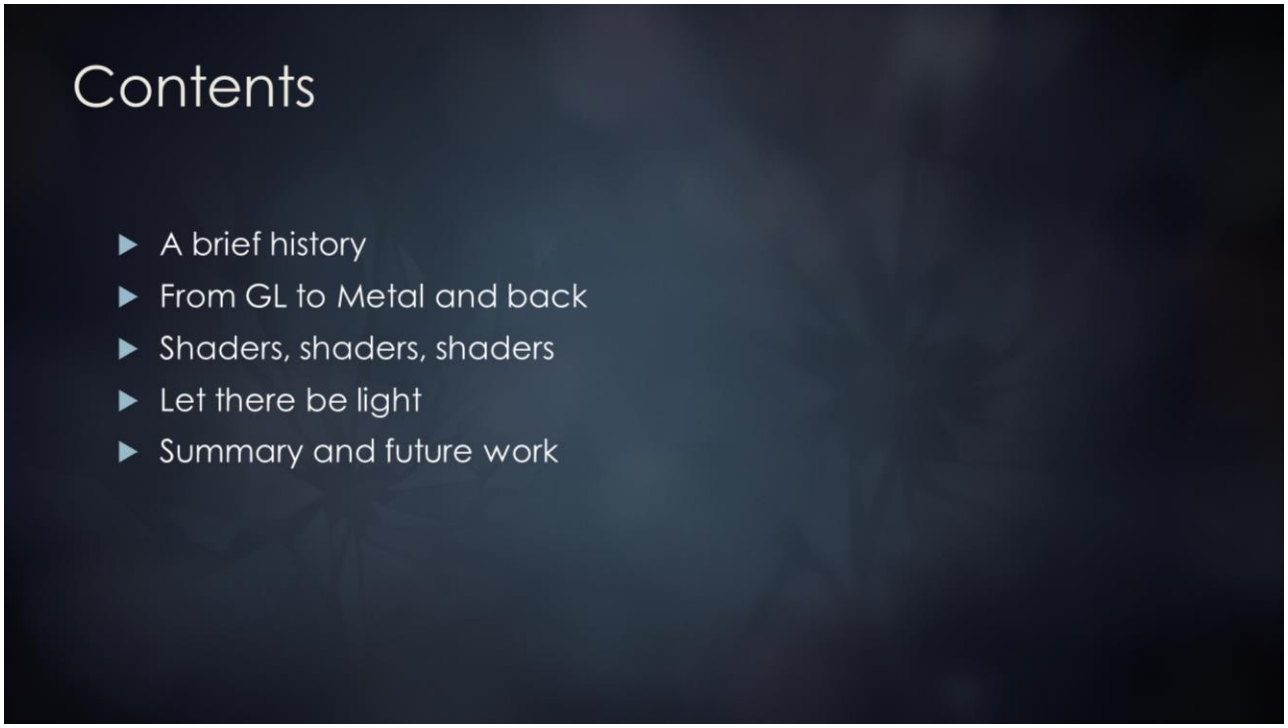
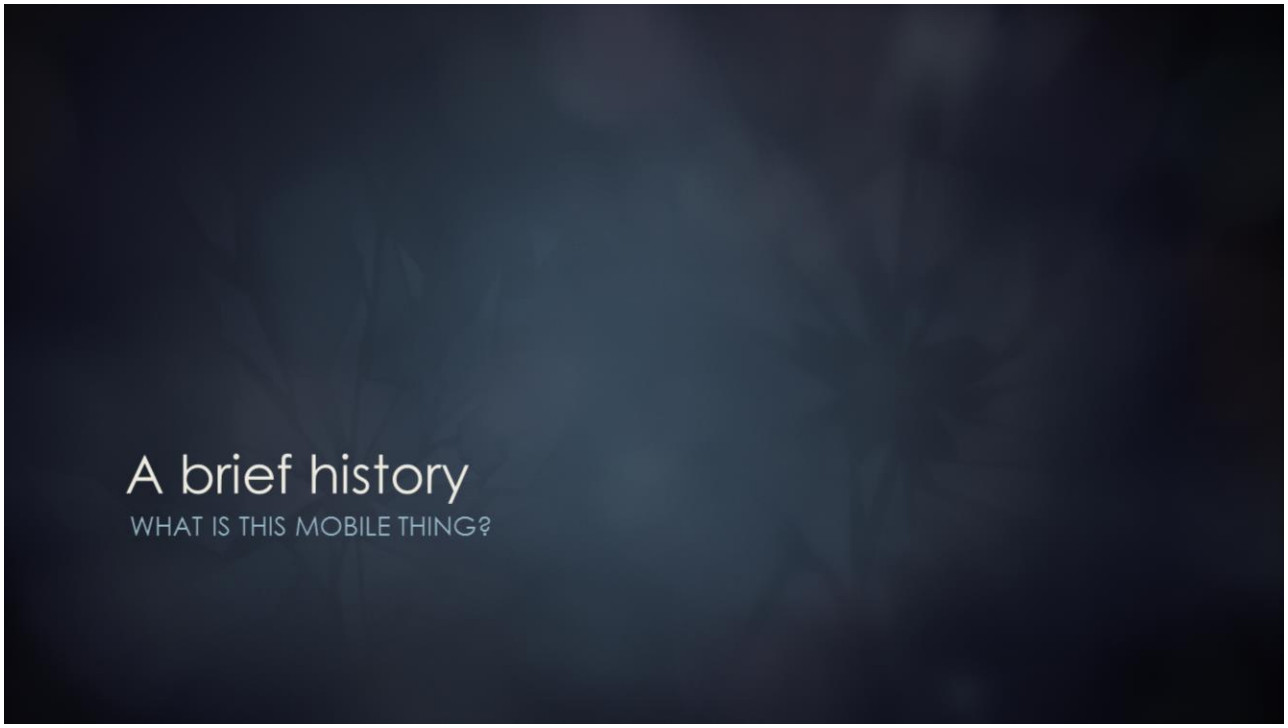

SIGGRAPH2015
Xroads of Discovery


SIGGRAPH2015
Xroads of Discovery
The 42nd International Conference and Exhibition
on Computer Graphics and Interactive Techniques
Frostbite on mobile
Niklas Nummelin
Electronic Arts / Frostbite

Hi everyone! My name is Niklas and I've been working at EA for the last 2.5 years. My main responsibility there have been to bring up the graphics side of the Frostbite engine on mobile devices.

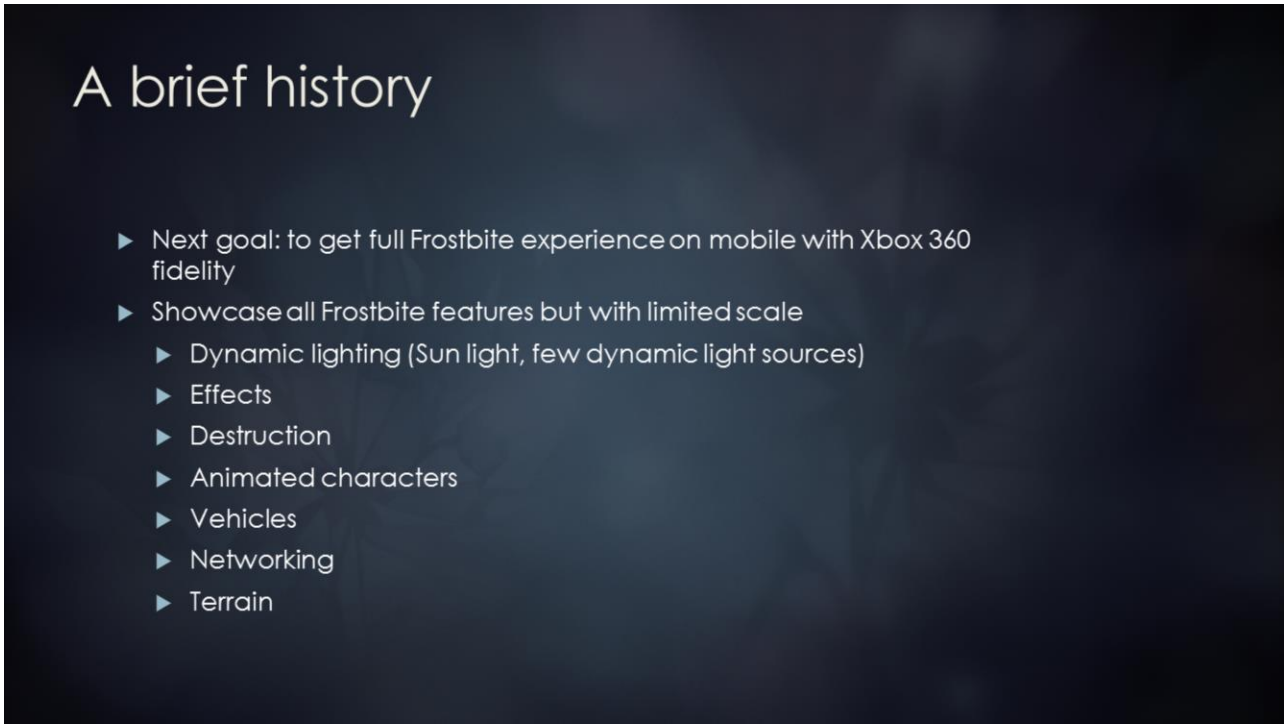


This talk will focus on our journey so far. I'll start off by running through some history. Then talk about what we've done the last year and share some of our findings.

