SPU-based Deferred Shading for Battlefield 3 on Playstation 3

Christina Coffin
Platform Specialist, Senior Engineer
DICE

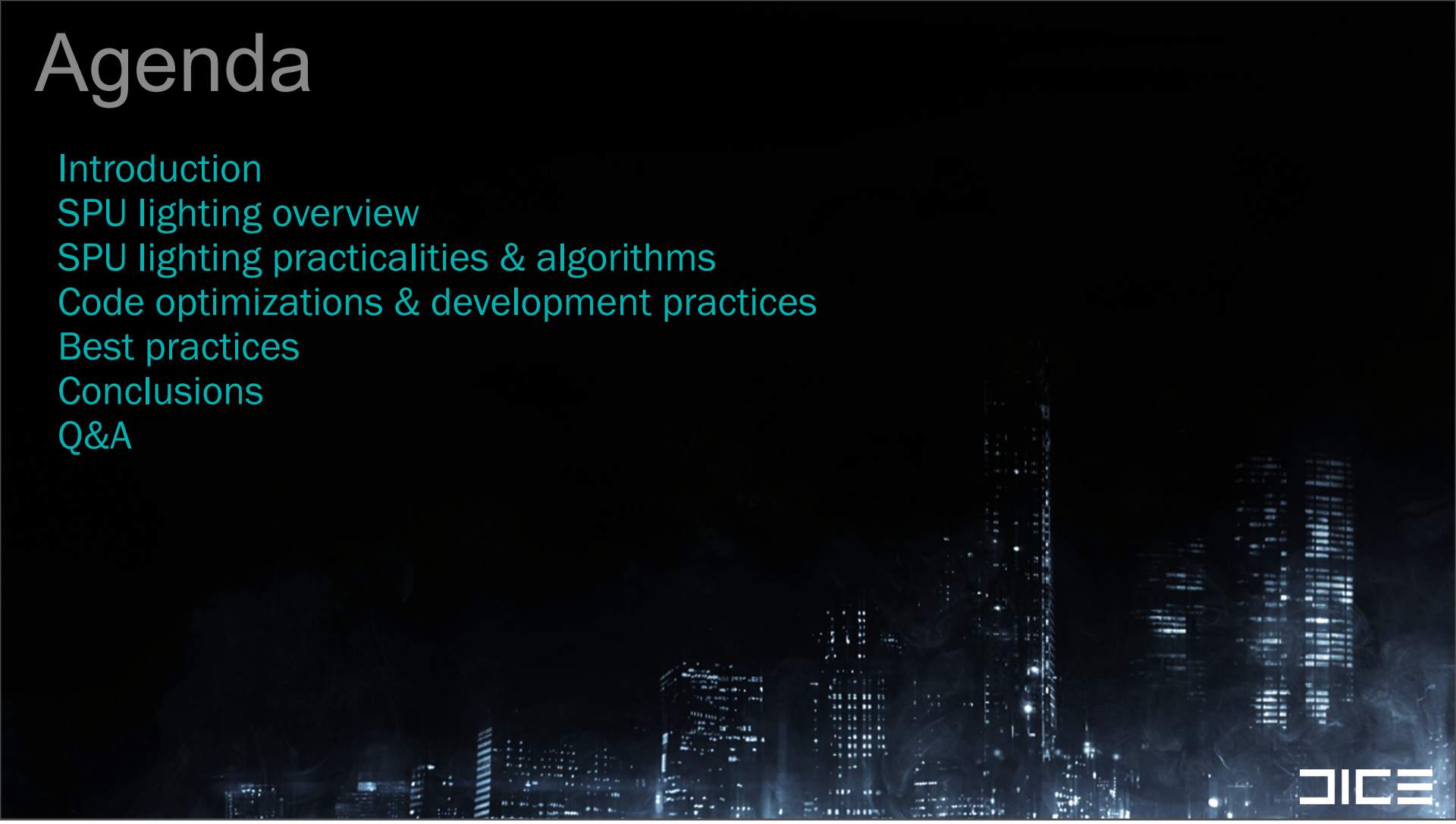
GDC

DICE

Tuesday, March 8, 2011

Agenda

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



Tuesday, March 8, 2011

Before getting into the lowlevel bits of SPU based deferred shading in the frostbite 2 engine, I want to briefly cover the motivations behind this.

Introduction

Maxxing out mature consoles

Tuesday, March 8, 2011

Playstation 3 turns 5 years old this fall = mature console,

For some of us, that means we've been working with the hardware a year or two longer than that.

Developers want more out of fixed hardware, time to get creative! Everyone has already employed basic optimization strategies

Such as the SPU Edge libraries for mesh vertex processing, using half floats in gpu shaders and so on...

Days of writing code once that runs on all platforms is over for some things, especially rendering.

>>We must be competitive versus First party / single console developers that have the luxury of only worrying about 1 platform.

Past: Frostbite 1

Tuesday, March 8, 2011

Before getting into the lowlevel bits of SPU based deferred shading in the frostbite 2 engine, I want to briefly cover the motivations behind this.

Past History:

Frostbite 1 (Bad Company 2 / Medal of Honor MP)

- Forward rendered
- Limitations with Night time and indoor lighting
- Limited Light sources

Past: Frostbite 1

Forward Rendered + Destruction + Limited Lighting

DICE

Tuesday, March 8, 2011

Before getting into the lowlevel bits of SPU based deferred shading in the frostbite 2 engine, I want to briefly cover the motivations behind this.

Past History:

Frostbite 1 (Bad Company 2 / Medal of Honor MP)

- Forward rendered
- Limitations with Night time and indoor lighting
- Limited Light sources

Tuesday, March 8, 2011

For Mirror's edge using a modified version of UE, the precomputed global illumination looks beautiful, but it is static.

Editing the world required regenerating lightmaps which took time, which made things slow for artists and designers building and tweaking level layouts or lighting.

Battlefield 3 has destructible environments which also doesn't work with static prebaked lighting.

So we need a new way to do our lighting.....

If you've played Battlefield Bad Company 2, then you are familiar with our destruction tech. Precomputed lighting just won't do.

We needed buildings to be destructible and have their attached light sources turn on or off as needed.

Tuesday, March 8, 2011

For Mirror's edge using a modified version of UE, the precomputed global illumination looks beautiful, but it is static.

Editing the world required regenerating lightmaps which took time, which made things slow for artists and designers building and tweaking level layouts or lighting.

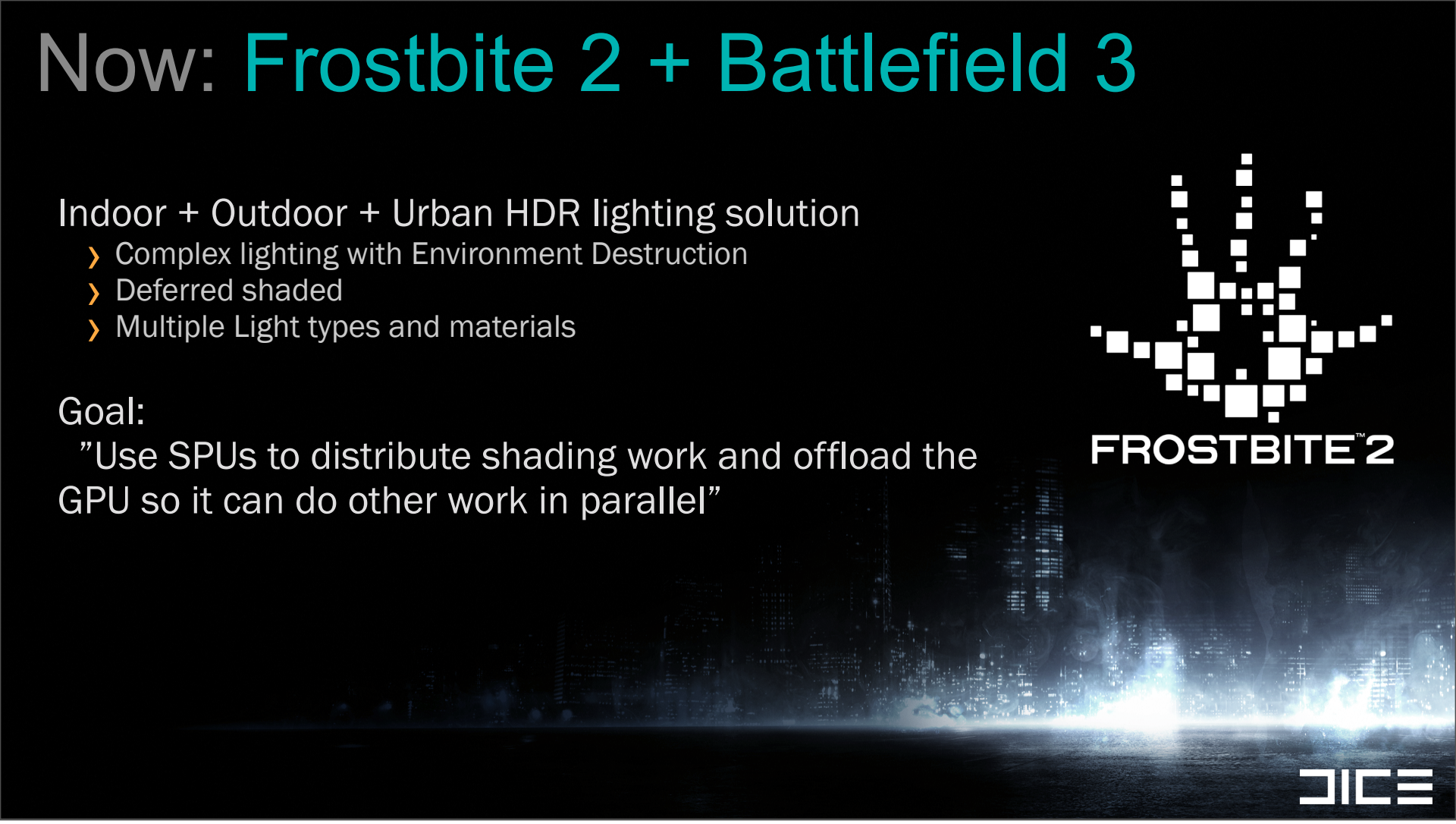
Battlefield 3 has destructible environments which also doesn't work with static prebaked lighting.

So we need a new way to do our lighting.....

If you've played Battlefield Bad Company 2, then you are familiar with our destruction tech. Precomputed lighting just won't do.

We needed buildings to be destructible and have their attached light sources turn on or off as needed.

Now: Frostbite 2 + Battlefield 3



Tuesday, March 8, 2011

Indoor+Outdoor+Urban HDR lighting solution

That works with environment destruction!

Global Illumination

A lot more light sources & light types

Multiple lighting models

Real-time level building with dynamic lighting so lighting artists get immediate feedback

<insert a lot of images>

For more information about the Art direction of Battlefield 3:

Kenny Magnussons's ”Lighting you up in Battlefield 3” here at GDC'11

Why SPU-based Deferred Shading?

Want more interesting visual lighting + FX

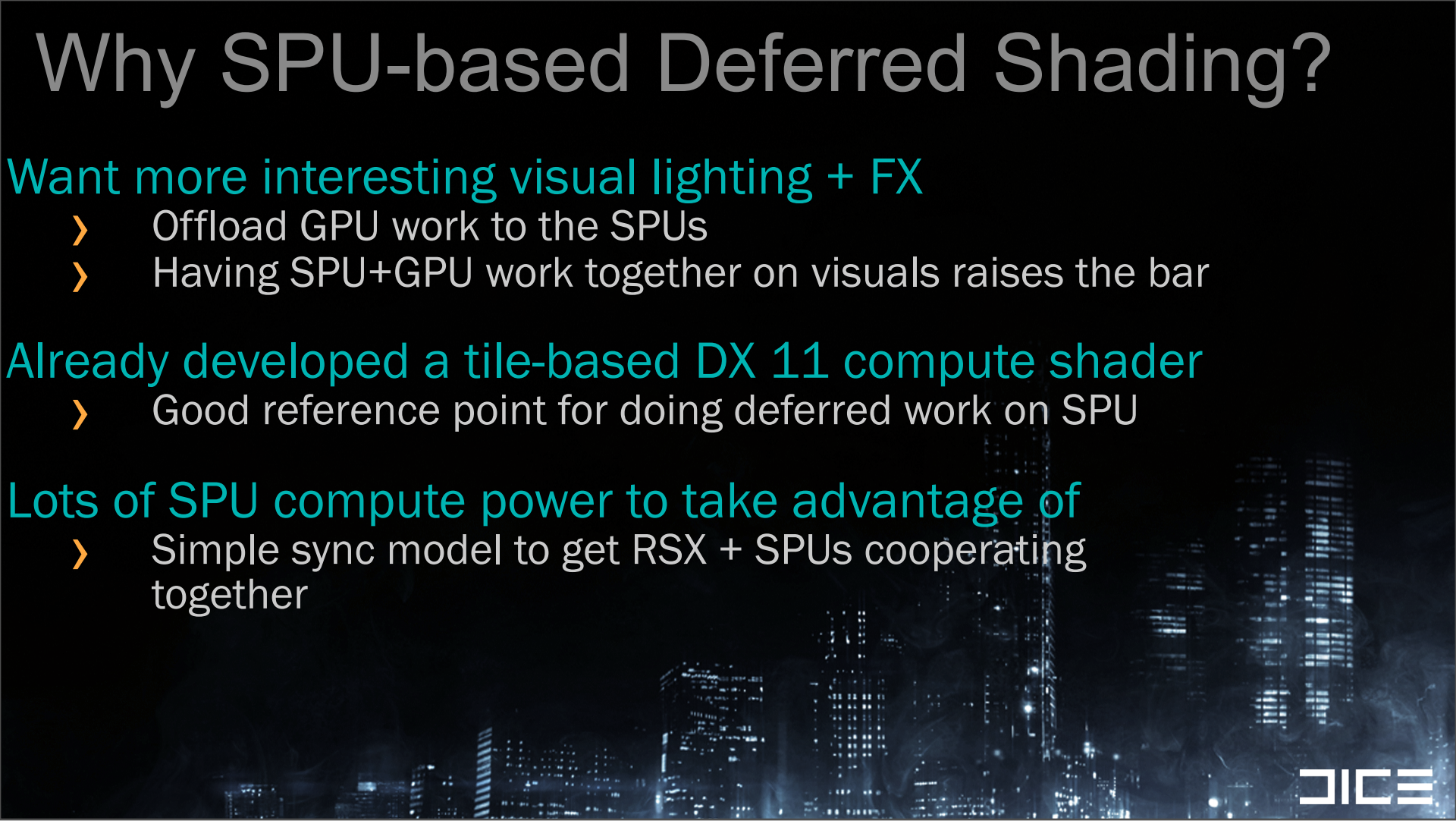
- Offload GPU work to the SPUs
- Having SPU+GPU work together on visuals raises the bar

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



Tuesday, March 8, 2011

A fair question is why do this stuff on SPU:

For our PC SKU, we support DX 11 and developed a tile based compute shader that was doing things similar to what we needed to do if we were going to do deferred shading processing on SPUs which made us more confident in the approach.

From an optimistic viewpoint, doing traditional GPU work on SPU work means you can make everything better, because you don't have to only rely on the GPU for better rendered visuals.

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

Tuesday, March 8, 2011

- An important point to note with our lighting is that it matches the visual quality of its RSX counterpart because it is performed at full float precision so there are no banding artifacts or cheap approximations causing any visual divergence versus the pure GPU implementation.

