
  - Cycles per shader
  - Force 2x2 texture
  - Null view rectangle
- Runs on Windows, OSX, Linux
- Very useful!

On Tegra, NVIDIA provides excellent performance tools which does everything you want. And it does not require using Windows either!



PerfHUD ES

PerfHUD ES

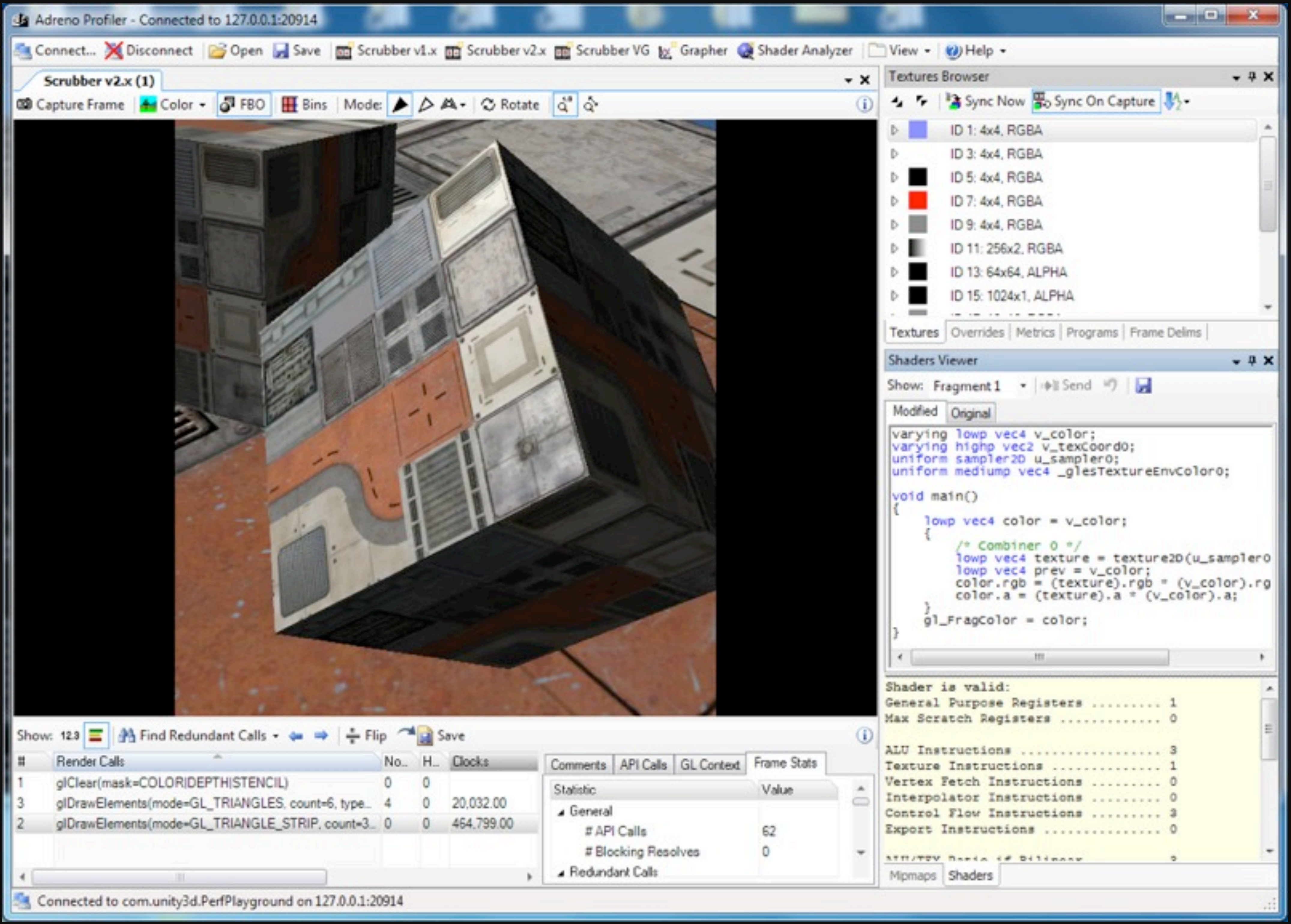
- Works with NVIDIA Development Boards (Ventana)
  - You can get it running on some consumer level Tegra devices but...
  - **Bricked** Motorola Atrix 4G just yesterday!

PerfHUD ES downside: does not easily work with consumer devices; you need the development board from NVIDIA :(

Tools: Android + Adreno

- Adreno Profiler (Windows only)
  - Timeline graphs, frame capture
  - Frame debug, API calls
  - Shader analyzer, live editing
- Works with consumer devices
- Nice!

Qualcomm provides excellent profiler as well (Windows only however). Even works with consumer devices!


Adreno Profiler - Connected to 127.0.0.1:20914
Connect... Disconnect Open Save Scrubber v1.x Scrubber v2.x Scrubber VG Grapher Shader Analyzer View Help
Scrubber v2.x (1)
Capture Frame Color FBO Bins Mode: Rotate
Textures Browser
Sync Now Sync On Capture
ID 1: 4x4, RGBA
ID 3: 4x4, RGBA
ID 5: 4x4, RGBA
ID 7: 4x4, RGBA
ID 9: 4x4, RGBA
ID 11: 256x2, RGBA
ID 13: 64x64, ALPHA
ID 15: 1024x1, ALPHA
Textures Overrides Metrics Programs Frame Delims
Shaders Viewer
Show: Fragment 1 Send
Modified Original
Show: 12.3 Find Redundant Calls Flip Save
Comments API Calls GL Context Frame Stats
Mipmaps Shaders

```
varying lowp vec4 v_color;
varying highp vec2 v_texCoord0;
uniform sampler2D u_sampler0;
uniform mediump vec4 _glesTextureEnvColor0;

void main()
{
     lowp vec4 color = v_color;
     {
          /* Combiner 0 */
          lowp vec4 texture = texture2D(u_sampler0
          lowp vec4 prev = v_color;
          color.rgb = (texture).rgb * (v_color).rg
          color.a = (texture).a * (v_color).a;
     }
     gl_FragColor = color;
}
```

```
Shader is valid:
General Purpose Registers ......... 1
Max Scratch Registers ............. 0

ALU Instructions .................. 3
Texture Instructions .............. 1
Vertex Fetch Instructions ......... 0
Interpolator Instructions ......... 0
Control Flow Instructions ......... 3
Export Instructions ............... 0
```

#	Render Calls	No...	H...	Clocks
1	glClear(mask=COLOR|DEPTH|STENCIL)	0	0	
3	glDrawElements(mode=GL_TRIANGLES, count=6, type...	4	0	20,032.00
2	glDrawElements(mode=GL_TRIANGLE_STRIP, count=3...	0	0	464,799.00

Statistic	Value
General	
# API Calls	62
# Blocking Resolves	0
Redundant Calls	

Connected to com.unity3d.PerfPlayground on 127.0.0.1:20914

Adreno Profiler

Shader Performance

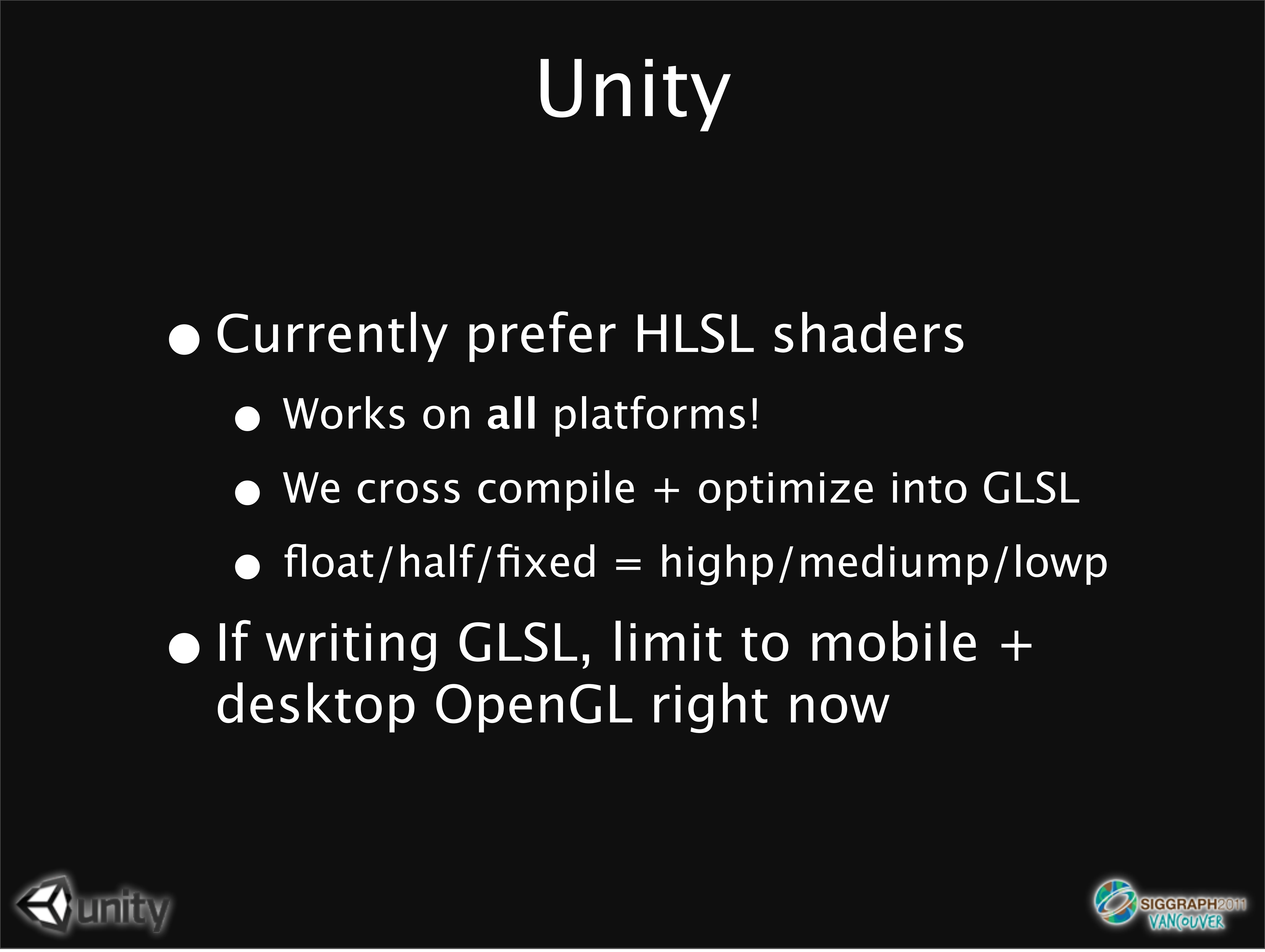
Shaders on mobile

- Most platforms OpenGL ES 2.0
  - Except Vita, 3DS, WP7
- GLSL ES shading language
  - Similar to HLSL, but different ;)

Most platforms we (or Unity) cares about are OpenGL ES 2.0 based, so let’s talk about that.

GLSL ES specific

- Precision qualifiers
  - Cg's float/half/fixed from GeForce FX days? Same stuff.
  - highp/mediump/lowp



Unity

- Currently prefer HLSL shaders
  - Works on **all** platforms!
  - We cross compile + optimize into GLSL
  - float/half/fixed = highp/mediump/lowp
- If writing GLSL, limit to mobile + desktop OpenGL right now

In Unity, shaders most often written in HLSL (aka Cg); we cross compile into GLSL ES for mobile platforms.

When you use float/half/fixed types in HLSL, they end up highp/mediump/lowp precision qualifiers in GLSL ES.

We find no performance penalty for the cross-compilation step. If you find any shaders that are suboptimal after HLSL->GLSL conversion, please let us know!

You can also write GLSL directly if you want to, but doing that limits you to OpenGL-like platforms (e.g. mobile + Mac) since we currently do not have GLSL->HLSL translation tools.

General Intuition

- Note: Hand-wavy
- Note: Can be GPU specific
- Profile, measure, don't trust!
    - Often shader optimization highly non intuitive
        - Especially when no shader profilers <cough> iOS </cough>

Some "general intuition" rules that however should not be applied blindly...

Precision: fixed / lowp

- -2.0 .. 2.0, 8 bit fixed point
- Colors, some normalized vectors
- Avoid swizzles and scalar lowp

lowp good for most colors and some normalized vectors.
Note that at least -2.0 .. 2.0 range is guaranteed, but clamping to +-2 may or might not happen depending on the GPU and the driver (i.e. GPU might have much larger range for lowp).
Avoid lowp if it will need fancy swizzles or be involved in purely scalar computations.

Precision: half / mediump

- 16 bit float; like DX9 HLSL’s half
- Most UVs, 2D vectors, general “no high precision needed” use

mediump just like “half” in DX9. This is good for most general stuff that does not explicitly need higher precision.

Precision: float / highp

- 24-32 bit float
- World coordinates, scalars, UVs with large textures/tiling or strong indirect offsets

highp is more like the regular floating point (which can be lower precision than full 32 bits though). Use when mediump is not enough.

Precision

- Minimize conversion between precisions
  - Can cost a lot!
- Think 4x, 2x, 1x performance for lowp, mediump, highp
  - But some things cost, e.g. swizzles on lowp
- GPU specific
  - Qualcomm Adreno much less sensitive to precision

Varyings / Interpolants

- Pass data from vertex to fragment
- Most GPUs: minimize count & pack
  - PowerVR, not so much!
  - Usually **much** cheaper to use more varyings if it helps avoiding indirect texture reads
  - Likewise, do not pack 2 UVs into one float4/vec4 varying for PowerVR

On PowerVR, varyings are very cheap (NOT like almost all other GPUs). Using more varyings can be a big win there if it helps avoid doing some math in the shader.

One particular case: texture2D(...,var.xy) is good on PVR; texture2D(...,var.zw) is bad – the GPU treats it as a dependent texture read which can't be prefetched. So do NOT pack two UVs into one vec4 varying if you care about PowerVR performance!

