BATTLEFIELD 3

SPU-based Deferred Shading for Battlefield 3 on Playstation 3

Christina Coffin
Platform Specialist, Senior Engineer
DICE

Agenda

- Introduction
- SPU lighting overview
- SPU lighting practicalities & algorithms
- Code optimizations & development practices
- Best practices
- Conclusions
- Q&A

Introduction

Maxxing out mature consoles

Past: Frostbite 1

Forward Rendered + Destruction + Limited Lighting

Precomputed + Static = ☹

Now: Frostbite 2 + Battlefield 3

Indoor + Outdoor + Urban HDR lighting solution

- Complex lighting with Environment Destruction
- Deferred shaded
- Multiple Light types and materials

Goal:
 ”Use SPUs to distribute shading work and offload the GPU so it can do other work in parallel”

Why SPU-based Deferred Shading?

Want more interesting visual lighting + FX

- Offload GPU work to the SPUs
- Having SPU+GPU work together on visuals raises the bar

Already developed a tile-based DX 11 compute shader

- Good reference point for doing deferred work on SPU

Lots of SPU compute power to take advantage of

- Simple sync model to get RSX + SPUs cooperating together

SPU Shading Overview

Physically based specular shading model

- Energy Conserving specular , specular power from 2 to 2048

Lighting performed in camera relative worldspace, float precision

fp16 HDR Output

Multiple Materials / Lighting models

Runs on 5-6 SPUs

Multiple lighting models + materials

- Standard
- Metallic
- Skin
- Translucency

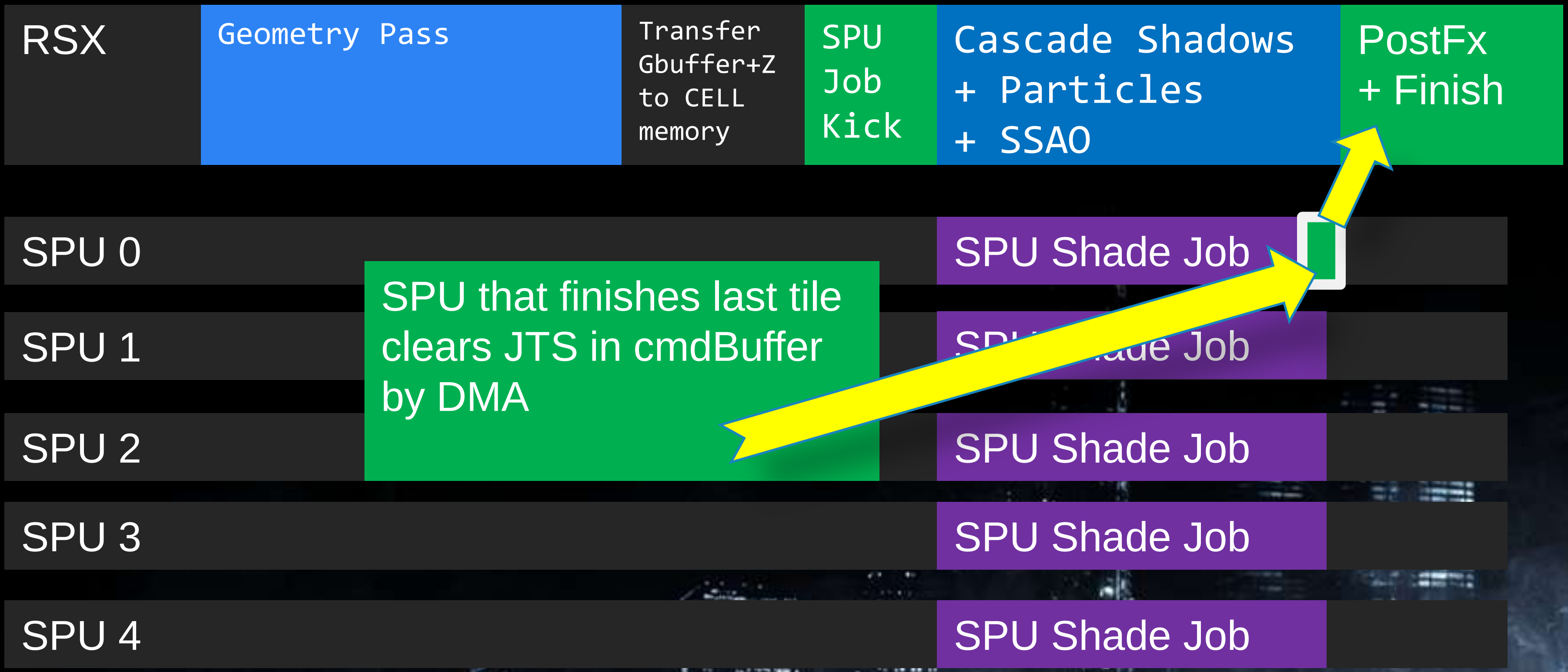
Rendering Frame Timeline

Note: Sizes not proportional to time taken!

RSX	Geometry Pass	Transfer Gbuffer+Z to CELL memory	SPU Job Kick	Cascade Shadows + Particles + SSAO	JTS Barrier
SPU 0				SPU Shade Job	
SPU 1				SPU Shade Job	
SPU 2				SPU Shade Job	
SPU 3				SPU Shade Job	
SPU 4				SPU Shade Job	

Rendering Frame Timeline

Note: Sizes not proportional to time taken!



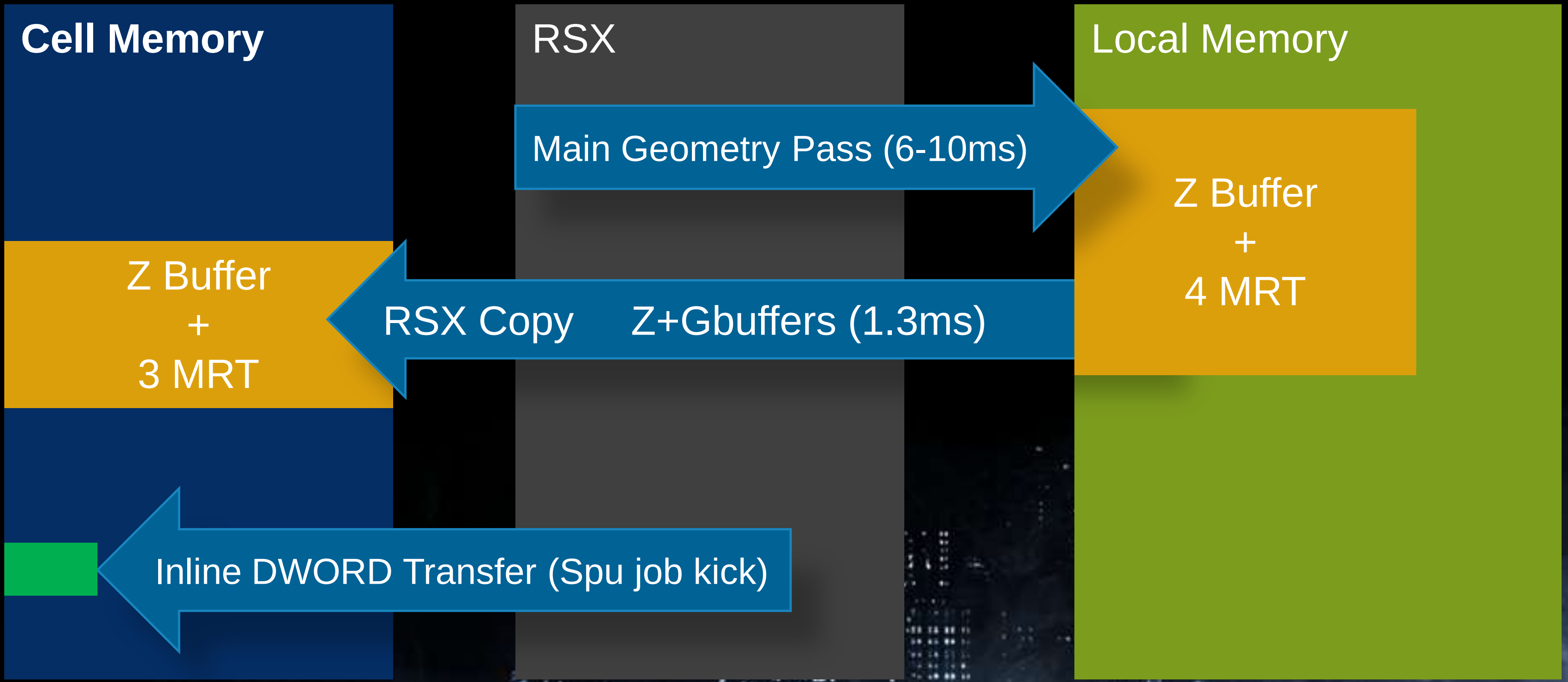
GPU Renders GBuffer

RSX render to local memory GBuffer Data

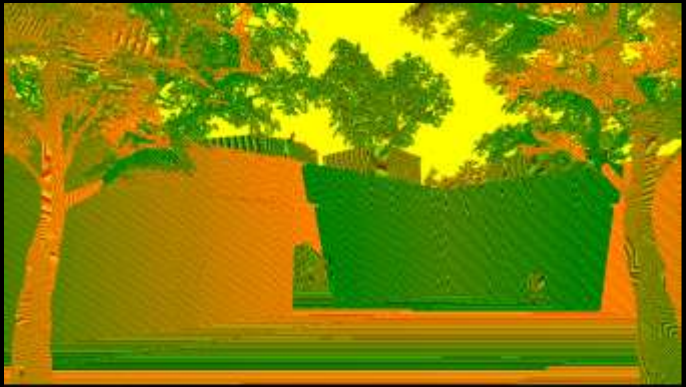
- 4x MRT ARGB8888 + Z24S8
- Tiled Memory

