SIGGRAPH2015

Xroads of Discovery

SIGGRAPH2015
Xroads of Discovery
The 42nd International Conference and Exhibition
on Computer Graphics and Interactive Techniques

Frostbite on mobile

Niklas Nummelin
Electronic Arts / Frostbite

Me

- Niklas Nummelin
- Worked at EA for 2.5 years
- Mobile industry 9 years
- Passion for computer graphics and music
- Responsible for bringing up Frostbite graphics on mobile

Contents

- A brief history
- From GL to Metal and back
- Shaders, shaders, shaders
- Let there be light
- Summary and future work

A brief history

WHAT IS THIS MOBILE THING?

A brief history

- Initiative to bring Frostbite to mobile started 2.5 years ago
- Battlefield 4 was in development
- First goal: Supply tech for Tablet Commander


MENU
A B D C E
US 300
00:45:23
CN 099
SCOREBOARD
Feffalump [G36C] LeCrap
Anpanfisk [G36C] IlCarpentero
US
SCAN UAV
EMP UAV
EVAC ORDER
HIGH VALUE TARGET
B
VEHICLE SCAN
C
CRUISE MISSILE
D
INFANTRY SCAN
VEHICLE DROP
RAPID DEPLOY
SUPPLY DROP
PROMOTE SQUAD
TheHelgon
Manualmartin
OhPlz
Megalodon
A
B
C
D
E
US
ALPHA
BETA
CHARLIE
DELTA
ECHO
FOXTROT
GOLF
50
CRUISE MISSILE LAUNCHED +50
SCAN BONUS +25
ORDER ACCEPTED +10
Manualmartin

A brief history

- Next goal: to get full Frostbite experience on mobile with Xbox 360 fidelity
- Showcase all Frostbite features but with limited scale
  - Dynamic lighting (Sun light, few dynamic light sources)
  - Effects
  - Destruction
  - Animated characters
  - Vehicles
  - Networking
  - Terrain

A brief history

- First OpenGL ES 2.0 + extensions

- Later OpenGL ES 3.0
- Exactly the same engine as running on console/PC

A brief history

- HLSL shaders ported to GLSL with custom optimizations
- Added GLSL target to graph-based shader generator
- Aras / Mesa’s GLSL-optimizer to the rescue!
- Added full forward rendering
  - Ended up with hybrid approach with Z-prepass

From GL to Metal and back

THEN WE WENT TO CUPERTINO

From GL to Metal and back

- Brought up initial Metal renderer in weeks
- Custom Metal backend for glsl-optimizer
- Reached steady 30 fps with SSAO, DOF, Radiosity, cascaded shadowmaps, HDR, color grading, FXAA and lots of draw calls on a stock iPad Air 1

From GL to Metal and back

- New idioms on mobile: Command buffers, pipeline/state objects, explicit lifetime management
- 5x lower cpu overhead
- Explicit control of render passes and tile memory lifetime