Shader Performance

- Some shaders
  - color
  - texture
  - texture * color
  - diffuse normal map w/ dir. light
    - no light/material colors; light color; light+mat colors
  - specular Blinn-Phong w/ dir. light
    - approx. per vertex specular; full per pixel

Performance experiments with some shaders, from the simplest possible one to relatively complex for mobile (fully per-pixel Blinn-Phong)

Fill screen

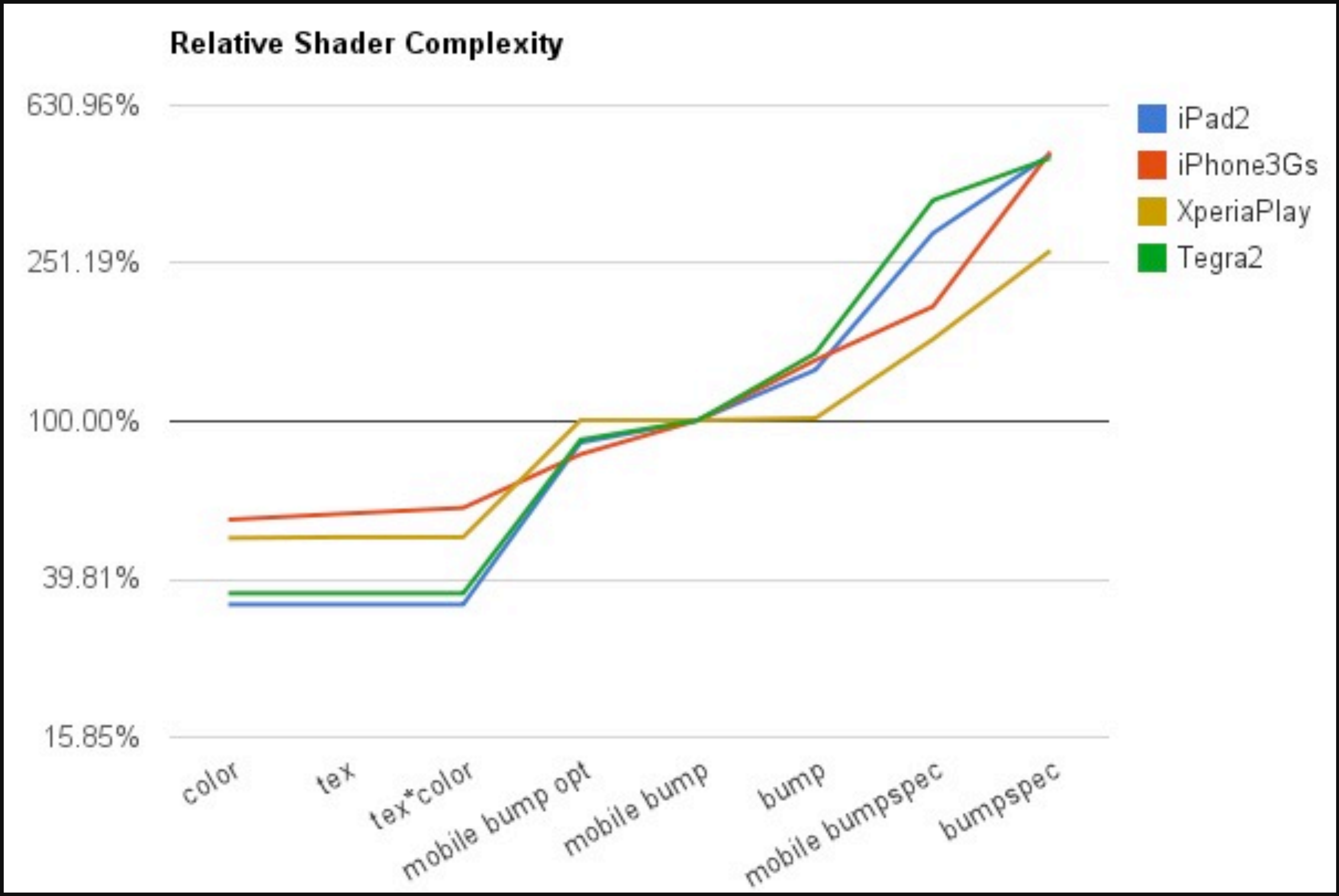
Draw a full screen worth of pixels, measure how long it takes

Shader Performance

- Normalized to iPad2 resolution
- From single color:
  - 1.4ms iPad2
  - 3.5ms XperiaPlay
  - 3.8ms Tegra2
  - 14.3ms iPhone3Gs
- To fully per-pixel bump spec:
  - 19.3ms iPad2
  - 18.4ms XperiaPlay
  - 47.7ms Tegra2
  - 122.4ms iPhone3Gs

If we'd normalize results to resolution of iPad2 (1024x768), these are the relative GPU speeds from the simplest possible shader to the fully per-pixel Blinn-Phong.

From these particular devices, iPad2 (SGX543) is the fastest, followed by XperiaPlay (Adreno205), then by Tegra2, and iPhone3Gs (SGX535) is the slowest.

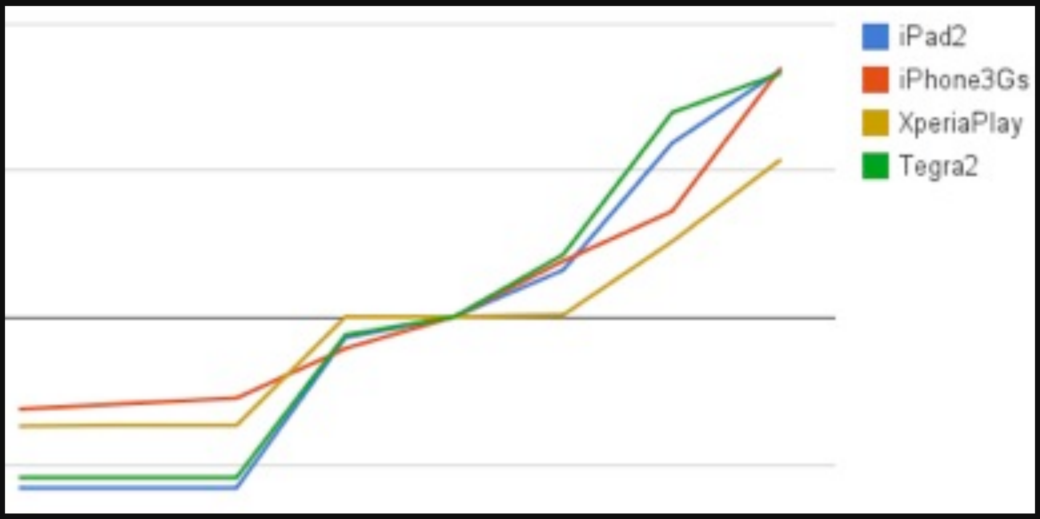


Relative to simple bump

Normalized graph for each GPU, relative to “mobile bump” shader; log vertical scale.

Relative Performance

- All lines increasing
  - Cheaper is cheaper everywhere
- XperiaPlay (Adreno) much less sensitive to shader complexity
  - Could be limited elsewhere
  - Or GPU has many ops “for free”
- Benefits differ across GPUs!



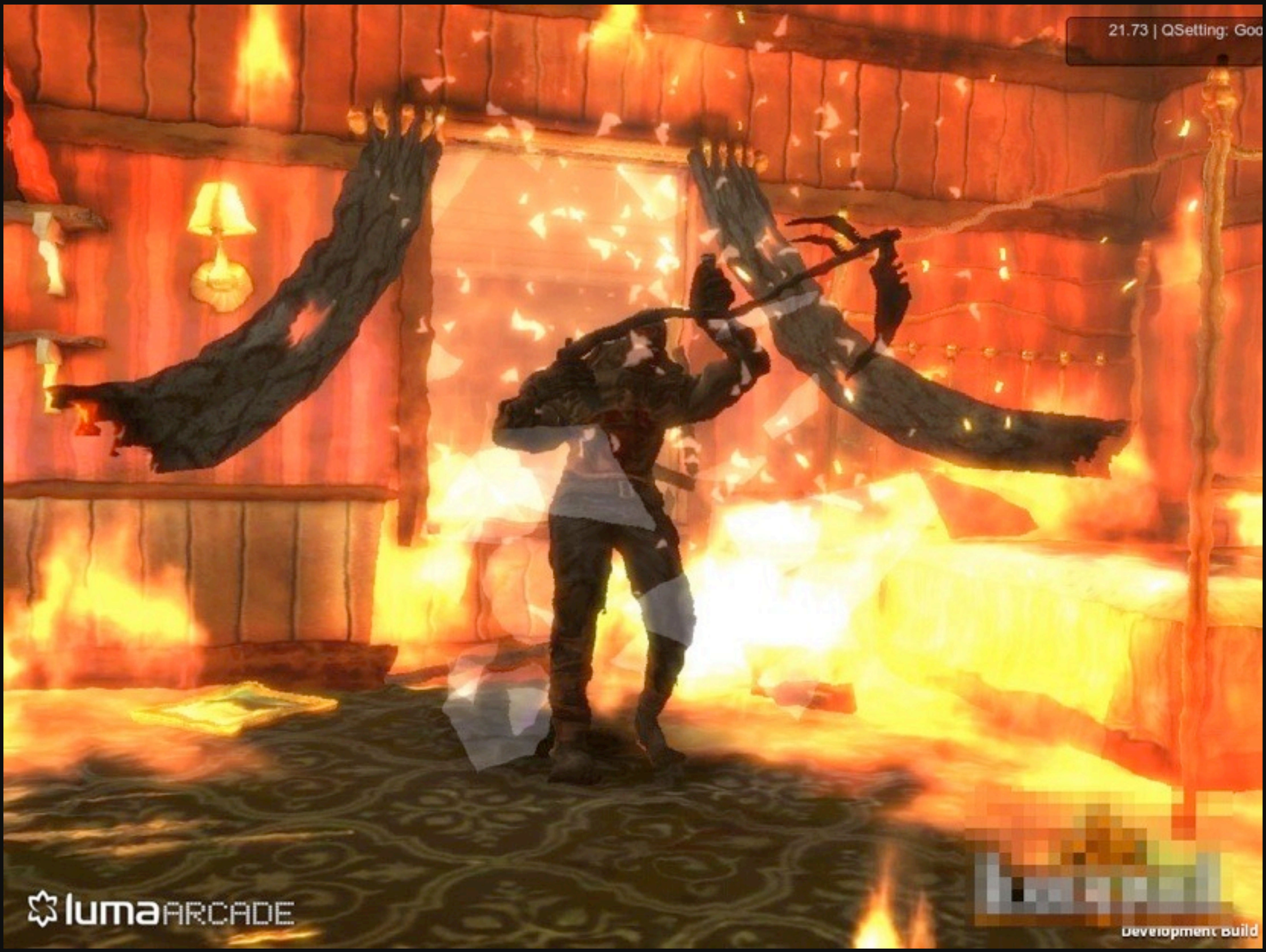
What can be seen:

Cheaper shaders are cheaper across all GPUs (not really surprising)

Adreno GPU is much less sensitive to the shader complexity; it could be that it’s limited elsewhere or has quite strong ALU units and/or has many operations which are essentially “free”

Case: Optimizing <game2> for iPad2

Case study: optimizing <game2> for iPad2



Unannouced Title

High poly, lots of particles, complex shaders, postprocessing.

In the beginning...

- 18 FPS, GPU bound. Ouch!
- Hard to profile GPU on iOS

Current state

- 37FPS after optimizations
- Now runs 26 FPS with MORE stuff!
    - 90k vertices (lots skinned)
    - A ton of particles
    - 3-6 textures per fragment
    - Bloom, Heat Shimmer
    - 4xMSAA, Anisotropic

After some optimizations, went 18->37 FPS on iPad2. So got some room for more stuff (4xMSAA, Anisotropic filtering). After adding that runs at 26FPS right now, but we're sure it can reach 30.

GPU profiling on iOS

- Tools
  - XCode Instruments:
    - Device %
    - Tiler %
    - Rasterizer %
    - Split count
  - Unity: GPU time per frame

Getting GPU time on iOS

- Force CPU to wait for GPU
  - glFinish() or glReadPixels(...)
  - after presentRenderbuffer(...)
- Wait only every 3rd or 4th frame
  - to keep submission of GL ES commands run in parallel with GPU
- Can measure 1st draw call (glClear)

How to get GPU time per frame on iOS (this is what Unity's internal Xcode profiler does). Basically we need to somehow force the GPU to finish rendering; either with glFinish or by reading a pixel from the back buffer.

Post-processing

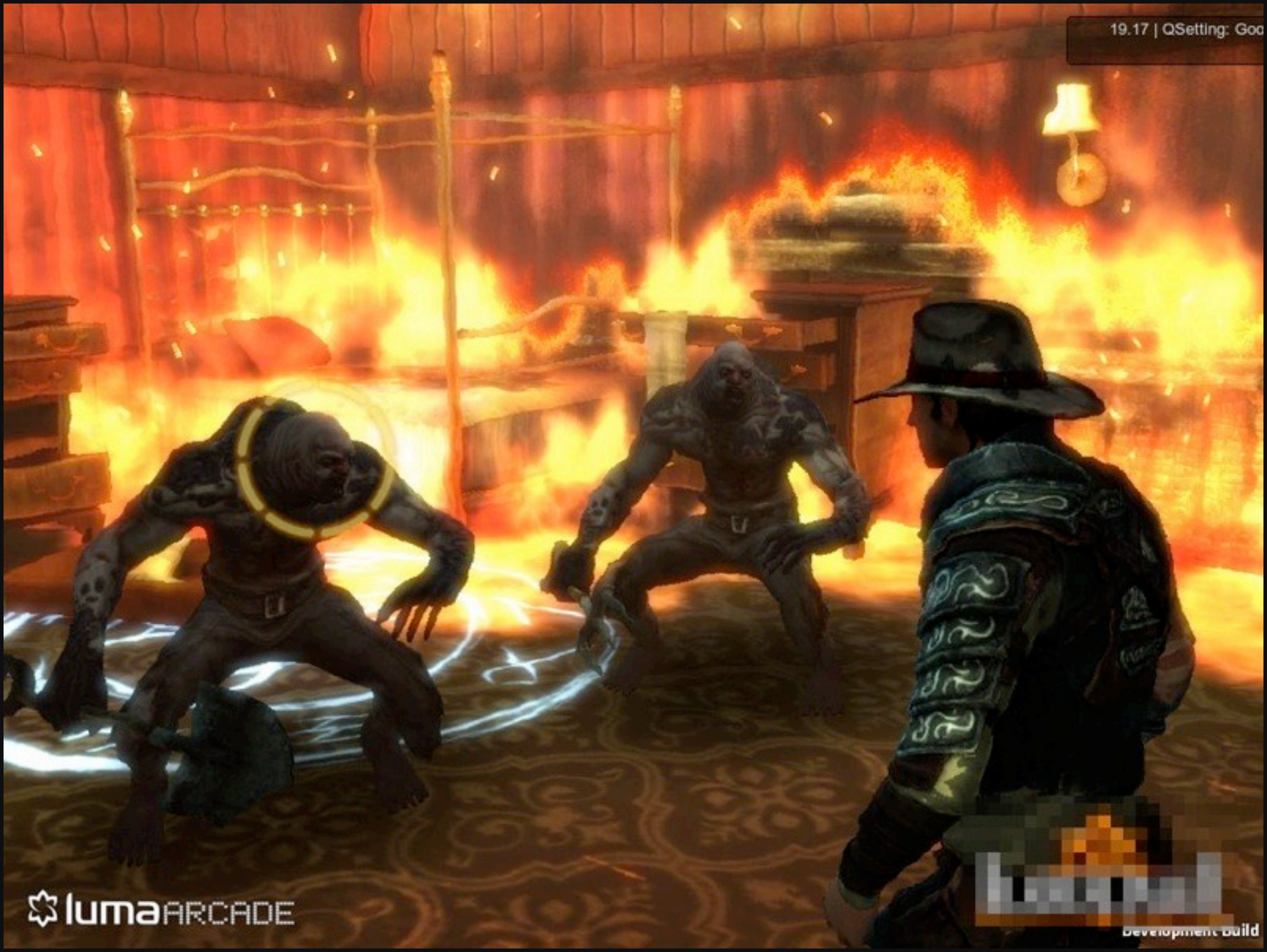
- Bloom and Heat Shimmer
- Took almost 1/3rd of the frame