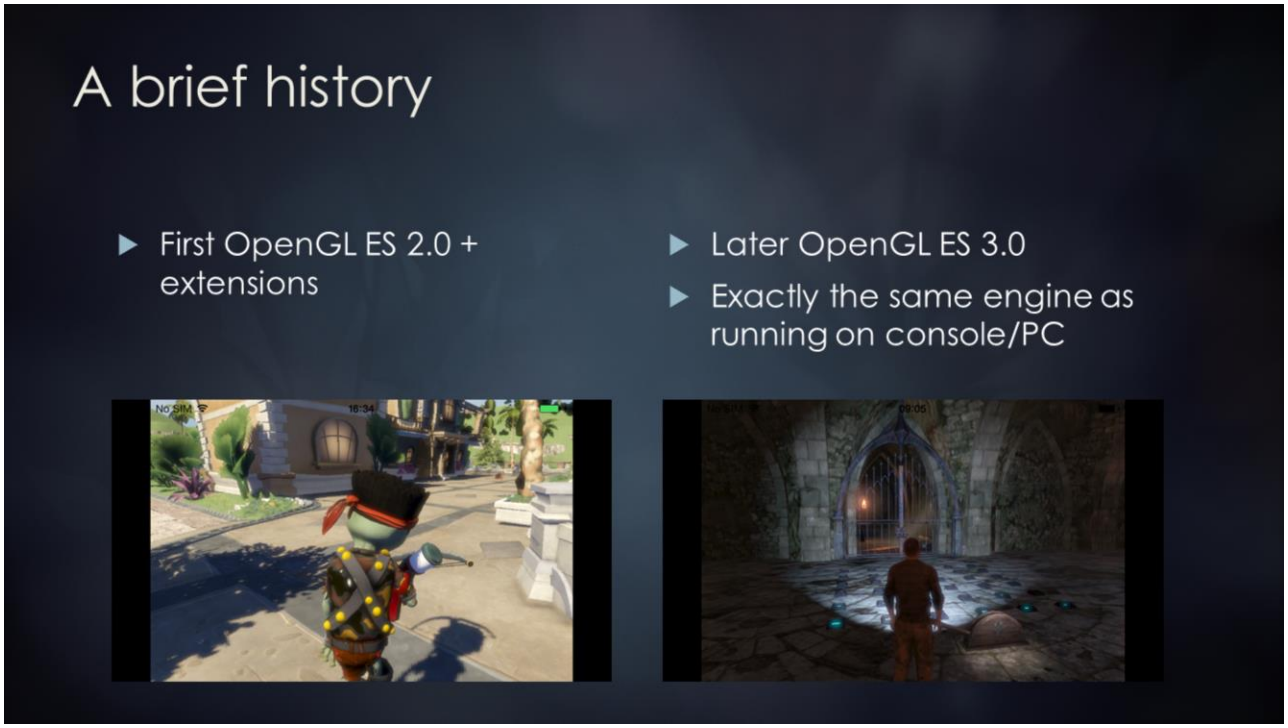

A brief history
WHAT IS THIS MOBILE THING?

When I started 2.5 years ago, Battlefield 4 was in full production. We were a small team consisting on 4 people that had taken on the task to get selected parts of Frostbite running on tablets. The pilot project that we were building this for was Battlefield 4 Tablet Commander, which was a game where you could join battlefield servers and play as a commander.

This is what it ended up looking like. The commander mode had actually been available in earlier battlefield games, but was brought back into Battlefield 4. By supporting Frostbite on tablets we could re-use large parts of that code, making the time to build the game a lot shorter.



While doing this we had started to develop a more long-term plan for what Frostbite on Mobile could be in the future. We had the strong feeling that we could bring the full frostbite experience to mobile with xbox 360 fidelity within a year, so we set out our next goal. Also we wanted to see if the dynamic nature of the Frostbite engine could play well on mobile and as the engine already scaled from 360 to gen4 console and PC quality, we predicted this should be possible.

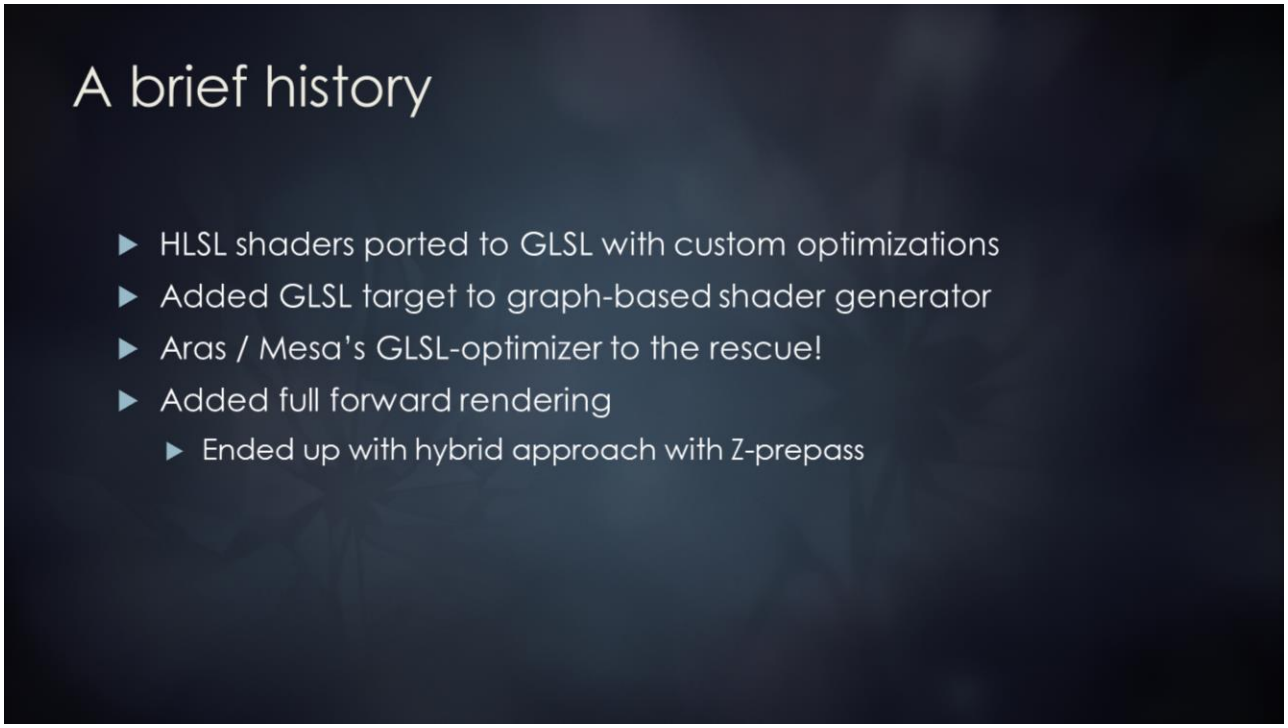


At the time we started development were still limited by OpenGL|ES 2.0, but with a wide variety of extensions.

We got a lot of functionality up, adding support for extensions to utilize hardware shadow sampling and instancing.
It was hard to keep all the various devices running at all times due to the fragmentation landscape and quality of driver implementations.

Then came OpenGL|ES 3.0 and we could almost remove all of the extensions we used and just use plain OpenGL|ES 3.0. Compatibiilty started to become much better.

As we didn’t have a legacy to maintain, we placed our bets on ES3 and beyond. Similar to how we bet on DX10 early for PC.



So to understand what kind of work we had to do:
Frostbite have two main types of shaders.
Shader programs, which are hand-written shaders that are written by programmers for solving a specific task.
Surface shaders, which are shaders represented by graphs authored by artists in our editor usually used as materials for objects.

All the hand-written shaders, like post processing etc, we manually ported to GLSL as it left us the opportunity to optimize them
fully for our targets.

We did a lot of optimizations like removing dependant texture reads and moving heavy computations out to the vertex shaders.
For the surface material shaders we just added GLSL emitting code to our shader generator.

As many others we found this nice tool by Aras of Unity called GLSL-optimizer and made use of that to strip our final code down to reasonable sizes.

One of the biggest changes to the engine we had to do then was to add support for forward rendering of both opaque and transparent objects. The engine had been exclusively using a full deferred renderer on all platforms,

but due to the bandwidth restrictions on mobile devices we had to add support for forward rendering.

We got the best performance by using a hybrid approach where we did a z prepass and used that depth buffer for calculating all screen space depth buffer effects, such as accumulating the shadow map cascades in a screen space shadow buffer, computing ssao and downsampling depth buffer to a half-res depth buffer for the half-res transparency pass. The biggest benefit here was that we reduced the bandwidth by not having to store / load the large hdr buffer to/from the tile memory after the opaque pass, but could instead continue with the transparent passes keeping data in tile memory as long as possible.

Everything was looking pretty promising but we were still lacking features that the engine had started to rely heavily on.

Then about a year ago we got the opportunity to test pilot Metal. The plan was to have demo for WWDC so we went there and luckily we finished it in time.