- Alpha tested objects kept to a minimum
- Instancing not as beneficial anymore

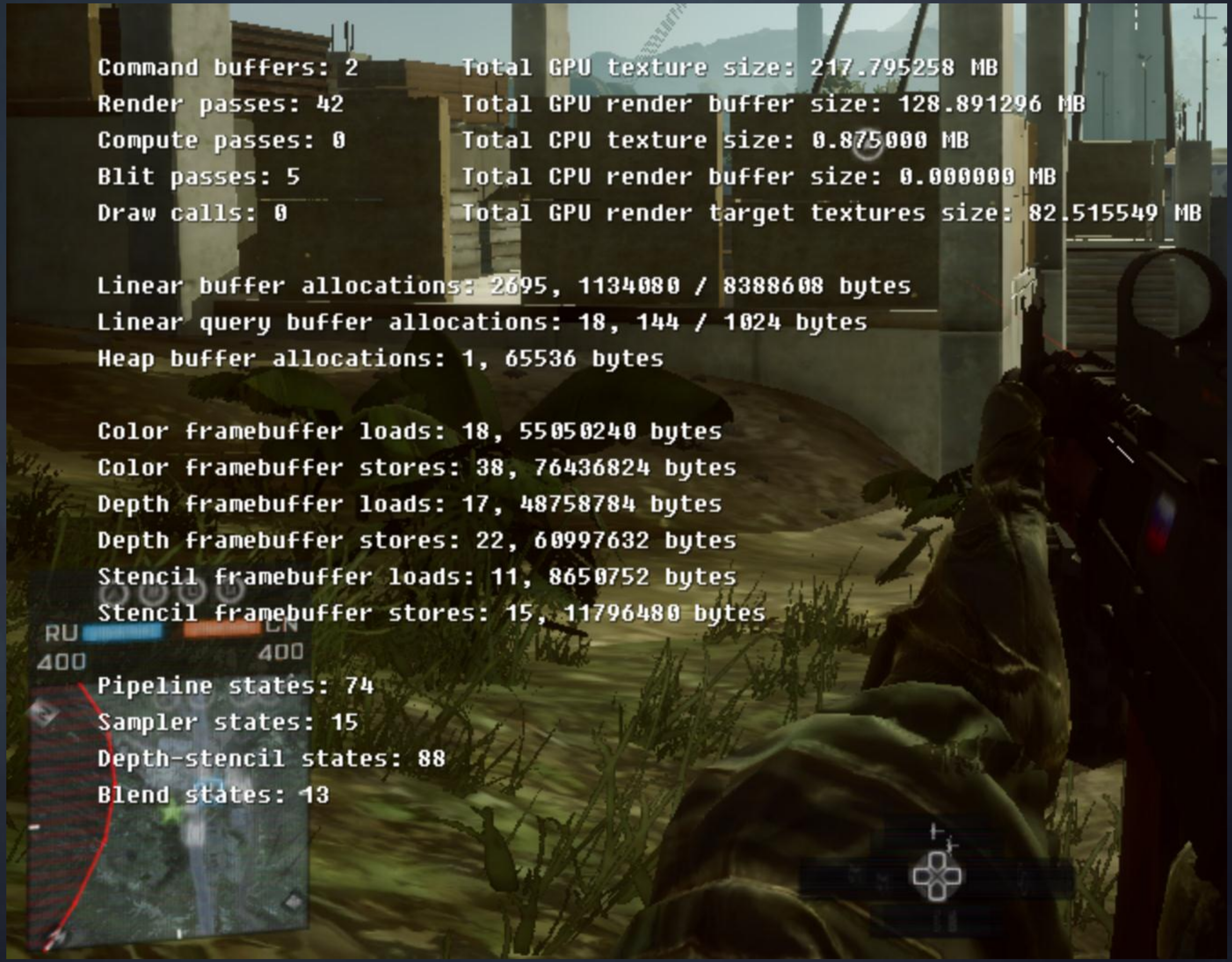
From GL to Metal and back

- Next goal: Test the limits of the engine (and hardware)
- Get a vertical slice of Battlefield 4 running on iPad Air 2

From GL to Metal and back

- Two major challenges:
  1. Engine had started to diverge in terms of memory consumption from the xbox 360 days

From GL to Metal and back



From GL to Metal and back

- Two major challenges:
  1. Engine had started to diverge in terms of memory consumption from the xbox 360 days
  2. Lots of shaders were written in pure HLSL

YACCGLO™ (Yet another cross-compiler based on glsl-optimizer)