	R8	G8	B8	A8
GB0	Normal .xyz			Spec. Smoothness
GB1	Diffuse albedo .rgb			Specular albedo
GB2	Sky visibility	Custom envmap ID	Material Param.	Material ID
GB3	Irradiance (dynamic radiosity)			

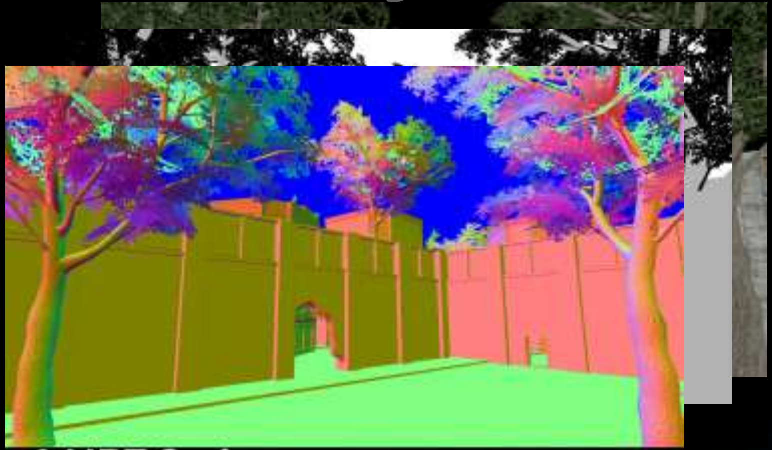
Setup Rendering Data Flow



Source data in CELL memory



Z Buffer
+ Stencil

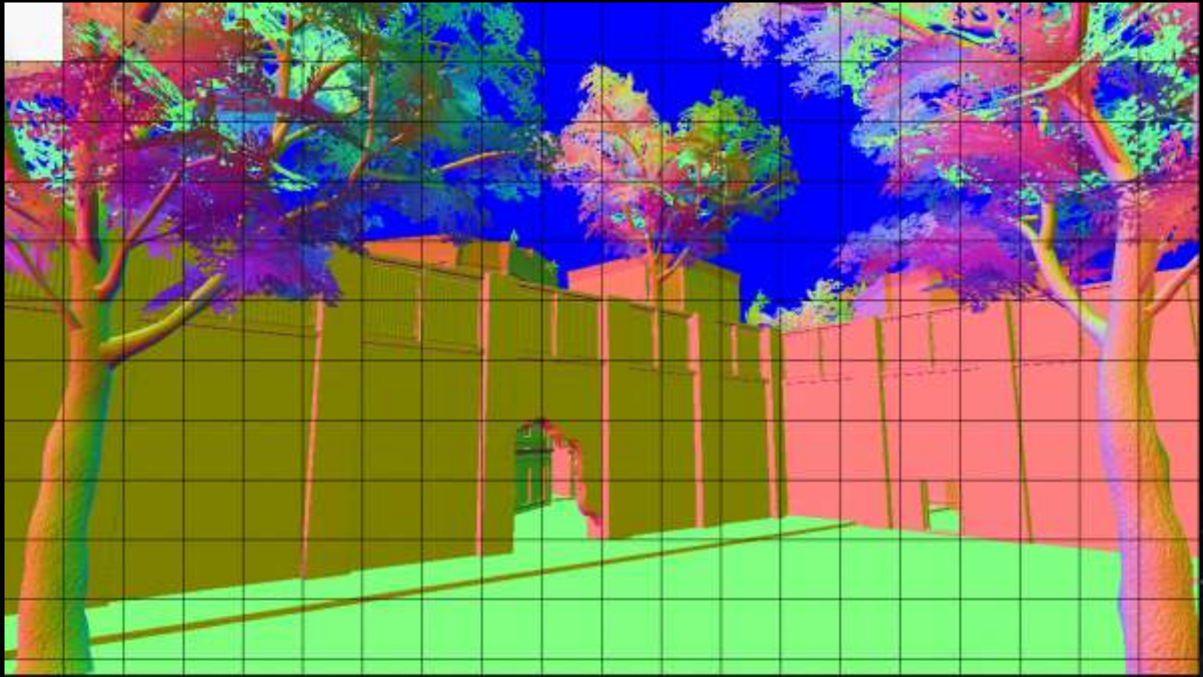


+3 MRT Surfaces

Packed array of all lights visible in camera (1000+)

SPU SPU SPU
SPU SPU SPU

SPU Tile Based Shading work units



64x64 pixel tiles = 1 SPU work unit

SPU Shading Flow Overview

For each 64x64 pixel screen tile region:

1. Reserve a tile
2. Transfer & detile data
3. Cull lights
4. Unpack & Shade pixels
5. Transfer shaded pixels to output framebuffer

SPU Tile Work Allocation

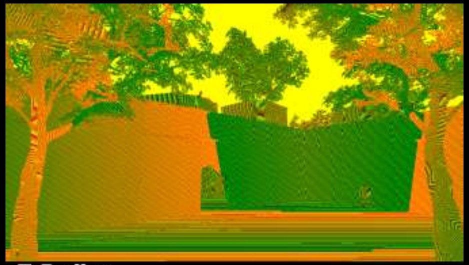
SPUs determine their tile to process by atomically incrementing a shared tile index value

- Index value maps to fetch address of Z+Gbuffers per tile
- Simple sync model to keep SPUs working
- Auto Load balancing between SPUs
  - Not all tiles take equal time = variable material+lighting complexity

Getting the Tile Data onto SPUs....

DATA TRANSFER + DE-TILING

FB Tile Data Fetch

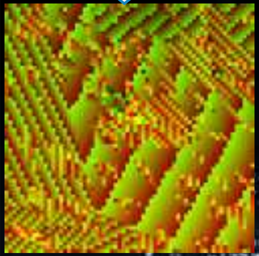


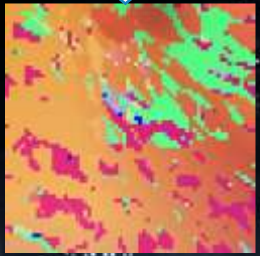
Z Buffer

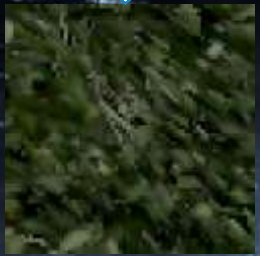
+3 MRT Surfaces

DMA Traffic: Get ZBuffer | Get MRT 0 | Get MRT 1 | Get MRT 2

SPU Code: Detile Z | Detile 0 | Detile 1 | Detile 2







SPU LS

- 64x64 Pixels 32bpp (x4)
- Output Linear Formatted
- Ready for Shading Work

SPU LIGHT TILE CULLING

SPU Cull lights

Determine visible lights that intersect the tile volume in worldspace

Tile frusta generation from min/max z-depth extents

- Ignores ’sky’ depth (Z >= 0xFFFFFF00)
- Each tile has a different near/far plane based on its pixels
- SPU code generates frusta bounding volume for culling

Point-, Spot- & Line-light types supported

- specialized culling vs. tile volumes

SPU SHADING LOOP

SPU Shade pixels

SPU Tile Based Shading

- We do the same things the GPU does in shaders, but written for SPU ISA
- Vectorize GPU .hlsl / compute shader to get started
- Negligable differences in float rounding RSX vs SPU

Core Steps:

Unpack Gbuffer+Z Data -> Shade -> Pack to fp16 -> DMA out

Core shading components

3MRT + Z = 128 bits per pixel of source data

- Distance attenuation
- Light Volume Clipping
- Light Shape Attenuation
- Attenuation by Surface Normal

- Diffuse Lighting
- Specular Lighting
- Wraparound Lighting
- Fresnel
- Material Type Shading
- Blend in Diffuse Albedo

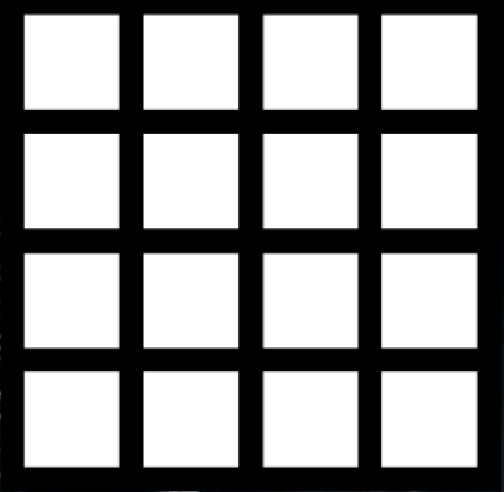
- Mask on Stencil Data
- Texture Sampling (Limited)

SPU Shading - 4x4 Pixel Quads

Core shading loop

- Operates on 16 pixels at a time
- Float32 precison
- Spatially Coherent
- Lit in worldspace

- Unpack source data to Structure of Arrays (SoA) format



Gbuffer data expansion to SoA for shading

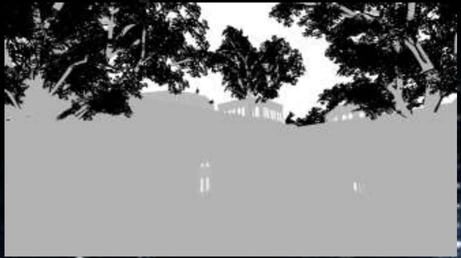

Depth + Stencil


Spec Albedo


Diffuse Albedo


Normals


Smoothness


Material ID

= Lots **shufb** + **csflt** instructions for swizzle /mask / converting to float

SPU Light Tile Job Loop

for (all pixels)

- Unpack 16 Pixels of Z+Gbuffer data
- Apply all PointLights
- Apply all SpotLights
- Apply all LineLights
- Convert lighting output
  to fp16 and store to LS

DMA output finished pixels

Finished 32x16 pixel tiles output to RSX memory by DMA list

- 1 List entry per 32x1 pixel scanline
- Required due to Linear buffer destination !