Initially Bloom and Heat Shimmer post-fx took around 16ms

Post-processing

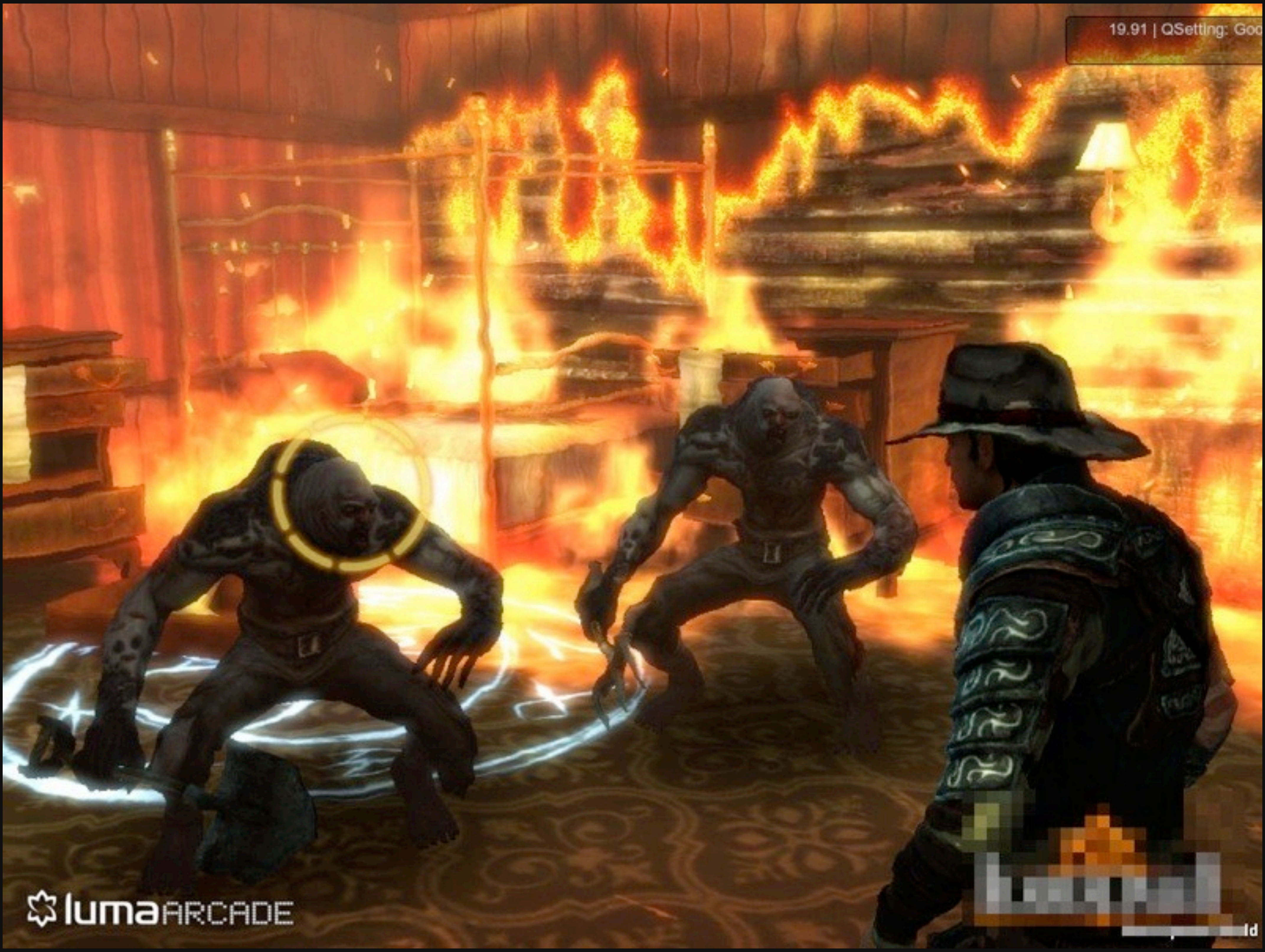
- Precision: lowp/fixed for everything
  - except texcoords for indirect texture fetches - use mediump/half
- Combine Bloom + Heat into one
  - Final fullscreen composite pass most expensive
- Saved ~10ms per frame

Some easy optimizations; runs at ~6ms



Burning Walls shader

Almost all objects have this “burning” shader that has fire-y thing changing over time.



Burning Walls shader

Burning Walls shader

- Texture or ALU bound?
  - ALU bound
- Start with high precision
  - Move complex | highp math into texture lookups
  - Move highp into vertex shader
- Refactor to reduce swizzles on lowp
- PVRUniSCo for rough estimate
  - But **very** approximate! (no SGX543 profile)

Initially shader was written by artists, and while it did what it was supposed to do, it was quite expensive.

Use lowp where possible; move math into either texture lookups or into vertex shader (approximate but was fine). Refactor some math to avoid swizzles on lowp data.

Burning Walls shader

- Saved ~8ms per frame

Reduce Vertex count

- Too many vertices
- Were causing scene splits
  - Was losing ~3ms
- Use less vertices :)

Too many vertices with complex varying data were causing “scene splits” (Apple’s Instruments show this).

Optimize Particles

- Reduce overdraw
- Simplest possible shaders

<game2>

- 37 FPS after optimizations
  - total save ~25ms
- Room for adding more
  - 4xMSAA
  - Aniso
  - 26FPS
    - will optimize some more

Case: AngryBots

Full in Unity 3.4!

AngryBots is the default example project that ships with Unity 3.4. All the assets, levels, scripts and shaders are there, take a look!

AngryBots

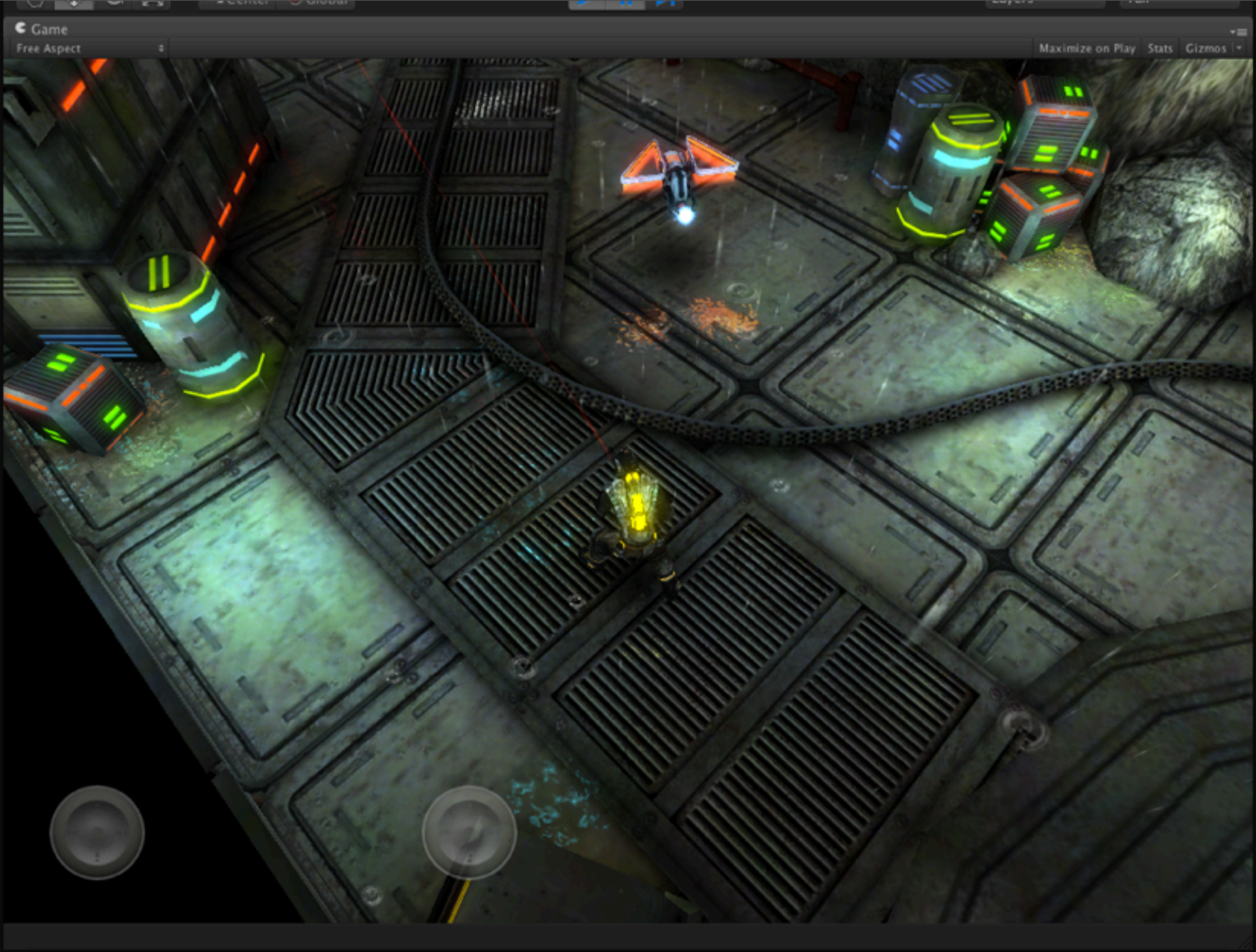
- Not much **optimization** per se
- Start simple & add stuff while can!
    - Texture, lightmap, bloom post-fx
        - Bloom can be disabled for slow GPUs

Added Stuff

- Wet surfaces, rain
- Reflective floors
  - Actual reflection, fake glossy reflection
- Noise overlay

Added for PC

- Depth of Field
- Height fog
- Normal mapped cubemap reflections
- More complex water fx
- ...