- Aka. Hellslinger
- DX11 HLSL frontend, Metal and GLSL backends
- Supports full compute, vertex and pixel shaders
- Supports all the various types of buffers/textures
- Separate textures and samplers on Metal
- FP16/FP32 (half/float)
- Cross-compiling all Frostbite shaders
- Outputs source file and json file with reflection meta data

```
Texture2D<float4> inputData : register (t0);

float4 constant;
static float4 global = float4(0.25f, 0.5f, 0.
    75f, 1.0f);

struct Input
{
    uint vertexId : SV_VertexID;
    uint instanceId : SV_InstanceID;
    float4 pos : TEXCOORD0;
};

struct Output
{
    float4 pos : SV_Position;
    float4 data : TEXCOORD0;
    float4 data2 : TEXCOORD1;
};

float getDefaultValue(uint def=8)
{
    return def;
}

Output vsSiggraph(Input input) : SV_Position
{
    Output output = (Output)0;

    output.pos = input.pos;
    output.data = inputData[uint2(input.
        vertexId, input.instanceId)] *
        getDefaultValue() * constant *
        global;

    return output;
}
```

```
// GLSL

uniform sampler2D inputData;
uniform vec4 constant;
layout(location=0) in vec4 TEXCOORD0_in;
out vec4 TEXCOORD0_out;
out vec4 TEXCOORD1_out;
void main ()
{
  uvec2 tmpvar_1;
  tmpvar_1.x = uint(gl_VertexID);
  tmpvar_1.y = uint(gl_InstanceID);
  gl_Position = TEXCOORD0_in;
  TEXCOORD0_out = ((vec4(2.0, 4.0, 6.0, 8.0) *
      constant) * texelFetch (inputData, ivec2(
      tmpvar_1), int(0)));
  TEXCOORD1_out = vec4(0.0, 0.0, 0.0, 0.0);
}

// Metal

struct Input {
    float4 TEXCOORD0_in [[attribute(0)]];
};
struct Output {
    float4 gl_Position [[position]];
    float4 TEXCOORD0_out [[user(TEXCOORD0_out)]];
    float4 TEXCOORD1_out [[user(TEXCOORD1_out)]];
};
struct UniformBuffer {
    float4 constant;
};
vertex Output blsl_main (
    Input inputs [[stage_in]],
    constant UniformBuffer& uniforms [[buffer(0)]],
    uint gl_InstanceID [[instance_id]],
    uint gl_VertexID [[vertex_id]],
    texture2d<float, access::sample>
        texture_inputData [[texture(0)]])
{
  Output outputs;
  uint2 tmpvar_1 = uint2();
  tmpvar_1.x = uint(gl_VertexID);
  tmpvar_1.y = uint(gl_InstanceID);
  outputs.gl_Position = inputs.TEXCOORD0_in;
  outputs.TEXCOORD0_out = ((float4(2.0, 4.0, 6.0, 8.0
      ) * float4(uniforms.constant)) * float4(
      texture_inputData.read(tmpvar_1, 0)));
  outputs.TEXCOORD1_out = float4();
  return outputs;
}
```


```
{
    "reflection" : {
        "compute_info" : {
            "num_threads_x": 0,
            "num_threads_y": 0,
            "num_threads_z": 0
        },
        "textures" : [
            {"name": "inputData", "dst_index": 0, "
                src_index": 0, "src_register_type":
                "t"}
        ],
        "samplers" : [
        ],
        "buffers" : [
            {"name": "uniforms", "dst_index": 0, "
                src_index": 0, "src_register_type":
                "b"}
        ],
        "uniforms" : [
            {"name": "constant", "offset": 0, "size"
                : 16}

        ]
    },
    "metrics" : {
    "alu" : 5,
    "tex" : 1,
    "flow" : 0
    }
}
```

From GL to Metal and back

- Used our Metal experience to improve our OpenGL|ES 3.0 backend
- Spent time aligning the Metal and GL backends with the consoles / PC
- Manage tile memory: glInvalidateFramebuffer, glClear
- Deferred rendering / Forward rendering on all platforms
- Most features on ES, but lower performance