Once all tiles are done & transfered:

- SPU finishing the last tile, clears 'wait for SPUs' JTS in cmdbuffer

GPU is free to continue rendering for the frame

- Transparent objects
- Blend-In Particles
- Post-process
- Tonemapping

Meanwhile, back in RSX Land....

RSX is busy doing something (useful) while the SPUs compute the fp16 radiance for tiles.

- Planar Reflections
- Cascade and Spotlight Shadow Rendering
- GPU Lighting that mixes texture projection/sampling
- Offscreen buffer Particle Rendering
- Z downsamples for postFX + SSAO
- Virtual Texturing / Compositing
- Occlusion queries

ALGORITHMIC OPTIMIZATIONS

Tile Light Culling

Tile Based Culling System

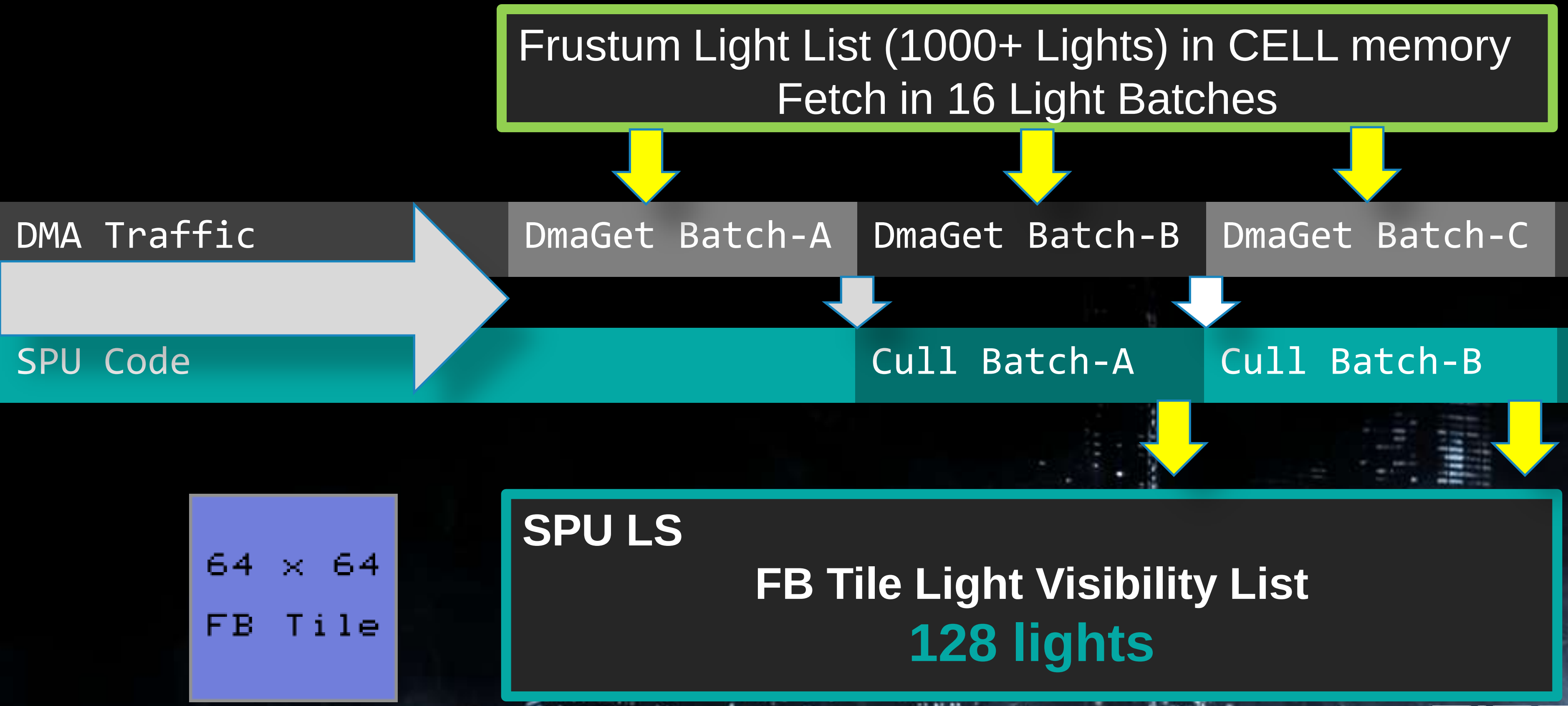
- Designed to handle extreme lighting loads
- Shading Budget
  - 40ms (split across 5 SPUs)
- Culling system overhead
  - 1-4ms (1000+ lights)

Tile Light Culling

2 Light Culling passes:

- FB Tile Cull, 64x64 pixel tiles
- SubTile Cull, 32x16 pixel tiles

FB Tile Light Culling



SubTile Light Culling

64 x 64
FB Tile

SPU LS

FB Tile Light Visibility List
128 lights

for(8 SubTiles)

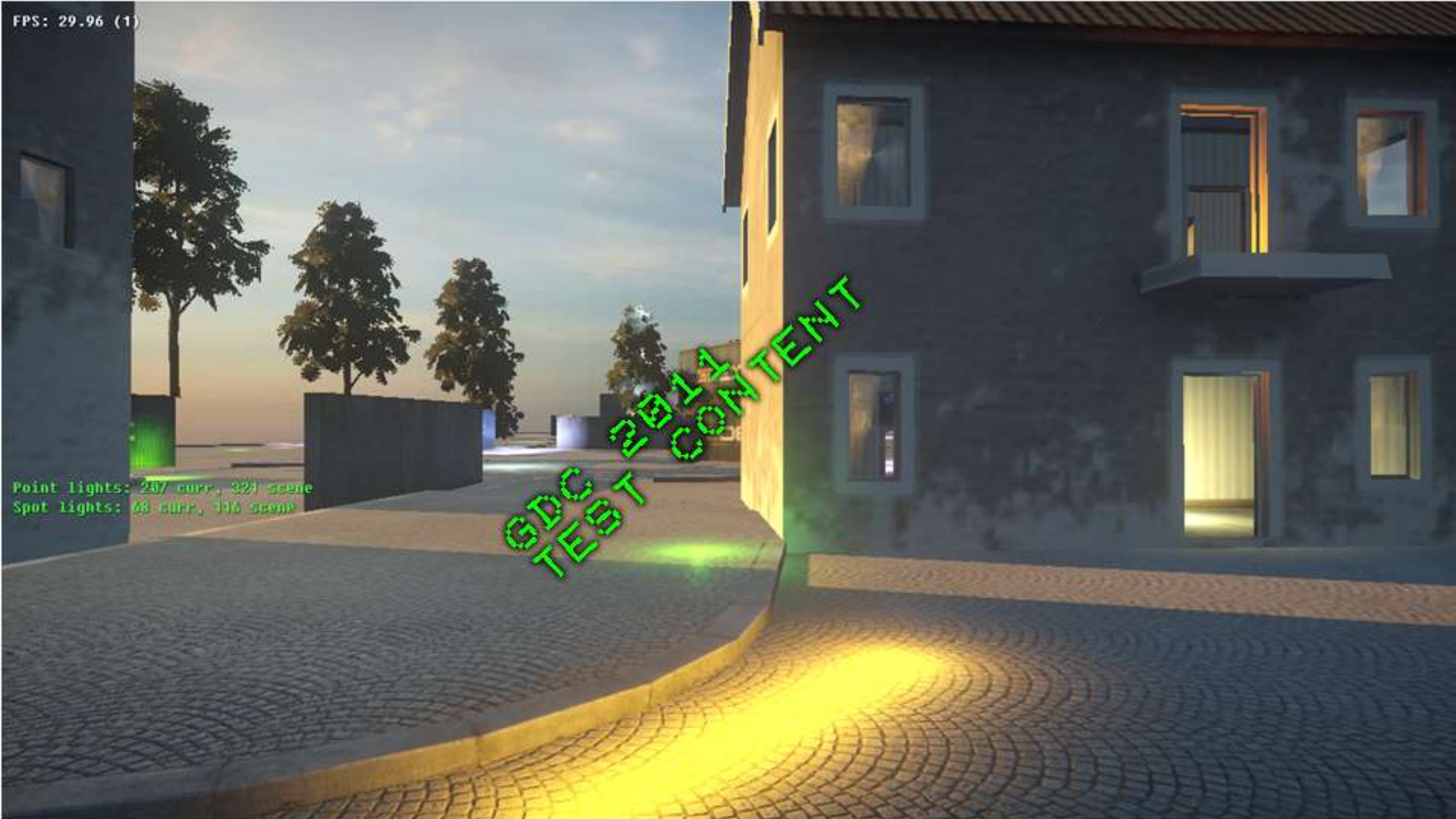
SubTile 0 Cull | SubTile 1 Cull | SubTile 2 Cull