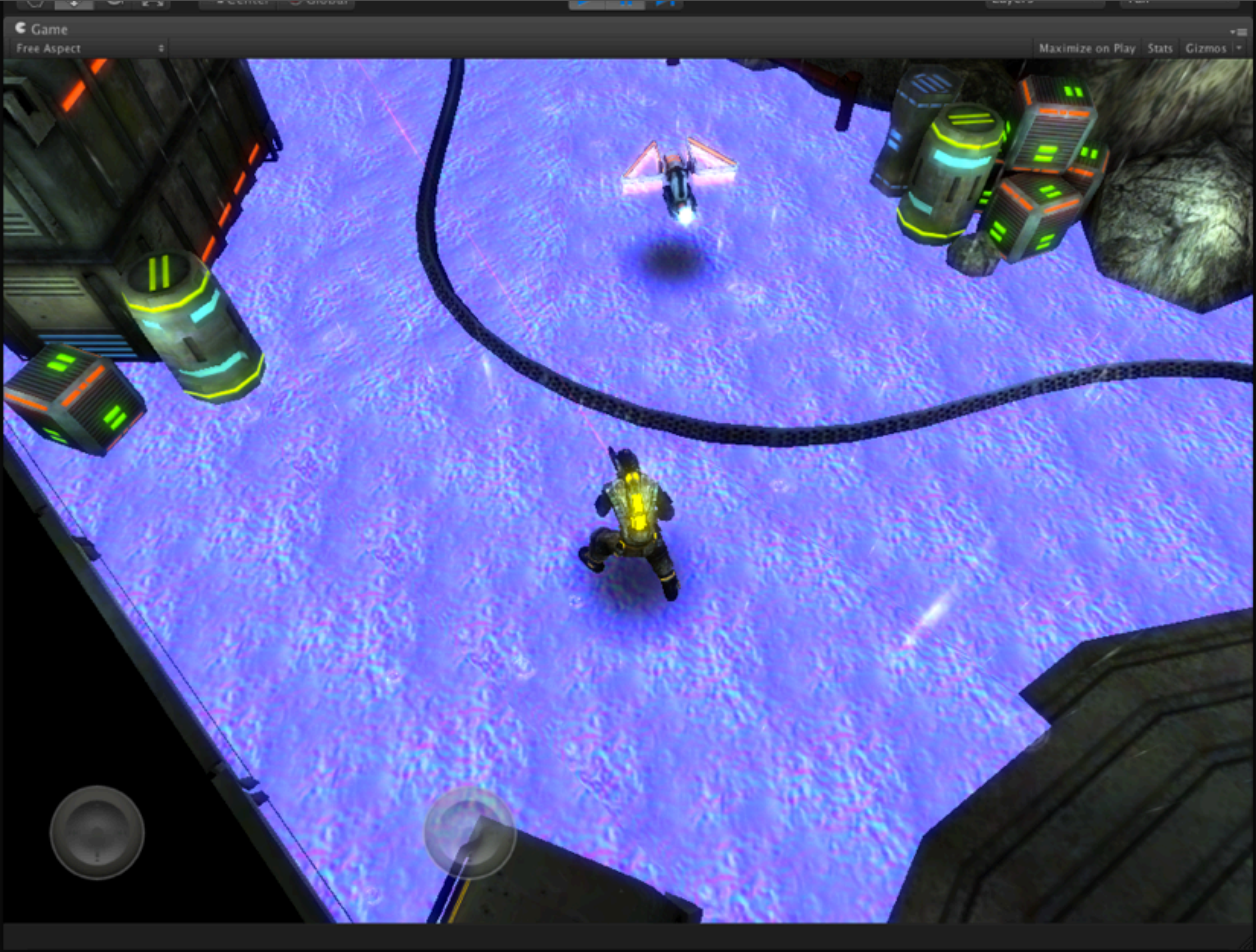


Wet floor

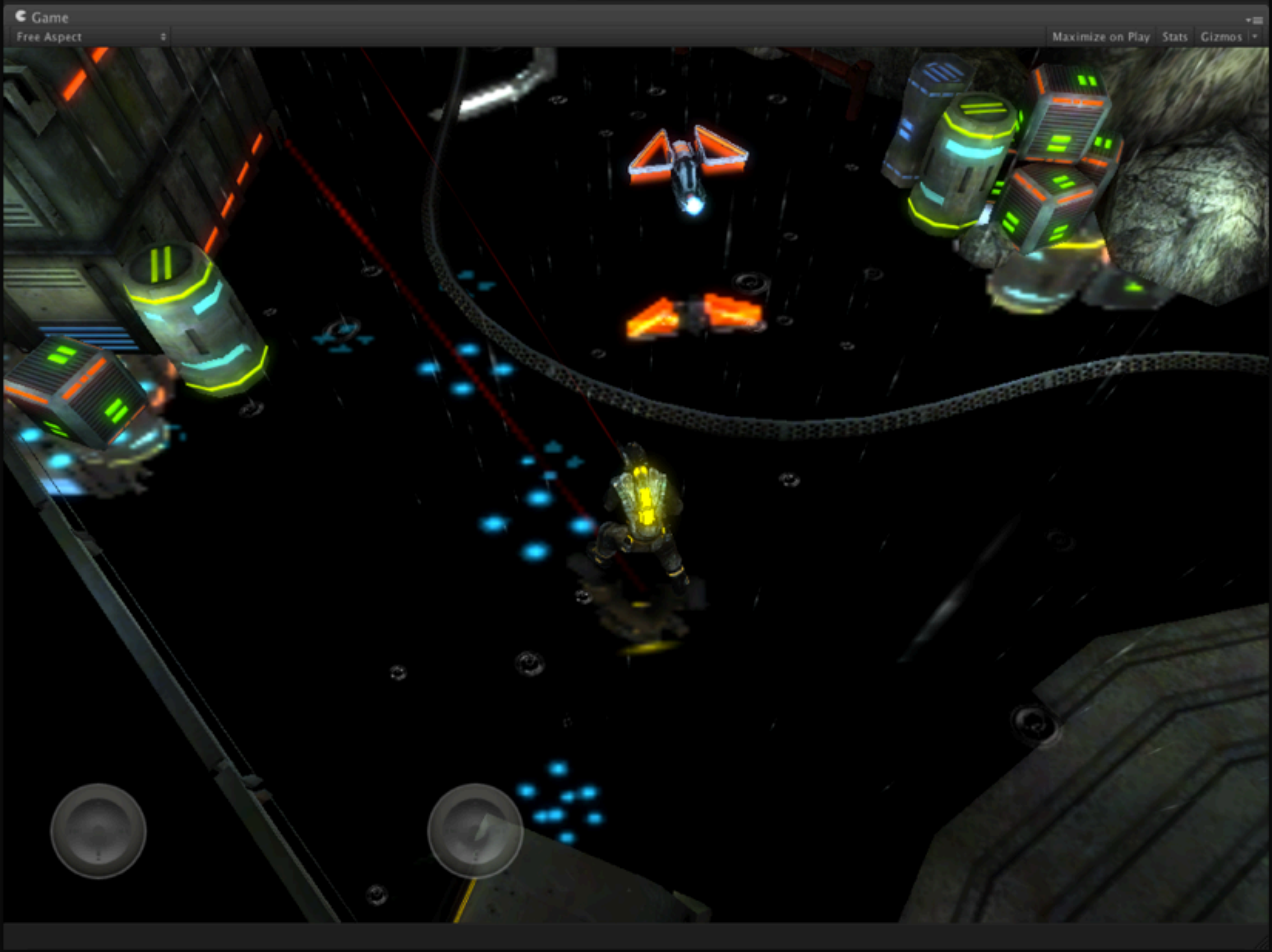
Breakdown of the wet floor shader

Wet floor

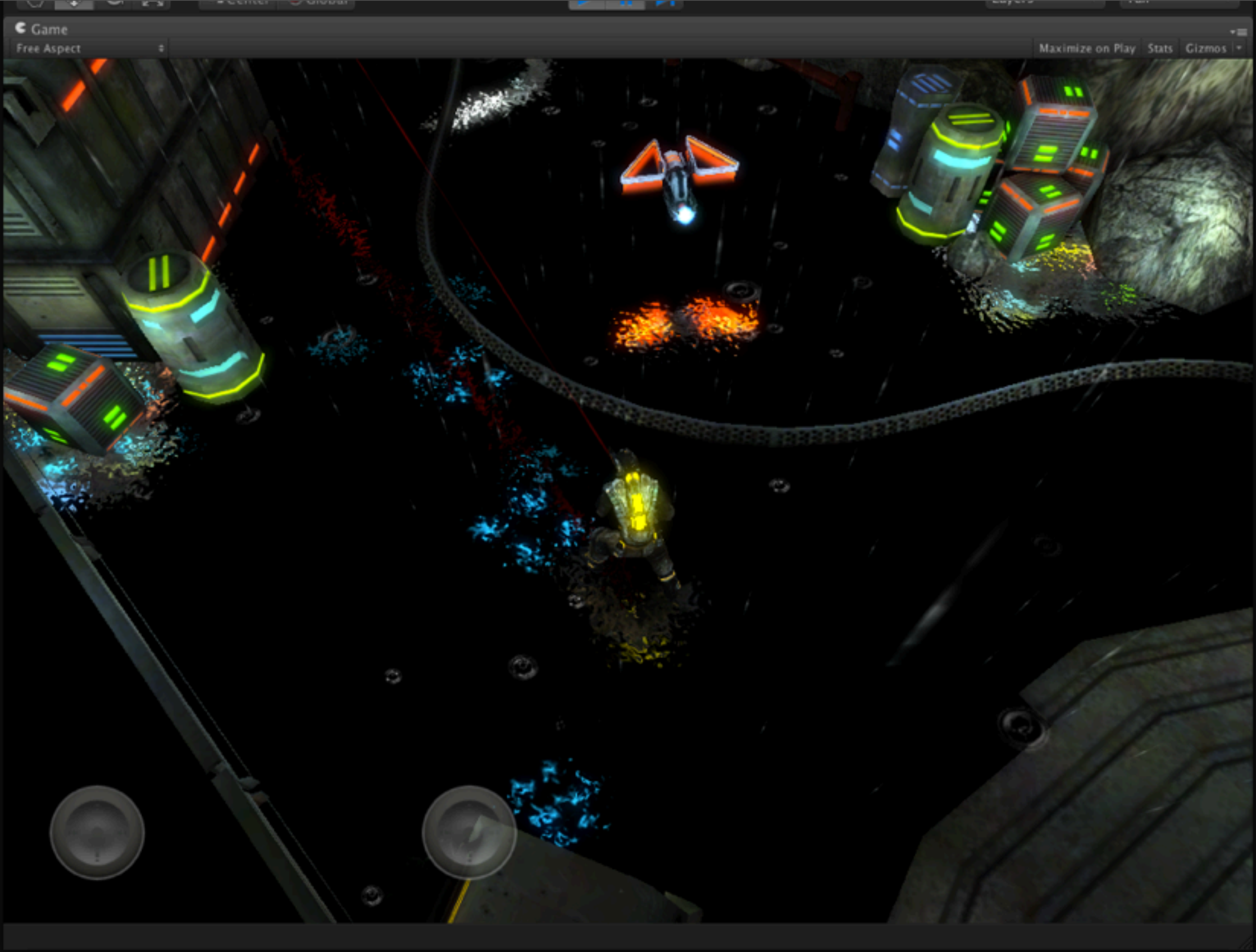
- Render subset of scene into realtime reflection texture
  - Simple shaders, just threshold bright colors
- Scrolling normal map to perturb it