Let there be light

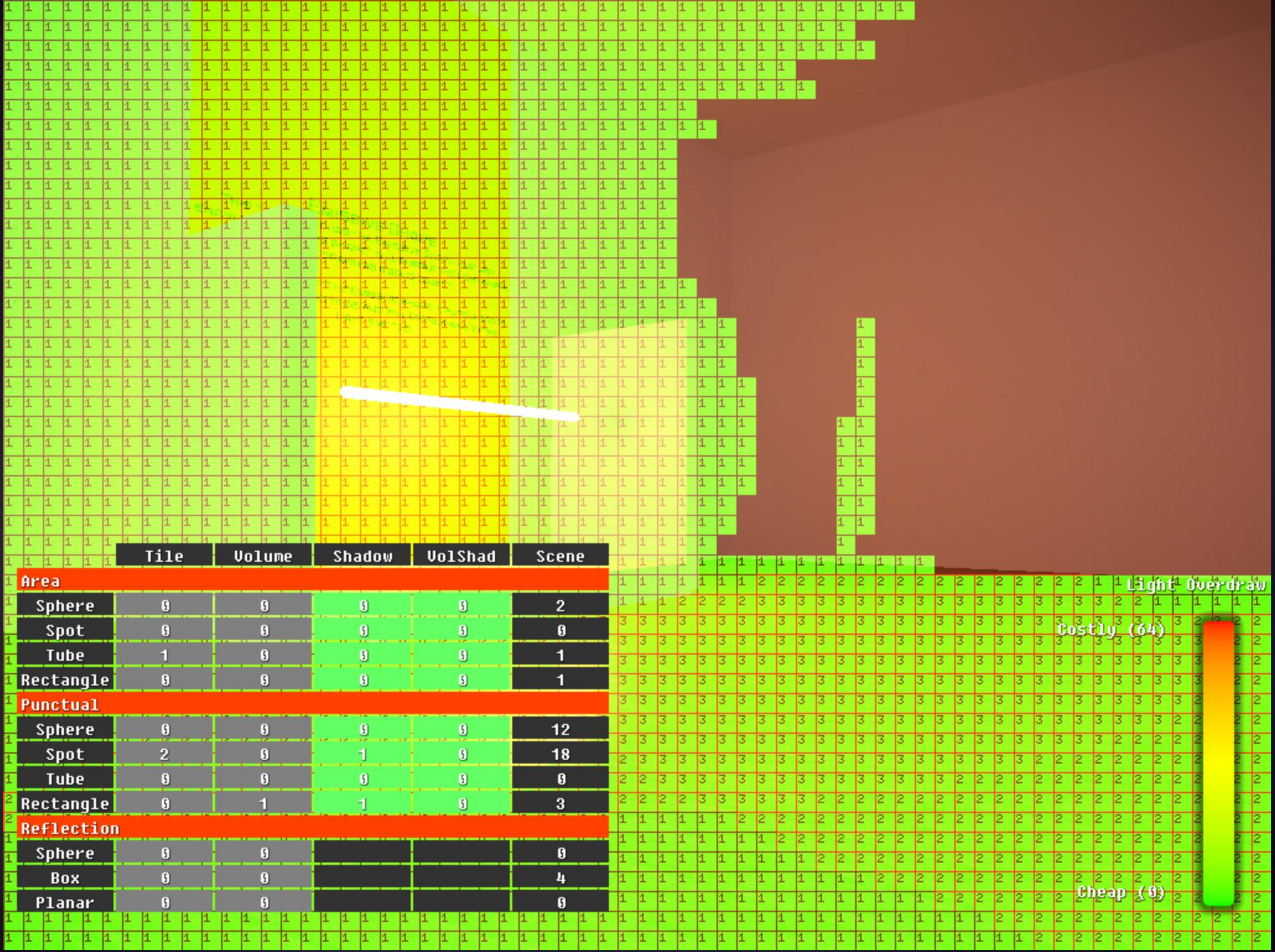
LOTS OF PRETTY PHYSICAL LIGHTS

Let there be light

- Many lights supported using light tiling optimization
  - See “Parallel graphics in Frostbite” SIGGRAPH 09 [1]
- All games moving to Physically Based Rendering
  - See “Moving Frostbite to PBR” SIGGRAPH 14 [2]