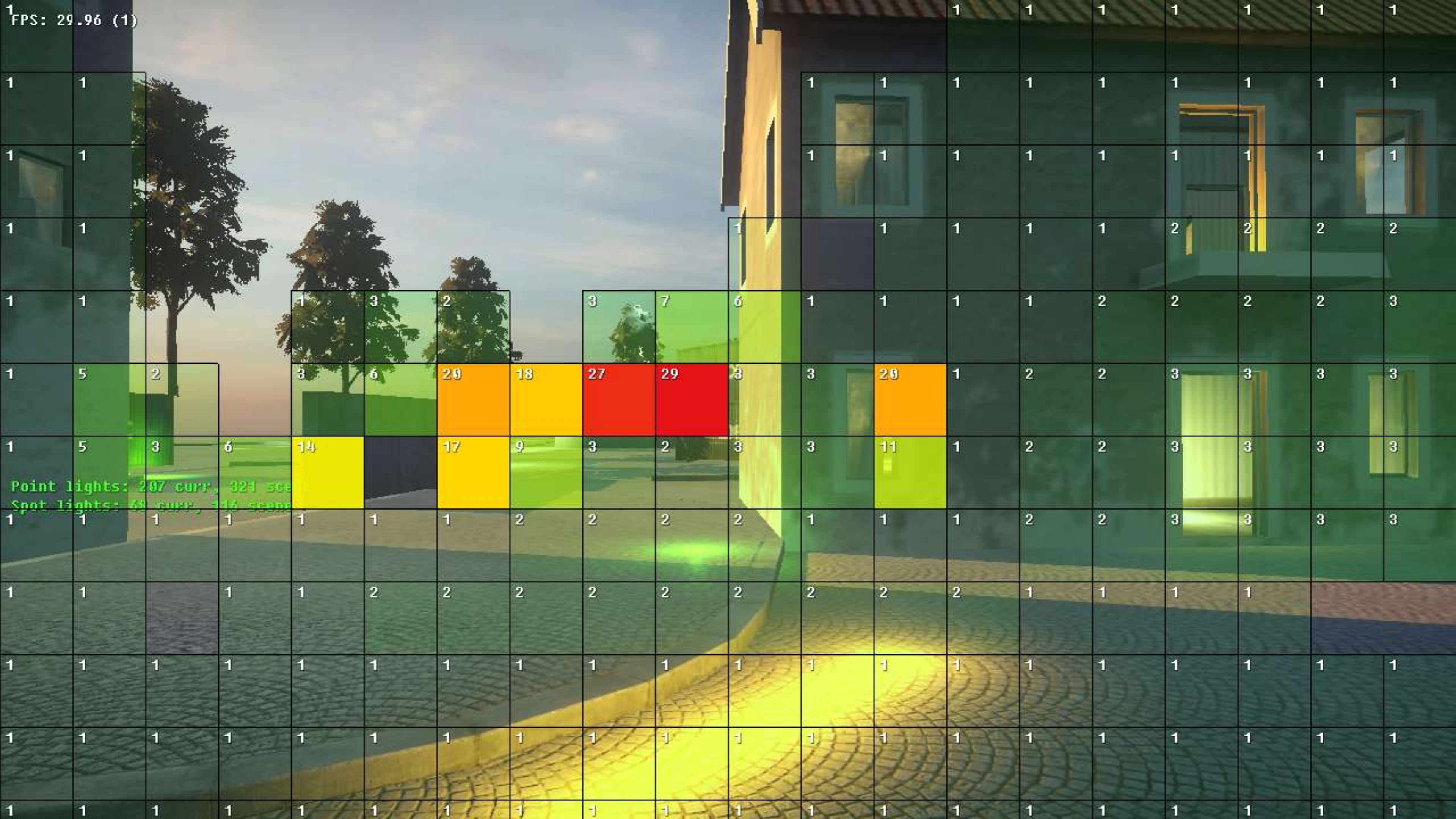
Subtile 0 Light Index List

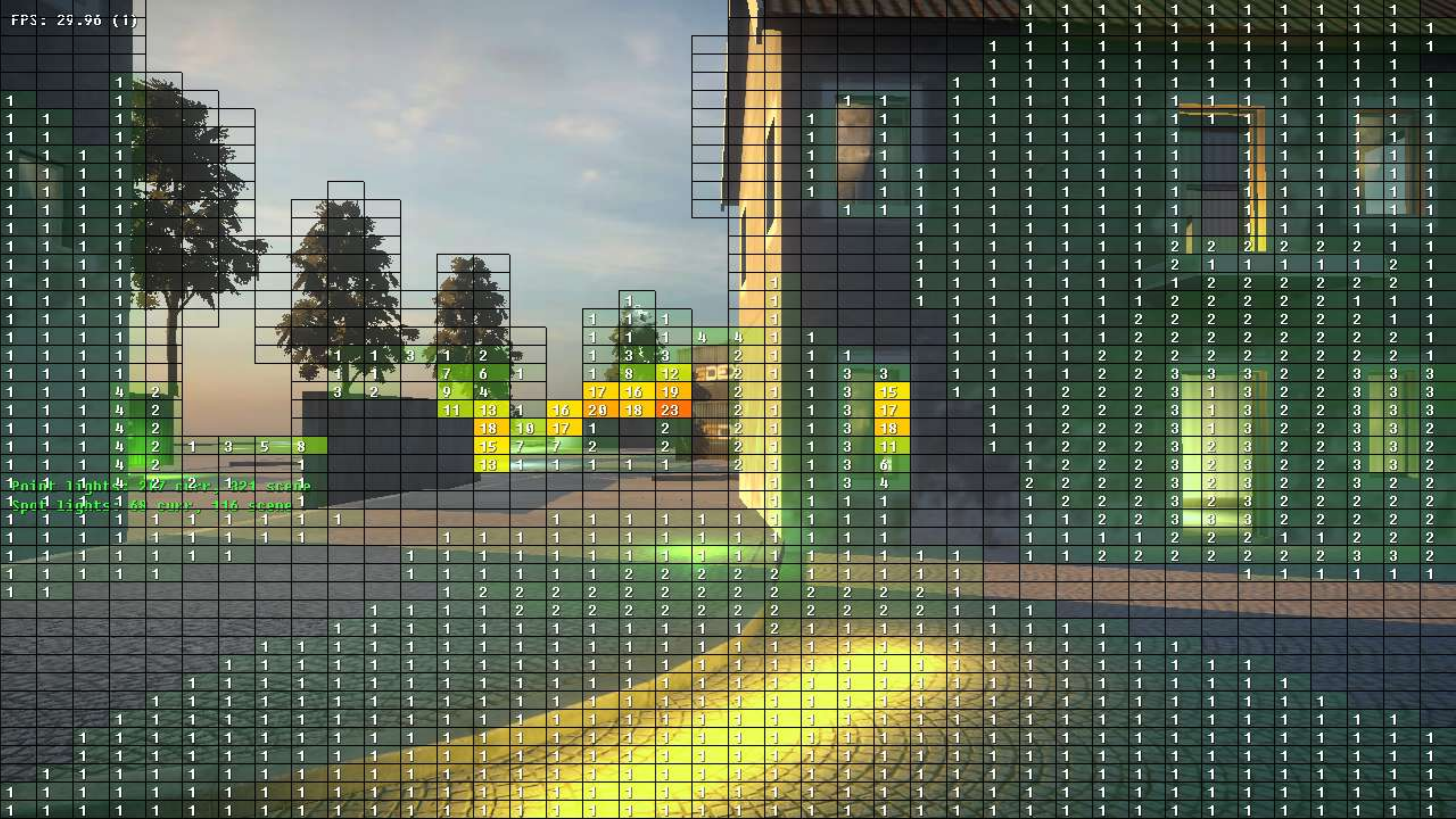
Subtile 1 Light Index List

Subtile 2 Light Index List

SubTiles
32x16
pixels

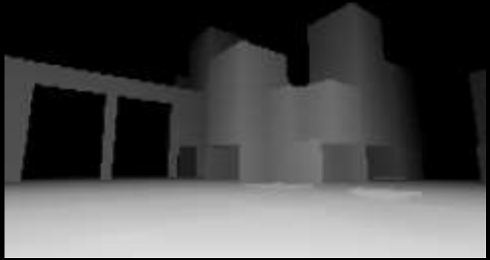

FPS: 29.96 (1)
GDC 2011
TEST CONTENT


FPS: 29.96 (1)

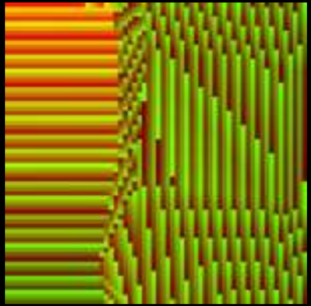

FPS: 29.96 (1)

Light Culling Optimizations - Hierarchy

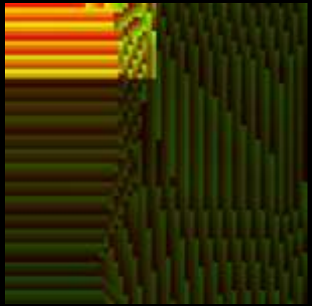
Camera Frustum



Light Volume
Coarse Z-Occlusion



FB Tile
SPU Z-Cull
64x64 Pixels



SubTile
SPU Z-Cull
32x16 Pixels

Coarse to Fine Grained Culling

Culling Optimizations - Takeaway

Complex scenes require aggressive culling

- Avoids bad performance edge cases
- Stabilizes performance cost
- Mix of brute force + simple hierarchy

Good Debug Visualizations is key

- Help guide content optimization and validate culling

Algorithmic optimization #0

Material Classification

- Knowing which materials reside in a tile = choose optimal SPU code permutation that avoids unneeded work.