Multiple lighting models + materials

- Standard
- Metallic
- Skin
- Translucency

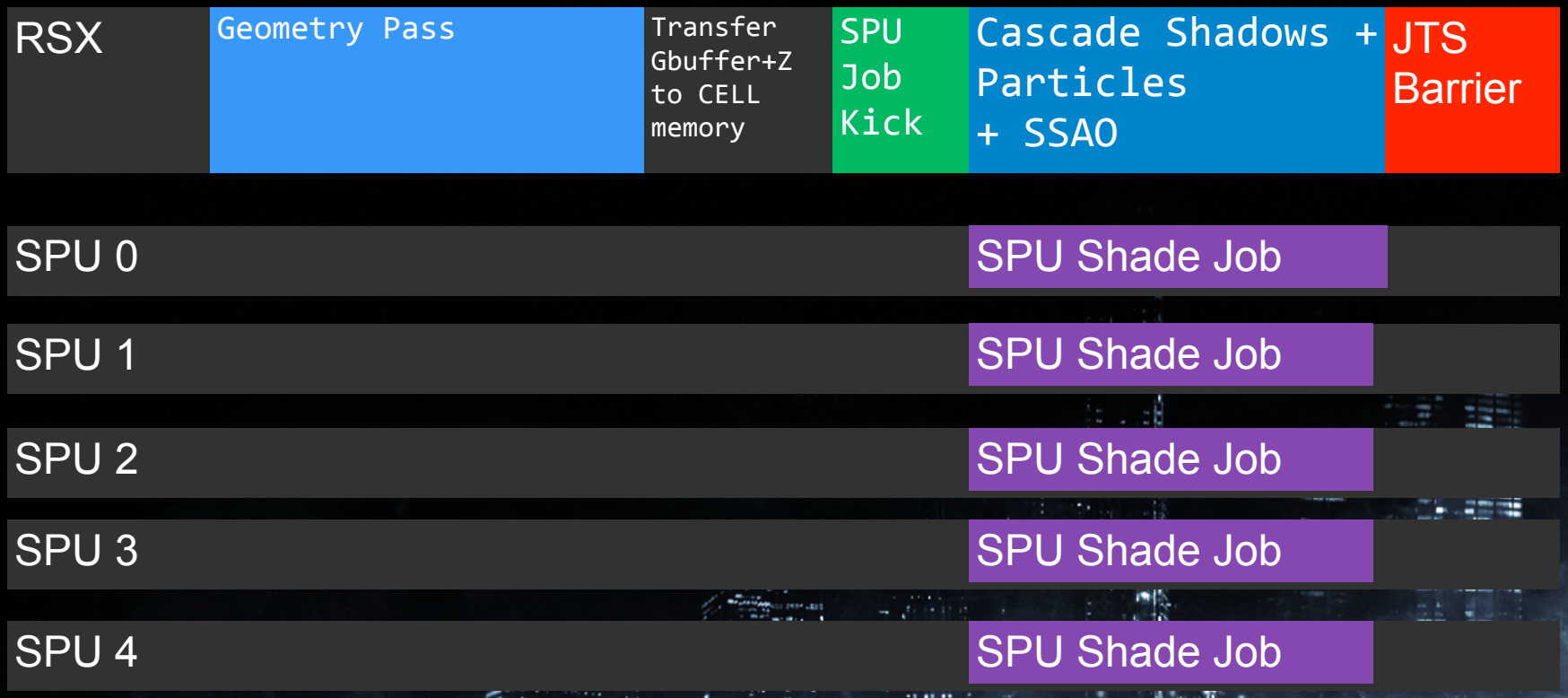
DICE

Tuesday, March 8, 2011

We want good lighting and material variety to realize our goals, so we devoted enough bits to the material related fields so multiple projects using frostbite 2 could realize their game specific Art style and visual goals

Rendering Frame Timeline

Note: Sizes not proportional to time taken!



Tuesday, March 8, 2011

SPUs doing shading are like little tiny GPU ALU units. (SPUs) They aren't really made for doing things like sampling textures, but they are good for number crunching, which is what we are making them do when we ask them to shade our gbuffer data with lots of light sources.

Opaque Geometry Pass

+ AlphaTest

GPU render gbuffer

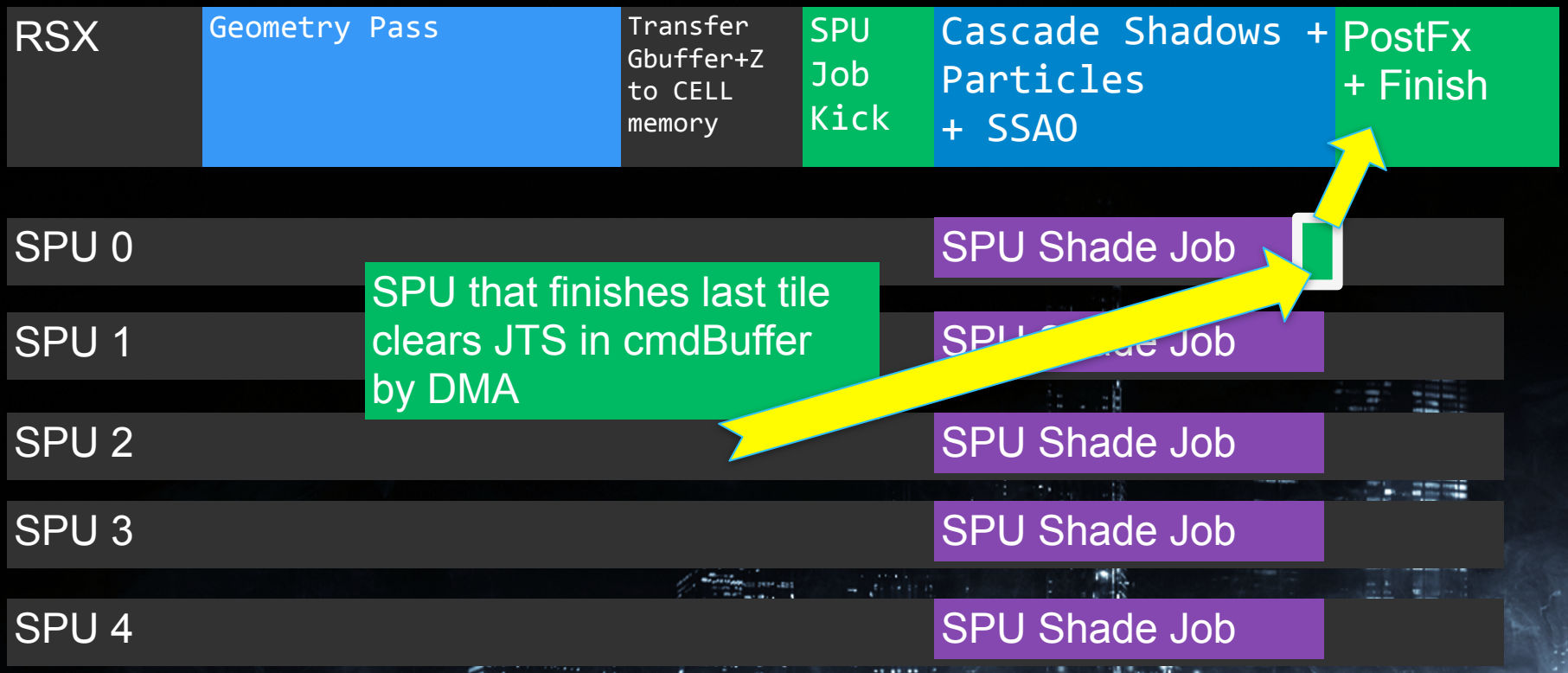
GPU starts SPU jobs

GPU does other work while SPU Shading runs

GPU waits for SPU job completion (JTS)

Rendering Frame Timeline

Note: Sizes not proportional to time taken!



Tuesday, March 8, 2011

SPUs doing shading are like little tiny GPU ALU units. (SPUs) They aren't really made for doing things like sampling textures, but they are good for number crunching, which is what we are making them do when we ask them to shade our gbuffer data with lots of light sources.

Opaque Geometry Pass

+ AlphaTest

GPU render gbuffer

GPU starts SPU jobs

GPU does other work while SPU Shading runs

GPU waits for SPU job completion (JTS)

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)			

The top 3 of these buffers are the ones that I will cover in this talk, the dynamic radiosity part is worthy of an entire talk all on its own.

Its important to note that all of the lighting I talk about here does contribute to the radiosity solver.

For more information, please refer to this talk:

http://www.slideshare.net/DICEStudio/siggraph10-arrrealtime-radiosityarchitecture

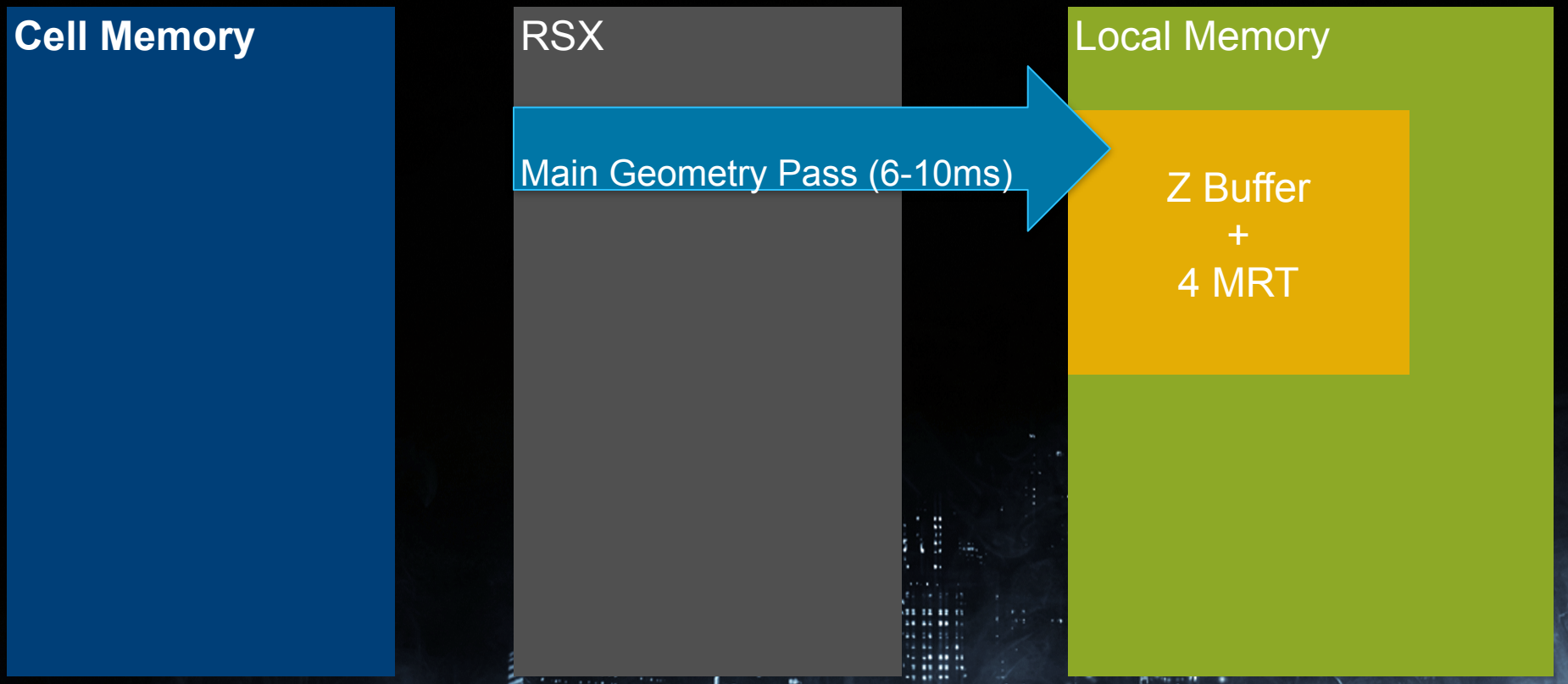
First of all, the RSX has to rasterize the gbuffer (4x ARGB8888) + zbuffer(1280x720 Z24S8) data.

This data also needs to get into main memory where the SPUs can quickly grab it and start working on it in tiles.

We begin by rasterizing to 4 tiled MRTs (multiple render targets) to populate the zbuffer+gbuffer which resides in RSX memory.

At this time, SPUs use the edge libraries and our SPU coarse depth rasterizer to keep this step efficient, removing unnecessary draw batches and triangles from going to the RSX.

Setup Rendering Data Flow



Tuesday, March 8, 2011

RSX transfer the data to main memory. Cheaper to render to rsx mem then render xfer to cell memory for SPU access

1.3ms RSX time to copy, 2-3ms faster overall

Due to 4MRT performance cliff rendering directly to cell memory.

Bonus:

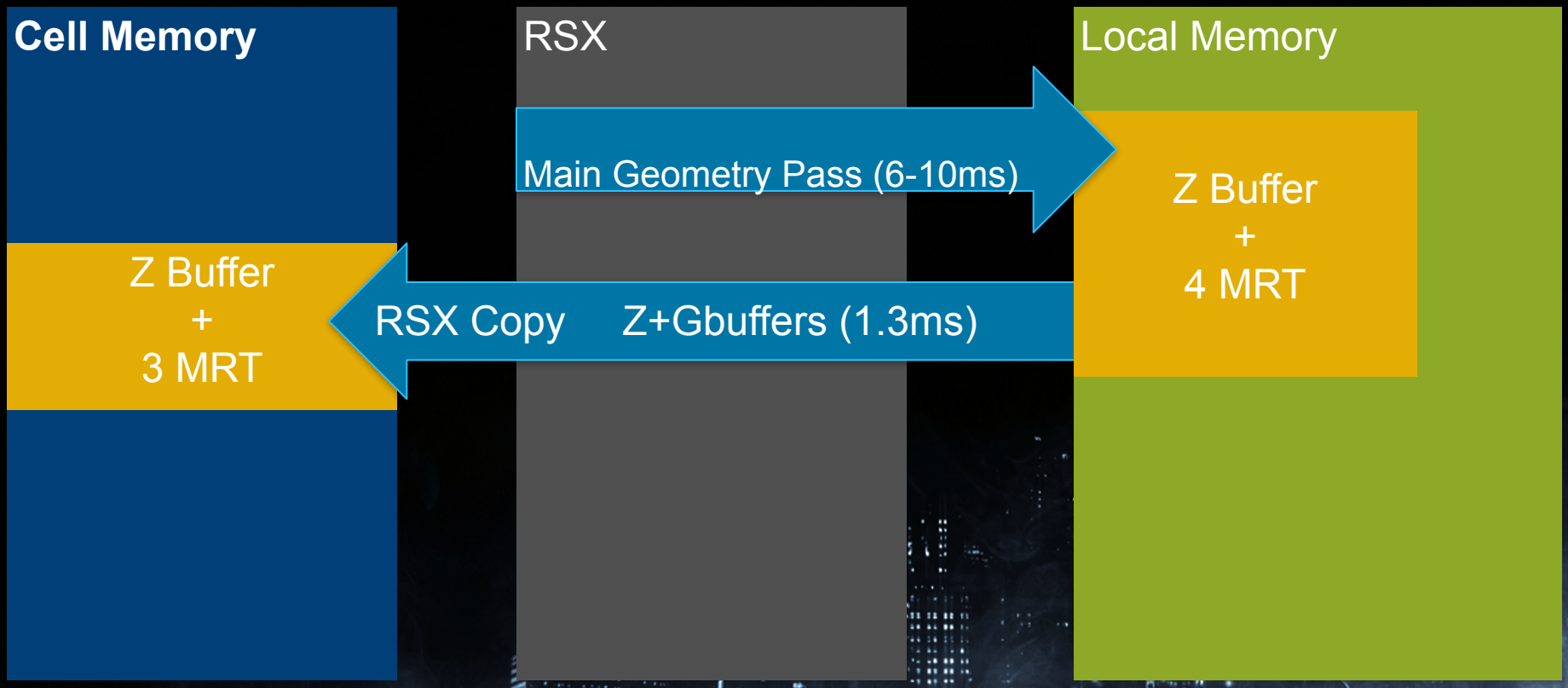
RSX can safely do lighting work on its own copy in parallel with SPU work.

At the start of the frame, we added our SPU shading jobs to the scheduler as a 'disabled' job.

The job becomes enabled by the RSX via an inline dword transfer to a cell memory address that tells the kernel this job can now run.

Lighting job is added at the beginning of the frame as a 'disabled job' until the dword transfer to the specified CE

Setup Rendering Data Flow



Tuesday, March 8, 2011

RSX transfer the data to main memory. Cheaper to render to rsx mem then render xfer to cell memory for SPU access

1.3ms RSX time to copy, 2-3ms faster overall

Due to 4MRT performance cliff rendering directly to cell memory.

Bonus:

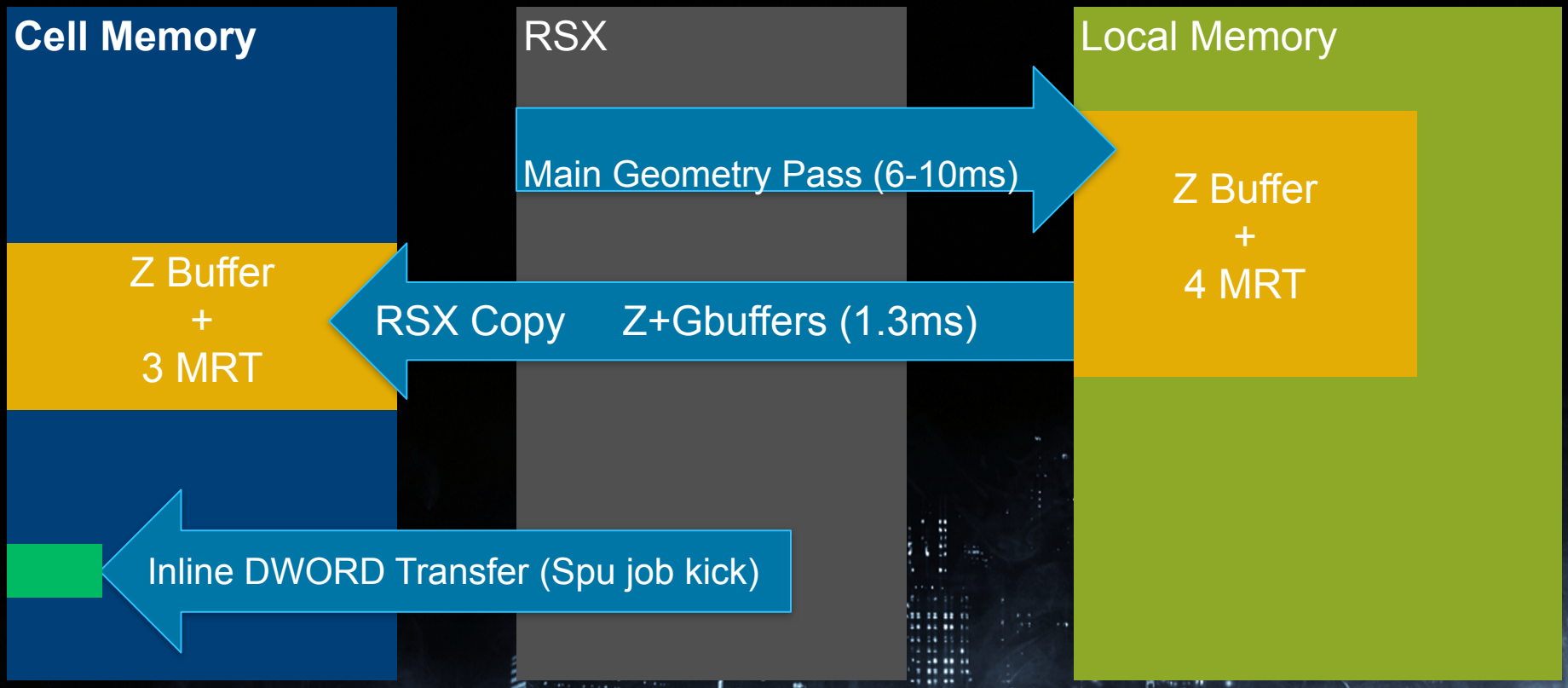
RSX can safely do lighting work on its own copy in parallel with SPU work.