Let there be light

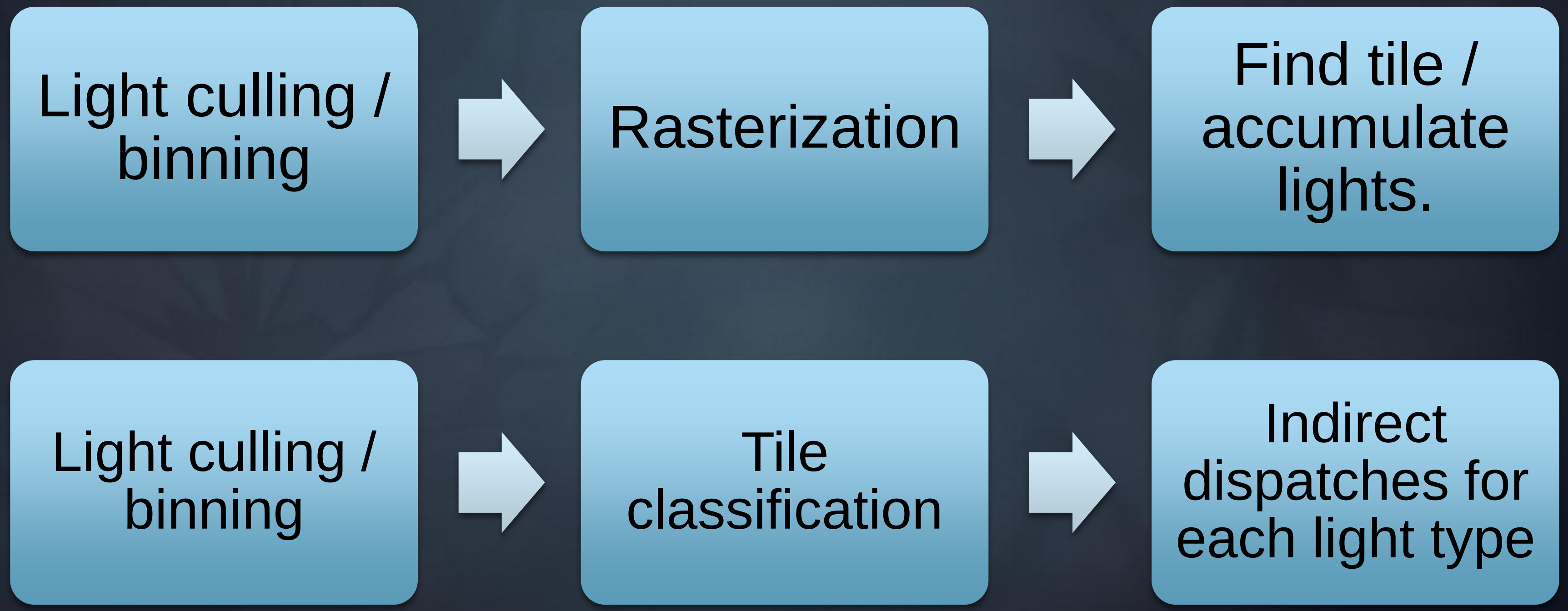
- Light types
  - Point lights, spot lights, area lights, shadow casting equivalents, planar reflections, local reflection volumes