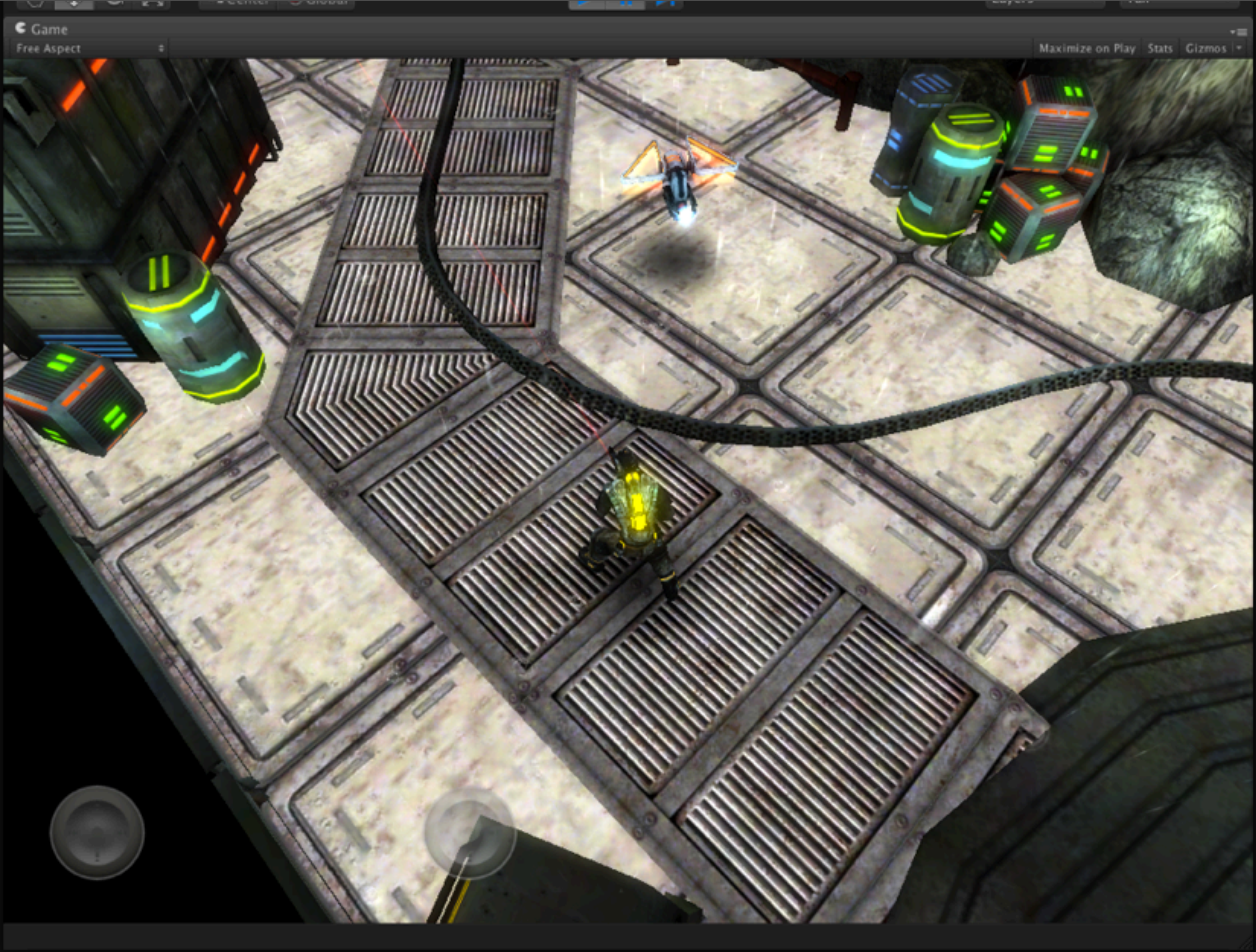
Water normal map



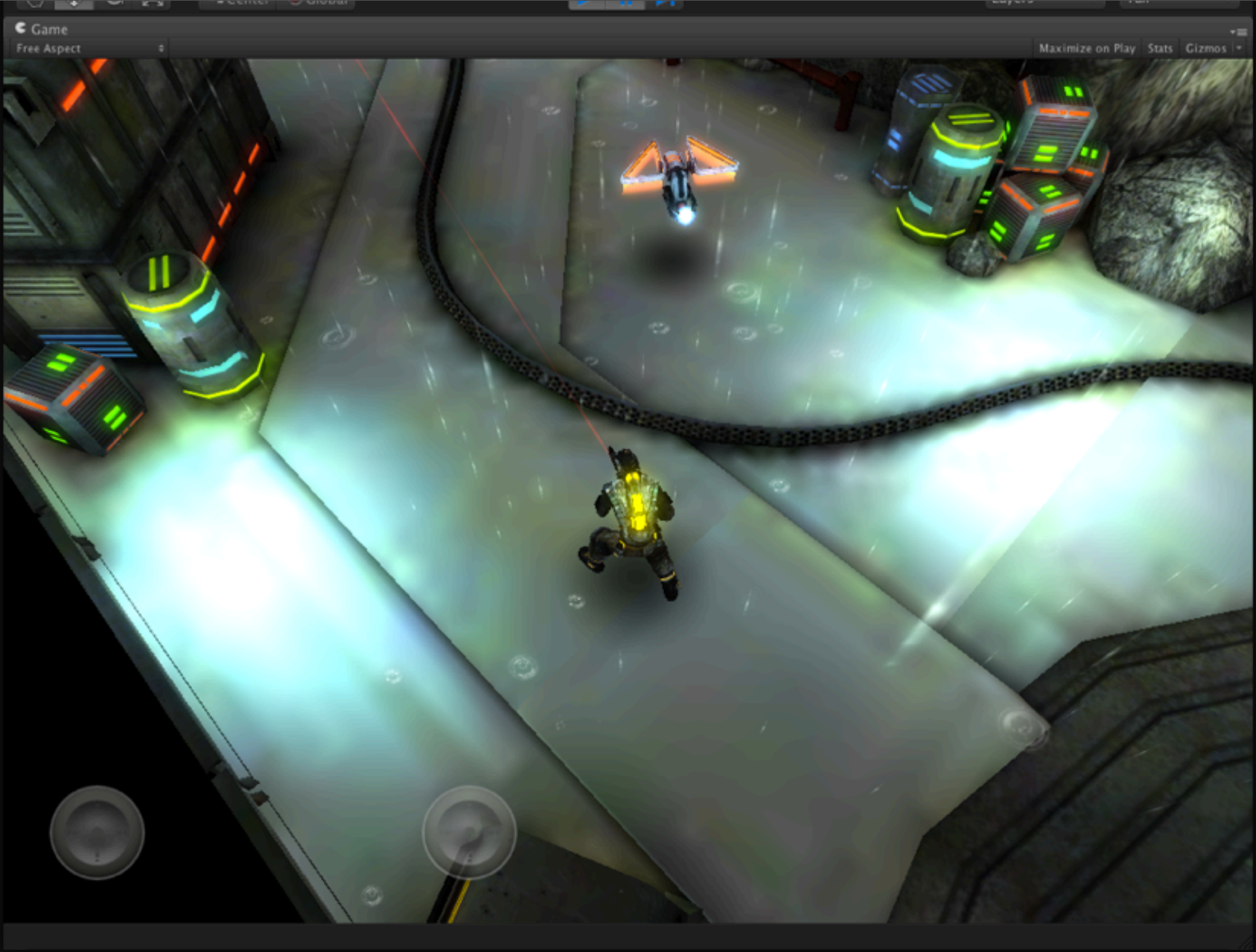
Planar reflection



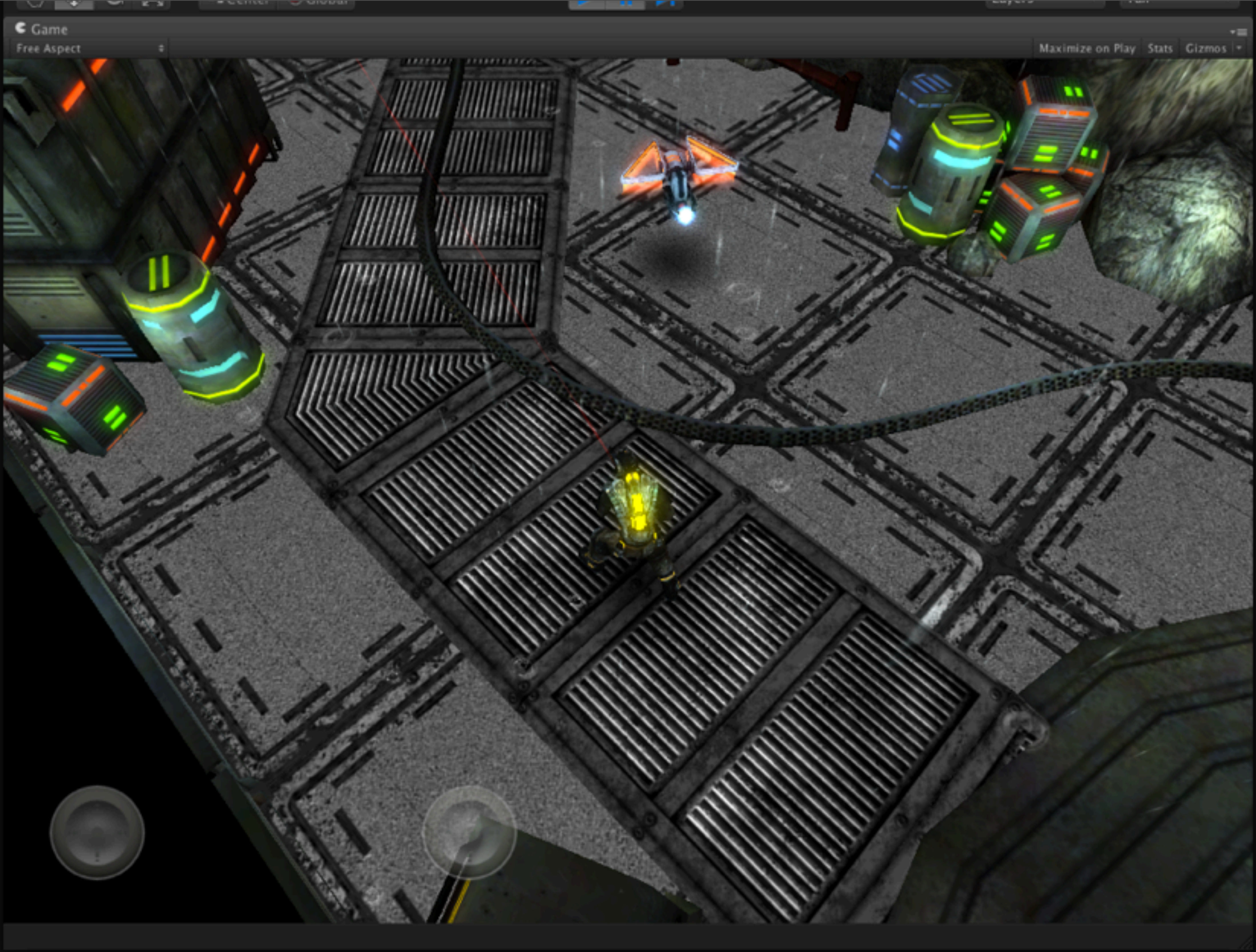
Perturbed reflection



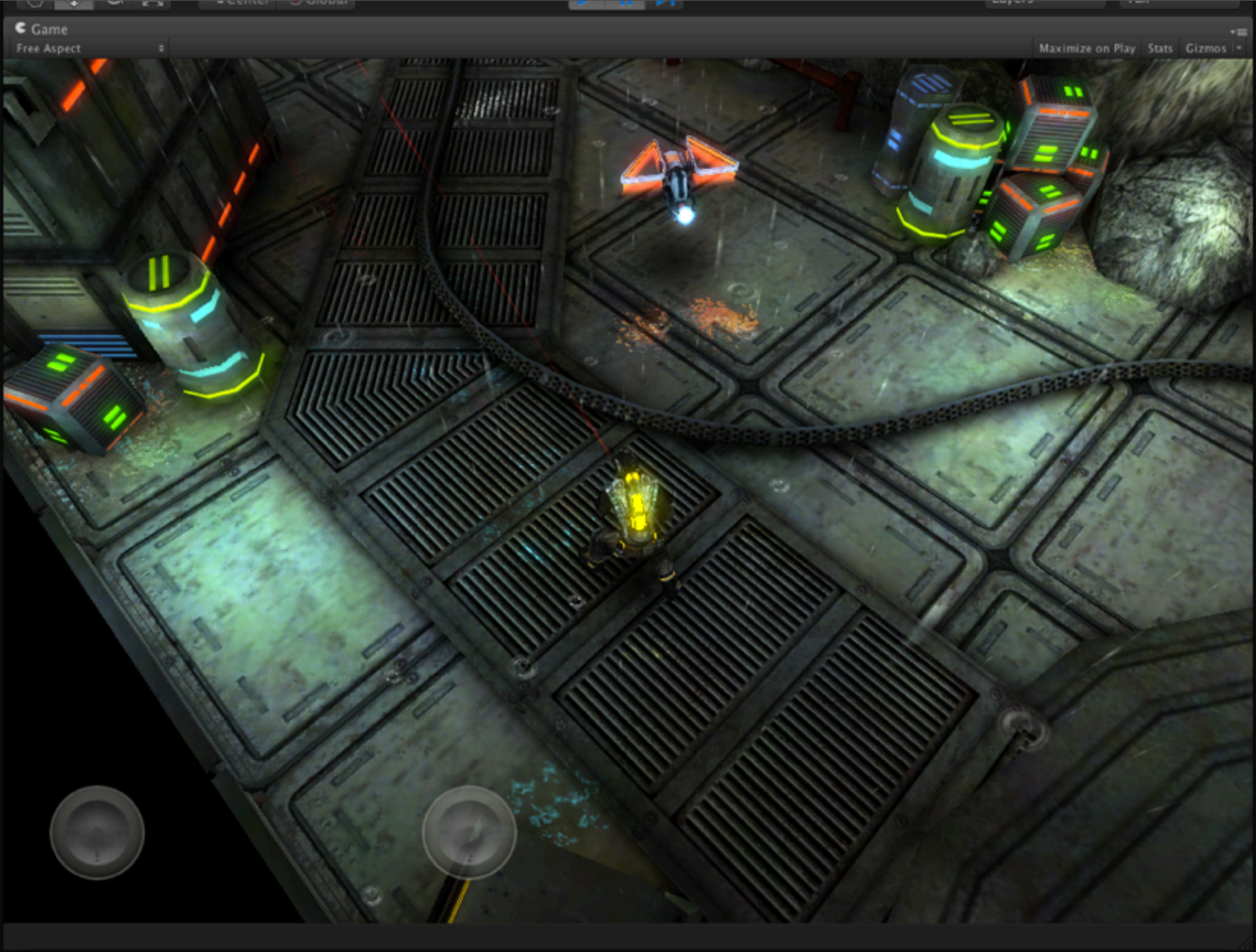
Albedo



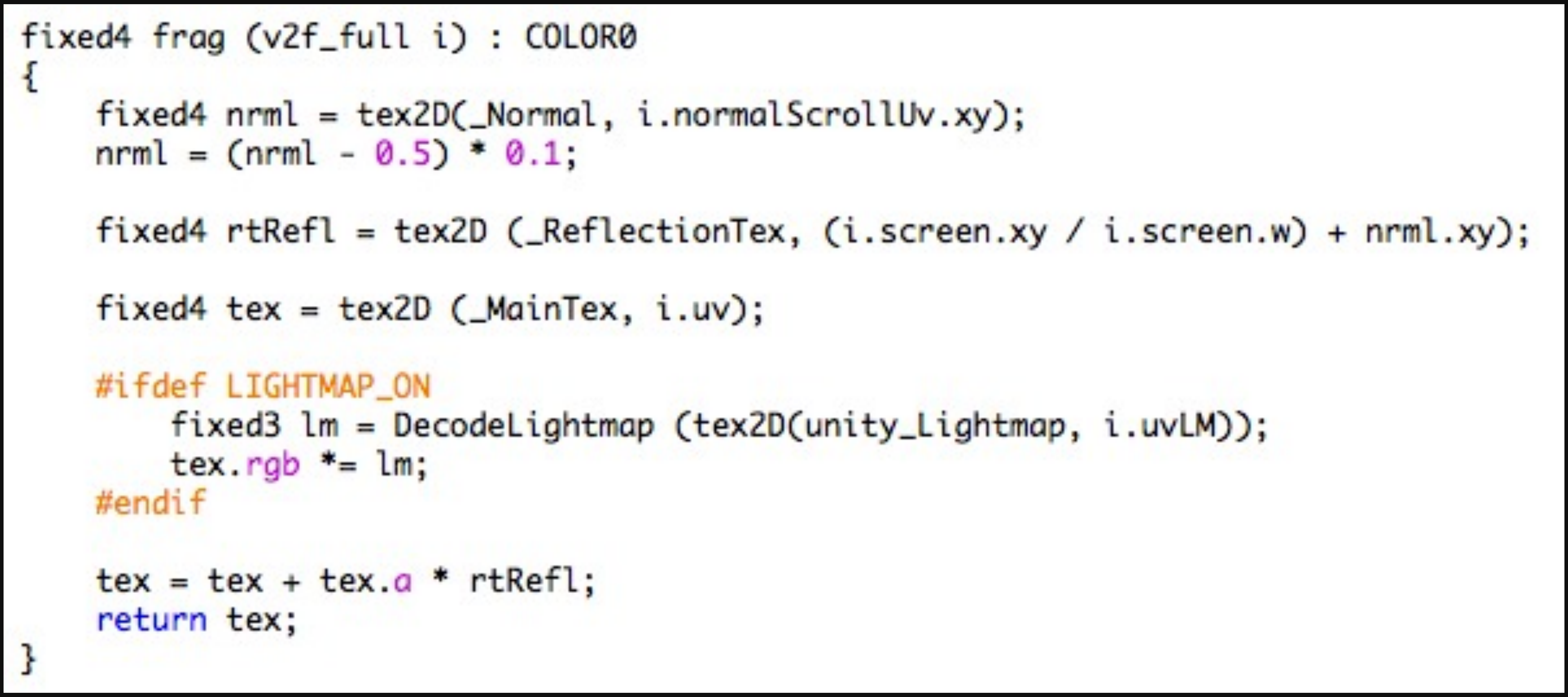
Lightmap



Gloss (A of albedo tex)

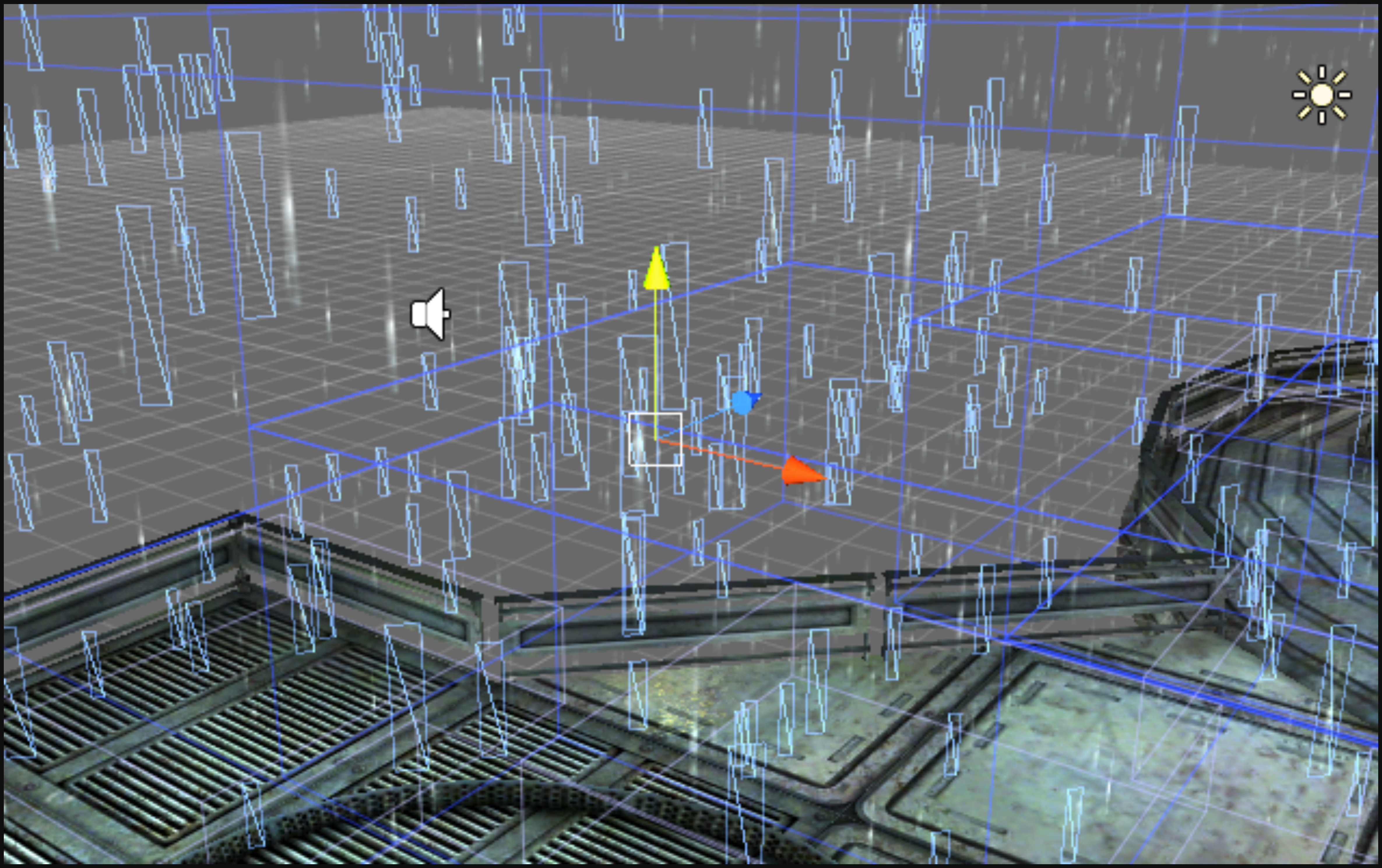


Result

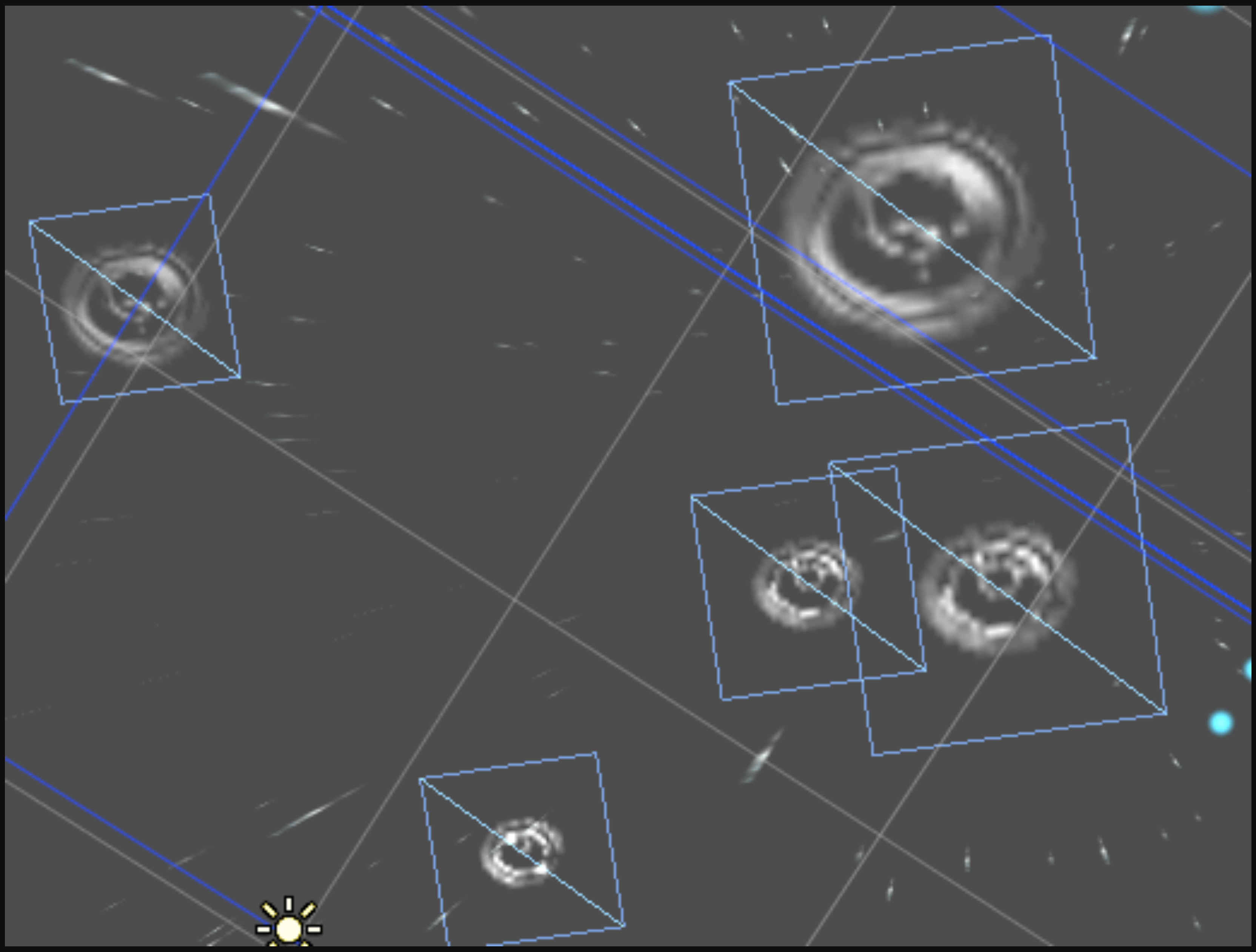


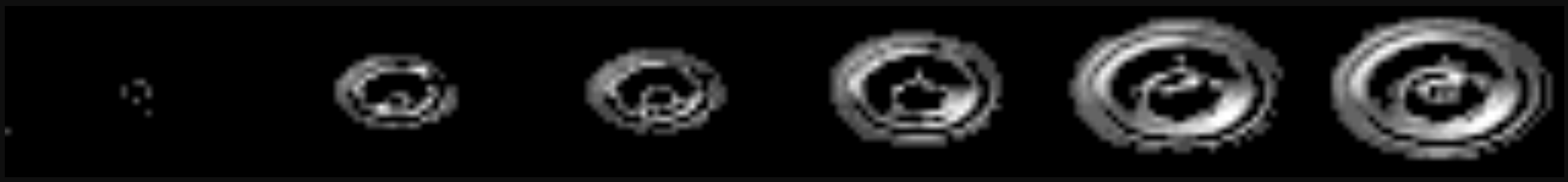
```
fixed4 frag (v2f_full i) : COLOR0
{
    fixed4 nrml = tex2D(_Normal, i.normalScrollUv.xy);
    nrml = (nrml - 0.5) * 0.1;

    fixed4 rtRefl = tex2D (_ReflectionTex, (i.screen.xy / i.screen.w) + nrml.xy);

    fixed4 tex = tex2D (_MainTex, i.uv);

    #ifdef LIGHTMAP_ON
        fixed3 lm = DecodeLightmap (tex2D(unity_Lightmap, i.uvLM));
        tex.rgb *= lm;
    #endif

    tex = tex + tex.a * rtRefl;
    return tex;
}
```

Pixel shader



Rain streaks: just geom





Splashes: geom + atlas

Water splashes are completely static geometry. Each “splash quad” loops over regions of a texture atlas. Different vertex colors for each quad offset the animation time of each splash.