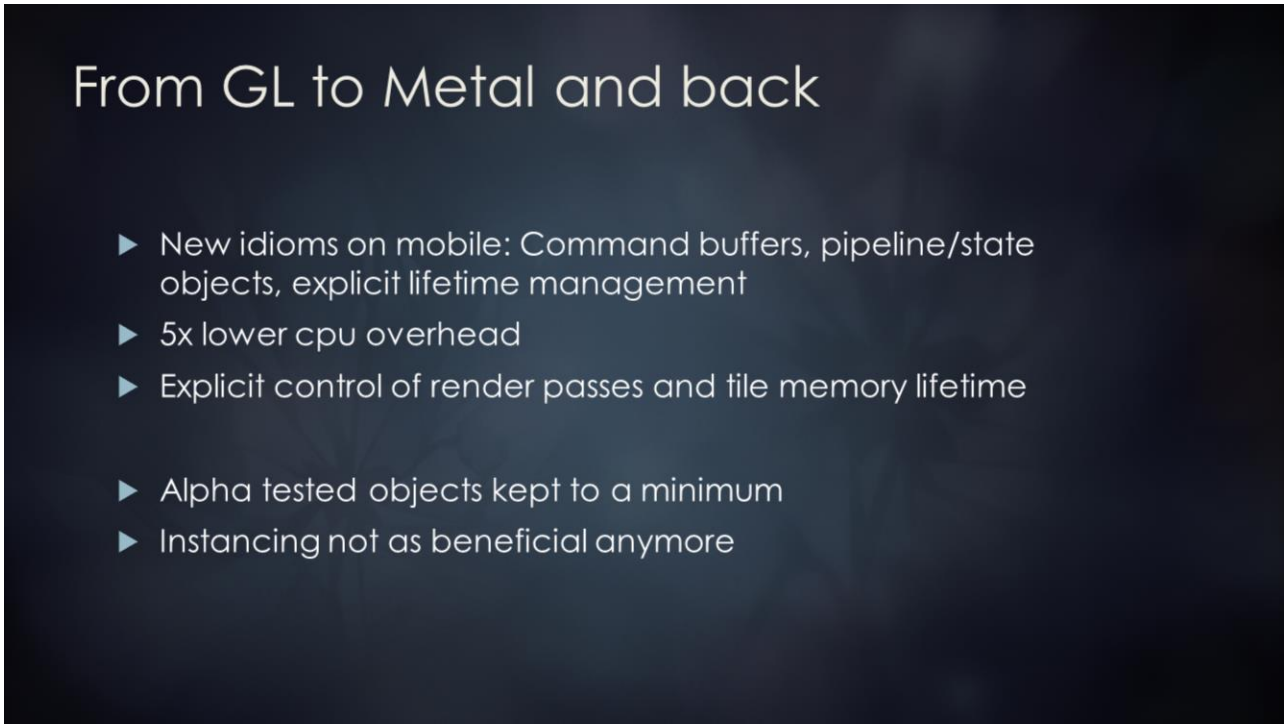
This a capture from the demo we showed at WWDC.

We went to Cupertino and worked on this for a number of weeks together with a small team from Popcap and our co-workers at home.

As we already used the glsl-optimizer compiler, we quickly realized that the easiest way to add support for all the shaders we needed would be to add a backend that generated Metal code.

After implementing the Metal API and creating the compiler backend all of this turned out to be working really well and we could toggle on all the features needed.

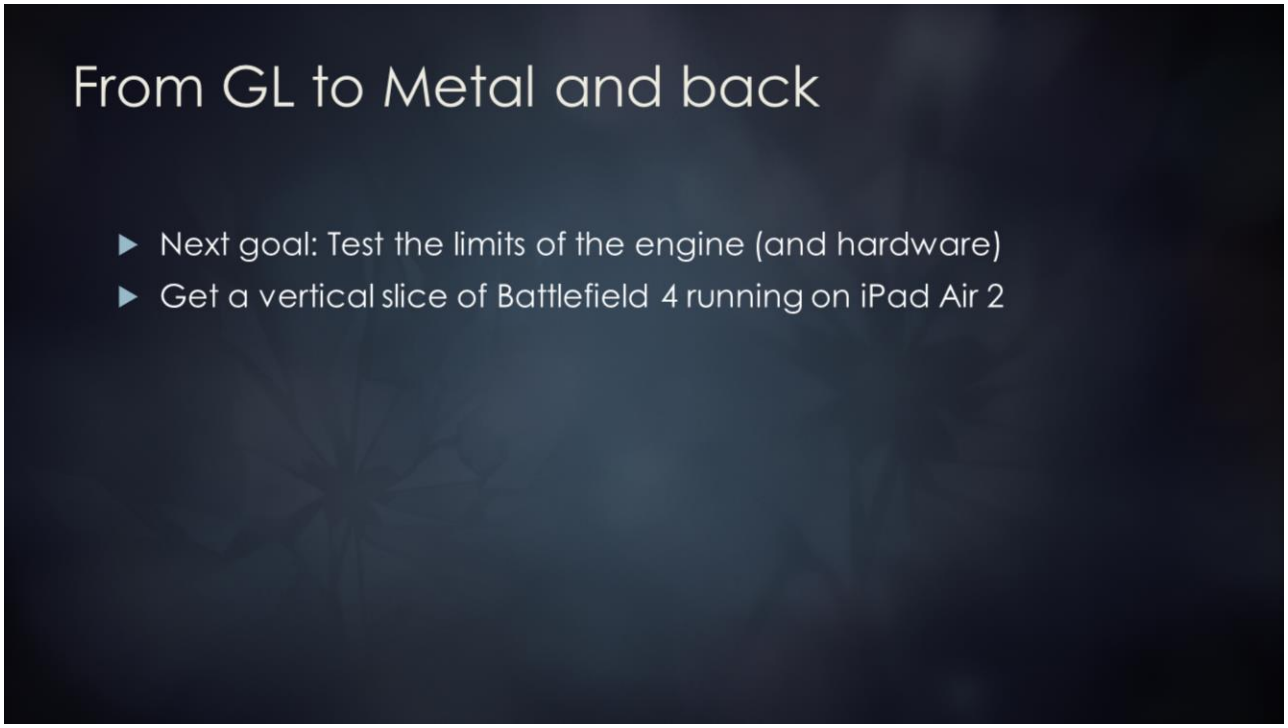
It enabled us to reach steady 30 fps with SSAO, DOF, Radiosity, cascaded shadowmaps, HDR color grading, FXAA and lots of draw calls on a stock iPad Air 1.



Apple had taken a lot of inspiration from how API:s were starting to be designed for the traditional platforms and put these idioms on to mobile. They also added new mobile specific concepts like being able to control the lifetime of renderpasses and tile memory. The Metal API turned out to be a good fit for Frostbite as the engine had continously been moving towards more explicit graphics api:s and memory management.

Due to the low-overhead nature of the api we could see a 5x lower cpu overhead than what we had with OpenGL. Also using the api for controlling the tile memory lifetime we could guarantee our bandwidth from/to tile-memory was kept to a minimum.

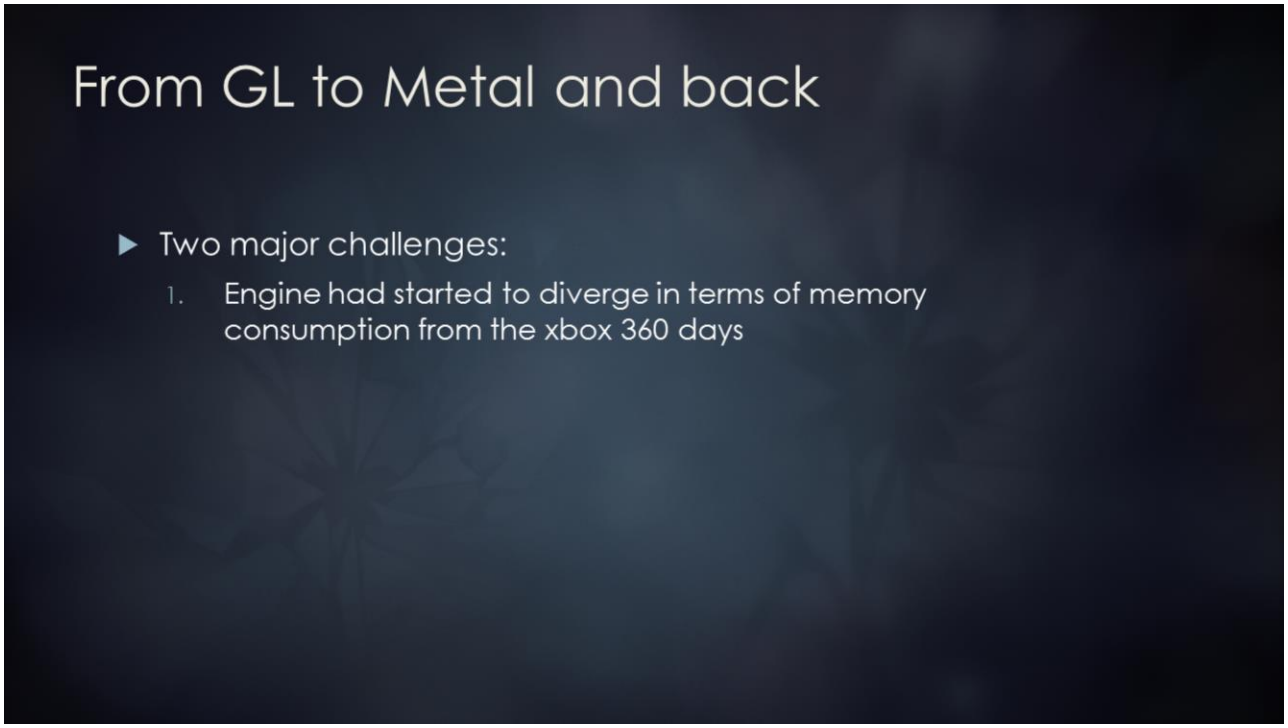
However the gpu bottlenecks stayed the same. We had to reduce our amount of alpha tested objects to a minimum. Also, due to the reduced cpu overhead, instancing didn't provide us with any major benefits in terms of performance on the cpu/gpu. At least not for the dataset for the demo.



At this point we were very excited to see how far we could push our Metal backend and the hardware, so we set out a new goal. This time pretty ambitious. We decided that we would try to get a vertical slice of Battlefield 4 up and running on an iPad Air 2 that had just been released.