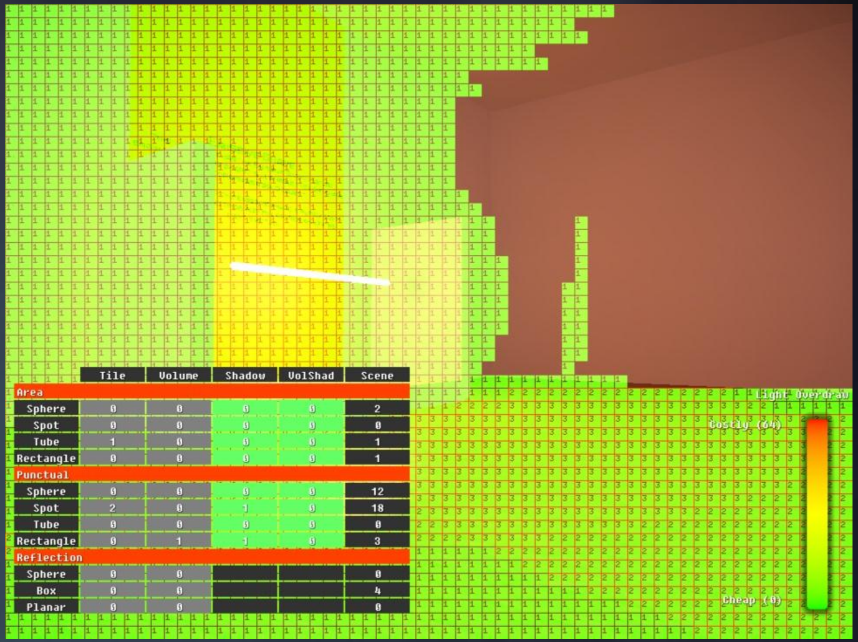


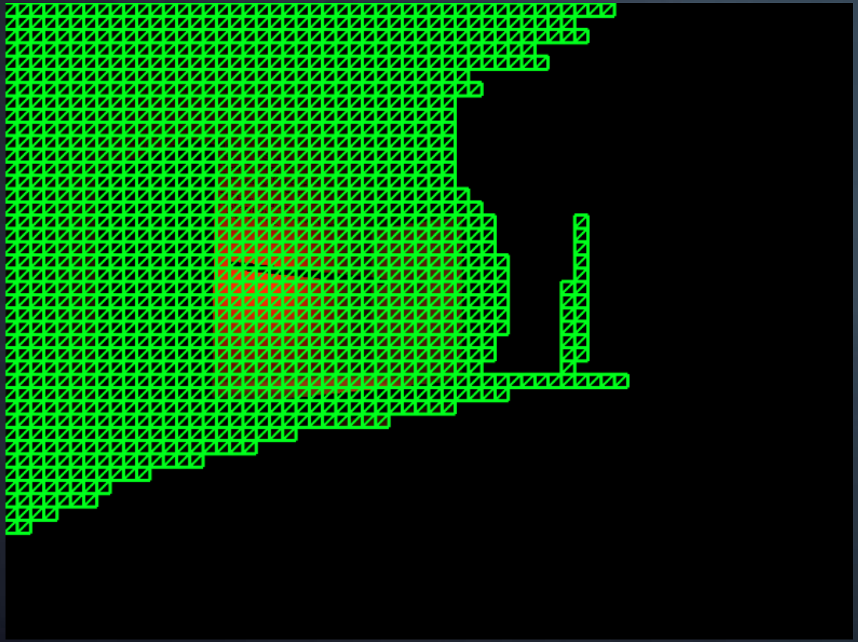
	Tile	Volume	Shadow	VolShad	Scene
Area					
Sphere	0	0	0	0	2
Spot	0	0	0	0	0
Tube	1	0	0	0	1
Rectangle	0	0	0	0	1
Punctual					
Sphere	0	0	0	0	12
Spot	2	0	1	0	18
Tube	0	0	0	0	0
Rectangle	0	1	1	0	3
Reflection					
Sphere	0	0			0
Box	0	0			4
Planar	0	0			0

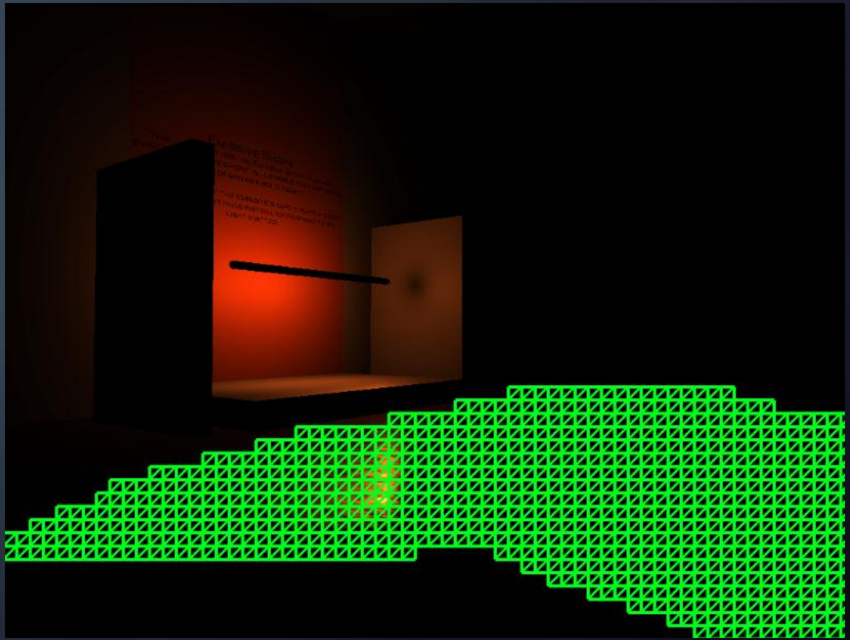
Let there be light

- Light tiling Forward vs Deferred.




Tile Volume Shadow VolShad Scene
Area
Sphere 0 0 0 0 2
Spot 0 0 0 0 0
Tube 1 0 0 0 1
Rectangle 0 0 0 0 1
Punctual
Sphere 0 0 0 0 12
Spot 2 0 1 0 18
Tube 0 0 0 0 0
Rectangle 0 1 1 0 3
Reflection
Sphere 0 0 0
Box 0 0 4
Planar 0 0 0
Light Overdraw
Costly (64)
Cheap (0)





Let there be light

- Cross-compiling many complex shaders
- Compute shaders for light culling / binning
- Toggle between: Deferred / Forward / Forward+