Case: Optimizing Post-fx

Post-fx costs

- At native device resolution
- Sepia tone: 3.6ms iPad2, 3.7ms 3Gs, 1.9ms XperiaPlay
- Bloom: 4.6ms iPad2, 7.4ms 3Gs, 6.2ms XperiaPlay
- Color Correction: 5.4ms iPad2, 7.6ms 3Gs, 7.0ms XperiaPlay
- Noise+scratches: 3.2ms iPad2, 4.6ms 3Gs, 5.3ms XperiaPlay

Post-processing effects aren't exactly cheap :)

Chain post-fx

- Chain individual post-fx: Bloom, Color Correction, Noise
  - 12.8ms iPad2, 19.1ms 3Gs, 18.5ms Play
  - Extra render targets, resolves, ...

Unity makes it very easy to add multiple effects on a single camera, effectively “chaining” them. This is good to mix & match them, however it gets expensive really really fast.

Combine post-fx

- Color Correction & Noise can be rolled into final composite pass of Bloom
  - Less RTs, less resolves, all better!
- **10.9**ms iPad2 (was **12.8**ms)

Once you know which postprocessing effects you need, try to combine all of them into one effect, effectively stitching the shaders together. E.g. take the final Bloom composition pass, and add Color Correction & Noise into the shader directly, instead of having them as separate effects.

Optimize it

- Slightly PowerVR specific...
- Use two vec2 UVs instead of packing into one vec4
- **9.5**ms iPad2 (was **10.9**ms)

Cheat a bit

- Scratch texture on noise not much visible, leave only noise
- **8.7**ms iPad2 (was 9.5ms)
- Final: 8.7ms iPad2, 12.5ms 3Gs, 12.9ms XperiaPlay
  - Initially was: 12.8, 19.1, 18.5ms
  - Shaved off 1/3rd of the cost!

Case: Optimizing FXAA

FXAA

- Fast Approximate AA
  - Post processing color buffer
  - “Intelligent blur”
- Too costly for current mobiles
- Old FXAA 2 version
  - HLSL variant, cross compiled into GLSL in Unity

FXAA is an anti-aliasing postprocessing effect by NVIDIA’s Timothy Lottes. Right now it’s too costly for mobile GPUs, but let’s just see if we can optimize it.

Initially I took old FXAA 2 version and used it’s HLSL variant, cross compiled into GLSL by Unity.

FXAA2 Initial

- **32.1**ms iPad2, **65.5**ms 3Gs, **30.4**ms XperiaPlay
- Use PVRUniSCo
    - It likes to crash a lot
    - When it does, keep on tweaking shader, sometimes that makes crash go away...

Yep, way too expensive!

Use PVRUniSCo to tweak the shader, however current version likes to crash on it a lot :(

Fail #1

- FXAA is all float/highp
    - Let’s use lower precision where can!
- Much slower :(
    - WTF?!
    - Maybe conversions cost?
    - No idea.

Many attempted optimization steps weren’t successful. Using “general intuition”, tried to replace all colors with lowp - much slower! Why - no idea.

Fail #2

- At end of translated shader, looked like a complex dynamic branch
  - Try to make it generate simpler code!
- Slightly slower :(
  - No idea again!

The end of the shader looked like a complex branch in GLSL that the driver may or might not be able to optimize. Tried replacing it with simpler code – result slower. Why – no idea again (PVRUniSCo was crashing at that point, couldn't see shader cycles).

Win #1

- Use more (5) vec2 varyings instead of vec2+vec4
- **28.9**ms iPad2 (was 32.1ms)

First win: using more smaller varyings instead of trying to pack into full vec4 ones.

Win #2

- Use mediump for all texcoords
- 28.2ms iPad2, 32.1ms 3Gs, 18.2ms Play
  - Was 32.1, 65.5, 30.4ms
  - Not much, but something

Second win: using mediump for all texture coordinates. So in the end, was able to optimize the shader a bit for iPad2, and almost made it twice as fast on iPhone3Gs and XperiaPlay.

FXAA 3.11

- Latest FXAA 3.1 “console-pc” variant
- Initial, **22.6**ms iPad2, 28.3ms 3Gs
  - Much faster!
  - Does not work on XperiaPlay :(
    - Needs GL_EXT_shader_texture_lod

Latest FXAA 3.11 “console” version even pre-optimization already much faster!

Does not work on XperiaPlay though, since it does not expose GL_EXT_shader_texture_lod (needed for texture lookups after a dynamic branch).

Optimizing FXAA 3.11

- Again some fails
- Largest gain: use more varyings
  - Again
- Tweak precision bit by bit, verifying if still faster
  - Tweaking all “by intuition” made it slower

Just like with FXAA 2, tried some steps which were not successful. Largest gains very similar: use more varyings, use mediump for all texture coordinates.

Used lowp for most of colors, but not all of them. When I used lowp for all colors, shader got slower; needed lots of iterations on the device to turn colors into lowp one by one, verifying if shader did not become slower.

Optimizing FXAA 3.11

- Down to **12.7**ms iPad2, 16.2ms 3Gs
  - Was **22.6**, 28.3
  - 4xMSAA on this scene: **2.5**ms iPad2
- If next GPUs would be 4-6x faster at same resolution, FXAA could be very attractive!

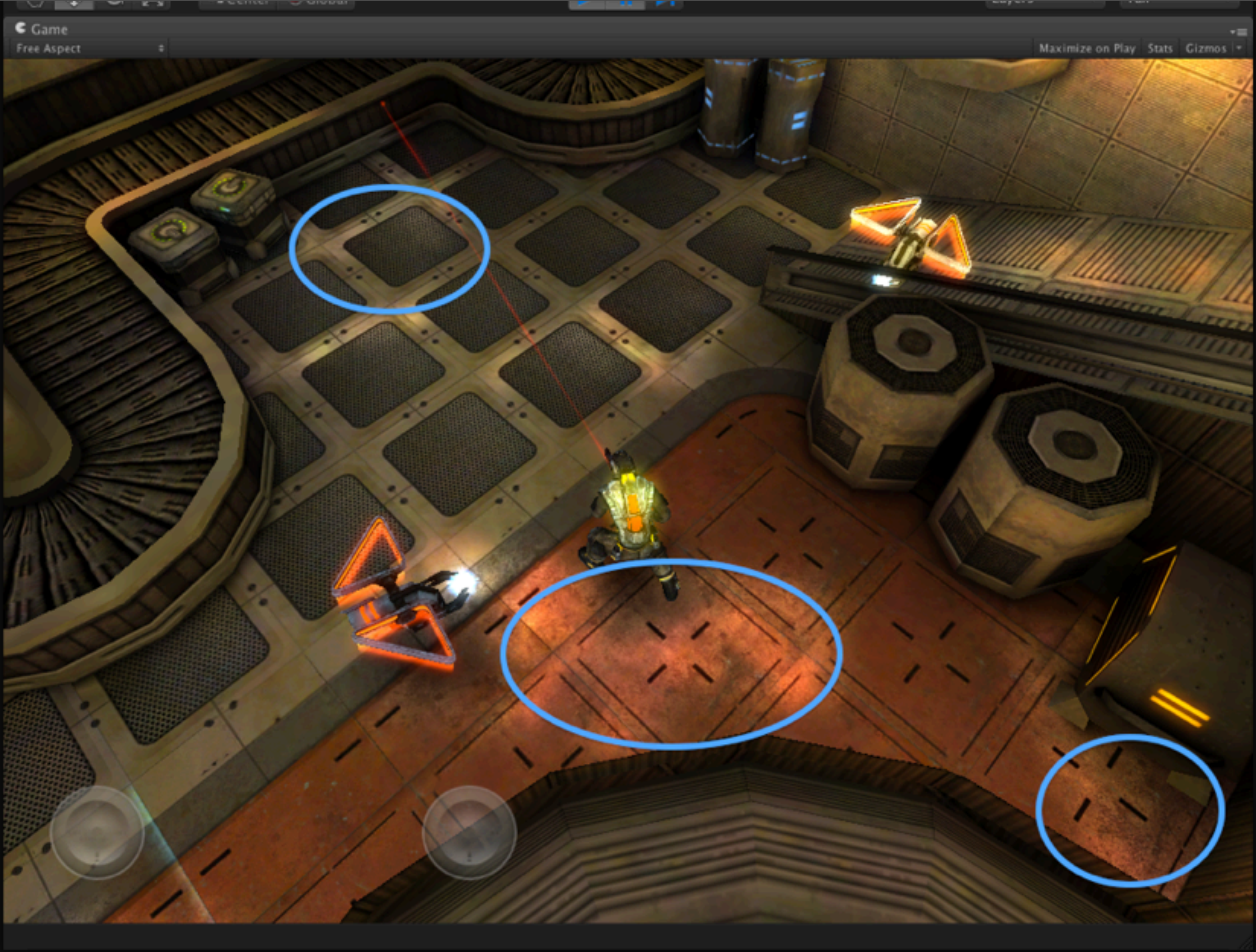
Almost 2x faster on iPad2!

12.7ms still way too expensive, but if next generation of mobile GPUs would be 4-6x faster while keeping the same resolution, FXAA for anti-aliasing could become very attractive.

That's it!

Extra Slides

Extra stuff we had initially prepared, but decided it's not really on topic; and we had time limit as well...



Glossy Reflection

Breakdown of fake glossy reflection in Unity 3.4 AngryBots

“Proper way”

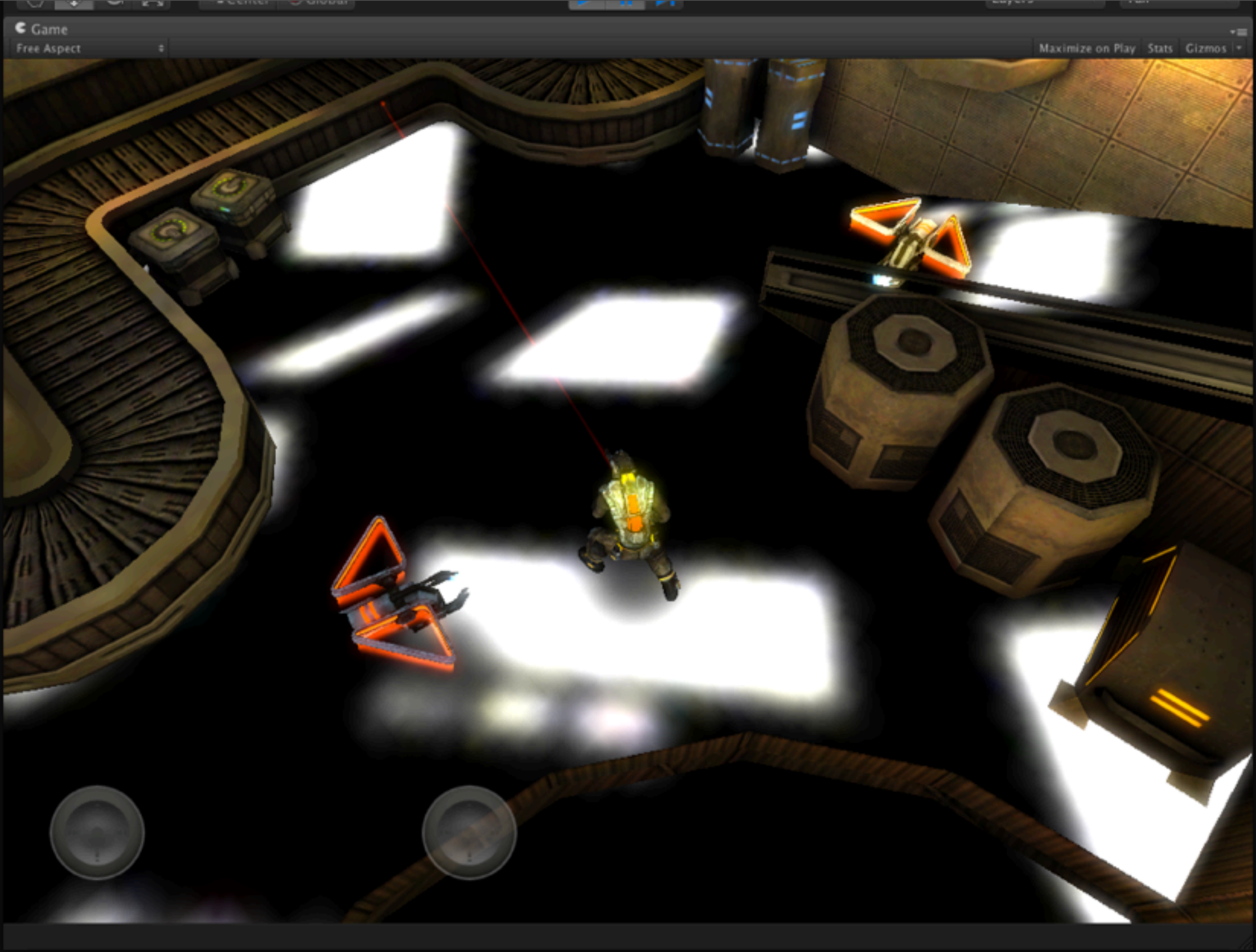
- Bake a bunch of blurred scene cubemaps
- Use one/more of them as reflection
- Can perturb with normal map
- Fine on PC, too costly on mobile!

“Proper way” would be baking a bunch of cubemaps, blur them, use them. This is what AngryBots does on the PC & consoles in fact. Was too costly for mobiles though.

Fake it!