At the start of the frame, we added our SPU shading jobs to the scheduler as a 'disabled' job.

The job becomes enabled by the RSX via an inline dword transfer to a cell memory address that tells the kernel this job can now run.

Lighting job is added at the beginning of the frame as a 'disabled job' until the dword transfer to the specified CE

Setup Rendering Data Flow



Tuesday, March 8, 2011

RSX transfer the data to main memory. Cheaper to render to rsx mem then render xfer to cell memory for SPU access

1.3ms RSX time to copy, 2-3ms faster overall

Due to 4MRT performance cliff rendering directly to cell memory.

Bonus:

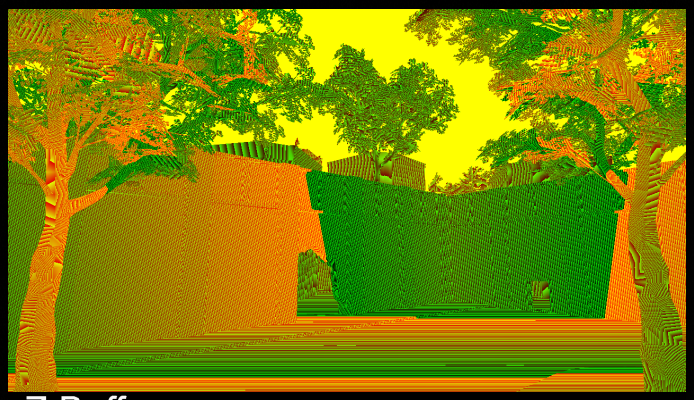
RSX can safely do lighting work on its own copy in parallel with SPU work.

At the start of the frame, we added our SPU shading jobs to the scheduler as a 'disabled' job.

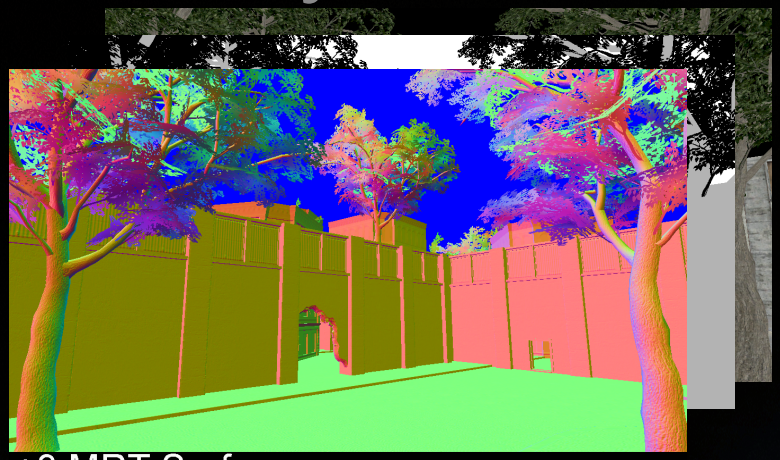
The job becomes enabled by the RSX via an inline dword transfer to a cell memory address that tells the kernel this job can now run.

Lighting job is added at the beginning of the frame as a 'disabled job' until the dword transfer to the specified CE

Source data in CELL memory



Z Buffer
+ Stencil



+3 MRT Surfaces

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

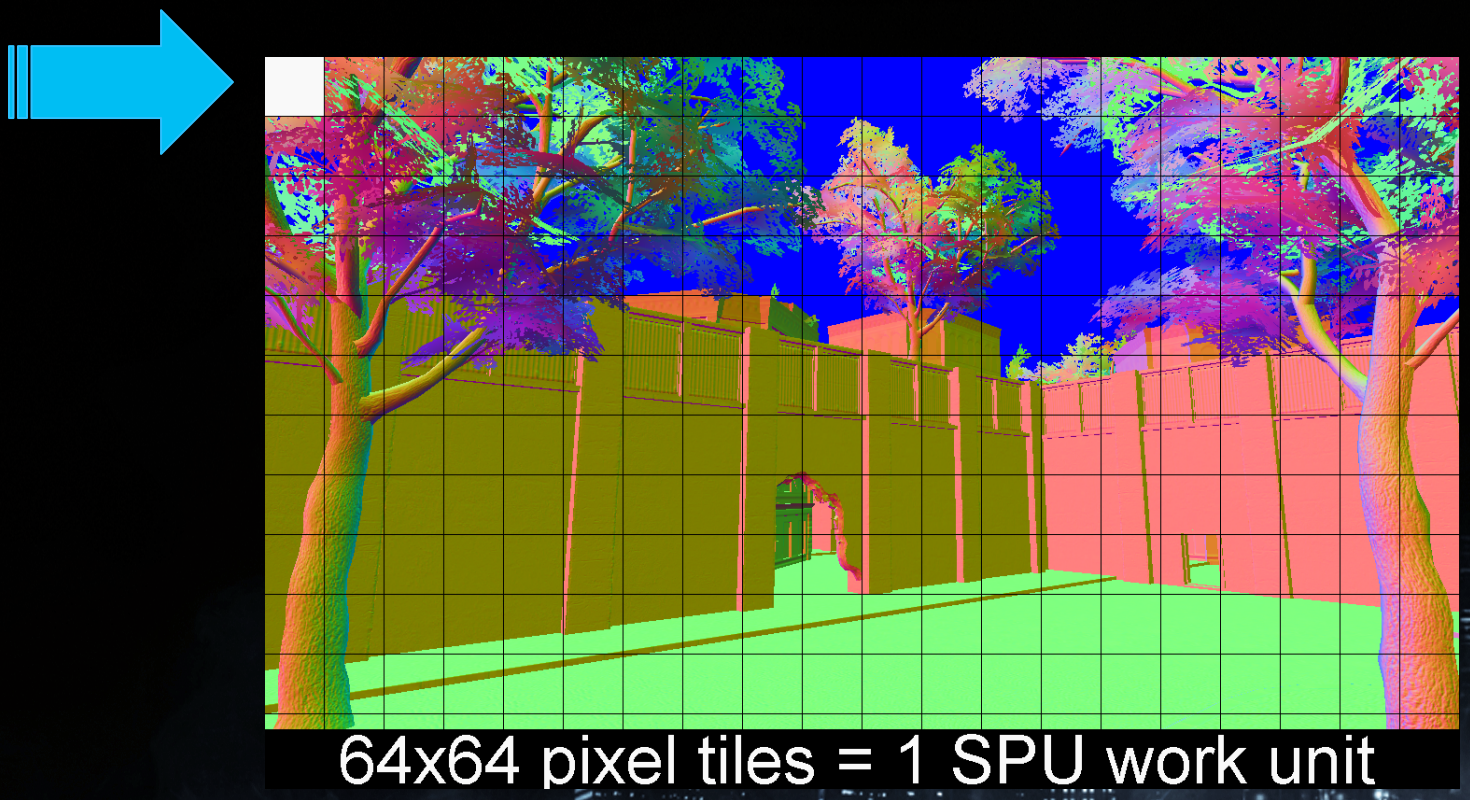
SPU SPU SPU
SPU SPU SPU

Tuesday, March 8, 2011

So now that the RSX has tranferred the required Zbuffer data and MRT data to CELL memory where the SPUs can access it,
And we also have a packed flat list of of all visible lights within our camera frustum, we need to feed this data to 6 hungry SPUs.

So now we need to figure out how to divide up the work and work within the hardware limitations of the SPUs that have very little Local store memory (256k) but have fast DMA.

SPU Tile Based Shading work units



Tuesday, March 8, 2011

We divide the screen into 64x64 pixel tiled regions, which becomes our core workload unit per SPU.

When an SPU reserves a tile, it is responsible for doing all the shading work on that tile.

SPU Shading Flow Overview

Tuesday, March 8, 2011

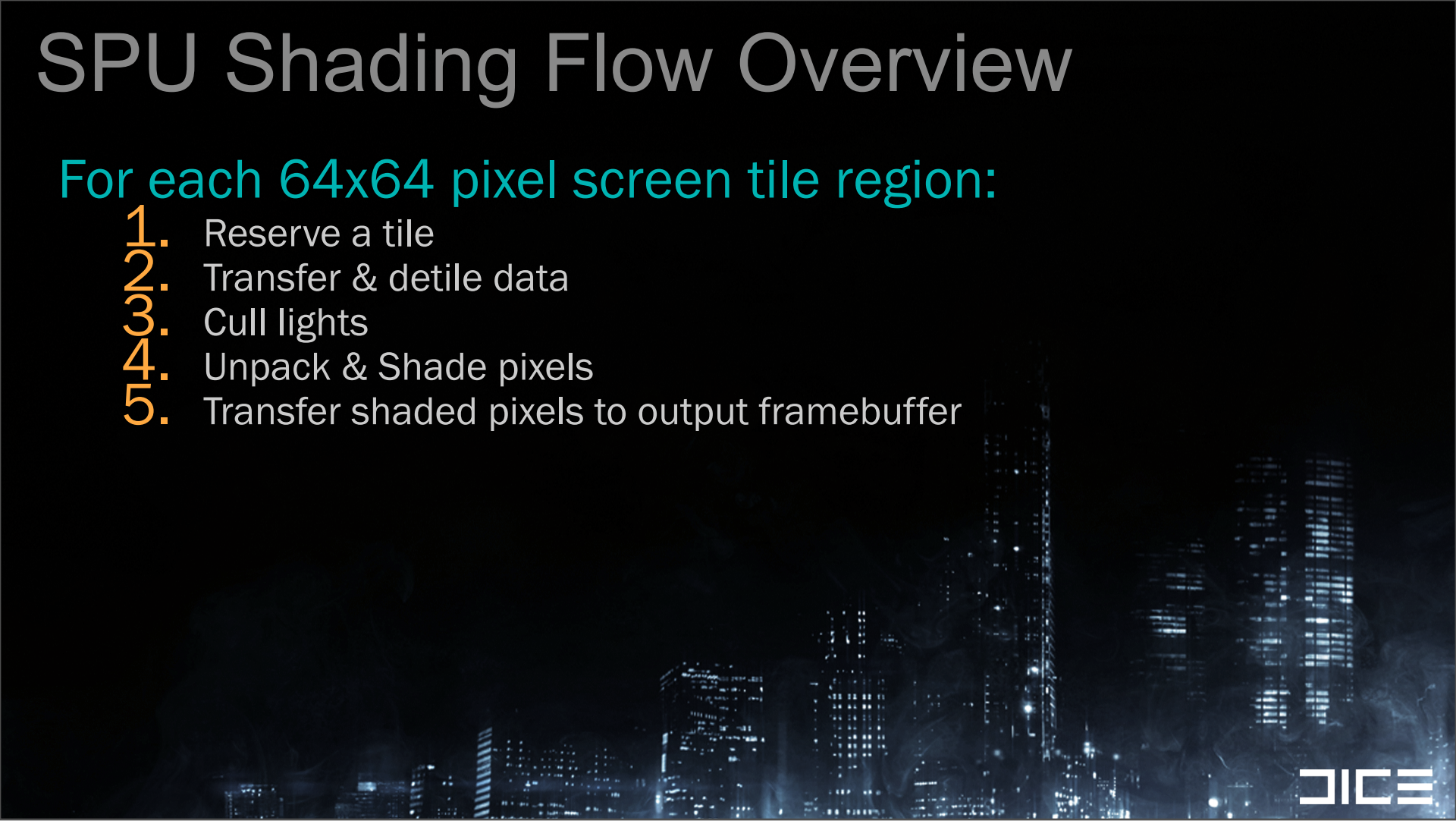
Here are the rough steps we do to calculate our shading work on tiled regions on the SPUs.

I will cover each one in some depth, then touch on a few of these steps again later in the optimization and algorithm section on how to make these steps more efficient.

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



Tuesday, March 8, 2011

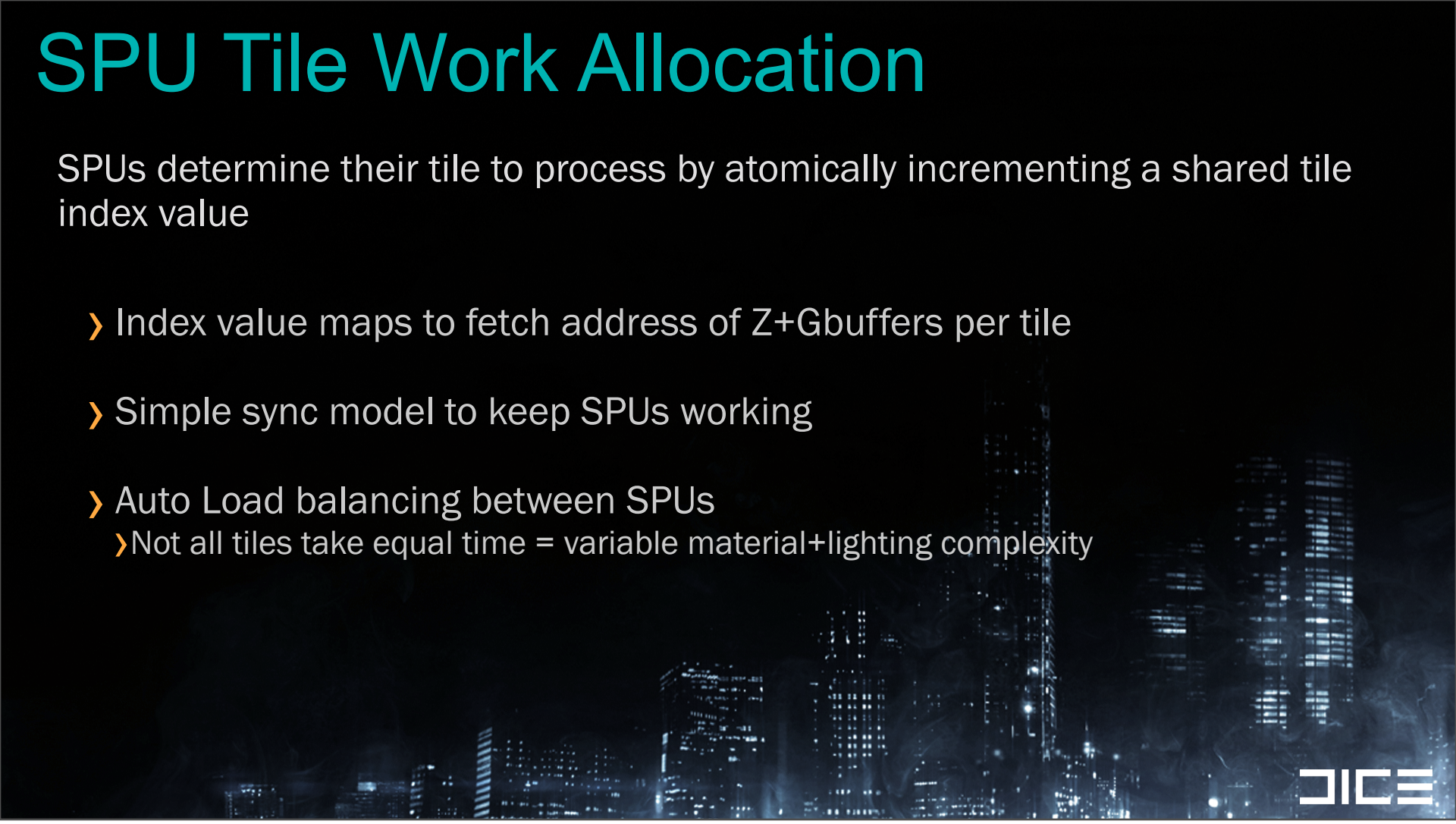
Here are the rough steps we do to calculate our shading work on tiled regions on the SPUs.

I will cover each one in some depth, then touch on a few of these steps again later in the optimization and algorithm section on how to make these steps more efficient.

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



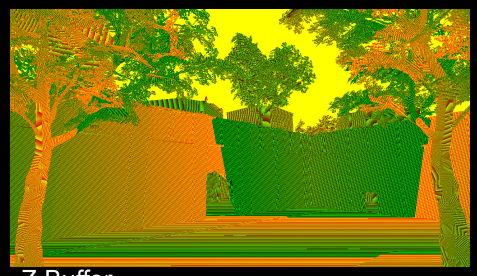
SPUs follow a dynamic tile arbitration scheme using atomic increment similar to (Tovey 2009) to determine the next tiled region of memory they will operate on.