Let there be light

- Cube map arrays for local reflections unw(ra/ar)ped to 2d lat-long texture arrays
- Some alu overhead when sampling, but supports hardware adressing/filtering/mipmaps

```
#if FB_SUPPORT_CUBE_ARRAY

float4 sampleIBLArray(TextureCubeArray tex, SamplerState samplerState, float4 vecAndSlice, float mipLevel)
{
    return tex.SampleLevel(samplerState, vecAndSlice, mipLevel);
}

#else

float4 sampleIBLArray(Texture2DArray tex, SamplerState samplerState, float4 vecAndSlice, float mipLevel)
{
    float3 direction = normalize(vecAndSlice.xyz);
    float2 thetaPhi = cartesianToLatLong(direction);
    return tex.SampleLevel(samplerState, float3(thetaPhi.y/(2.0 * FB_PI), thetaPhi.x/FB_PI, vecAndSlice.w), mipLevel);
}

#endif
```

Let there be light

- Rewrote deferred light accumulation from cs to vs/ps
- Accumulate lights in tile memory \0/
- No indirect drawcalls/dispatches on Metal -> emulated using early out vertex shader

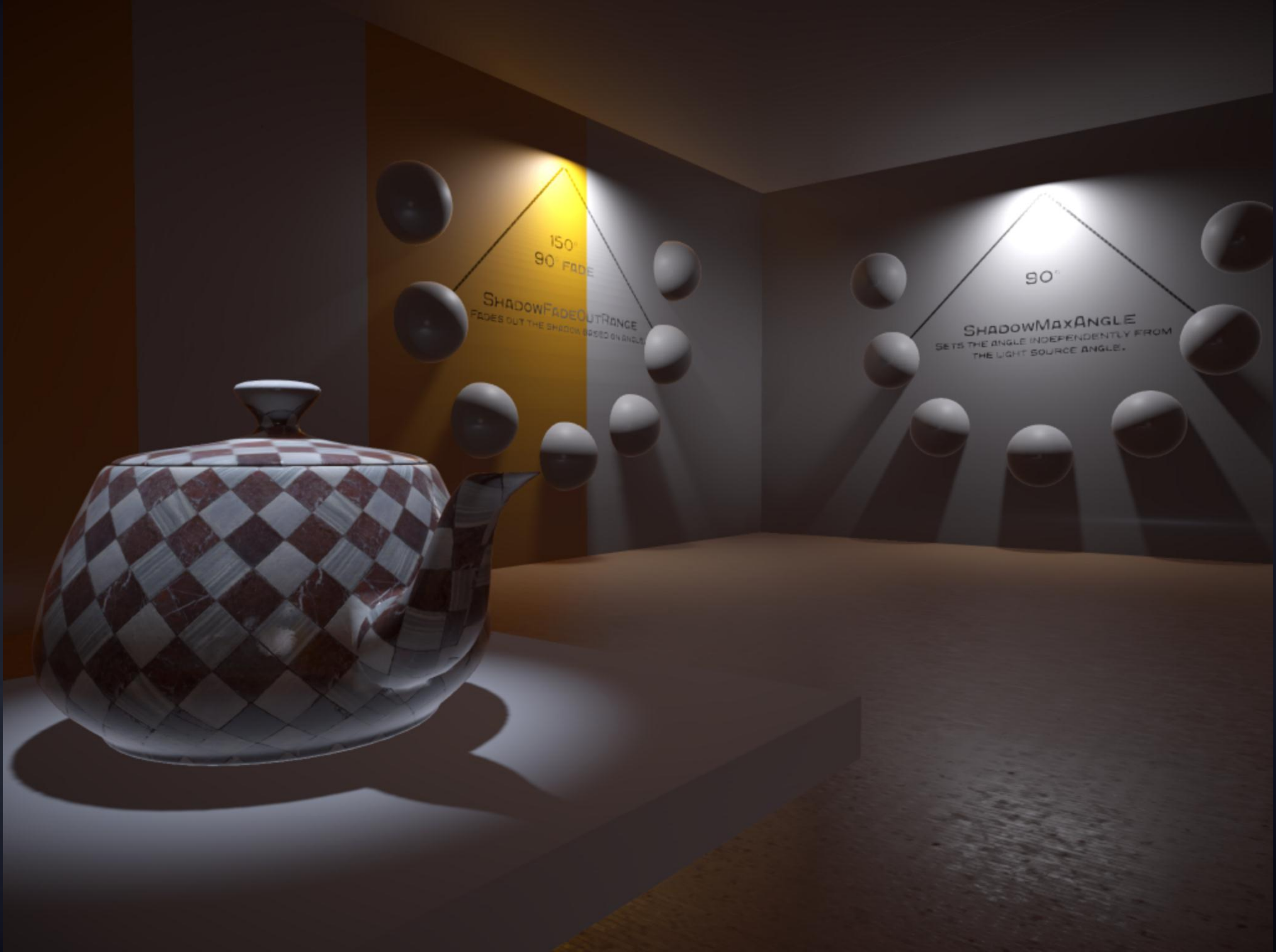
```
#if defined(VS)
VsTiledLightingOutput vsTiledLighting(uint tile : SV_InstanceID,
                                      uint tileCorner : SV_VertexID)
{
    VsTiledLightingOutput output = (VsTiledLightingOutput)0;

// No indirect on Metal.
#if defined(RENDERPATH_METAL)
    if (tile >= g_indirectArgs[0])
    {
        output.pos = float4(-1, -1, -1, 1);
        return output;
    }
#endif

    uint texCoordPacked = g_compactTileGridBuffer[tile];
    uint2 texCoordSrc = uint2(texCoordPacked >> 16, texCoordPacked & 0xFFFF);

    uint structuredBufferOffset = screenspaceCoordToLightGridOffset(texCoordSrc, g_tileResolutionX);
    uint4 packedLightGrid = getLightCullInput(g_lightCullInput, structuredBufferOffset);

    ....
```