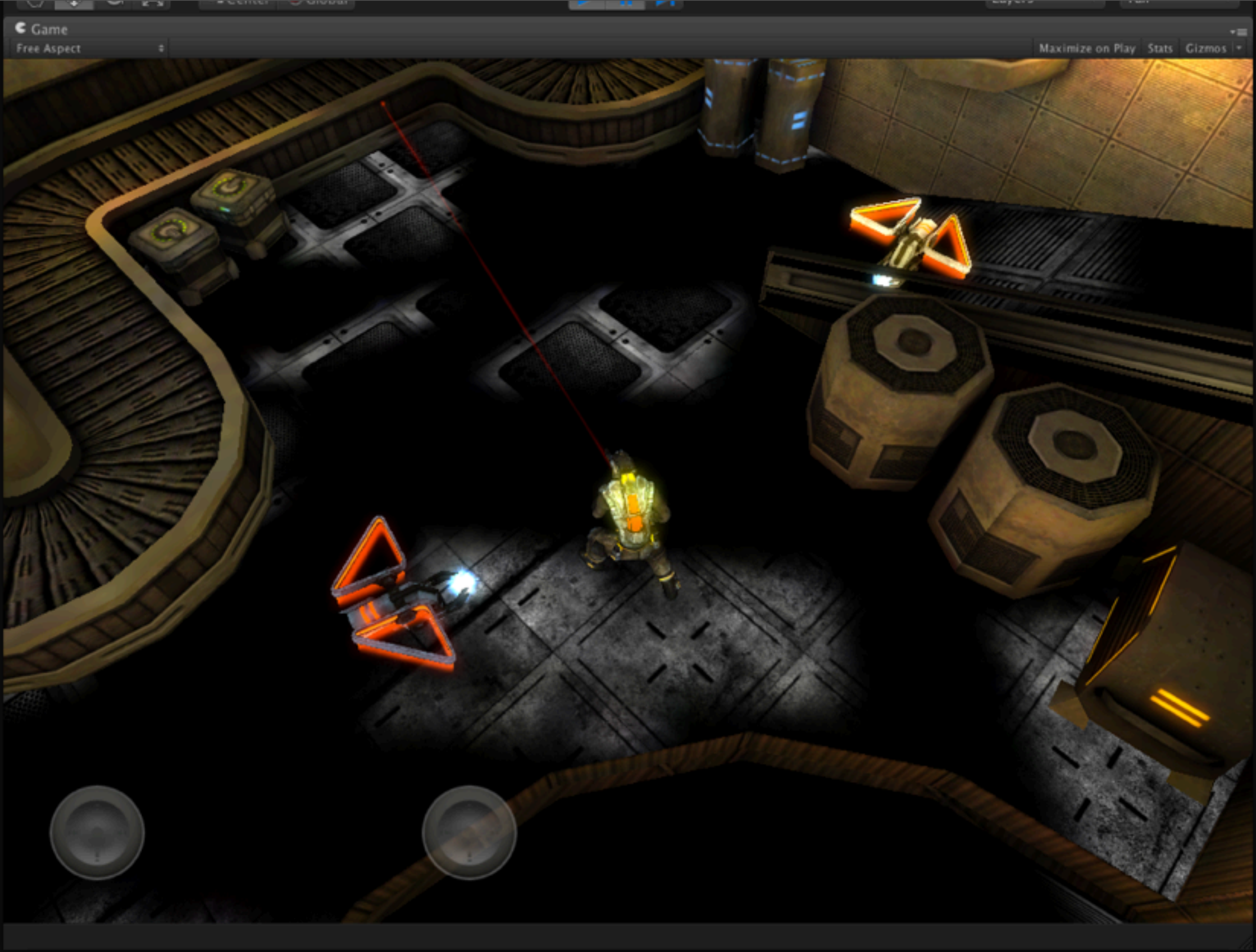
- Planar texture of “some stuff”
- UVs based on world pos & camera pos
  - Move a bit when camera moves
  - Apply on mostly horizontal surfaces
  - Works since fixed camera angle

“Let’s just reflect some generic 2D texture”

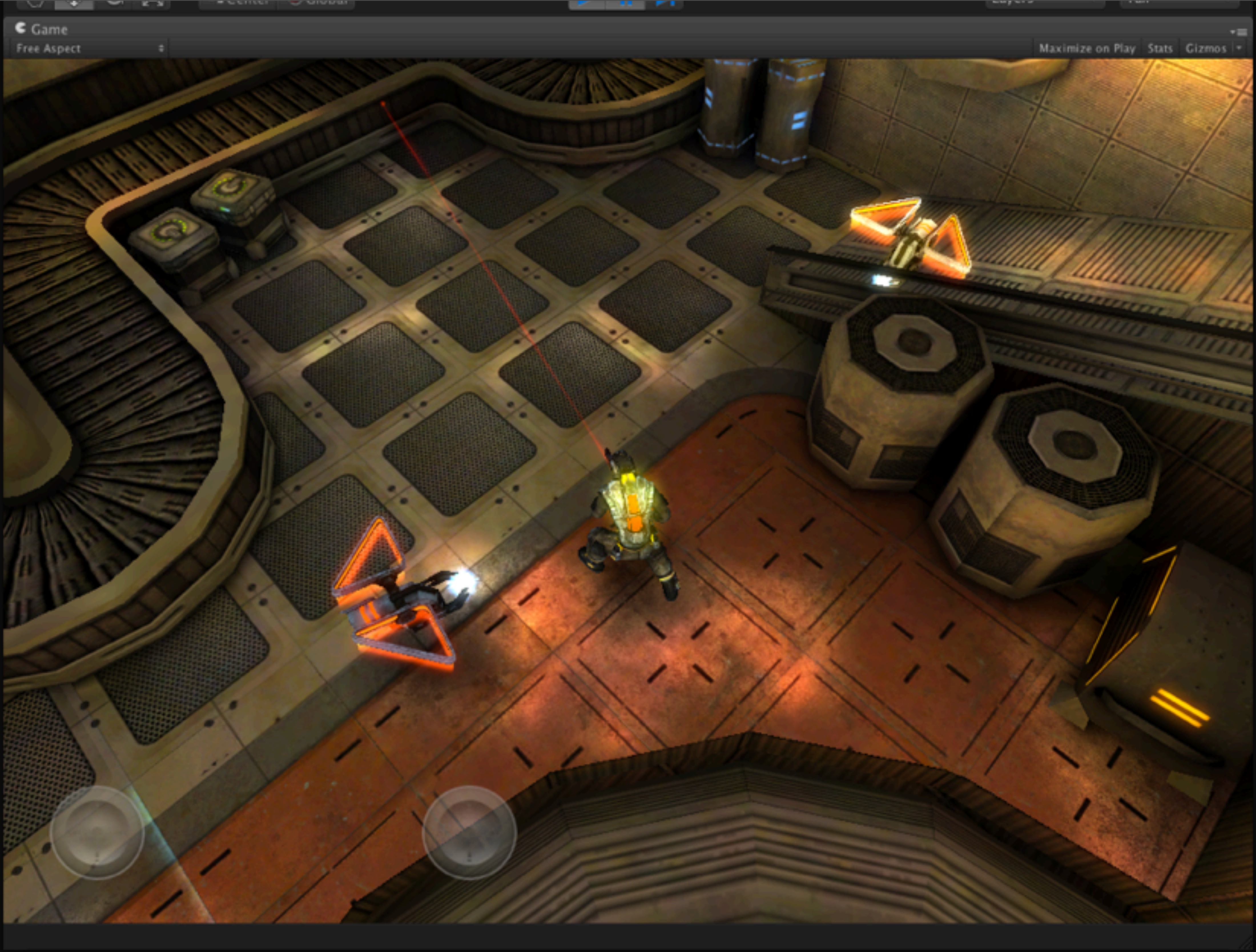


Actually reflected

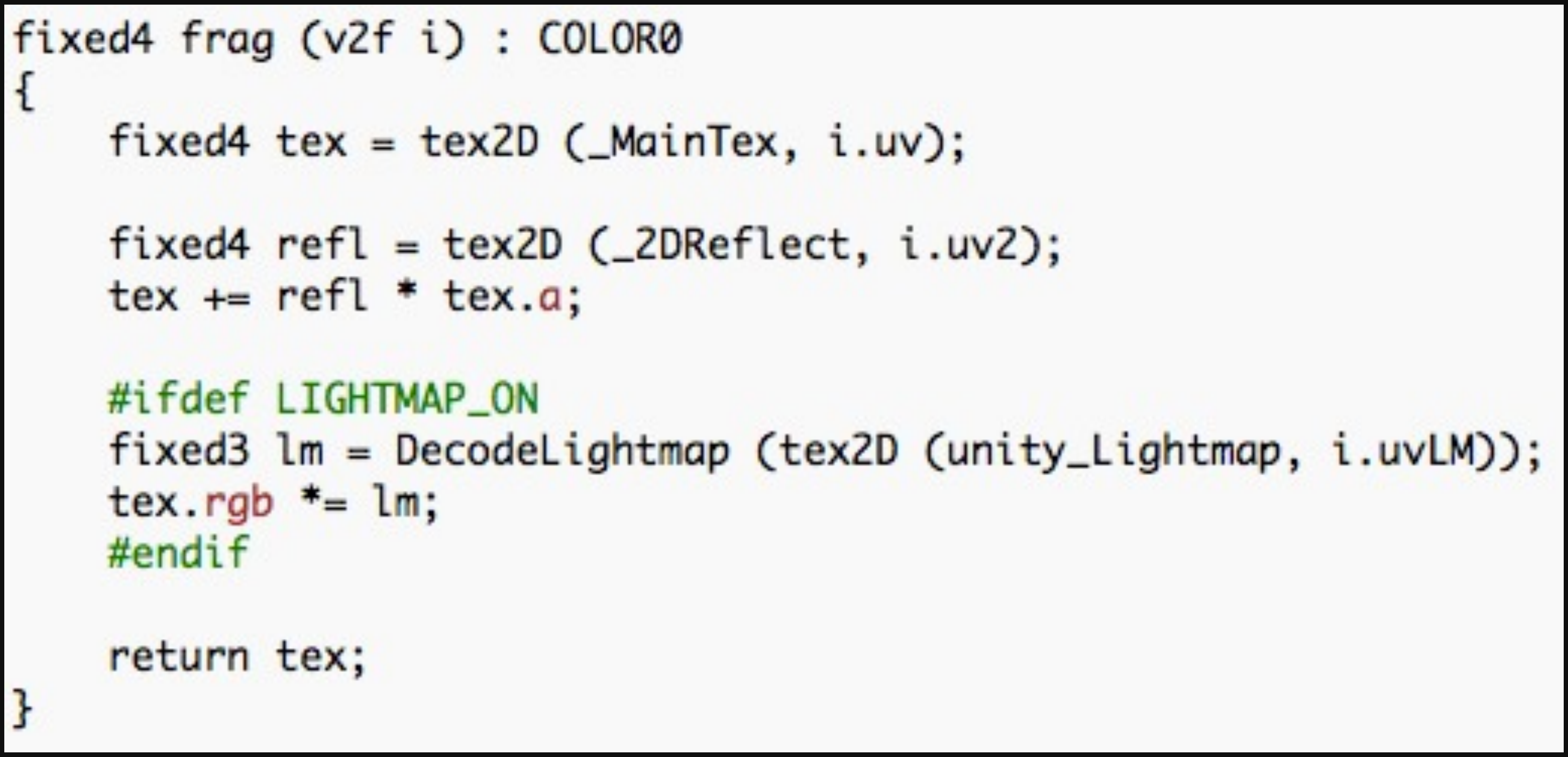
This is what is actually being reflected – it does not match what's in the scene at all. But it does result in some glossy highlights moving around when the player moves around – good enough!



With gloss map



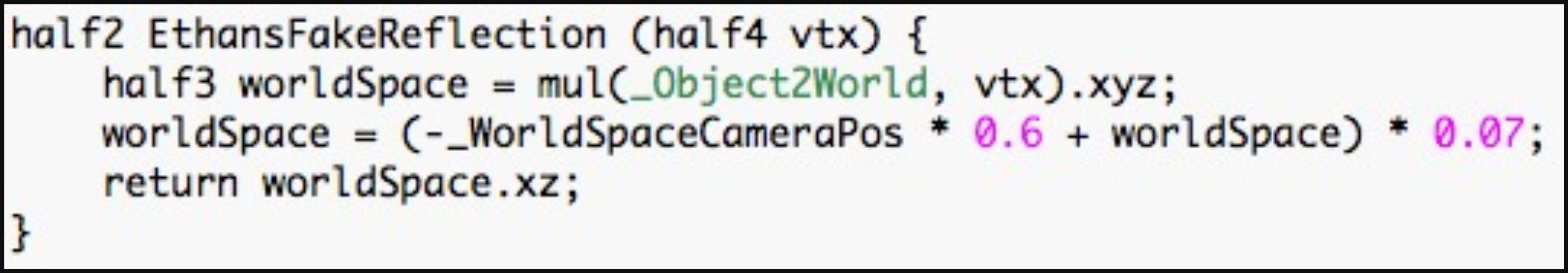
Ta-da!



```
fixed4 frag (v2f i) : COLOR0
{
    fixed4 tex = tex2D (_MainTex, i.uv);

    fixed4 refl = tex2D (_2DReflect, i.uv2);
    tex += refl * tex.a;

    #ifdef LIGHTMAP_ON
    fixed3 lm = DecodeLightmap (tex2D (unity_Lightmap, i.uvLM));
    tex.rgb *= lm;
    #endif

    return tex;
}
```

Pixel shader: simple

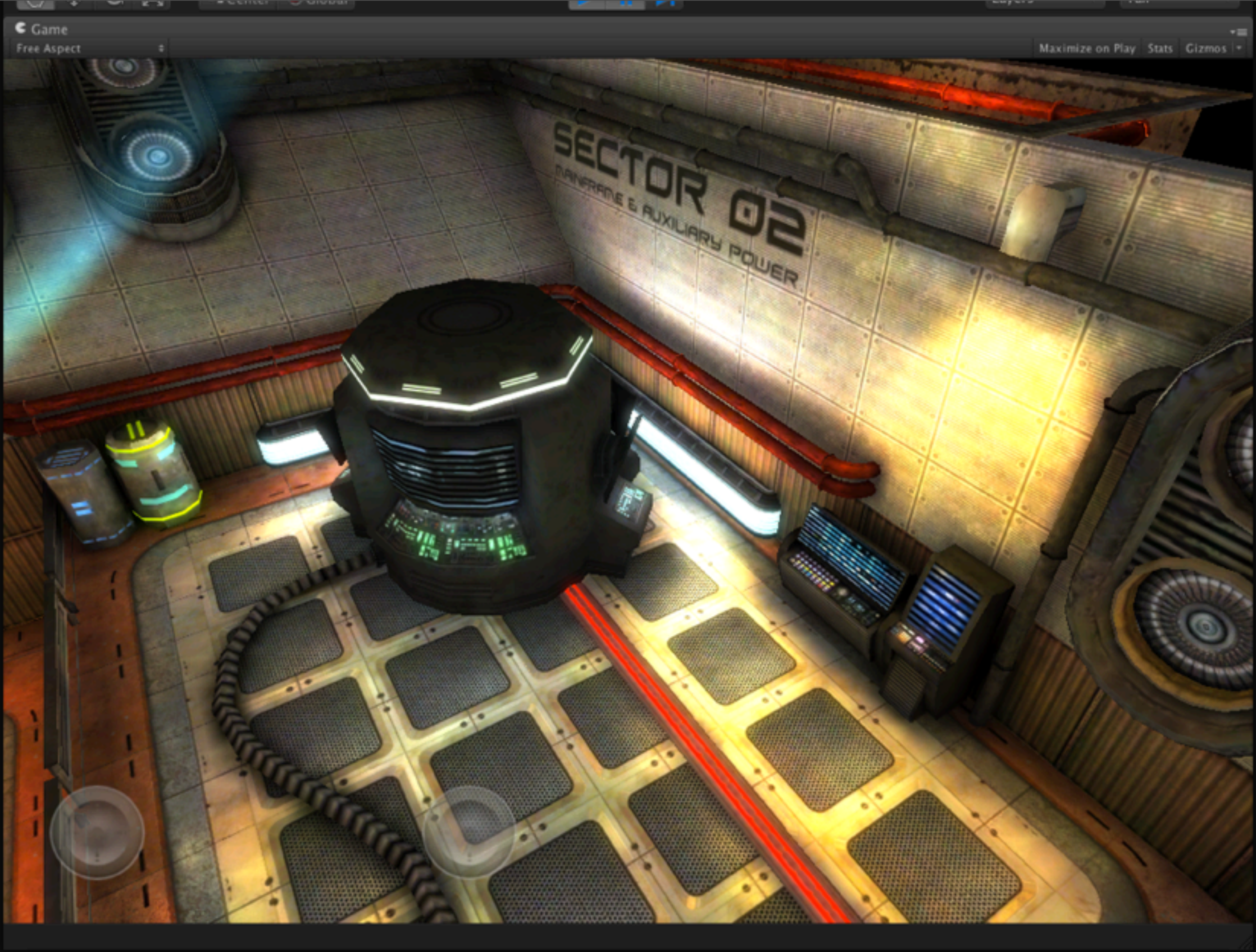
Reflection UVs



```
half2 EthansFakeReflection (half4 vtx) {
    half3 worldSpace = mul(_Object2World, vtx).xyz;
    worldSpace = (-_WorldSpaceCameraPos * 0.6 + worldSpace) * 0.07;
    return worldSpace.xz;
}
```