The shared tile index value resides in CELL Memory

Some tiles take longer to process than others due to light and material complexity varying per tile, so using this scheme to dish out work to SPUs is better than using fixed allocations of screen area to each SPU.


Getting the Tile Data onto SPUs....
DATA TRANSFER + DE-TILING

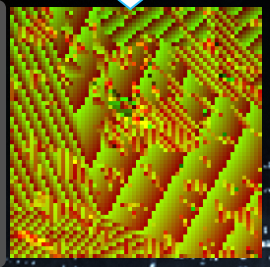
FB Tile Data Fetch

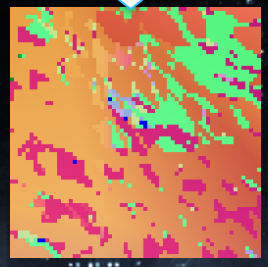


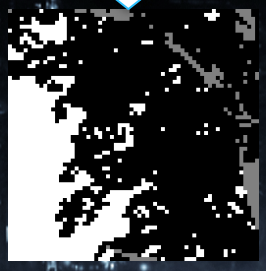
DMA Traffic | Get ZBuffer | Get MRT 0 | Get MRT 1 | Get MRT 2

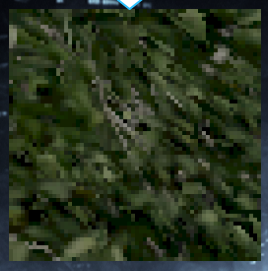
SPU Code | Detile Z | Detile 0 | Detile 1 | Detile 2

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









Tuesday, March 8, 2011

x4 DmaGet to SPU LS

3 Gbuffer Channels + Zbuffer

Detiled to linear format in LS

Overlap detiling code w/async DMA

We double buffer DMA get the light list so we can execute culling work on the lights while the DMA for the next light batch is in flight.

DmaGet batches of 16 lights into LS + cull vs FB Tile frusta + depth extents

Double Buffer DMA, do cull work while next batch in flight

SPU LIGHT TILE CULLING

Tuesday, March 8, 2011

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

At this point the SPU code does the remainder of its work in subtiles which is 32x16 pixels in size.

Using the zbuffer information in SPU local store, We determine a Subtile min/max (ignoring sky pixel depth) /full Z range extents.
The same culling optimization that we performed on the parent FB tile to skip the 'all sky pixels' case still applies here.

SPU SHADING LOOP

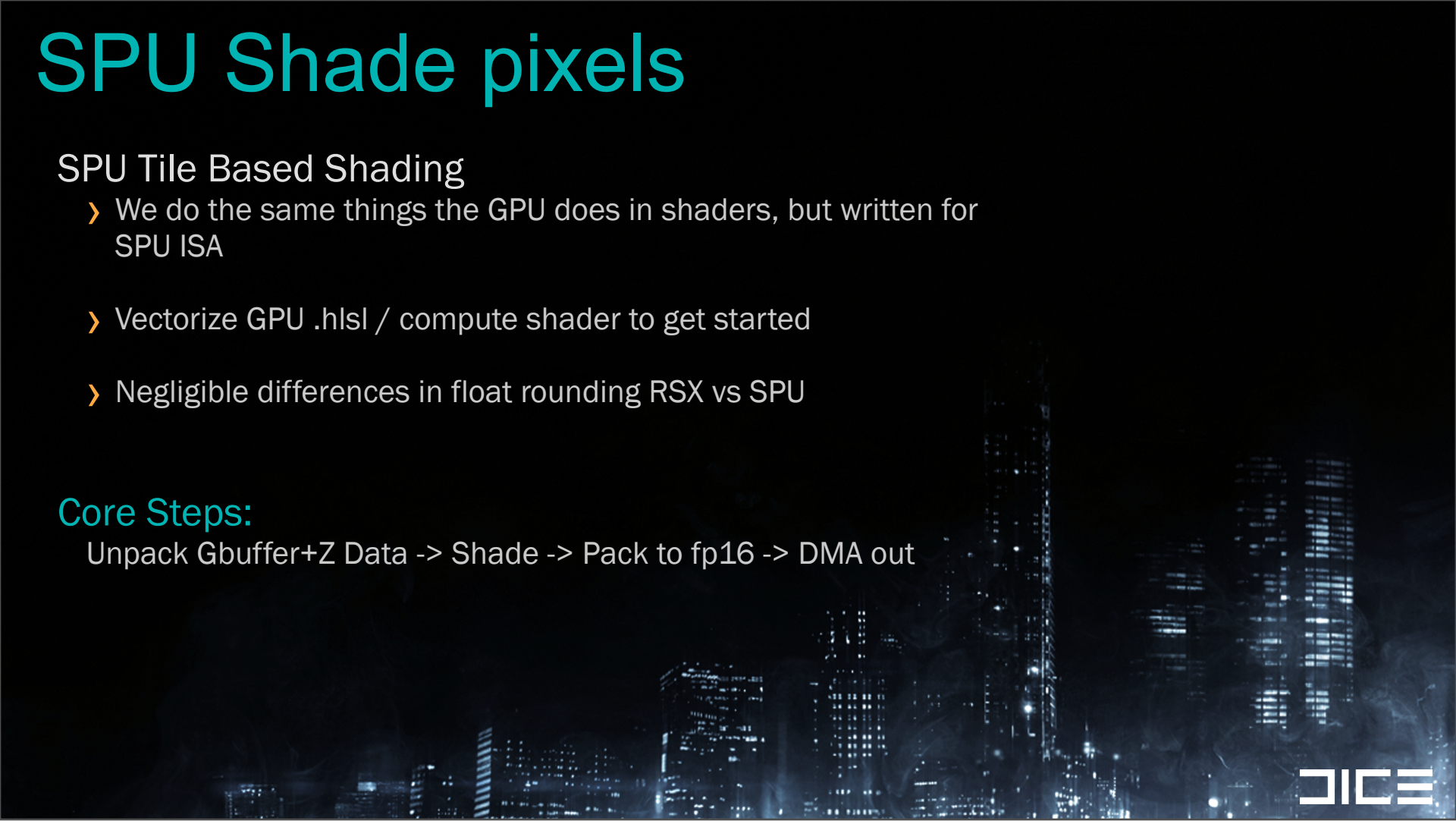
SPU Shade pixels

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
- Negligible differences in float rounding RSX vs SPU

Core Steps:

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



Tuesday, March 8, 2011

Difference in float rounding modes between RSX and SPU based shading doesn't create any significant divergence

Its common to use 'half' type in RSX shaders, but with SPU shading we always use 'float' precision so the SPU version is better ☺

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)

Tuesday, March 8, 2011

What exactly are we doing in our SPU shader?

Here are some example components of what shading work we do on SPUs.

A key point here: This isn't a comprehensive list of everything we do

Light Volume Clipping = explicit clipping planes combined with Light Shape Attenuation to limit the region of influence of where the light is applied.

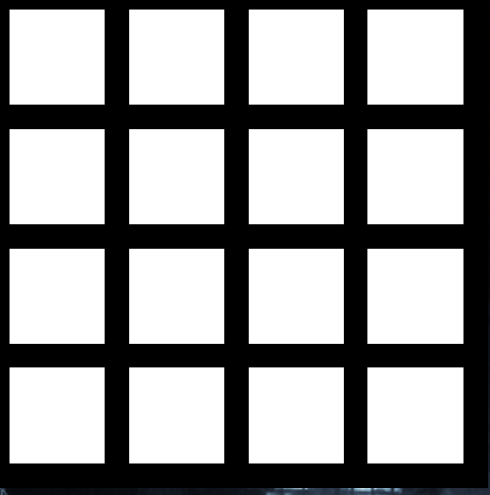
Light Shape Attenuation = Spot and Line light type shaping of the region the light influences

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



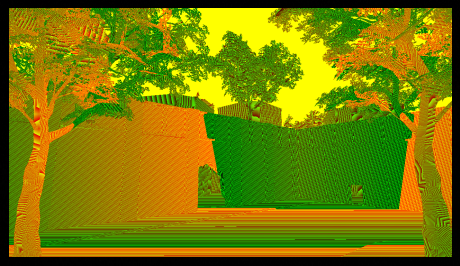
Tuesday, March 8, 2011

At the lowest level we have 4x4 pixel quads that we work on when we shade our pixels.

Notice that the 4x4 quad shape lends itself better to spatial coherency than a 16x1 horizontal span

Another important point is that unlike some GPU shader where we work at half precision, we stick to using floats which the SPU is designed to work on efficiently.

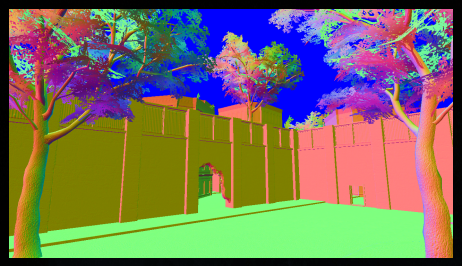
Gbuffer data expansion to SoA for shading



Depth + Stencil

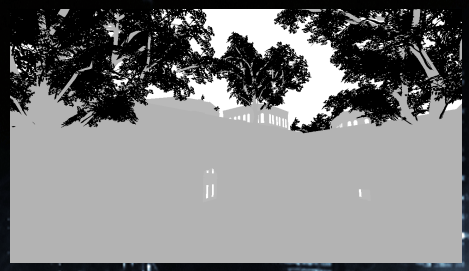
Spec Albedo

Diffuse Albedo



Normals

Smoothness



Material ID

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

Tuesday, March 8, 2011

Images: debug visualization of some gbuffer channel data:

The 3 gbuffers plus Z buffer tile data in SPU is full of many different channels of data that all has to undergo some form of shuffling and unpacking.

For it to be usable and also be processed efficiently on the SPU, nearly all of it has to be converted to floating point format and converted to Structure of Arrays formatting.

The SPUs shufb instruction masks and shuffles data into structure of arrays format and prepares it so we can use the csflt instruction to convert it from integer 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

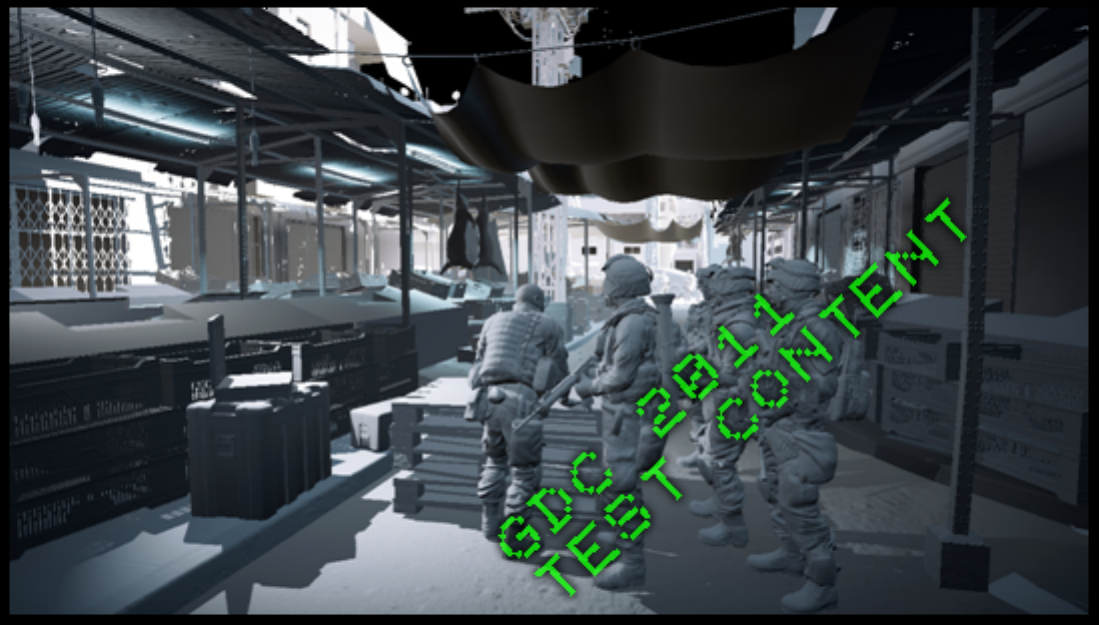


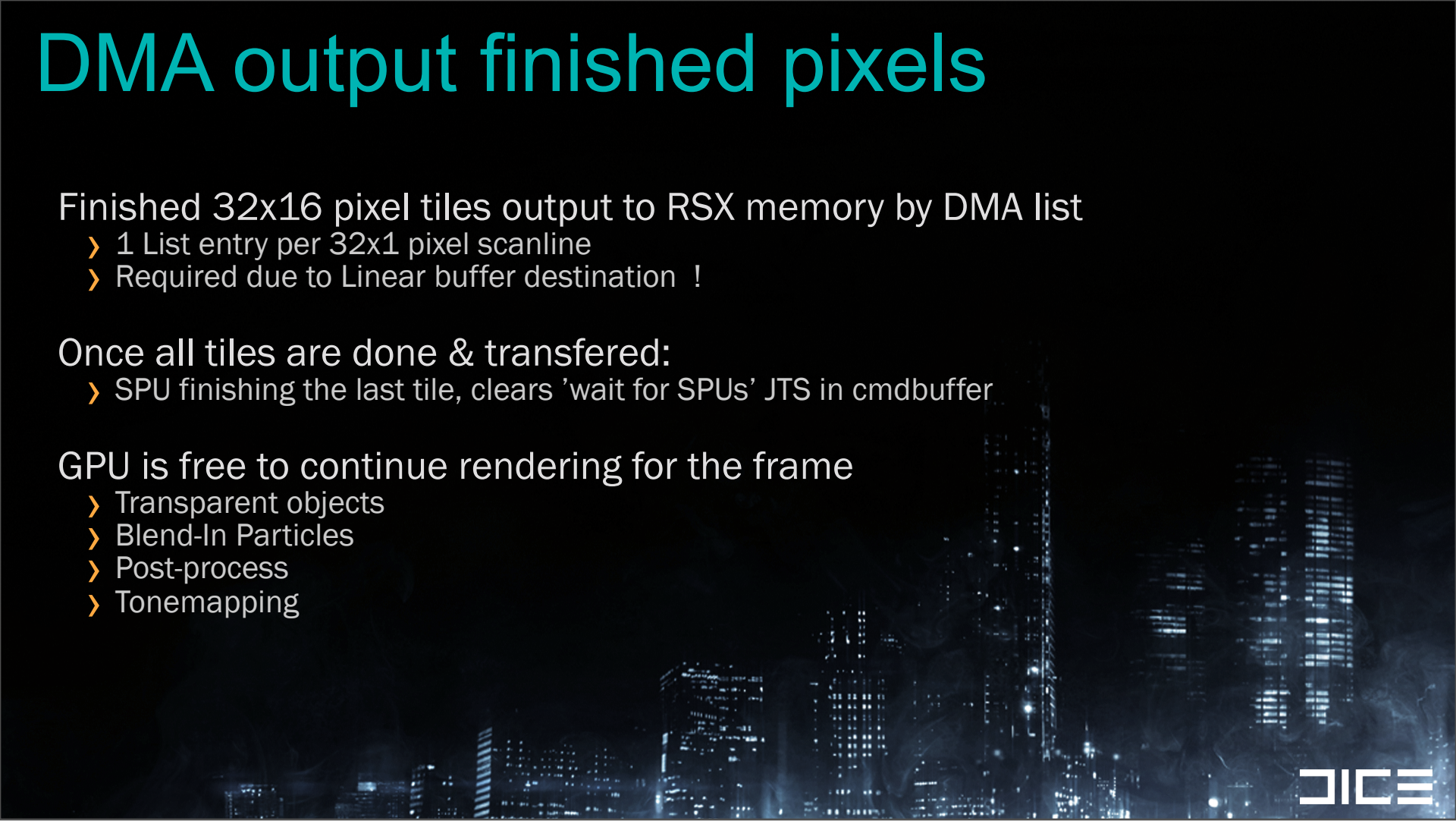
Image: deferred shadow pass+sunlight combines with calculated lighting from SPUs

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



Using a DMA list saves us about 1ms spu time per frame, otherwise we get too much overhead from issuing each 32x1 without dmalist

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

All this work can happen in parallel with the SPU Shading

Spotlights with texture projection or shadow projection work are done on the RSX and merged with the SPU work in another step