  - *E.g. No point in calculating skin shading if the material isnt present in a subtile.*

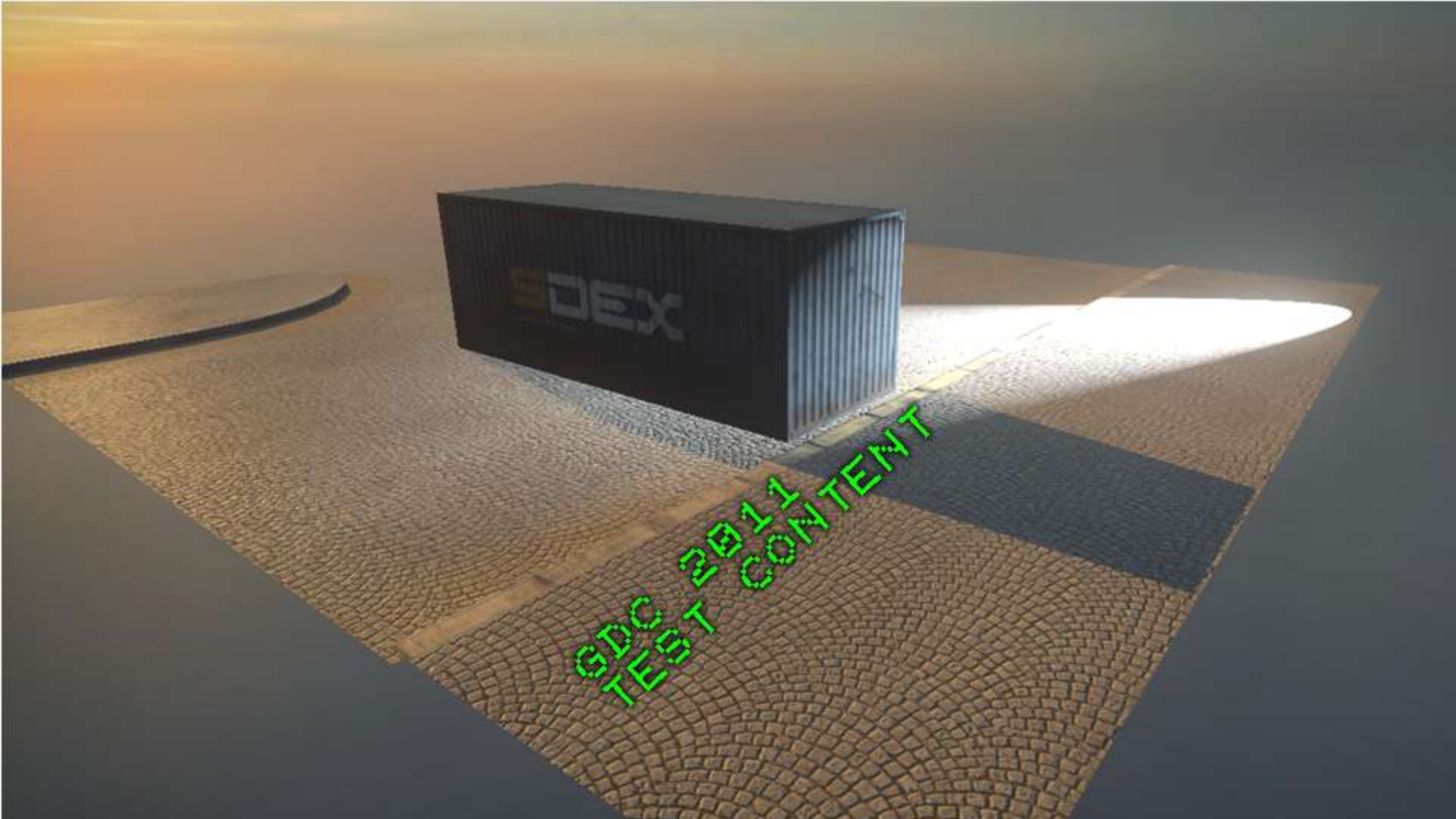
Use SPU shading code permutations!

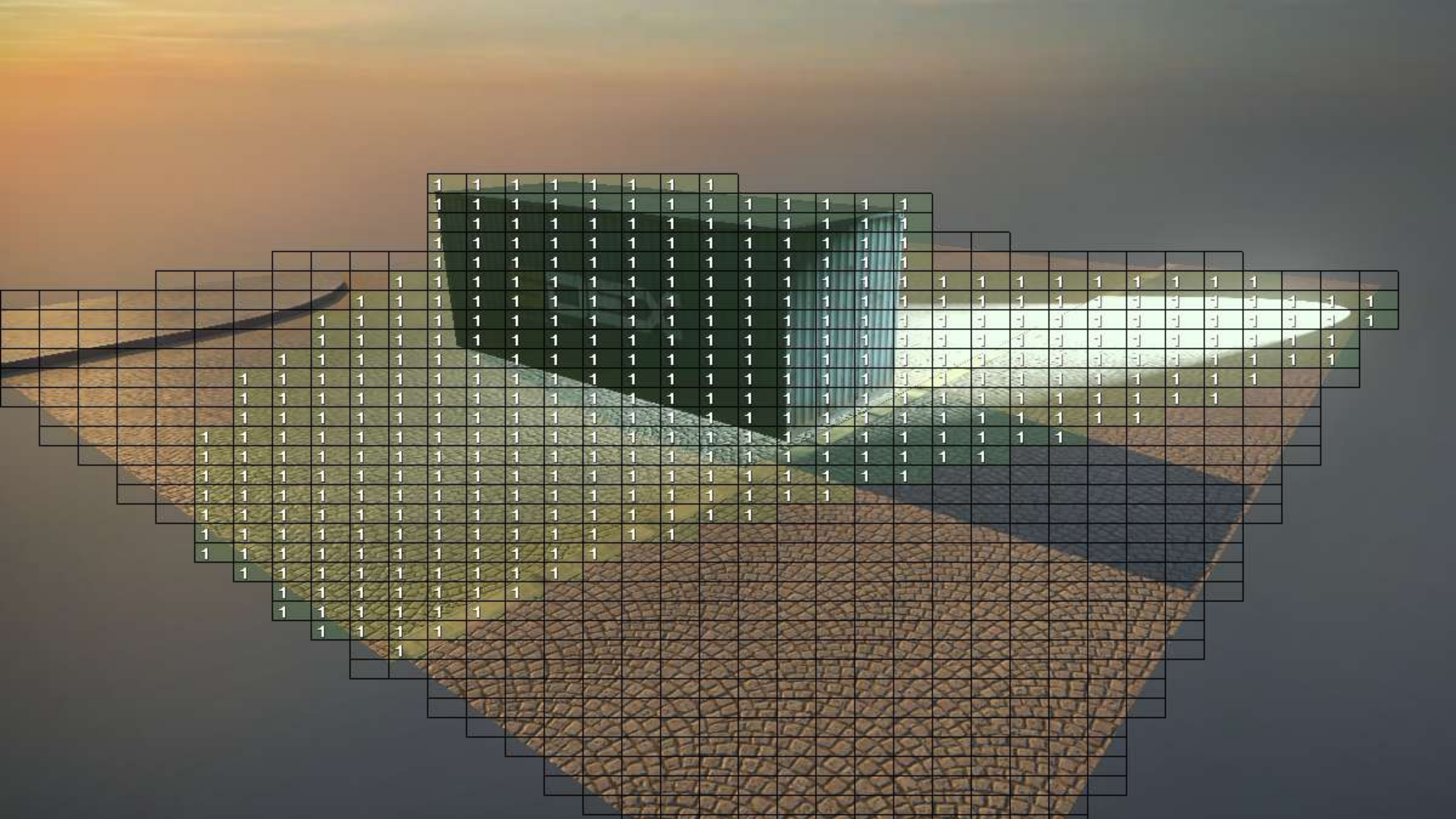
- Similar to GPU optimization via shader permutations
- SPU Local Store considerations with this approach