And we succeeded. The most complex new features we got up and running was our advanced terrain system that uses a virtual texture to be able to visualize large terrains and our mesh scattering system used to distribute and render vegetation and rocks.

As the content still had quality levels for the old consoles we could piggy back on that to get suitable levels of details. We got this running in 30 fps on an iPad Air 2.

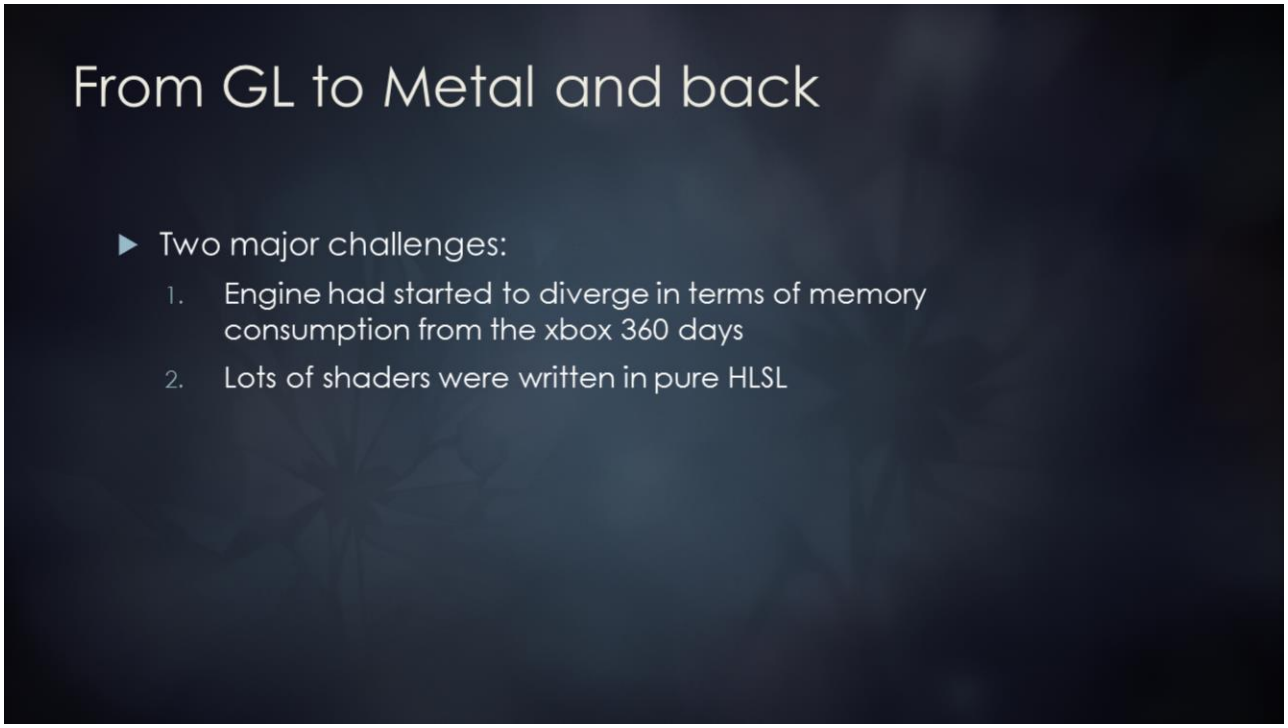


Doing this we had two major challenges. Ps4 and Xbox one had become the primary engine targets with xbox360/ps3 being removed from the engine. Keeping memory consumption down wasn't everyone's top priority anymore. Also, we had never tried to run anything of this size on our tablets so we were very concerned about memory.

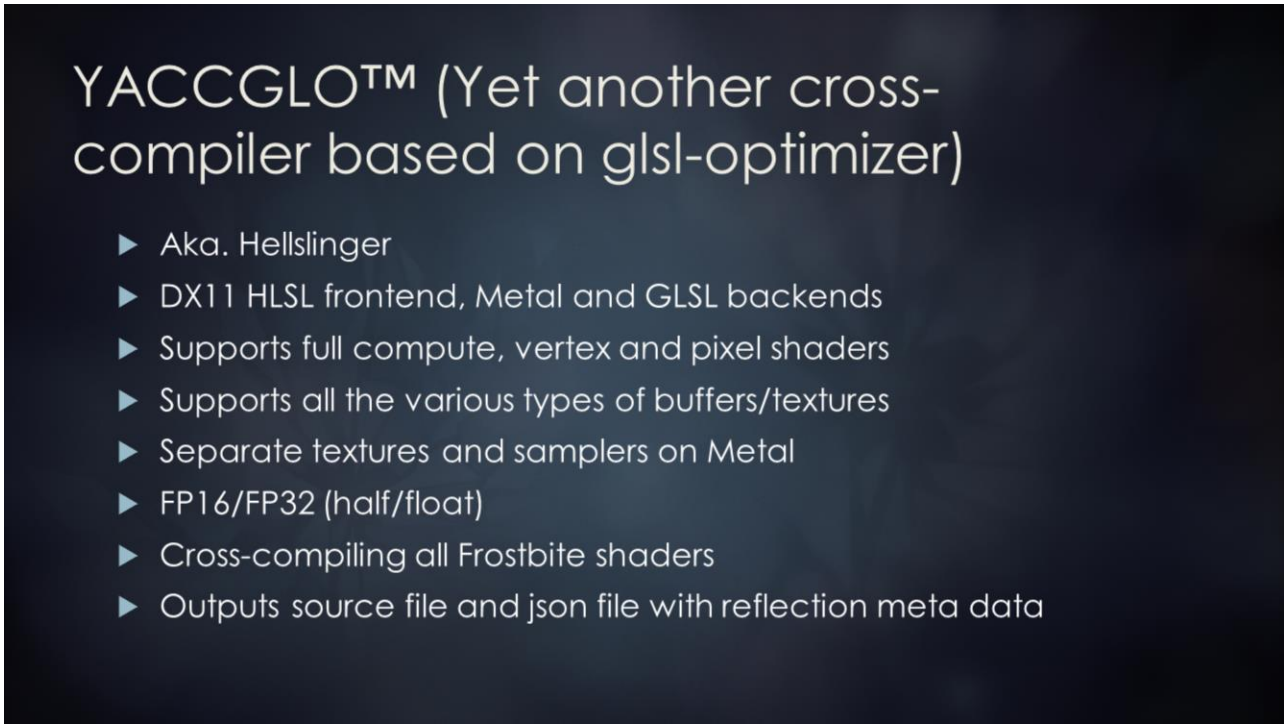
However as we knew BF4 shipped on 512 MB consoles we were hopeful.

To tackle this we added fine grained statistics for our gpu textures / buffers. The engine have tags for all the resources, so we added commands for printing all the resources by size and usage frequency to easily see what we should focus on and could get rid of. Here's an image of a statistics view that we've used quite a lot to keep track and get an overview of our resources. Due to the various quality of native profilers and the difficulties of setting up environments for them we wanted to be able to profile as much as possible with our own cross platform code

We had to compress all the textures, which took many hours for a single level. The virtual terrain texture however, as it was built in runtime and we had no suitable runtime compressor for mobiles, needed to be stored raw and took almost 100 mb.

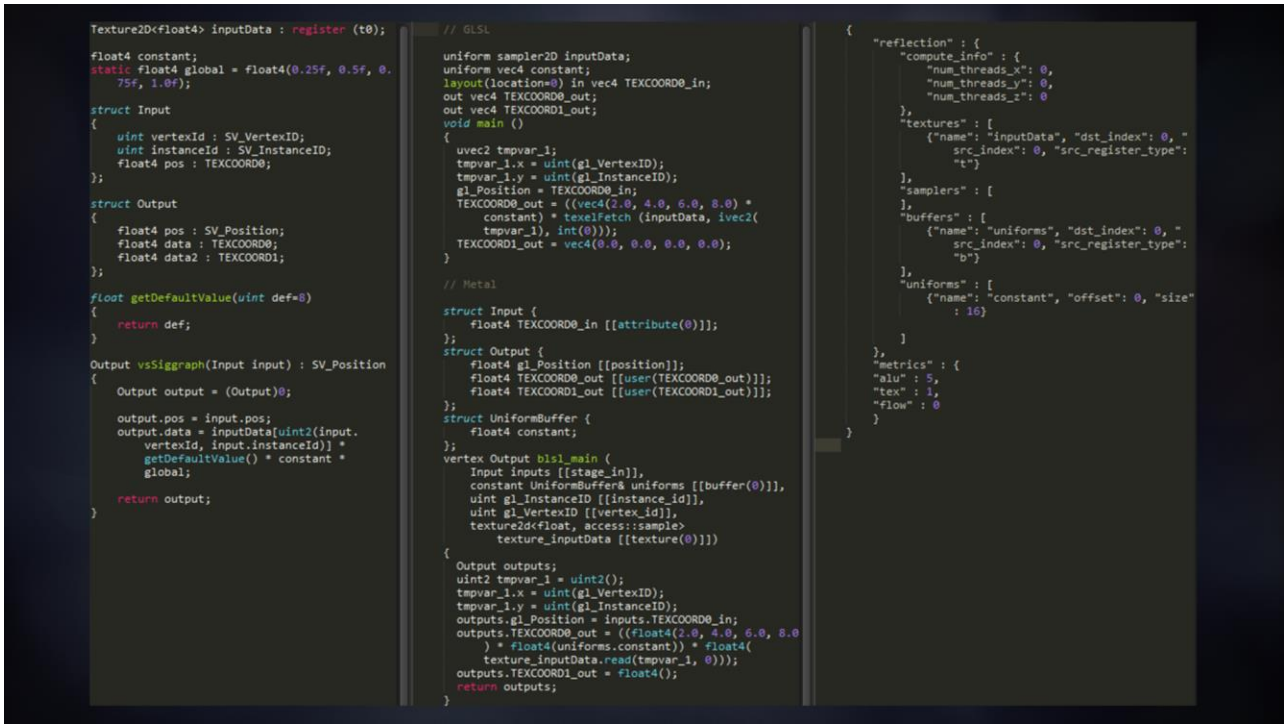


The biggest challenge was the fact that a lot of custom shaders were written in pure HLSL. We could of course just have ported all shaders to GLSL as we did previously, but as HLSL was the main shader language of the engine and we had noticed it is a pain to maintain multiple versions of shaders we decided it was time to revisit the way we manage shaders on mobile. Also the shader languages of the mobile platforms had matured.