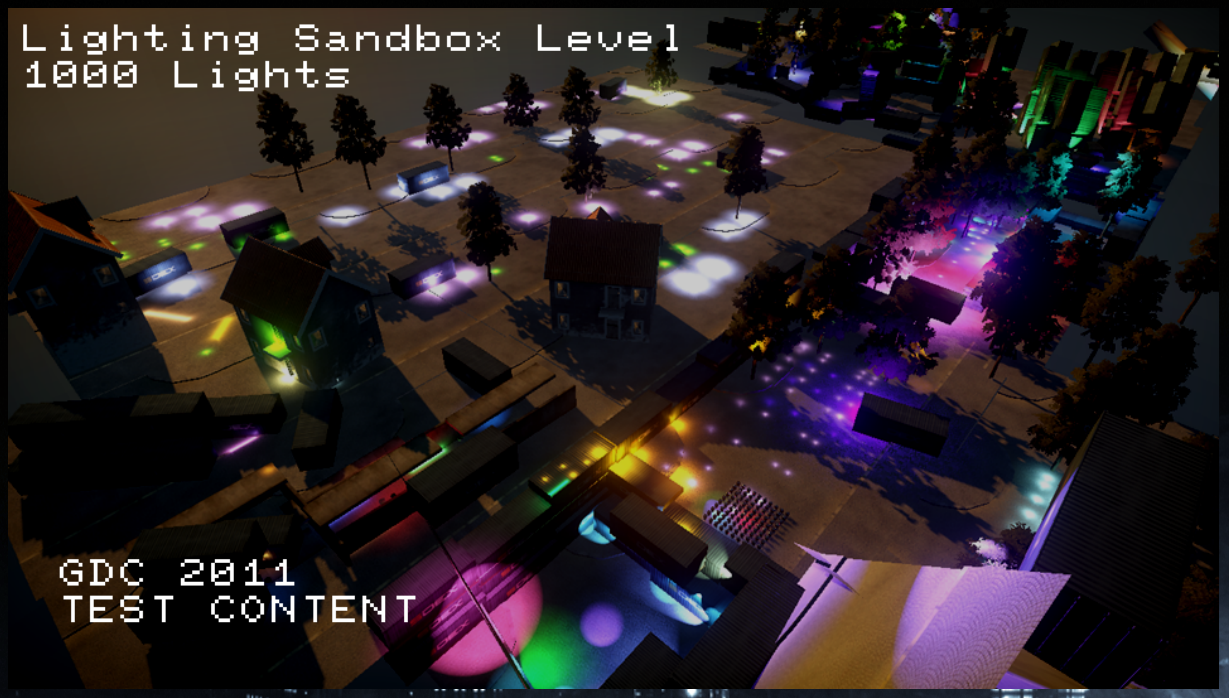
Tuesday, March 8, 2011

Algorithmic optimizations:

Tile Light Culling

Tile Based Culling System

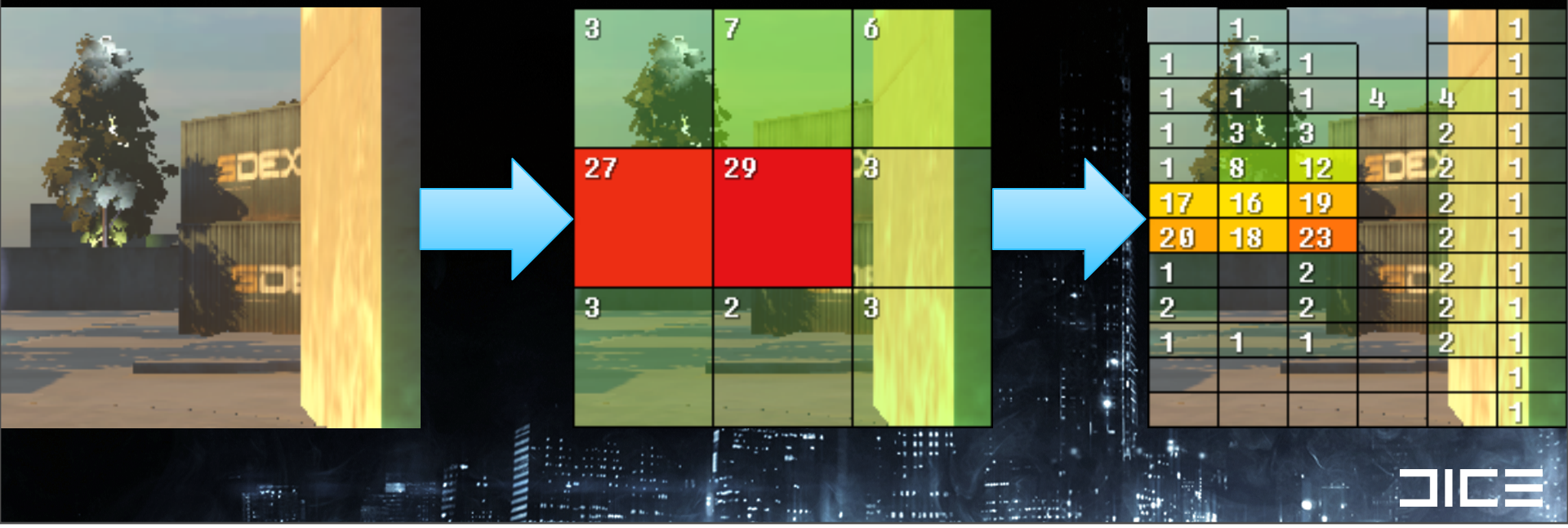
- Designed to handle extreme lighting loads
- Shading Budget
  - 40ms (split across 5 SPUs)
- Culling system overhead
  - 1-4ms (1000+ lights)



Tile Light Culling

2 Light Culling passes:

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



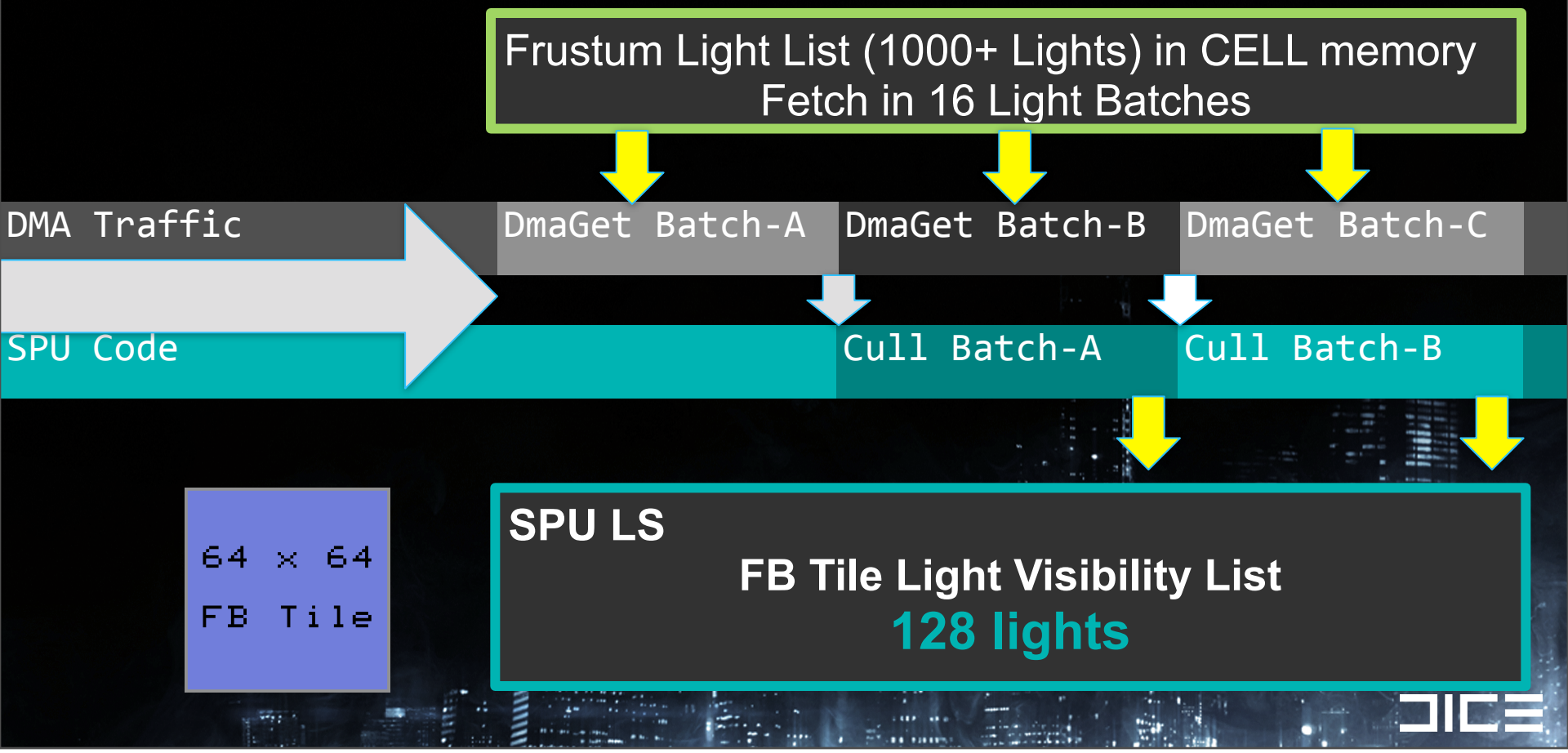
Tuesday, March 8, 2011

Here is an extreme usage case to illustrate how the culling system reduces the number of lights for 64x64 and 32x16 pixel subregions of the screen.

Stress that one of the big wins is completely removing all lights from a tile, because it removes unpacking, packing to fp16 and DMA out work among other things.

Once you have at least 1 light in a tile, your paying for unpacking+fp16 packing+dma out work (fixed overhead)

FB Tile Light Culling



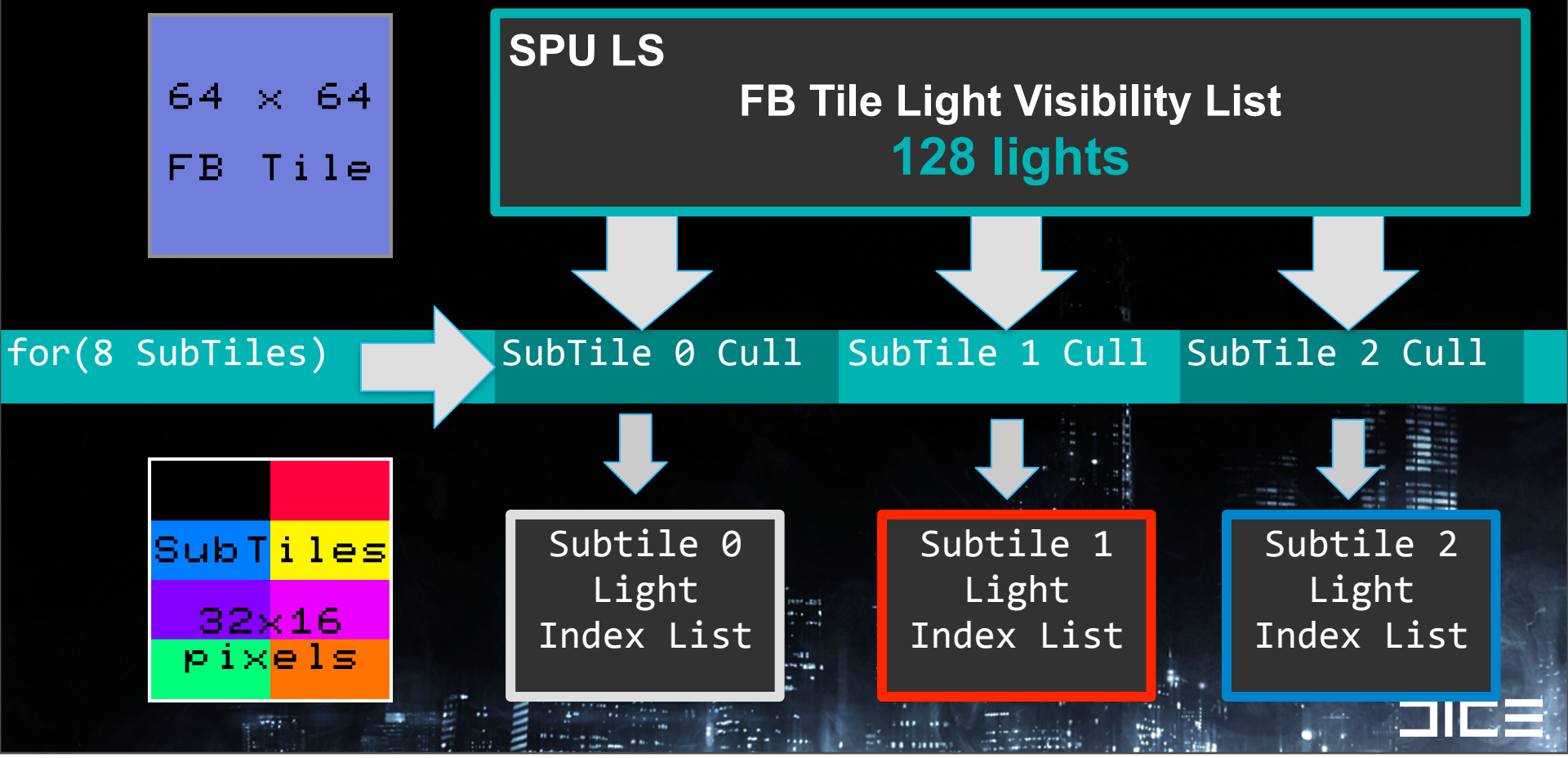
Tuesday, March 8, 2011

We double buffer DMA get the light list so we can execute culling work on the lights while the DMA for the next light batch is in flight.

DmaGet batches of 16 lights (single Block DMA) into LS + cull vs FB Tile frusta + depth extents

Double Buffer DMA, do cull work while next batch in flight !