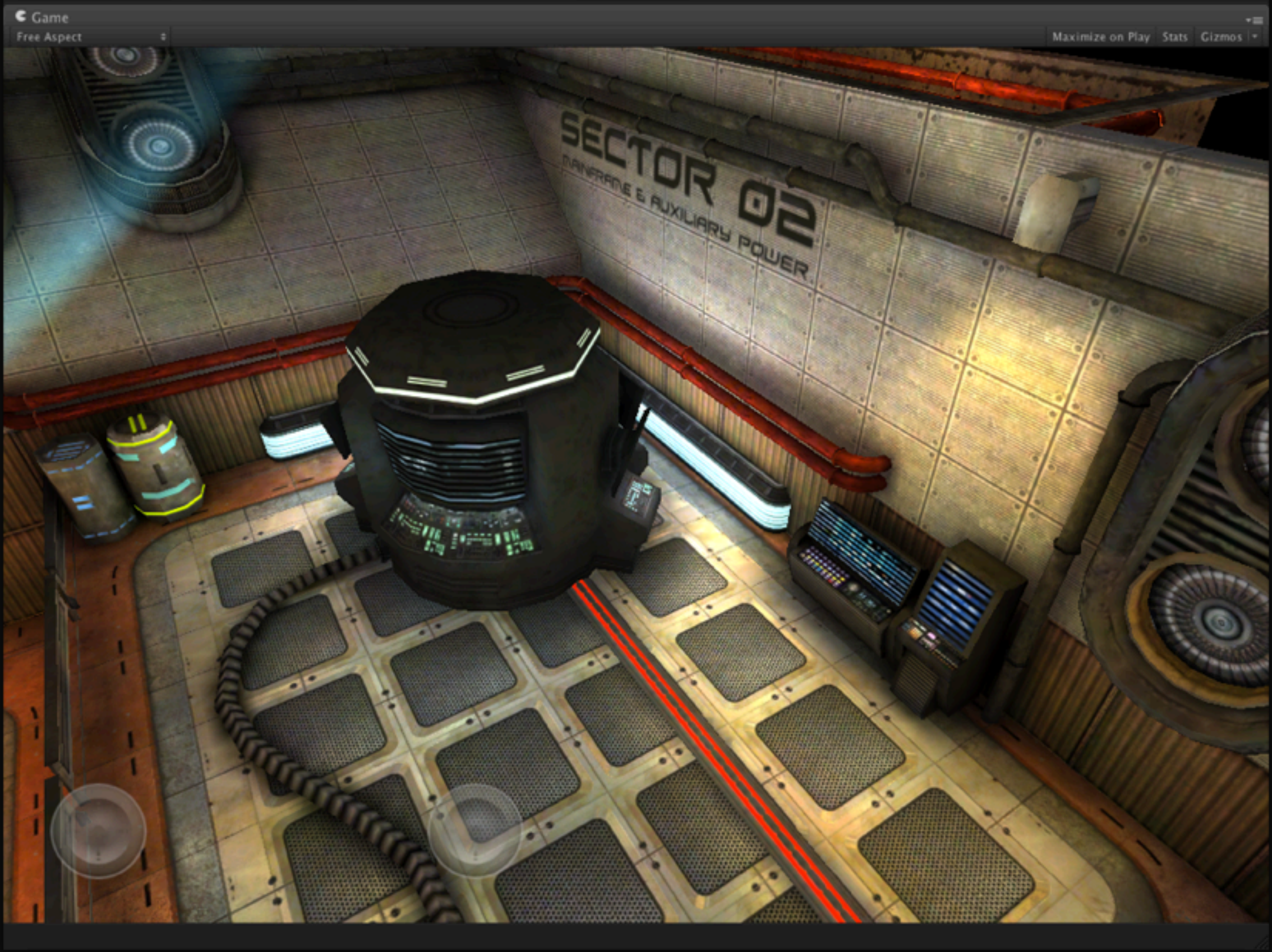
- Not much physical sense...
- ...but it looked fine!

Vertex shader code that computes UVs of this planar reflection texture.



Bloom Post-fx

Bloom in Unity 3.4 AngryBots



Without Bloom

Ye Olde Bloom

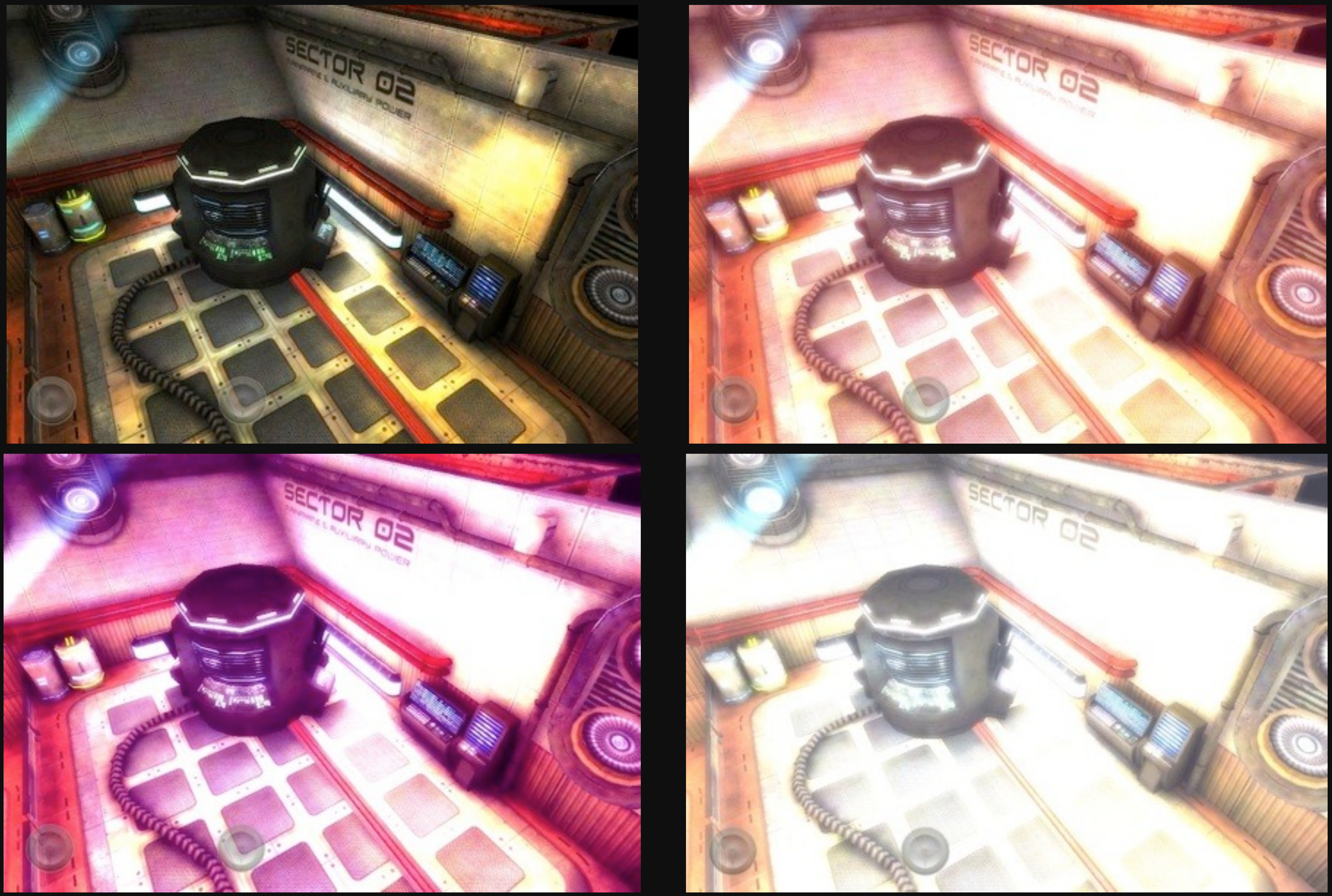
- Downsample into 4x4 smaller RT
  - Takes max() of 5 taps, thresholds
- Separable blur on small RT
  - 4 tap
  - Horizontal & vertical passes
- Add to original image

Just a regular old Bloom effect

Also Color Adj

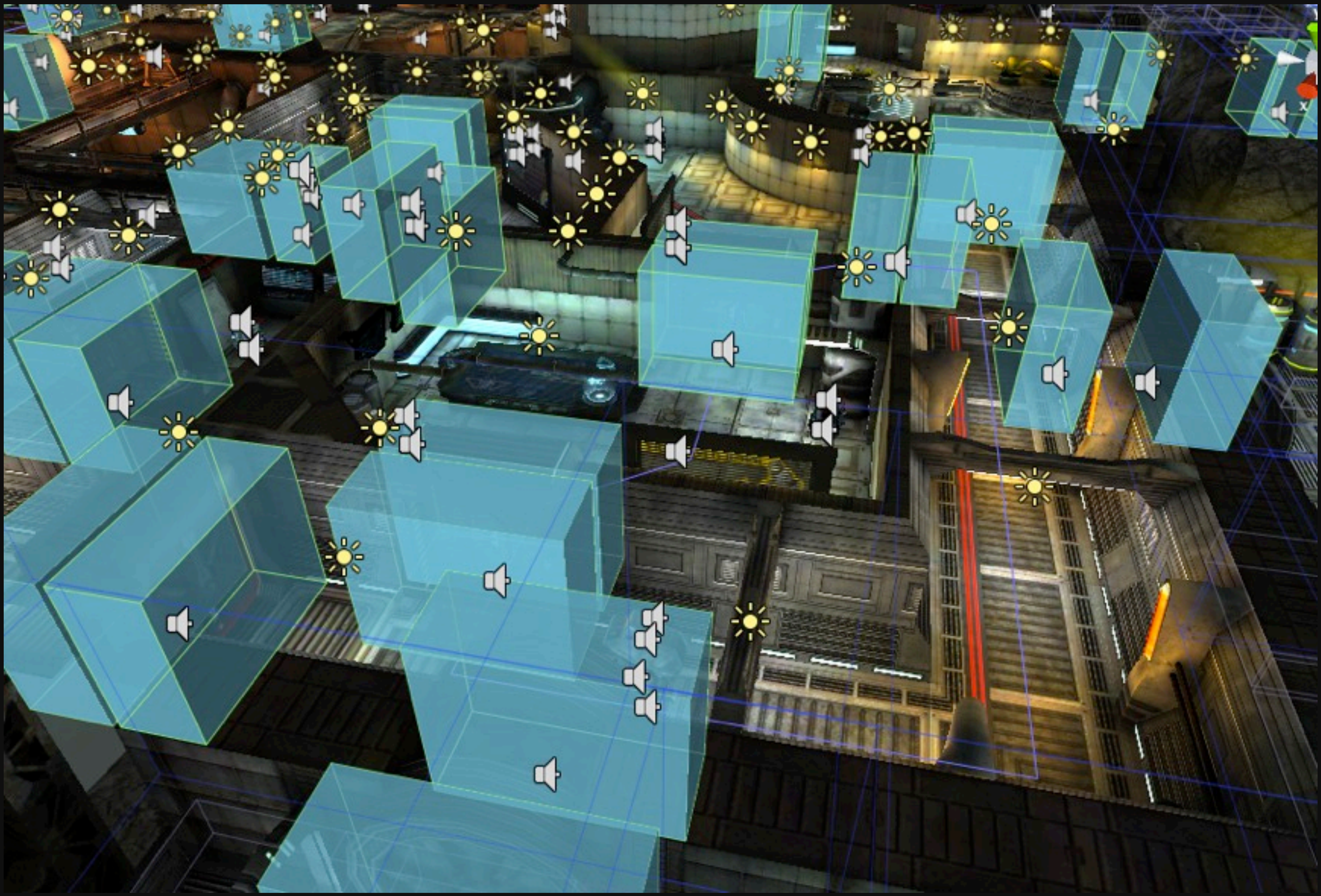
- Can also do simple color adjustiment
- Add/subtract color with alpha factor

It also does a very simple color adjustment in AngryBots



Used by game events

Used to slightly tint the scene depending on “mood”, or very strongly tint it when the player is hit, etc.



Mood Boxes

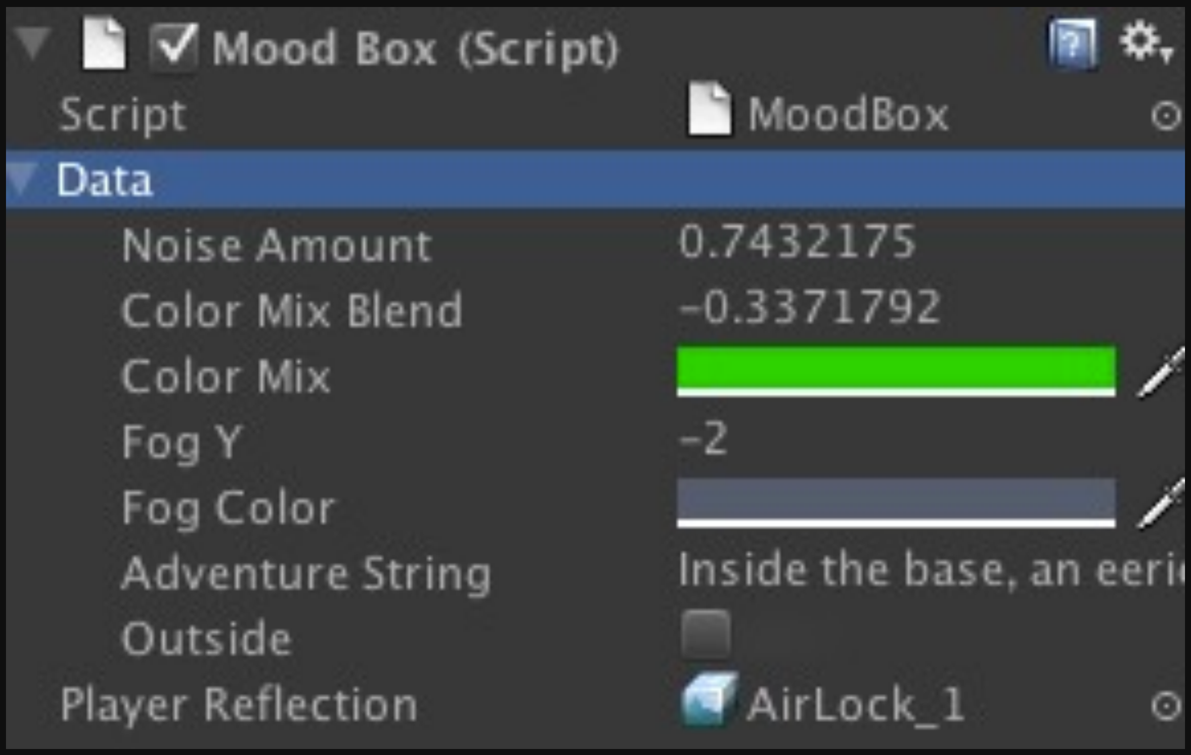
“Mood Boxes” in Unity 3.4 AngryBots

Mood Boxes

- Trigger areas mostly around doors
- Artistic stuff
  - Bloom color adjustment parameters
  - Height fog
  - Fake reflection texture used
- Optimization
  - Turn realtime reflection, height fog, ... on/off

Mood Boxes

- When entering new mood box, lerp towards it
  - No harsh parameter switches



Mood Box (Script)

Script	MoodBox
Data	
Noise Amount	0.7432175
Color Mix Blend	-0.3371792
Color Mix	🟩
Fog Y	-2
Fog Color	▨
Adventure String	Inside the base, an eeri
Outside	☐
Player Reflection	AirLock_1