As we already had backends for glsl-optimizer it made sense to use that as a basis for a cross-compiler. We had a look at the compiler source code and decided it would be possible to create a decent cross-compiler by adding a frontend that parses HLSL and some code to handle unique HLSL features. A few months later that was done.

It is a DX11 style syntax compiler that supports compiling all of our shaders, including compute shaders. The result of a compilation is a Metal or GLSL source file, with a json file containing reflection such as constant buffer names to offsets/sizes, resource mappings and number of compute threads per group.



```
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 bls1_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;
}
```

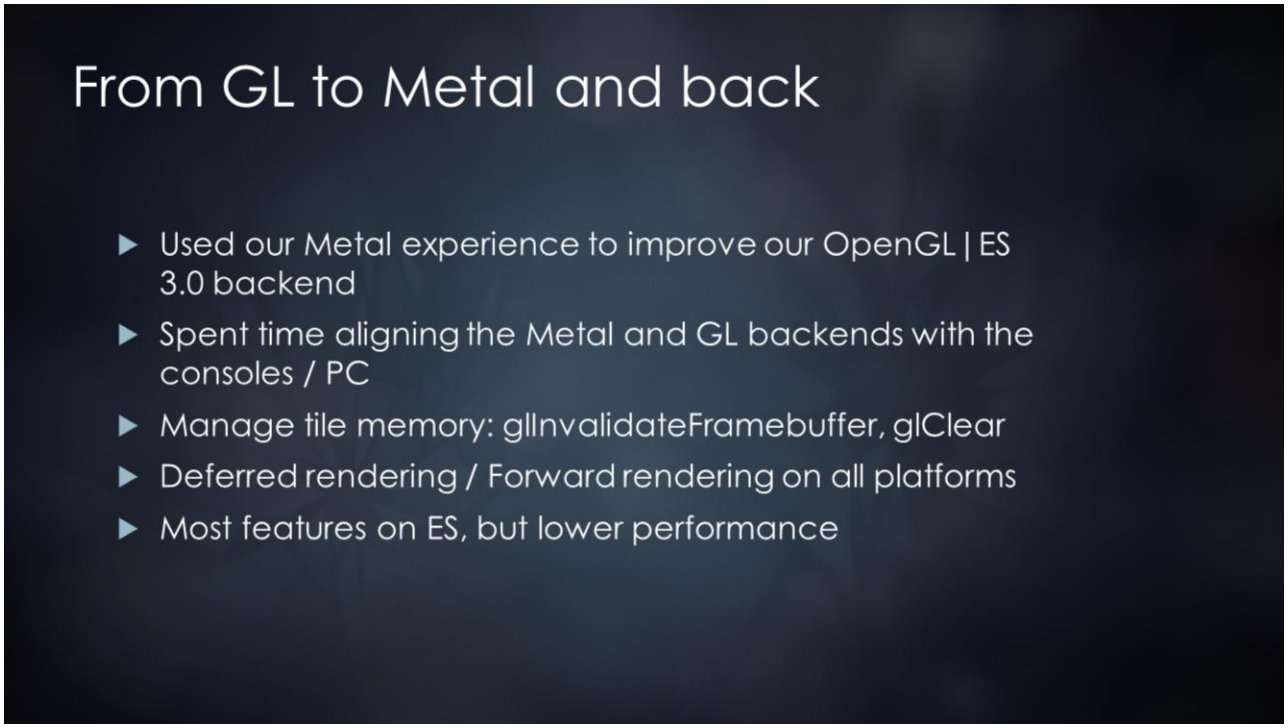
```
{
    "reflection" : {
        "compute_info" : {
            "num_threads_x": 0,
            "num_threads_y": 0,
            "num_threads_z": 0
        },
        "textures" : [
            {"name": "inputData", "dst_index": 0, "
                src_index": 0, "src_register_type":
                "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
    }
}
```

Here’s an image showing some of these features. It handles implicit type conversions, function default parameters, HLSL semantics, HLSL matrix syntax etc.

HLSL register name to actual resource index is provided in the reflection data.

Not shown here is support for compute shader writeable buffers, atomics and append/consume buffers through buffers + atomics.

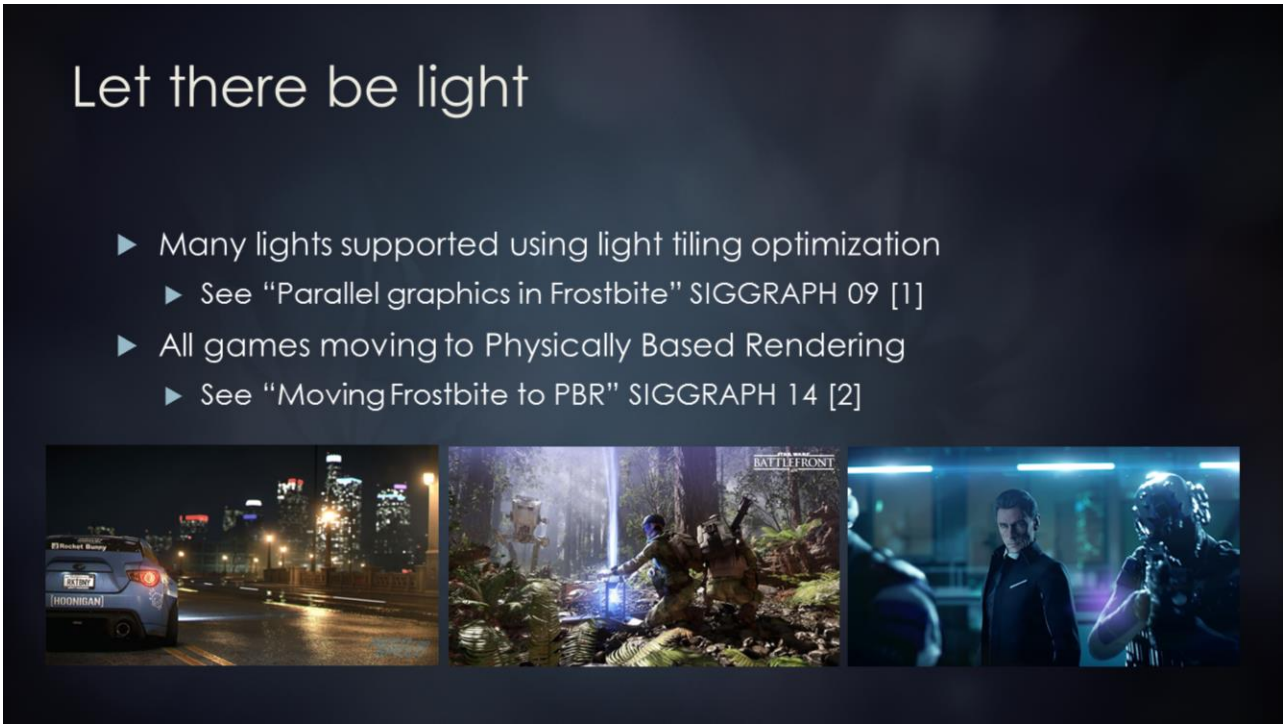
One nice side effect of having our own compiler is that we have been able to make workarounds for vendor specific bugs in the backends of our compiler.



With all the focus on Metal, we realised we could improve our OpenGL|ES backend, so we spent time fixing bugs and aligning the platforms even more. Metal, being a pretty transparent api, had learnt us a lot of how mobile GPU:s should be treated. We used this knowledge to refine our OpenGL|ES implementation. For instance when we created the Metal backend we added an explicit api for controlling if the frame buffer textures should be loaded to tile memory and stored back. We had started to make heavy use of this api in the code to reduce bandwidth. This optimization could be transitioned to ES by just implementing that api using glInvalidateFramebuffer and glClear.