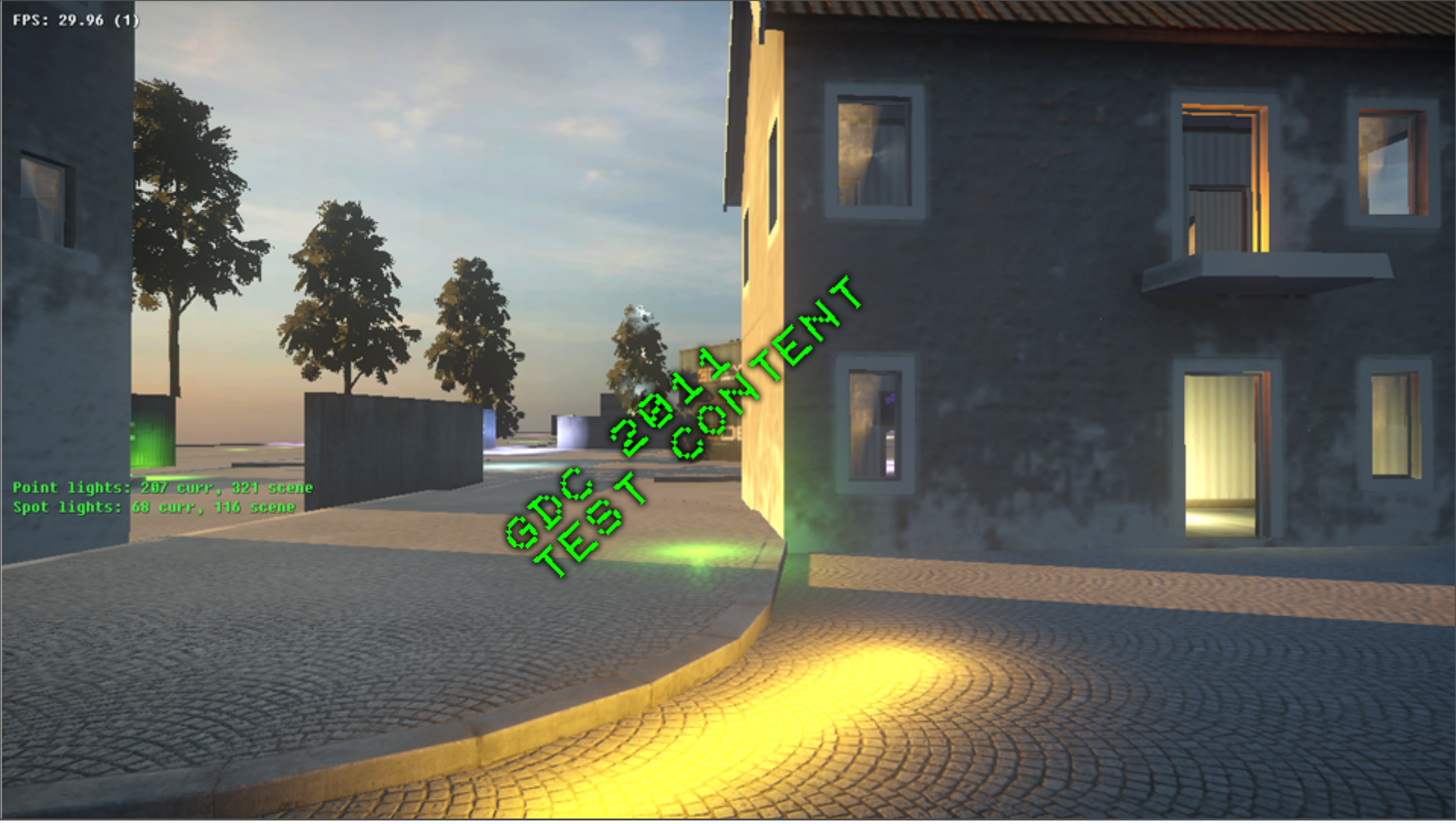
SubTile Light Culling



Tuesday, March 8, 2011

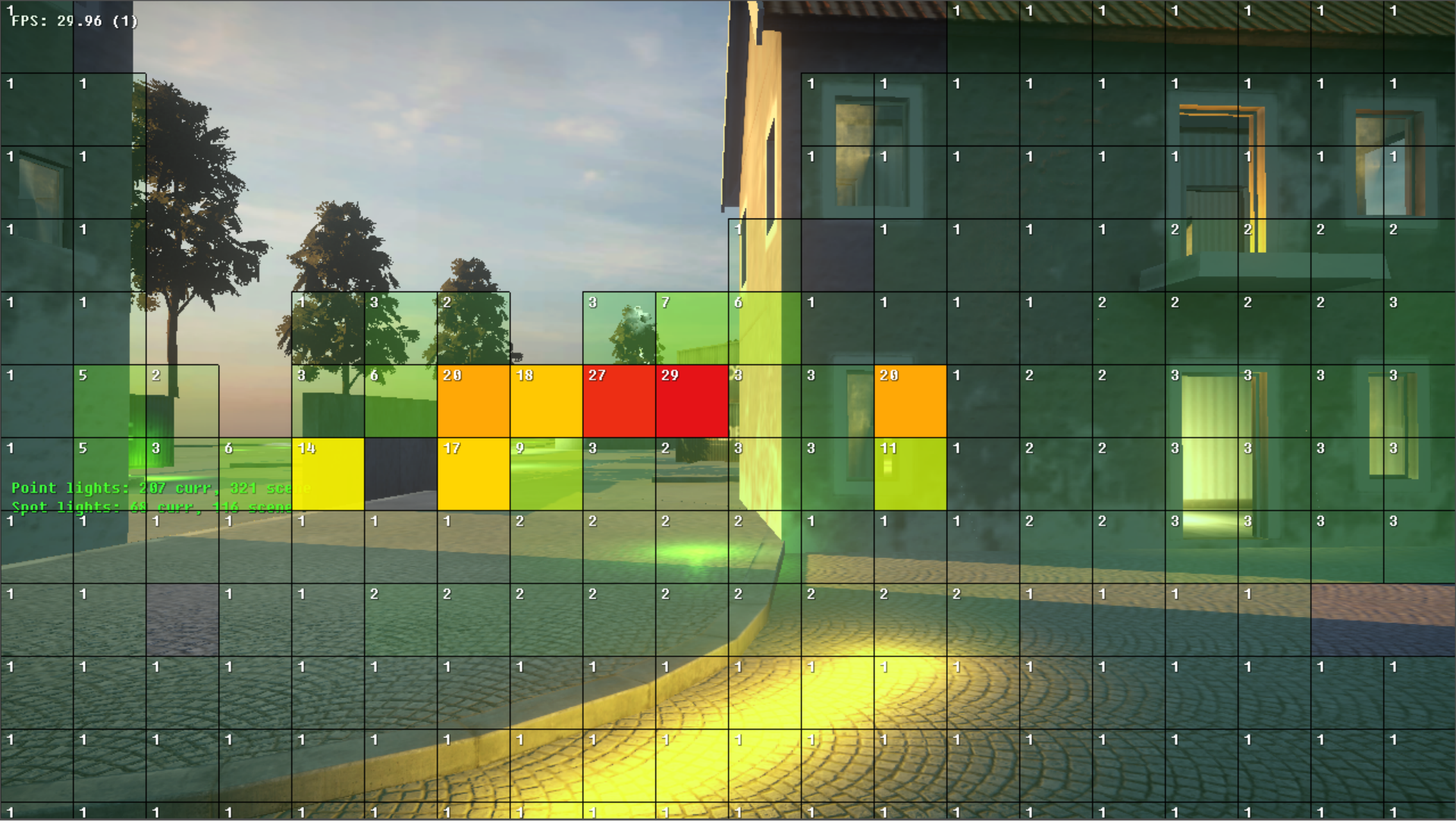
We subdivide the 64x64 pixel FB Tile into 8 smaller subtiles that are 32x16 pixels in size.

We do another culling pass on the lights at the subtile level to further reduce the amount of lighting work we need to do for each subtile.

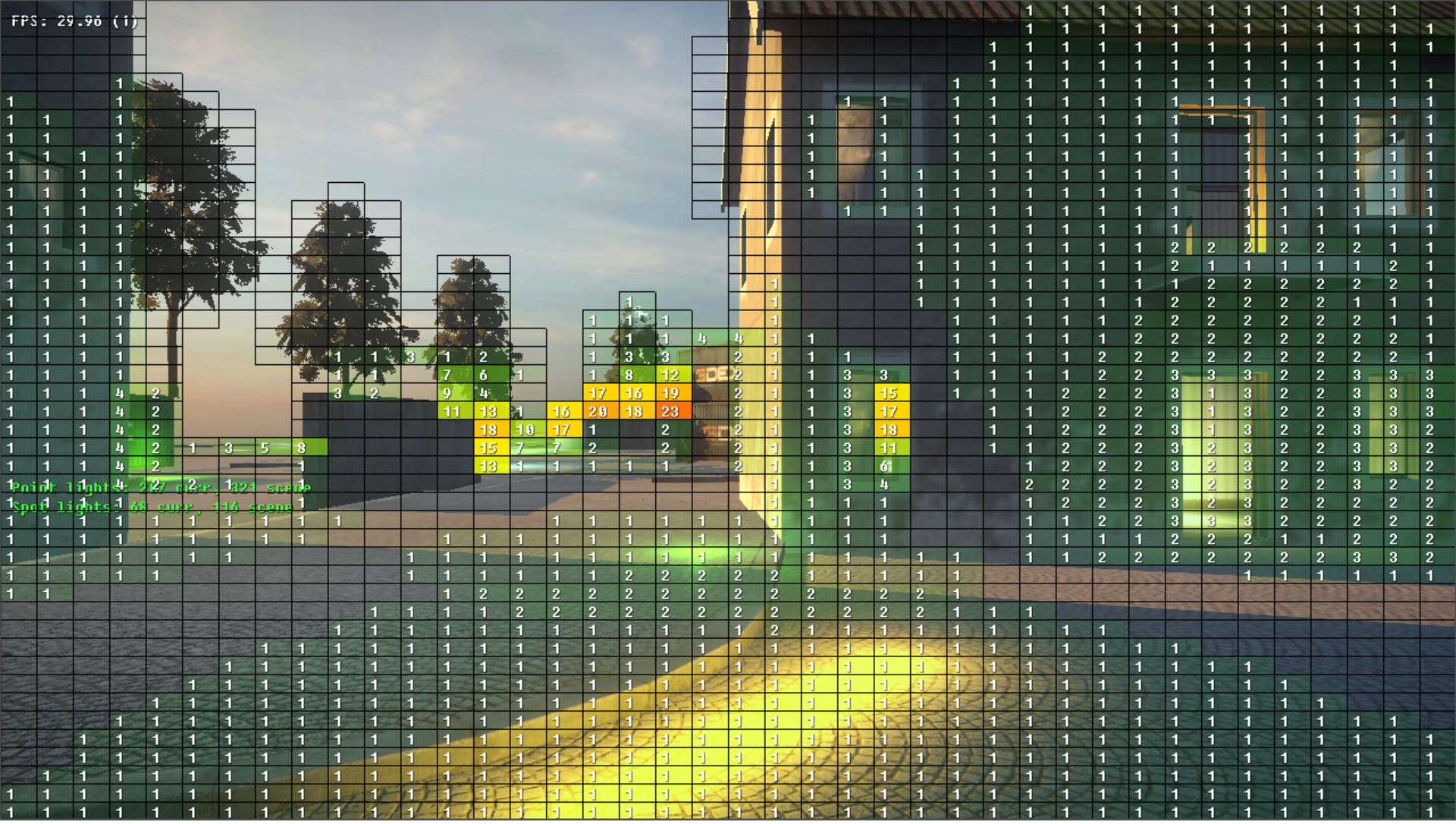


Tuesday, March 8, 2011

Sandbox test image using bc2 assets



Tuesday, March 8, 2011

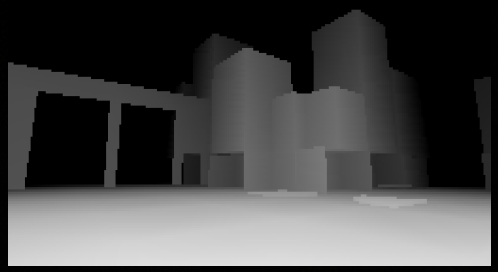


Tuesday, March 8, 2011

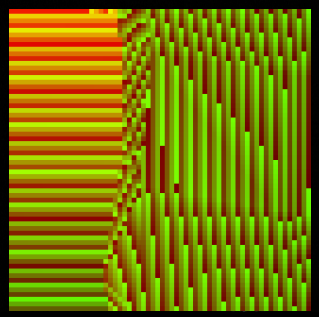
Further refinement of the culling pass for the 8 subtiles within a parent FB tile

Light Culling Optimizations - Hierarchy

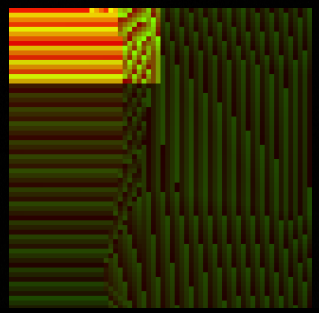
Camera Frustum



Light Volume
Coarse Z-Occlusion



FB Tile
SPU Z-Cull
64x64 Pixels



SubTile
SPU Z-Cull

Coarse to Fine Grained Culling

Tuesday, March 8, 2011

Camera frustum cull lights (PPU/SPU)

Software Occlusion Rasterizer occlude lights (SPU)

Conservative occlusion

LightTileJob Cull / Occlude Lights

64x64 pixel FB Tile region frusta + depth extents

DMA Traversal of all lights in camera frustum.

32x16 pixel SubTile region frusta + depth extents

Uses subset of lights from its parent FB Tile

Stress importance of testing your culling with lots of different camera angles when optimizing culling + low level lighting!

The upside of all of this culling work is that the steps are still seperable and if anything goes wrong, it goes wrong in such an obvious and spectacular fashion that its fairly easy to track down and fix ☺

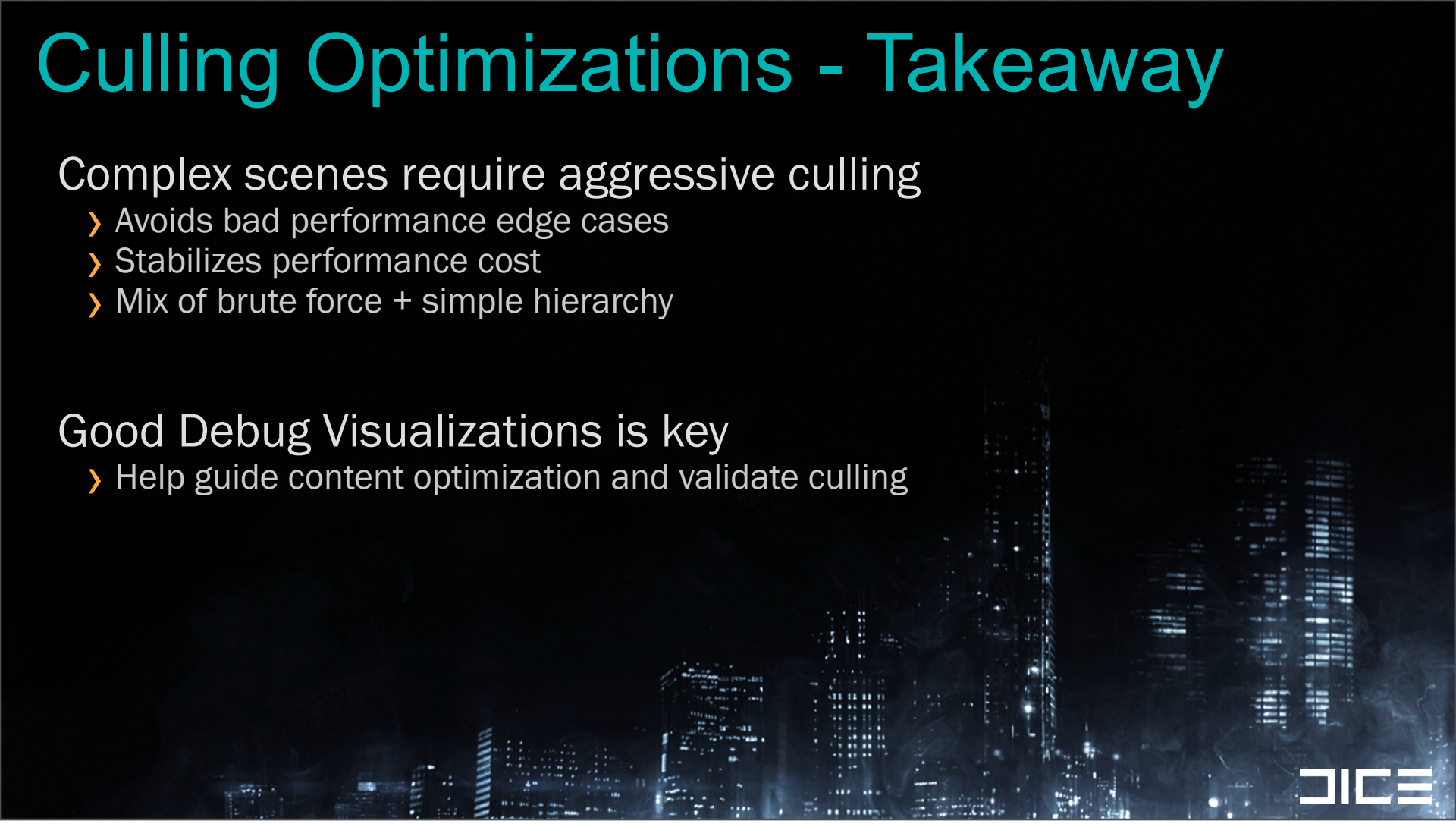
Culling Optimizations - Takeaway

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



Tuesday, March 8, 2011

Stress importance of testing your scenes to determine best culling

Lazy culling is fast for simple scenes (like a cornell box), but breaks down with more scene + depth + lighting complexity.

Know your use cases, content dependent

r culling with lots of different camera angles when optimizing culling + low level lighting!

Algorithmic optimization #0

Tuesday, March 8, 2011

In order to efficiently process all the pixels in a subtile, we need to know exactly what types of materials are present in a subtile.

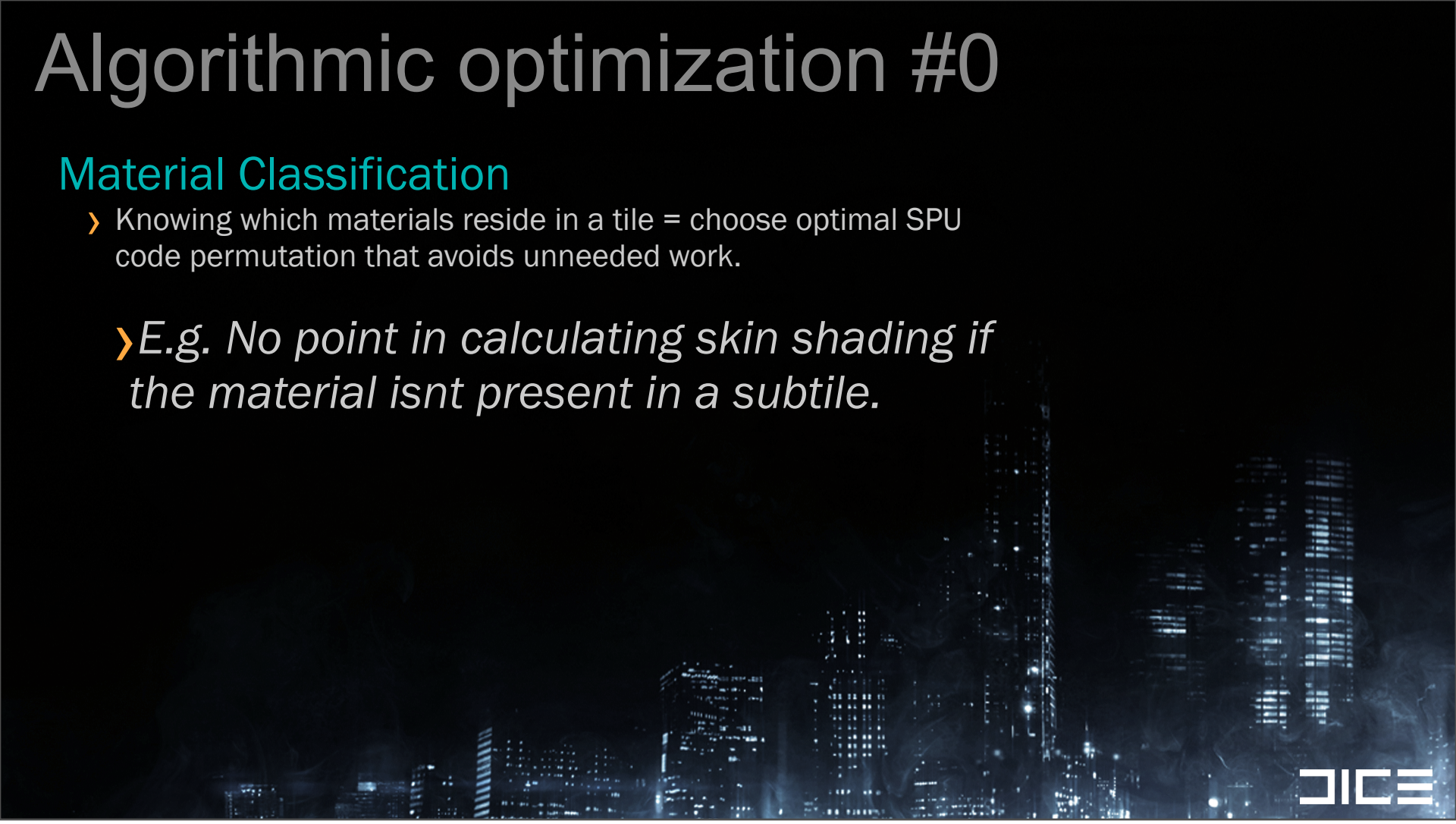
Motivation: A key goal is to execute SPU code that processes pixels efficiently,we don't want to execute code that does extra computation for things like skin shading if there is no skin material pixels in the tile.