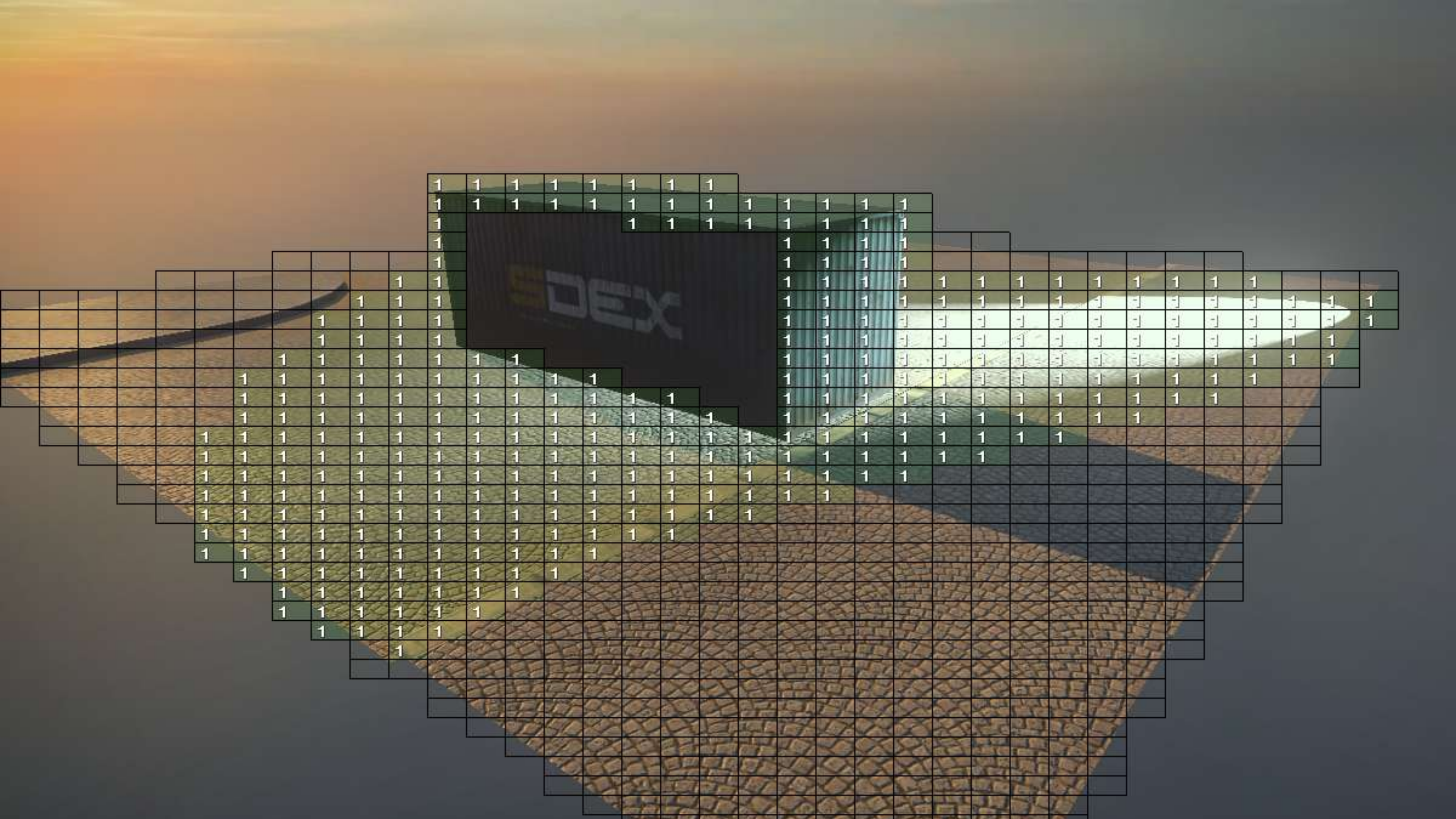
Algorithmic optimization #1

Normal Cone Culling

- Build a conservative bounding normal cone of all pixels in subtile,
- Cull lights against it to remove light for entire tile
- No materials with a wraparound lighting model in the subtile are allowed. (Tile classification)
- Flat versus heavily normal mapped surfaces dictate win factor


SDEX
GDC 2011
TEST CONTENT





Algorithmic optimization #2

Support diffuse only light sources

- Common practice in pure GPU rendered games
- Fill / Area lighting
- Only use specular contributing light sources where it counts.
- 2x+ faster
- Adds additional lighting loop + codesize considerations

Algorithmic optimization #3

Specular Albedo Present in a subtile?

If all pixels in a subtile have specular albedo of zero:

- Execute diffuse only lighting fast path for this case.
- If your artists like to make everything shiny, you might not see much of a win here

Algorithmic optimization #4

Branch on 4x4 pixel tile intersection with light based on the calculated lighting attenuation term

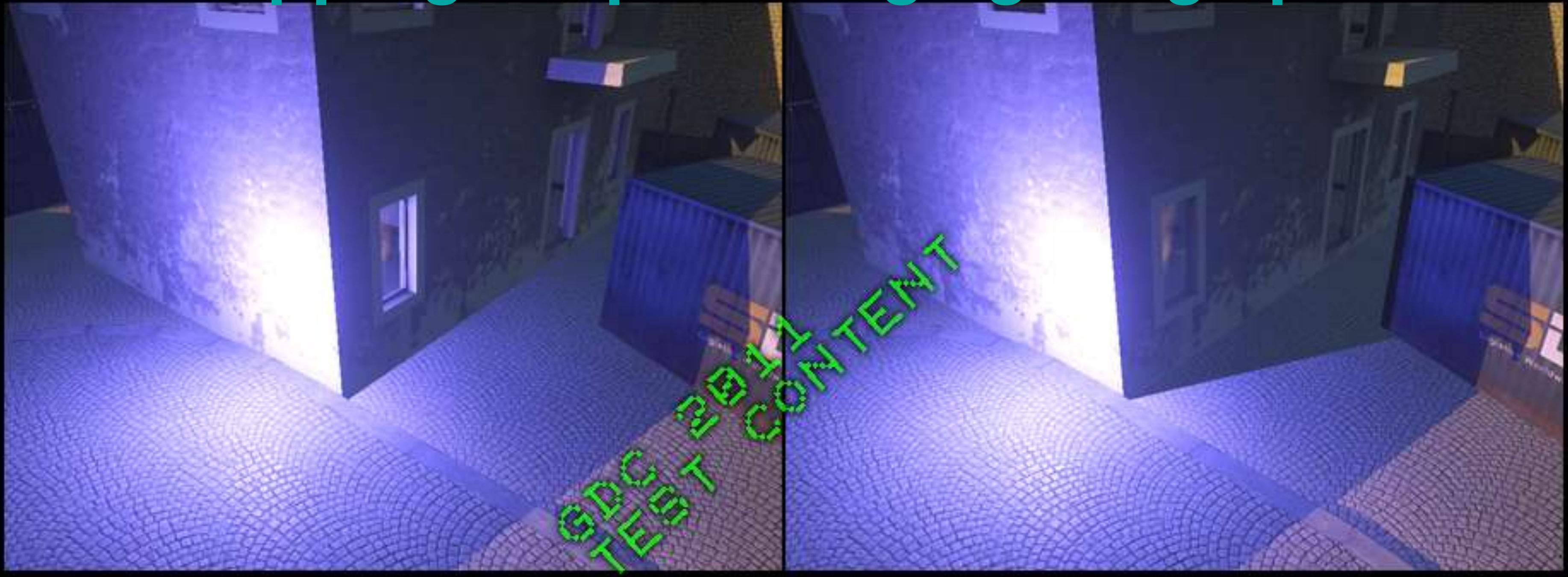
```
float   attenuation   = 1 / (0.01f + sqrDist);
        attenuation   = max( 0, attenuation + lightThreshold );



if( all 16 pixels have an attenuation value of 0 or less)
  (continue on to next light)
```

Branching if attenuation for 16 pixels < 0

```
// compare for greater than zero, can use this to saturate attenuation between 0-1
qword     attenMask_0          = si_fcgt( attenuation_0, const_0 );
qword     attenMask_1          = si_fcgt( attenuation_1, const_0 );
qword     attenMask_2          = si_fcgt( attenuation_2, const_0 );
qword     attenMask_3          = si_fcgt( attenuation_3, const_0 );

// 'or' merge masks from dwords in quadwords (odd pipe)
qword     attenMerged_0        = si_orx( attenMask_0 );
qword     attenMerged_1        = si_orx( attenMask_1 );
qword     attenMerged_2        = si_orx( attenMask_2 );
qword     attenMerged_3        = si_orx( attenMask_3 );

// final merge of 4 quadwords with horizonally merged masks
qword     attenMerge_01        = si_or( attenMerged_0, attenMerged_1 );
qword     attenMerge_23        = si_or( attenMerged_2, attenMerged_3 );
qword     attenMerge_0123      = si_or( attenMerge_01, attenMerge_23 );

if( !si_to_uint(attenMerge_0123))
     continue;// move to next light!
```