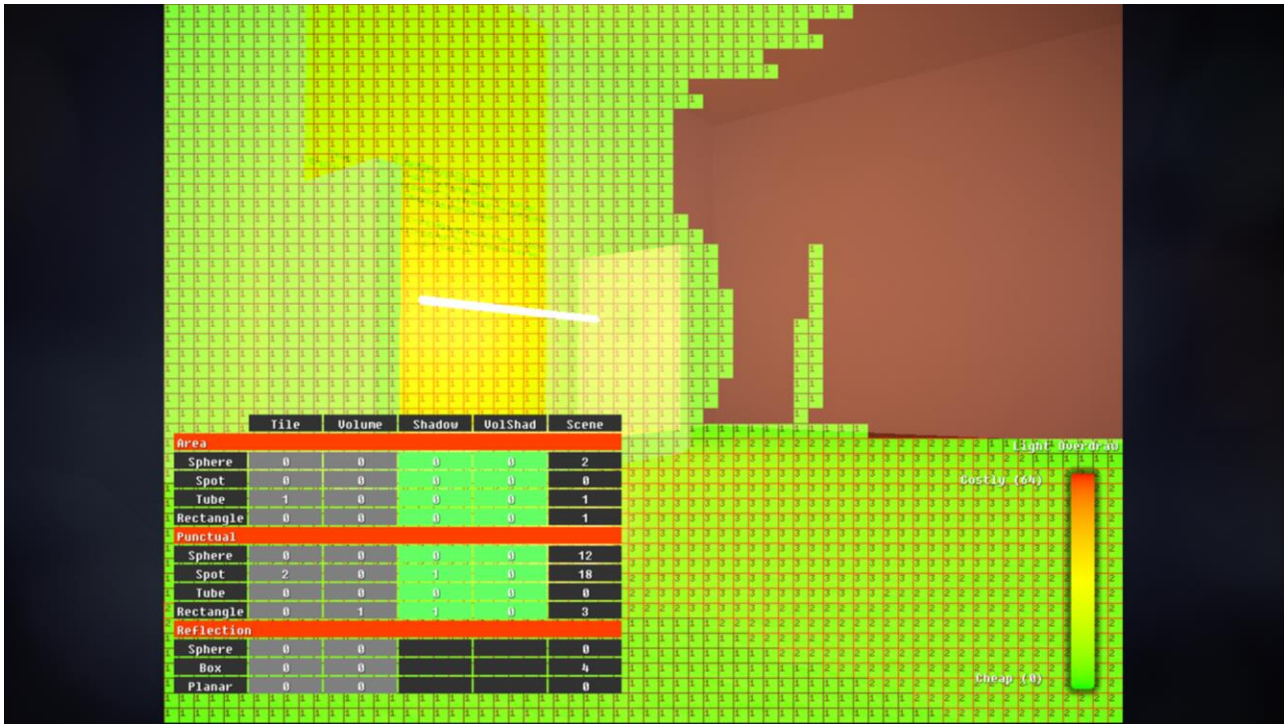
We also made an effort into supporting switching deferred and forward rendering in runtime for all our platforms, to be able to compare performance and try out features in both modes.

At this point the PBR transition of frostbite was nearly done. With everything we had done to close the gap between mobile and console/desktop, PBR and dynamic lighting seemed to be a pretty low hanging fruit. Also, all the Frostbite games were transitioning to our new PBR system for managing our materials and lights so we needed it.



There's a few good talks going into details of how this has been achieved in the engine. The images here shows some upcoming games and from now on all the Frostbite games are Gen4 console / PC only.

The engine supports multiple types of lights including lights with shadowmaps, planar reflections and local reflections through cube maps. Multiple local reflections can be baked in runtime or offline and stored in a database and loaded on demand into a texture array.

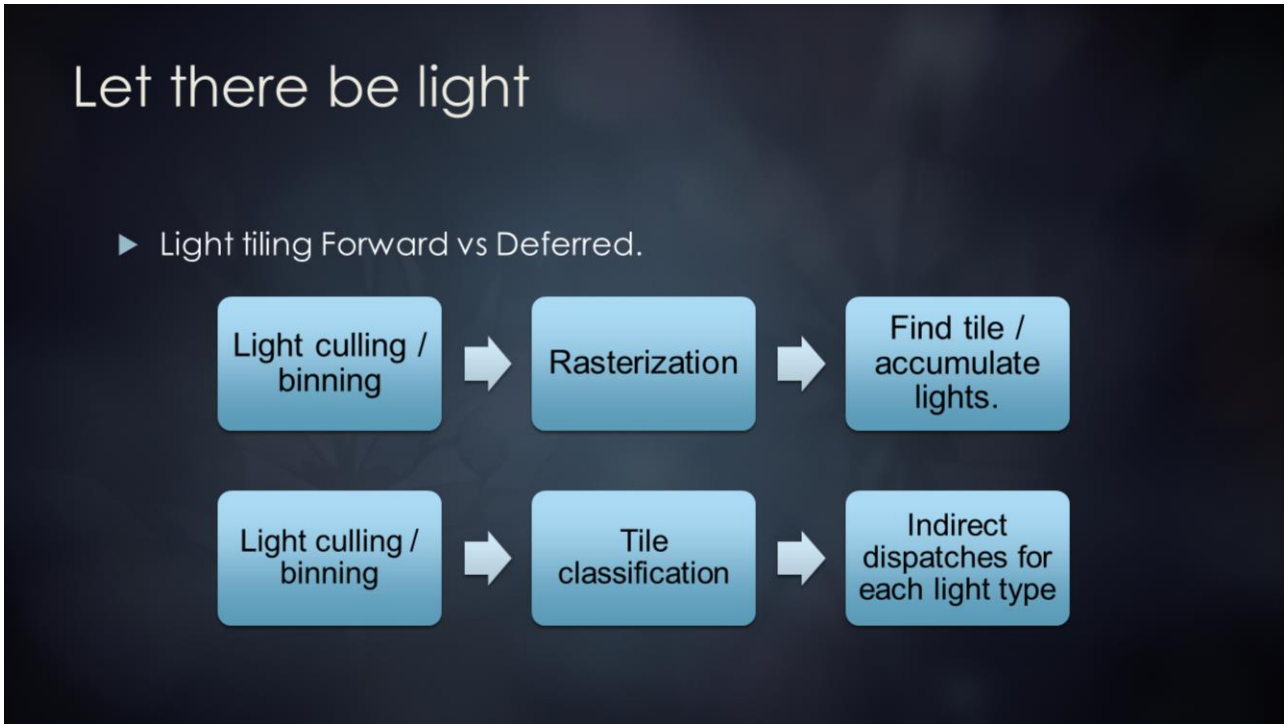


In order to be able to render many of these lights efficiently we need to create a spatial data structure.

We do this by taking the set of lights that are inside the view frustum and perform a pass that bins them into 2d-tiles, usually 16x16 pixels.
It tries to eliminate as many lights as possible by using various culling techniques against each tile frustum and a coarse z-buffer consisting of the min and max z per tile.

The result of this pass is a 2d-texture or buffer, containing light counts for each light type and an offset into a buffer of indices that in turn references the actual light data.

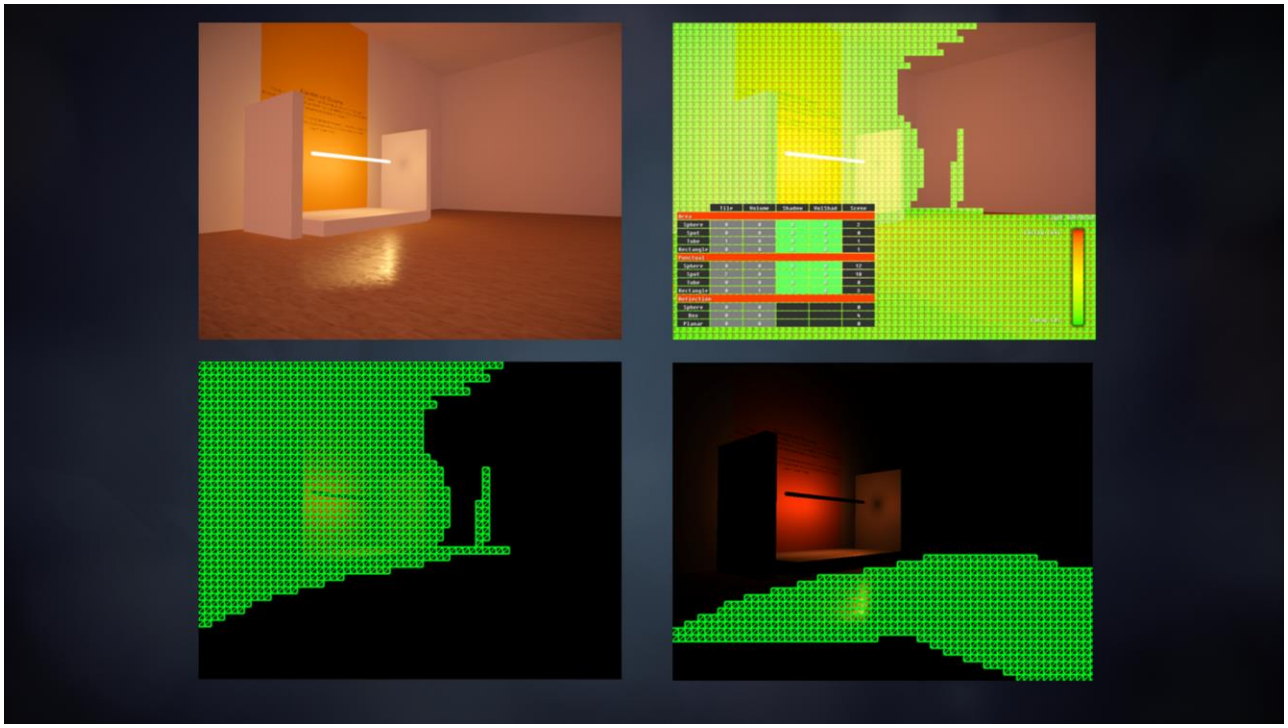
This extra indirection is used to reduce the amount of memory and copies needed for the light data.