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.*



In order to efficiently process all the pixels in a subtile, we need to know exactly what types of materials are present in a subtile.

Motivation: A key goal is to execute SPU code that processes pixels efficiently,we don't want to execute code that does extra computation for things like skin shading if there is no skin material pixels in the tile.

Algorithmic optimization #0

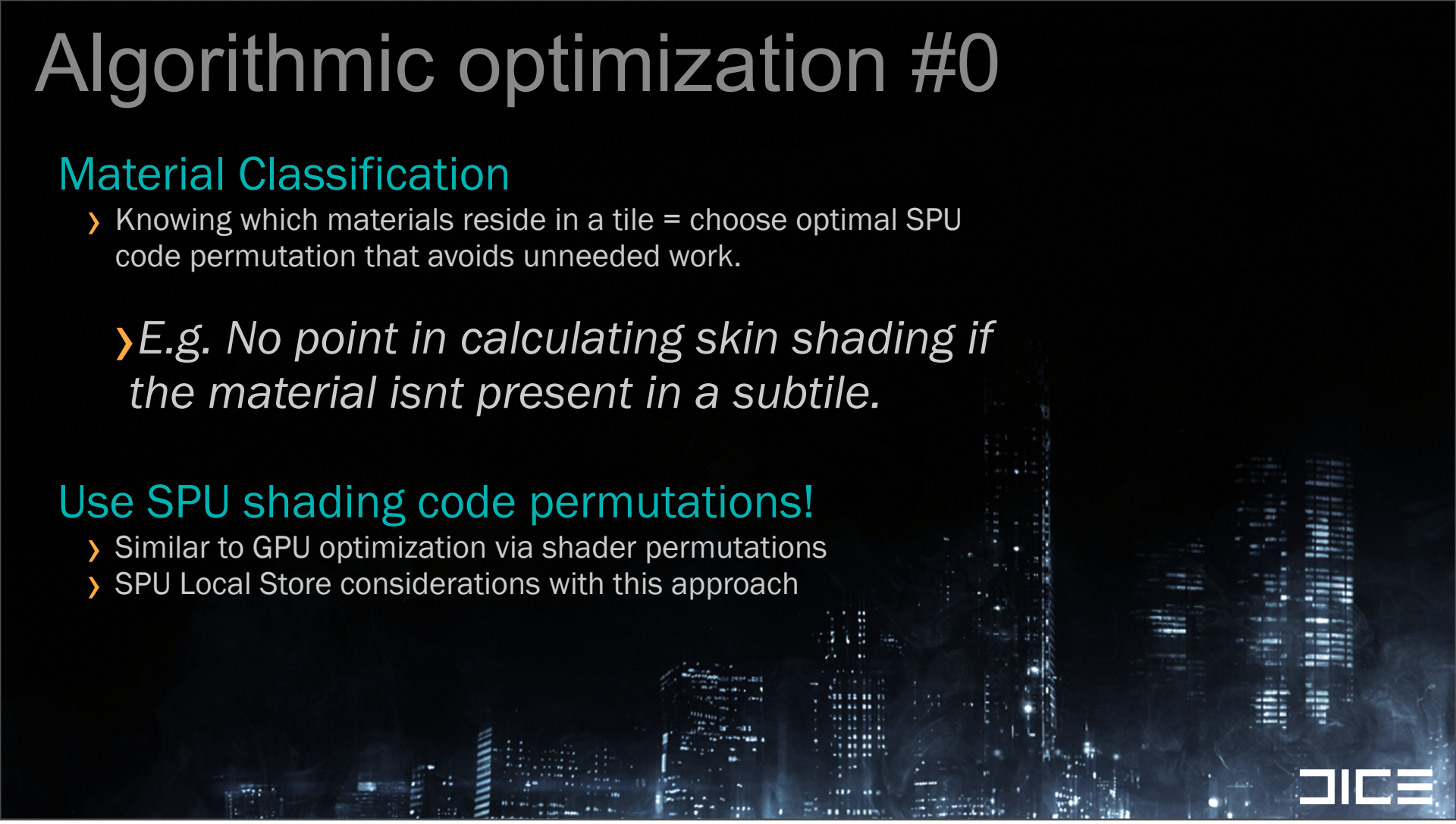
Material Classification

- 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



Tuesday, March 8, 2011

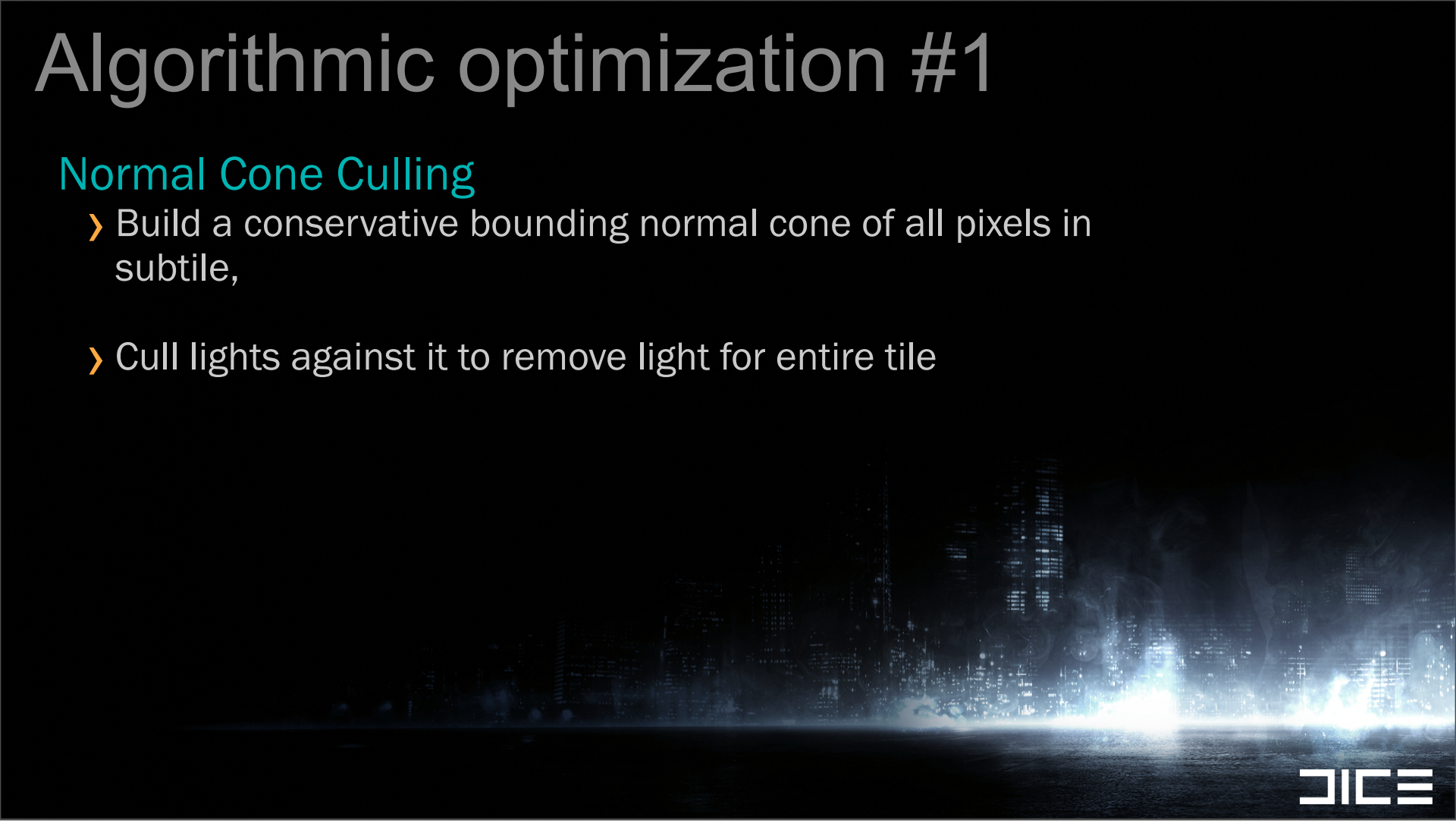
In order to efficiently process all the pixels in a subtile, we need to know exactly what types of materials are present in a subtile.

Motivation: A key goal is to execute SPU code that processes pixels efficiently,we don't want to execute code that does extra computation for things like skin shading if there is no skin material pixels in the tile.

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



Tuesday, March 8, 2011

As I mentioned earlier in the material classification step, one of the specific lighting cases we can optimize for is culling out the lighting influence of each lightsource intersecting the pixels for an entire tile if all those pixels point away from the lightsource.

To do this, we build a conservative bounding normal cone of all the pixels in a tile, then cull our lights against it.

(screenshot of normal culling on/off)

The efficiency of this technique depends on your lighting scenarios, its also more efficient with surfaces that aren't heavily normal mapped.

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

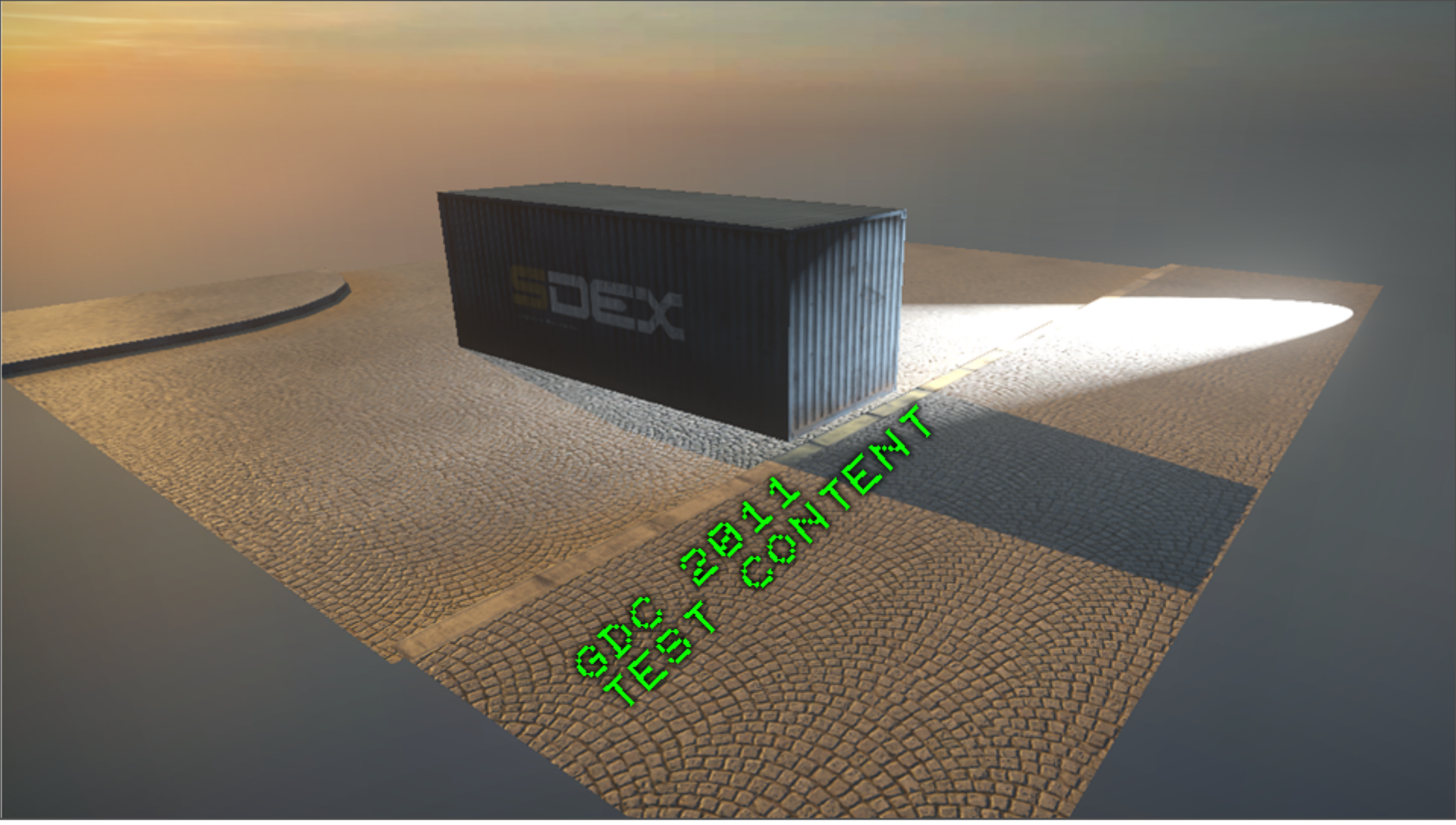
Tuesday, March 8, 2011

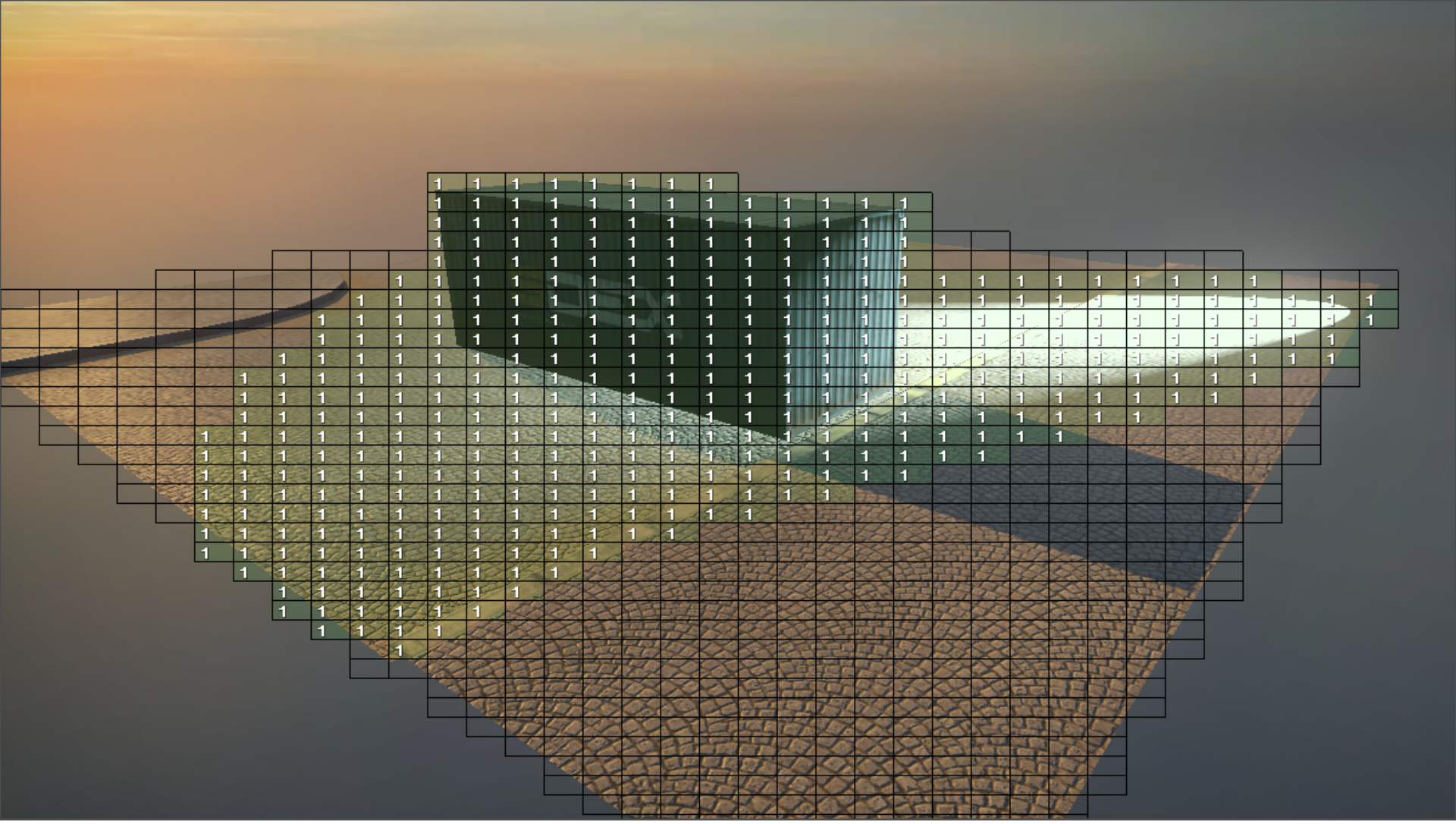
As I mentioned earlier in the material classification step, one of the specific lighting cases we can optimize for is culling out the lighting influence of each lightsource intersecting the pixels for an entire tile if all those pixels point away from the lightsource.

To do this, we build a conservative bounding normal cone of all the pixels in a tile, then cull our lights against it.