150°
90° FADE
SHADOWFADEOUTRANGE
FADES OUT THE SHADOW BASED ON ANGLE
90°
SHADOWMAXANGLE
SETS THE ANGLE INDEPENDENTLY FROM
THE LIGHT SOURCE ANGLE.

Optimizations

- Backend optimizations
  - Expose tiler hint api and use a lot (nop:s on non-tilers)
  - Merge as many render passes as possible
  - Reduce state changes

```
Color framebuffer loads: 18, 47.250 mb
Color framebuffer stores: 39, 70.664 mb
Depth framebuffer loads: 10, 27.750 mb
Depth framebuffer stores: 14, 36.422 mb
Stencil framebuffer loads: 8, 6.000 mb
Stencil framebuffer stores: 10, 7.500 mb
Total framebuffer loads: 36, 81.000 mb
Total framebuffer stores: 63, 114.586 mb
Total framebuffer loads/stores: 99, 195.586 mb
```

Auto merging

```
Color framebuffer loads: 10, 23.250 mb
Color framebuffer stores: 31, 46.664 mb
Depth framebuffer loads: 9, 24.750 mb
Depth framebuffer stores: 13, 33.422 mb
Stencil framebuffer loads: 7, 5.250 mb
Stencil framebuffer stores: 9, 6.750 mb
Total framebuffer loads: 26, 53.250 mb
Total framebuffer stores: 53, 86.836 mb
Total framebuffer loads/stores: 79, 140.086 mb
```

Optimizations

- Shader code
  - Use as many intrinsics / builtins as possible
  - Use scalar math
  - Careful packing / alignment of data

Summary and future work

WHERE DO WE GO NOW?

Summary and future work.

- Our approach: Get the full picture before diving into details
- Today’s mobile hardware and api:s enables full engine feature set
- Many tile memory specific optimizations can be done without diverging code bases from desktop / consoles
- If building for multiple platforms, use cross-compiler

Summary and future work.

- New API:s: Vulkan / ES 3.1, spir-v
- Tiler specific shader optimizations (deferred shading)
  - “Efficient Rendering with tile local storage” [3]
- Mobile specific shader optimizations (fp16/fp32 usage, alu / bandwidth balance)
- Future: Tesselation, async compute, indirect

The end

"The best way to predict the future is to invent it"

– Alan Kay / Abraham Lincoln / Peter F. Drucker

Thanks for feedback: Johan Andersson, Kristoffer Benjaminsson, Yuriy O'Donnell, Mathieu Guindon, Sébastien Hillaire, Sam Martin

For questions: niklas.nummelin@frostbite.com

References

- [1] Johan Andersson. “Parallel graphics in Frostbite – Current and Future”. Siggraph 2009.
- [2] Charles de Rousiers & Sébastien Lagarde. “Moving Frostbite to PBR”. Siggraph 2014.
- [3] M. Bjorge, S. Martin, S. Kakarlapudi, J-H. Fredriksen. “Efficient rendering with tile local storage”. Siggraph 2014.

FROSTBITE™

empowers game creators to shape the future of gaming