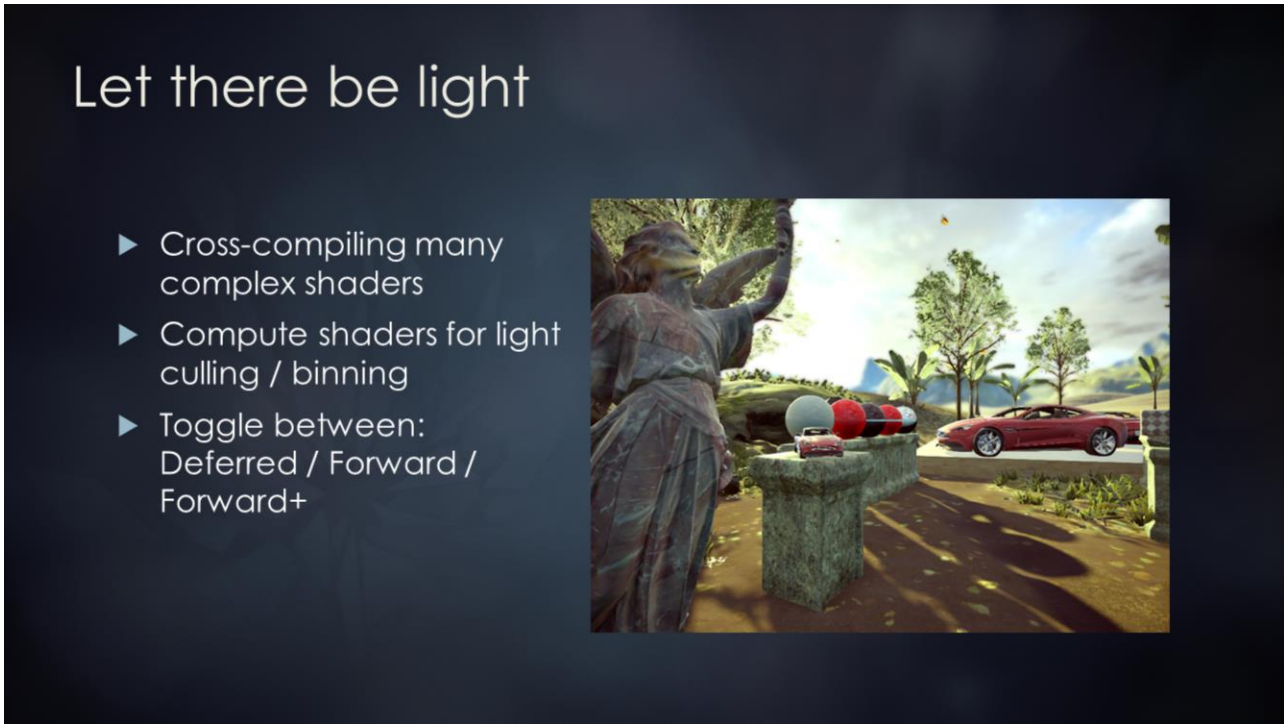
The data produced from the light culling pass can then be used to get the list of lights that can potentially affect each pixel.
We can use this data structure for both forward and deferred rendering.

In the forward case it's simple.
When rasterizing primitives we can just determine which tile we're in to get the list of lights and then iterate over them to accumulate their values.

In the deferred case, however, as we have already rasterized the g-buffers we need a way of not having to process tiles that don't contain lights.
Therefore we run a compute shader after the binning pass that finds out which types of lights are affecting each tile. This information is used to issue several indirect dispatches, that only process the tiles affected by lights, with optimized shaders for each permutation of lights.



Here’s an illustration of the steps involved when culling and applying lights using our deferred path. The first image shows the final image. The upper-right image visualizes the light grid, and the bottom images show two of the passes that are executed to apply lights when doing deferred rendering. As you can see we will only process the tiles that gets affected by the lights.



As usual it turned out to be more work than expected. Getting the complex cross-compiled shaders even to compile in the native compilers was non-trivial.

Currently the light culling / binning phase relies heavily on compute shaders, so as of today that is a requirement. Compute support on mobile isn't optimal yet, so we could probably benefit from doing the culling on the cpu early on in the frame. We won't be able to use the depth buffer for the the culling, but doing clustering on the cpu could erase the need.

As we had previously added deferred/forward toggle, we implemented support for being able to toggle a z prepass. This prepass helps when doing the light culling with forward rendering at the cost of rendering extra geometry.

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
```

There were a few things we had to do to support some of the new features in the engine on mobile.

As both vanilla ES3 and Metal were missing support for cubemap arrays we had to add pass that when baking the reflection volumes into cube maps converted them to 2d lat-long textures.

This adds a small alu overhead when getting a sample, but works properly with adressing modes, bilinear filtering and mipmaps. Also we can use exactly the same code with a different function for sampling the texture.

Having an array of reflection textures allows us to sample different reflection volumes per object or even several per object when doing deferred light accumulation.

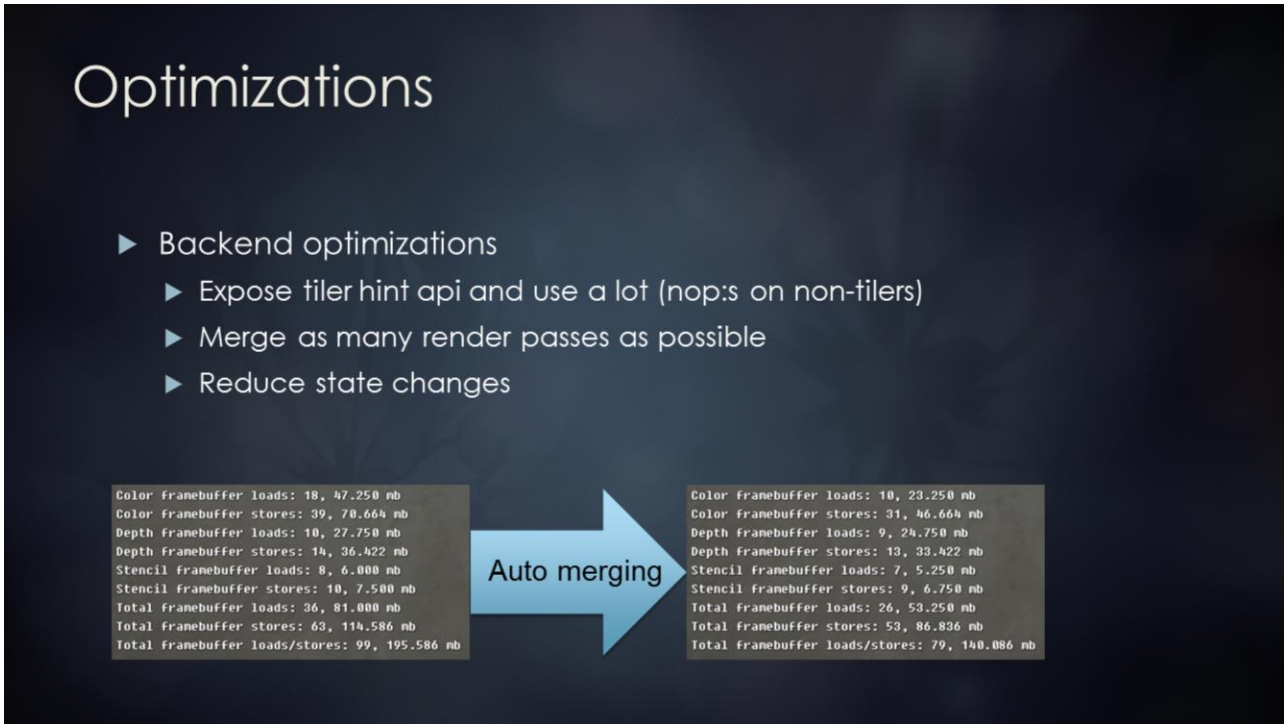
Let there be light

- 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);

    ...
```

We added a vertex/pixel-shader version of the deferred light accumulation pass in order to be able to emulate the indirect dispatches with optimized shaders for applying lights. Basically the vertex shader receives the indirect args buffer and emits an offscreen primitive if the tile to be rendered is out of bounds. Also this meant we could stay in tile memory when accumulating all the different light types.

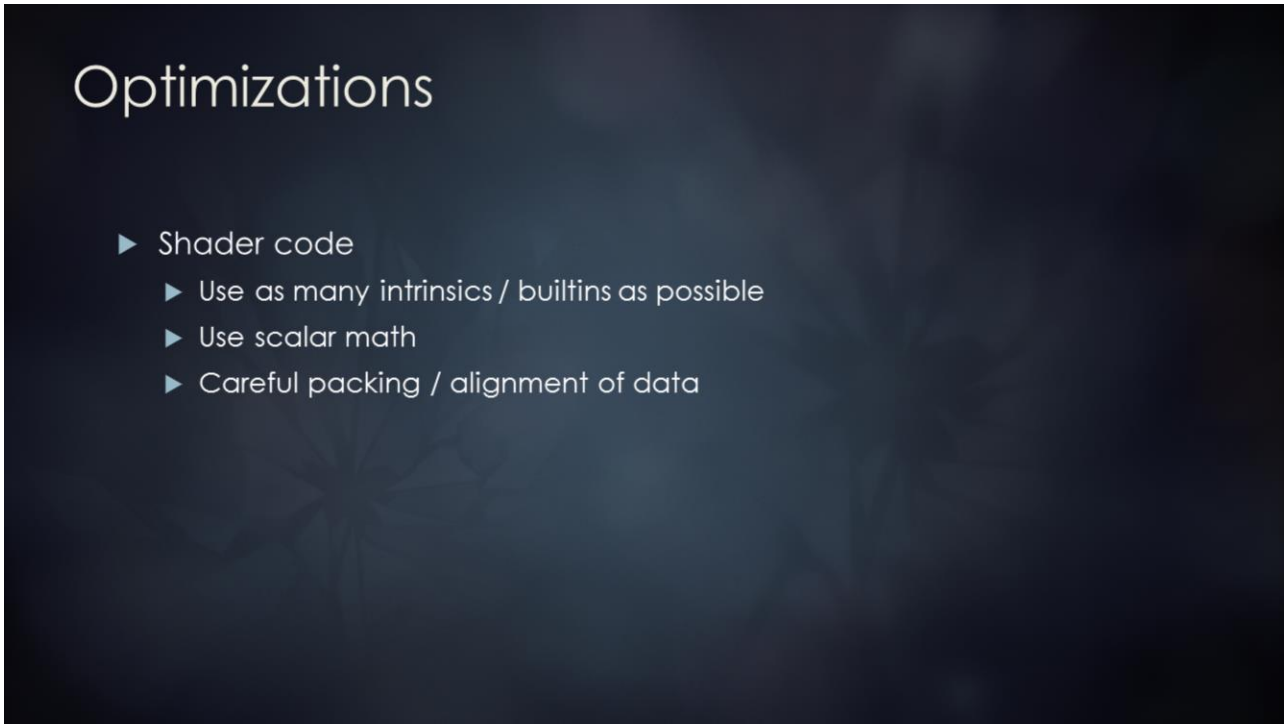
It’s worth mentioning that everything is dynamic, ranging from the lights to dynamic shadowmaps for every light, to dynamically updated reflection views.