Algorithmic optimization #5

Clipping + Optimizing lighting space

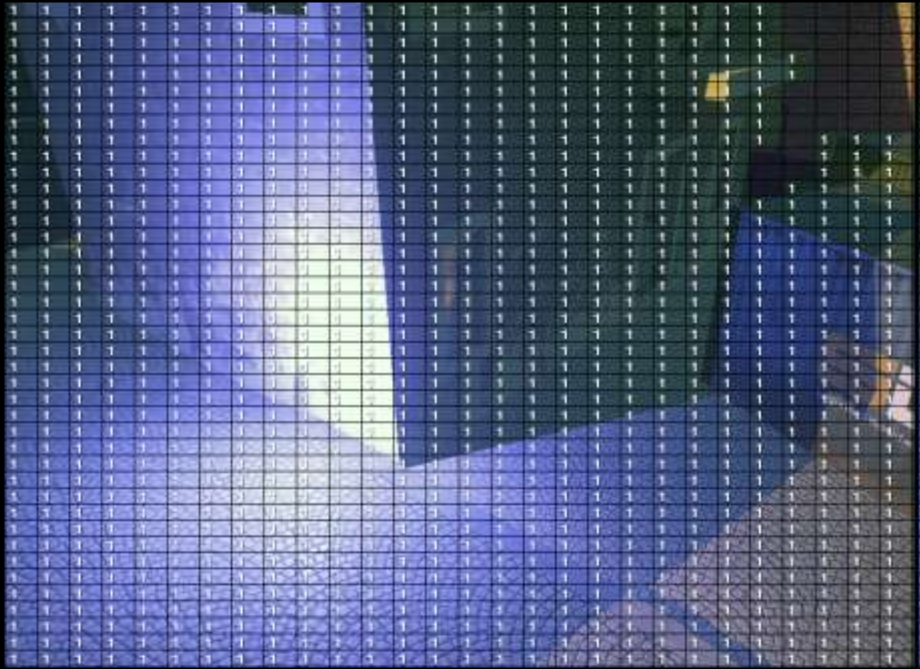


No Light Clipping

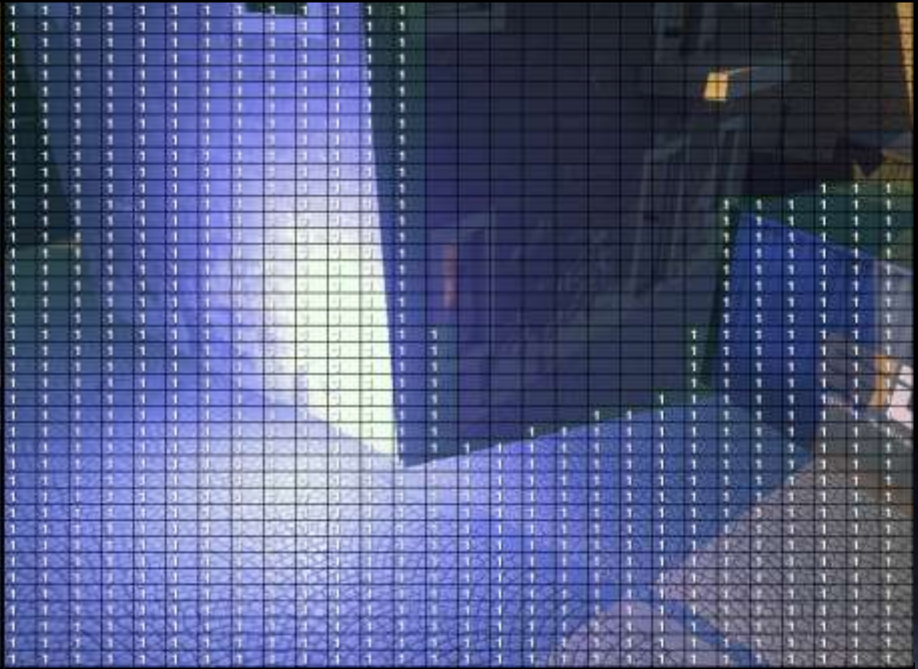
Light Clipping Against House Wall

Algorithmic optimization #5

Clipping + Optimizing lighting space



No Light Clipping



Light Clipping Against House Wall

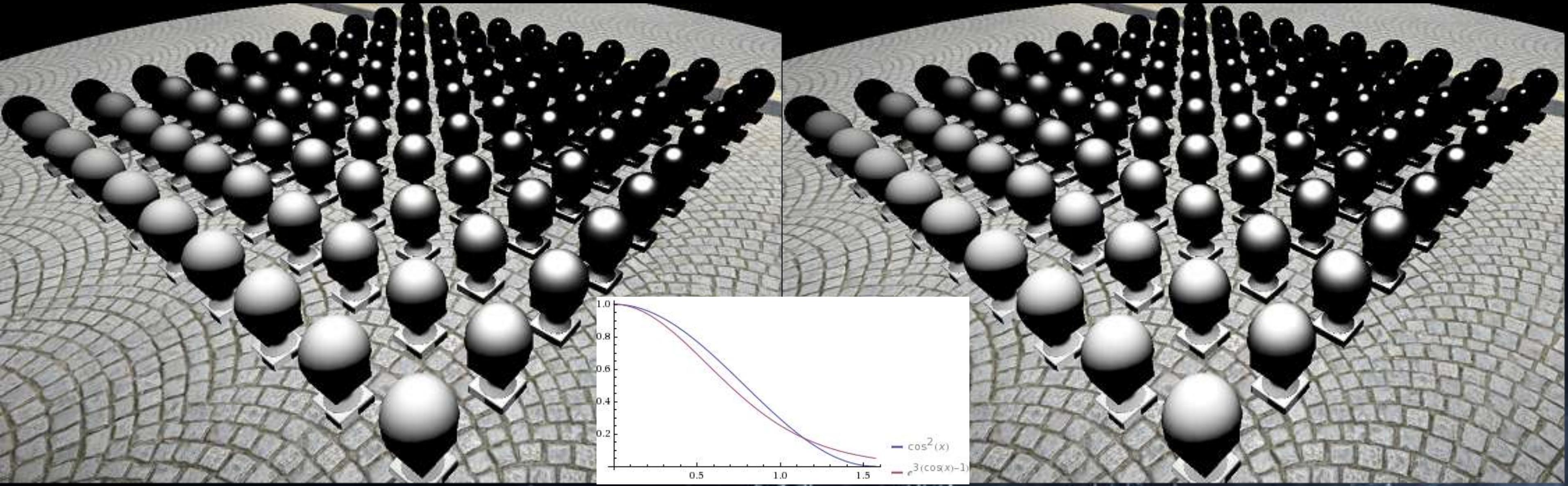
Why so much culling?

Why not adjust content to avoid bad cases?

- Highly destructible + dynamic environments
- Variable # of visible lights - depth ‘swiss cheese’ factor
- Solution must handle distant / scoped views

Algorithmic Optimization # 6

Spherical Gaussian Based Specular Model



CODE OPTIMIZATIONS

Code optimization #0

Unpack gbuffer data to Structure of Arrays (SoA) format

- Obvious must-have for those familiar with SPUs.
- SPUs are powerful, but crappy data breeds crappy code and wastes significant performance.

- shufb to get the data in the right format
- SoA gets us improved pipelining+ more efficient computation
  - 4 quadwords of data

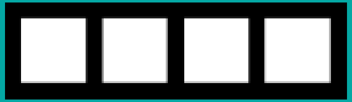
```
X0 Y0 Z0 W0
X1 Y1 Z1 W1
X2 Y2 Z2 W2
X3 Y3 Z3 W3
```

→

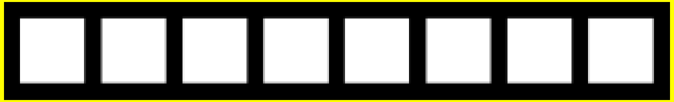
```
X0 X1 x2 x3
Y0 Y1 Y2 Y3
Z0 Z1 Z2 Z3
W0 W1 W2 W3
```

Code optimization #1: Loop Unrolling

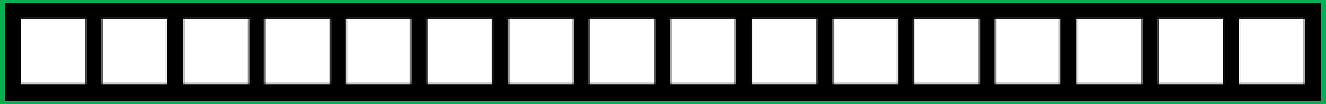
First versions worked on different sized horizontal pixel spans



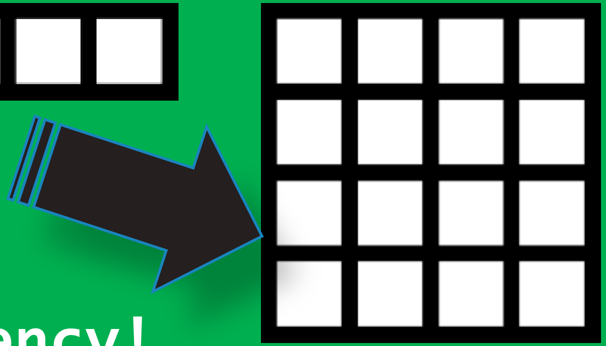
4x1 pixels = Minimum SoA implementation



8x1 pixels = 2x unrolled



16x1 pixels = 4x unrolled



4x4 pixels = 4x unrolled

+ Improved Spatial Coherency!

Code optimization #2

Branch on Sky pixels in 4x4 pixel processing loops

Branches are expensive, but can be a performance win
- Fully unpacking and shading 16 pixels = a lot of work to branch around

Also useful to branch on specific materials
- Depends on the cost of branching relative to just doing a compute + vectorized select

Code optimization #3

Instruction Pipe Balancing:

SPU shading code very heavy on *even* instruction pipe

Lots of fm,fma, fa, fsub, csflt ....

Avoid shli , or + and (*even* pipe),
use rotqbii + shufb (*odd* pipe) for shifting + masking

- Vanilla C code with GCC doesnt reliably do this which is why you should use explicitly use si intrinsics.

DICE

Code Optimization #4
Lookup tables for unpacking data

Can be done all in the odd instruction pipe

- Lighting Code is naturally Even pipe heavy
  - odd pipe is underutilized !

Huge wins for complex functions

- Minimum work for a win is ~21 cycles for 4 pixels when migrating to LUT.

Source gbuffer data channels are 8bit

- Converted to float and multiplied by constants or values w/ limited range
- 4k of LS can let us map 4 functions to convert 8bit ->float

Specular power LUT

From a GPU shader version .hlsl source:

```
half smoothness = gbuffer1.a;// 8 bit source

// Specular power from 2-2048 with a perceptually linear distribution
float specularPower = pow(2, 1+smoothness*10);

// Sloan & Hoffman normalized specular highlight
float specularNormalizationScale = (specularPower+8)/8;
```

	R8	G8	B8	A8
GB0	Normal .xyz			Smoothness
GB1	Diffuse albedo .rgb			Specular albedo
GB2	Sky visibility	Custom envmap ID	Material Param.	Material ID
GB3	Irradiance (dynamic radiosity)			

Remapping functions to Lookups

8bit gbuffer source value = 256 Quadword LUT (4k)

Store 4 different function output floats per LUT entry

- LUT code can use odd instruction pipe exclusively
- parent shading code is even pipe heavy = WIN

Total instructions to do 8 lookups for 4 different pixels:

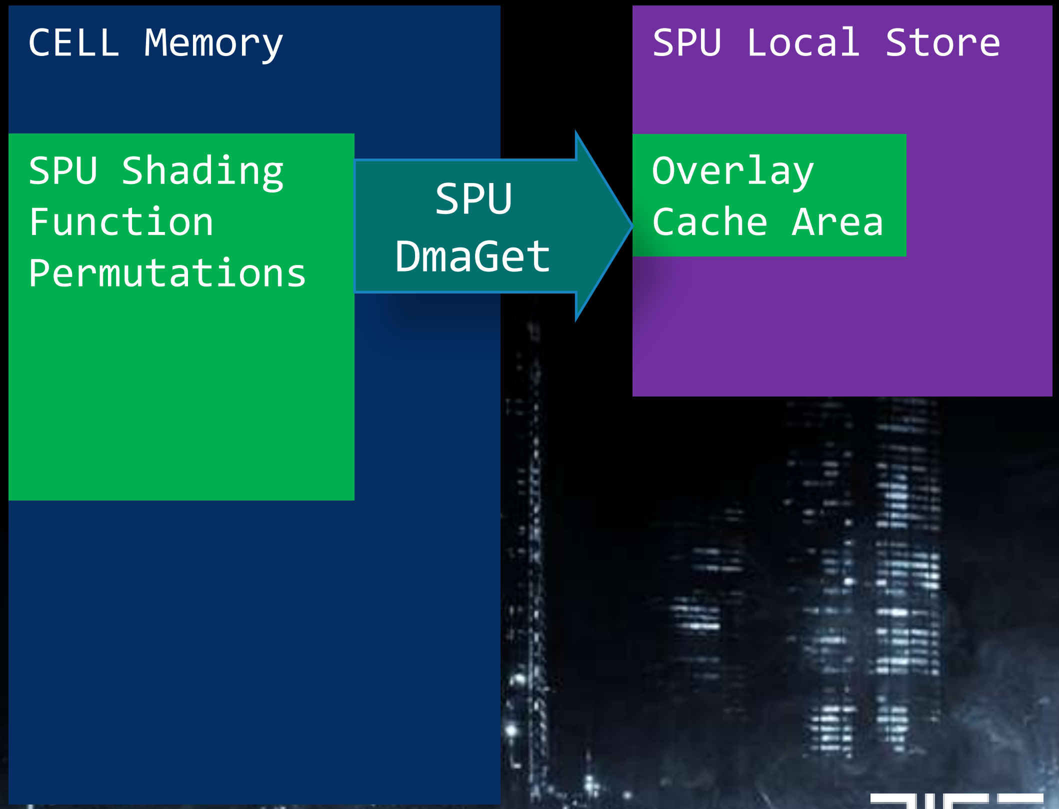
- 8 shufb, 4 lqx, 4 rotqbii (all odd pipe)
- ~21cycles

	X	Y	Z	W
LUT	float specularPower = pow(2, 1+smoothness*10); // This gives us specular power from 2-2048 with a perceptually linear distribution	float specularNormalizationScale = (specularPower+8)/8;	Shuffle + mask + Unpack normal component 0-255 to -1 to 1 float	Shuffle + mask + Unpack normal component 0-255 to -1 to 1 float squared

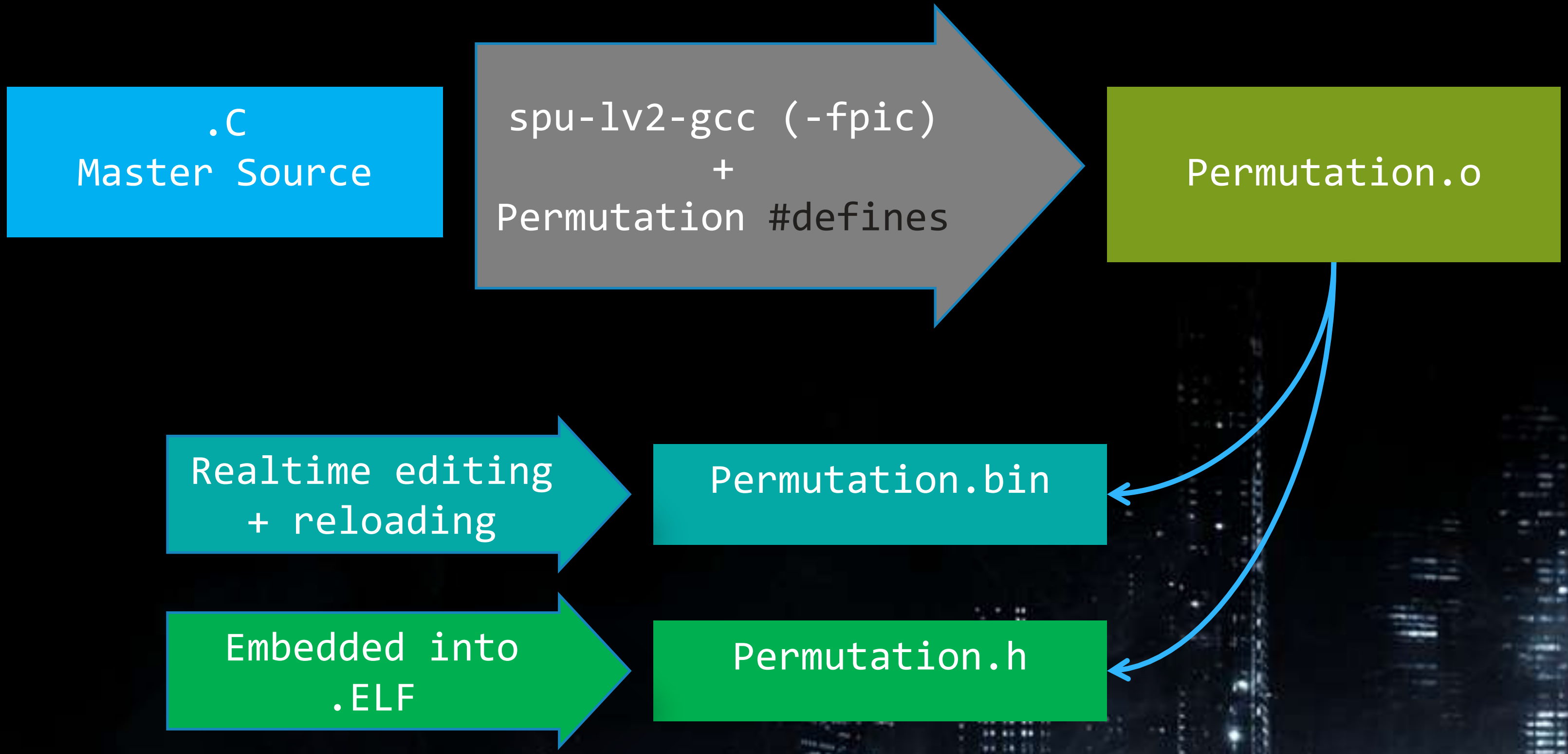
Code Optimization # 5

SPU Shading Code Overlays

- Avoids Limitations of limited SPU LS
- Position Independent Code
- SPU fetches permutations on demand



Overlay Permutation Building



BEST PRACTICES

Code + Development Environment

‘Must-haves’ for maintaining efficiency and *my* sanity:

Support toggle between pure RSX implementation and SPU version
- validate visual parity between versions

Runtime SPU job reloading
- build + reload = ~10 seconds

Runtime option to switch running SPU code on 1-6 SPUs

Maintain single non-overlay übershader version that compiles into Job
- Add/remove core features via #define
- Work out core dataflow and code structuring + debugging in 1 function.

Possible Code Permutations

Materials:

- Skin
- Translucent
- Metal
- Specular
- Foliage
- Emissive
- ‘Default’

Transformations:

- Different field of view projections

Light Types:

- Point light
- Spotlight
- Line Light
- Ellipsoid
- Polygonal Area

Lighting Styles:

- Diffuse only
- Specular + Diffuse Lighting
- Clip Planes
- Pixel Masking by Stencil

Code Permutations Best Practices

Material + Light Tile permutations

Still need a catch-all übershader

- To support worst case (all pixels have different materials + light styles)
- Fast dev sandboxing versus regenerating all permutations

Determining permutations needed is driven by performance

- Content dependent and relative costs between permutations

Managing codesize during dev

```
#define NO_FUNC_PERMUTATIONS // use only ubershader
```

Visualize permutation usage onscreen (color ID screen tiles)

'SPA' (SPU ASSEMBLER)

SPA is good for:

- Improving Performance*
- Measuring Cycle counts, dual issue
- Evaluating loop costs for many permutations
- Experimenting with variable amounts of loop unrolling

Don't jump too early into writing everything in SPA

- Smart data layout, C code w/unrolling, SI instrinsics , good culling are foundational elements that should come first.

Conclusions

SPUs are more than capable of performing shading work traditionally done by RSX

- Think of SPUs as another GPU compute resource

SPUs can do significantly better light culling than the RSX

RSX+SPU combined shading creates a great opportunity to raise the bar on graphics!

Special Thanks

- Johan Andersson
- Frostbite Rendering Team
- Daniel Collin
- Andreas Fredriksson
- Colin Barre-Brisebois
- Steven Tovey + Matt Swoboda @ SCEE
- Everyone at DICE

Questions?

Email: christina.coffin@dice.se
Blog: http://web.mac.com/christinacoffin/
Twitter: @christinacoffin

Battlefield 3 & Frostbite 2 talks at GDC'11:

Mon 1:45	*DX11 Rendering in Battlefield 3*	Johan Andersson
Wed 10:30	*SPU-based Deferred Shading in Battlefield 3 for PlayStation 3*	Christina Coffin
Wed 3:00	*Culling the Battlefield: Data Oriented Design in Practice*	Daniel Collin
Thu 1:30	*Lighting You Up in Battlefield 3*	Kenny Magnusson
Fri 4:05	*Approximating Translucency for a Fast, Cheap & Convincing Subsurface Scattering Look*	Colin Barré-Brisebois

For more DICE talks: http://publications.dice.se

References

A Bizarre Way to do Real-Time Lighting
http://www.spuify.co.uk/?p=323

Deferred Lighting and Post Processing on PLAYSTATION®3
http://www.technology.scee.net/files/presentations/gdc2009/DeferredLightingandPostProcessingonPS3.ppt

SPU Shaders - Mike Acton, Insomniac Games
*www.insomniacgames.com/tech/articles/0108/files/**spu_shaders**.pdf*

Deferred Rendering in Killzone 2
http://www.guerrilla-games.com/publications/dr_kz2_rsx_dev07.pdf

Bending the graphics pipeline
http://www.slideshare.net/DICEStudio/bending-the-graphics-pipeline

A Real-Time Radiosity Architecture
http://www.slideshare.net/DICEStudio/siggraph10-arrrealtime-radiosityarchitecture