In order to reach 30 fps with all the fancy new features we had to further optimize what we had. Instead of changing the algorithms we attacked the problem by trying to optimize our backends and shader code as much as possible.

As I previously mentioned we had implemented an api to tell the backend what to do with the frame buffer textures. Using this information we could also in the backend merge many render passes, without the user having to care about it. In GL drivers this already happens in many cases, but on Metal where things are more explicit we had to manage this. In some cases we restructured the order of passes to be able to merge them on mobile, which didn't affect the performance on non-mobile platforms.

We also added a lot of redundant state filtering, including keeping common buffers and textures at fixed register indicies to reduce state changes.



In terms of shader code we added a lot of intrinsics to the cross-compiler to get as efficient code as possible. In many cases these maps to single instructions. Also we disabled vector optimization passes in our cross-compiler to help optimizing for scalar alu:s and not loose any information for the native compilers. Also still don't expect compilers to optimize even the trivial things for you.

Our performance of some of the more advanced shaders were doubled by managing packing of structs ourselves in the cross-compiler.

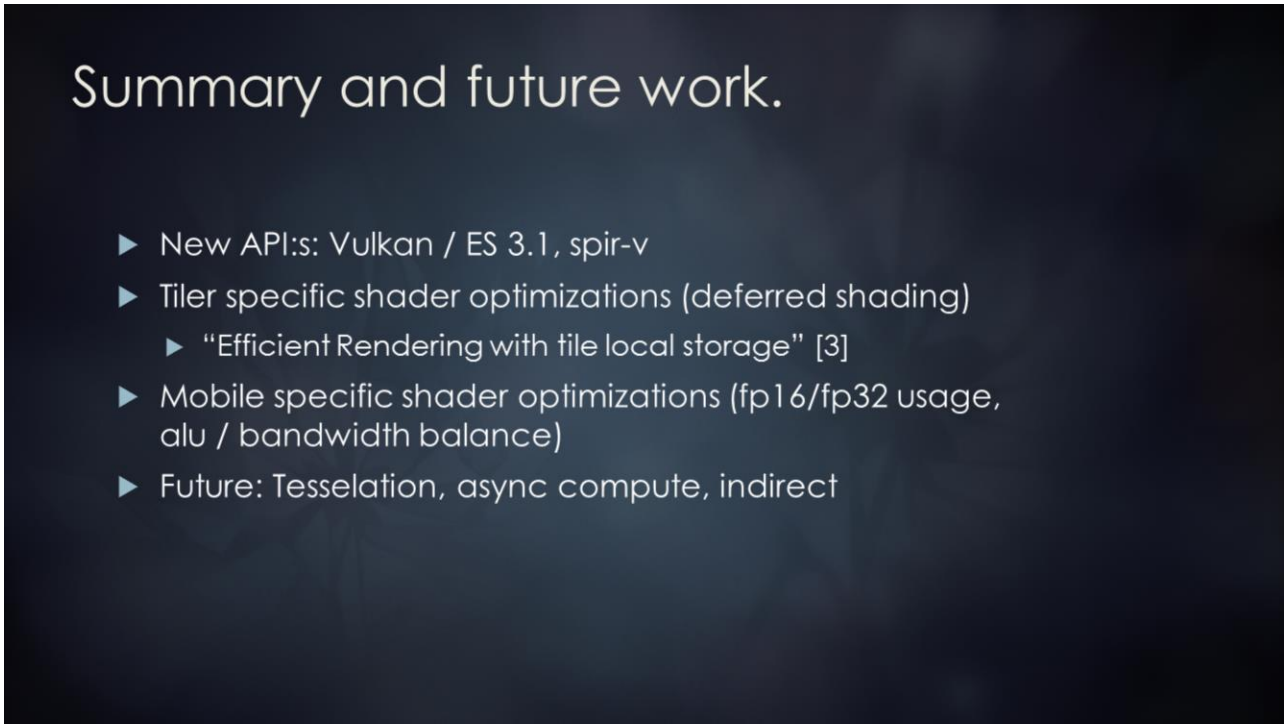

Summary and future work
WHERE DO WE GO NOW?

To summarize our work has shown that a bleeding edge engine can run on today's mobile hardware.

We've done a straight implementation of the engine, cross compiling all shaders with minor divergence.

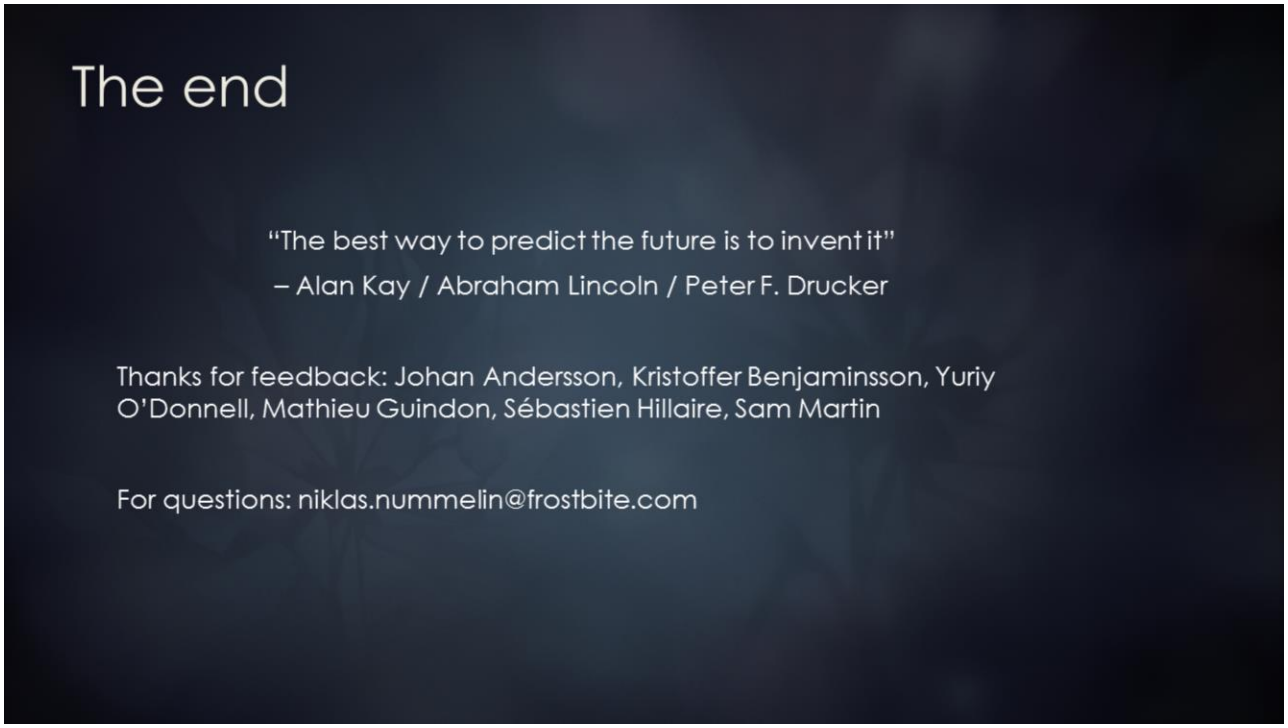
Our approach has been to get the full picture before starting to tweak the details. We prefer to disable slow features until we attack them and attack the ones giving the most bang for the buck first. We believe we're at a point now when we have the full picture and can start diving into the details.

Things we have noticed: Many tile memory specific optimizations such as merging render passes and specifying how frame buffer textures should be loaded and stored can be done without diverging code bases. Cross-compilation is key.



We hope that Vulkan will get a good uptake on Android as this style of api has proven to be a good fit for our engine. Until Vulkan has gained some momentum, we will stick to ES3 and experiment with ES 3.1. We haven't had time to experiment with the tile storage yet and really want to see what can be done it with it. There's a really interesting talk from last year going into the details on how this can be used to optimize deferred rendering. Also due to most mobile gpu:s having native support for both fp16 and fp32 operatons, and with newer gpu:s some alu may perform better than a single texture lookup we want to find a good balance there.

Frostbite has also started to utilize async compute, tesselation and indirect dispatches a lot so we hope to see more of that on Mobile in the future.



I'll wrap up with this quote, that apparently has been said by many famous persons, because I think it resonates well with what we all do in the mobile space.

If you have any questions, don't hesitate to mail me here.

Thank you!

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.

Here are the previously mentioned talks.


FROSTBITE™
empowers game creators to shape the future of gaming