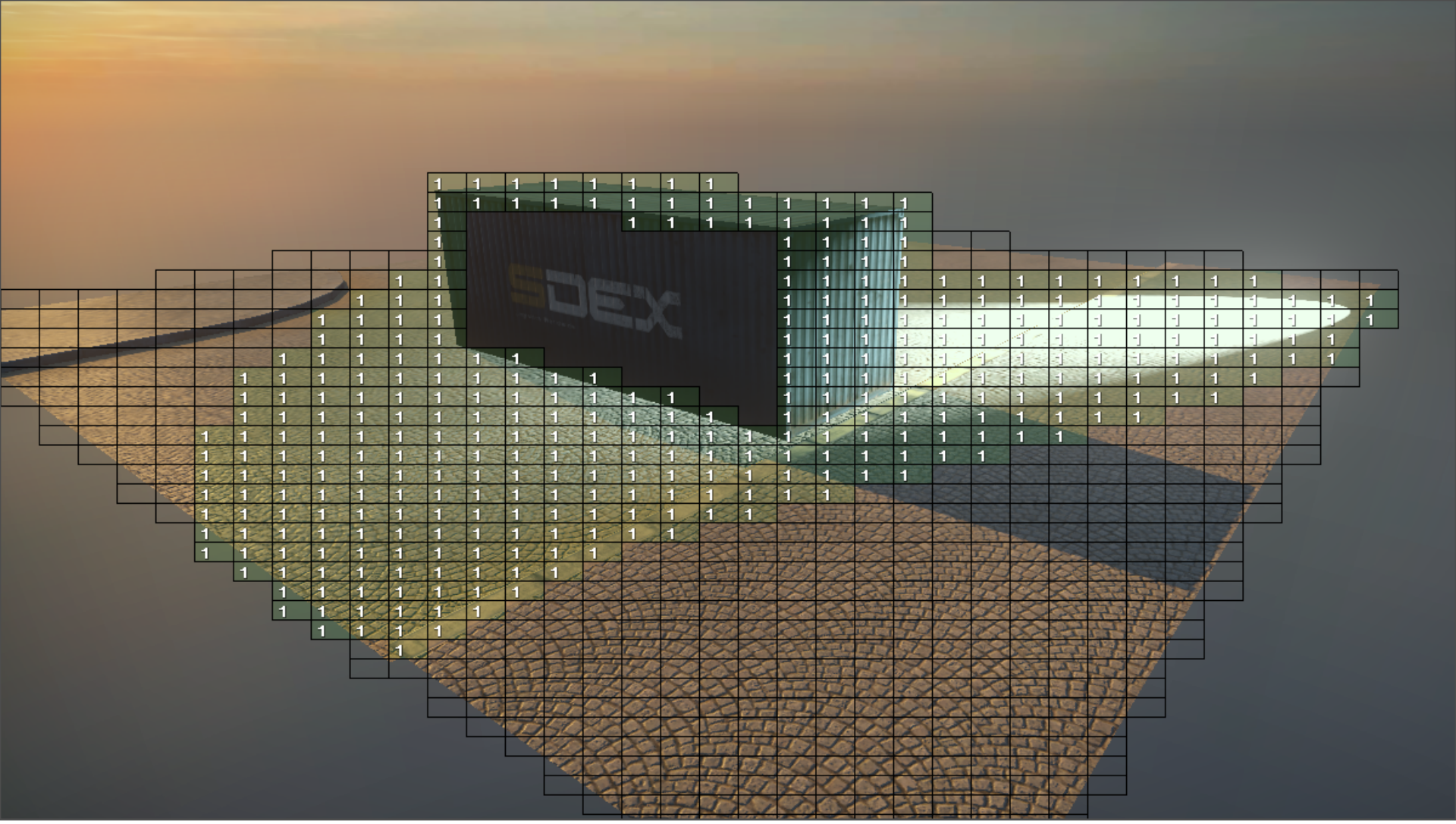
(screenshot of normal culling on/off)

The efficiency of this technique depends on your lighting scenarios, its also more efficient with surfaces that aren't heavily normal mapped.


DEX
GDC 2011
TEST CONTENT



Tuesday, March 8, 2011



Tuesday, March 8, 2011

Algorithmic optimization #2

Support diffuse only light sources

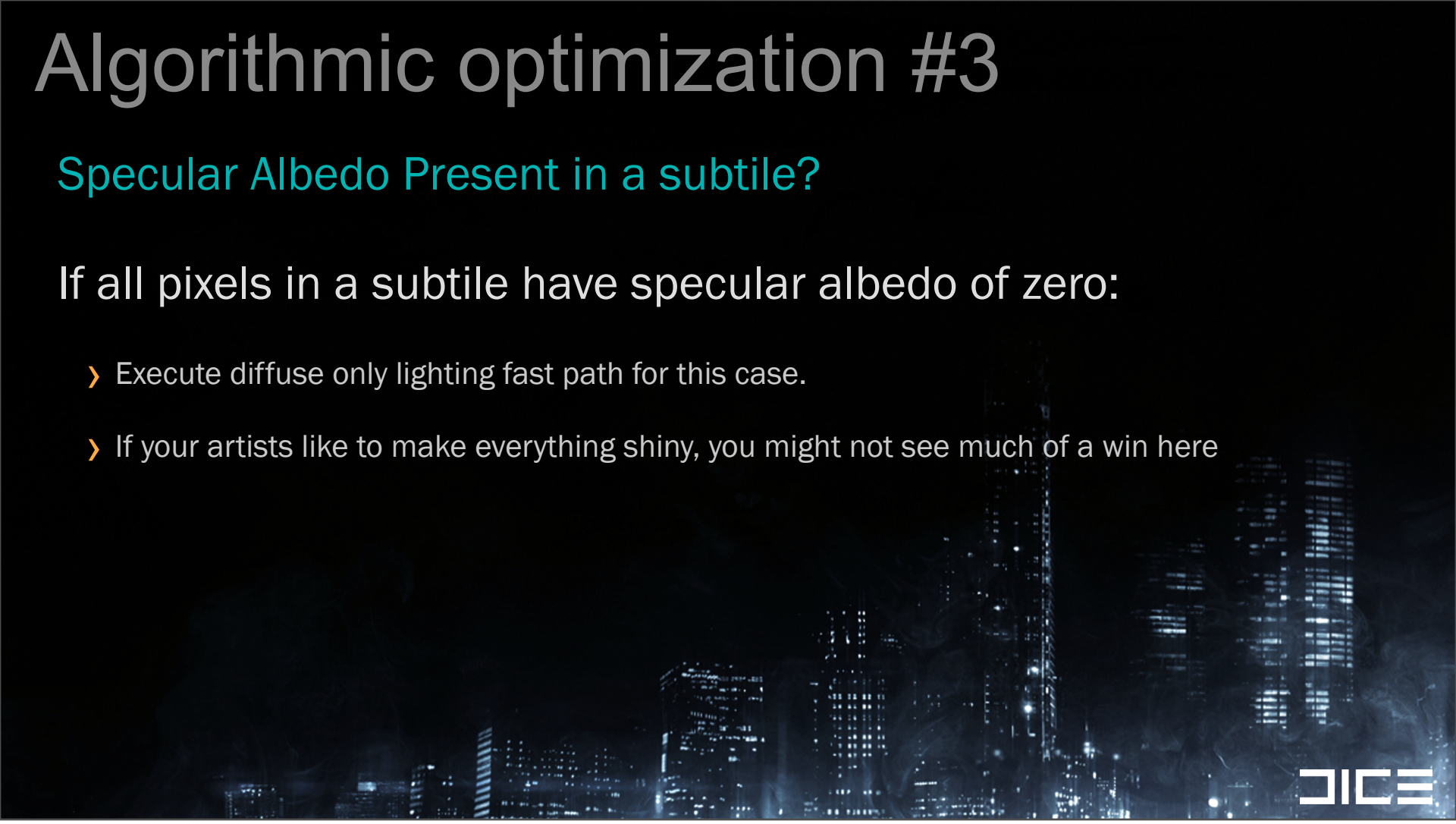
- Common practice in pure GPU rendered games
- Fill / Area lighting
- Only use specular contributing light sources where it counts.
- 2x+ faster
- Adds additional lighting loop + codesize considerations

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



Tuesday, March 8, 2011

If all pixels have zero specular albedo, then calculating specular lighting is pointless (diffuse only lighting fastpath).

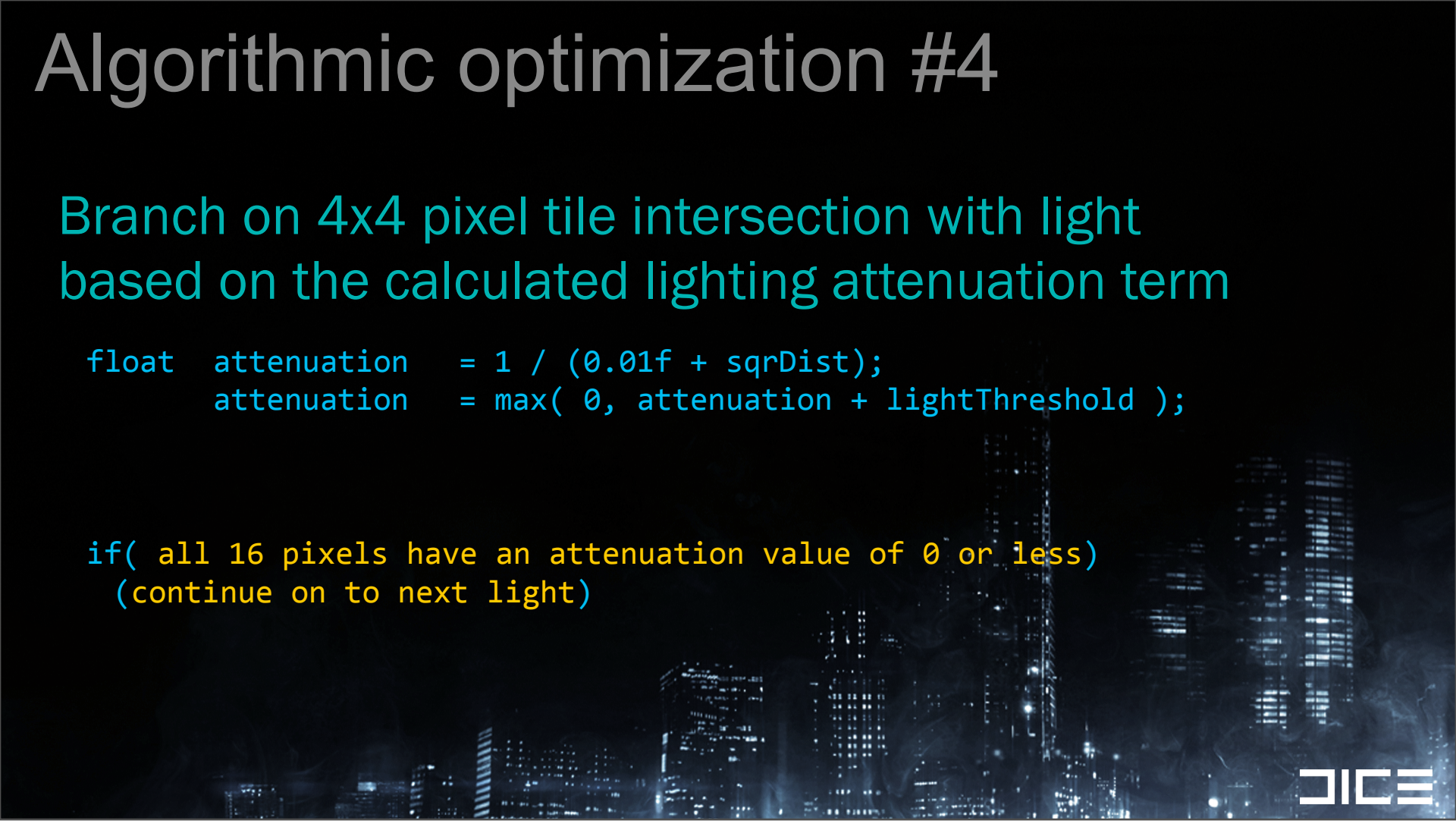
If you are making a FPS game with bald space marines where everything has shiny specular everywhere, you might not get much of a speedup with this optimization. For Battlefield 3, our environments tend to have dirty/dusty parts in their environment, so this optimization helps in some cases.

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)
```

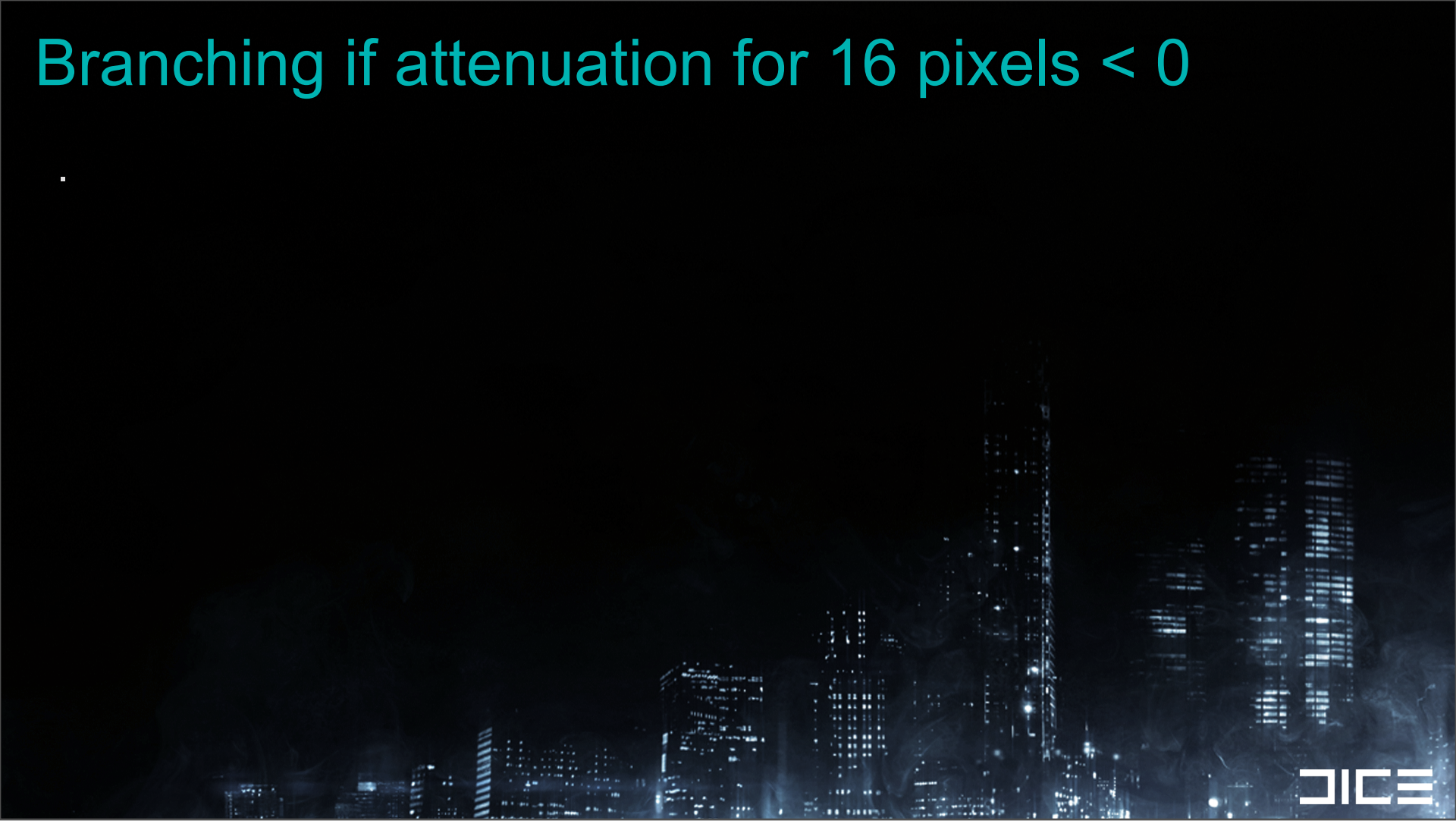


Key point here is to solve for the attenuation value from the light to all the pixels before you do any material specific work.

High depth extents + limited amount of light overlapping means huge wins with this optimization.

In general we got 20-30% speedup with this.
For super simple lighting with no materials you'll break even or spend at worst extra 1-2ms spu time per frame adding this test.

Branching if attenuation for 16 pixels < 0



Tuesday, March 8, 2011

A Key point here is to solve for the attenuation value from the light to all the pixels before you do any material specific work.

The floatingCompareGreater than instruction fcgt is something we already need for clamping attenuation so the orx and or instructions to collapse the 16 pixel selection masks is the core extra work needed for the branch test.

With large tile depth extents + limited amount of light overlapping in world/screenspace gets huge wins with this optimization.

In general we got 20-30% speedup with this.

For super simple lighting with no materials you'll break even or spend maybe an extra 1-2ms per frame adding this test.

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 );
```

A Key point here is to solve for the attenuation value from the light to all the pixels before you do any material specific work.

The floatingCompareGreater than instruction fcgt is something we already need for clamping attenuation so the orx and or instructions to collapse the 16 pixel selection masks is the core extra work needed for the branch test.

With large tile depth extents + limited amount of light overlapping in world/screenspace gets huge wins with this optimization.

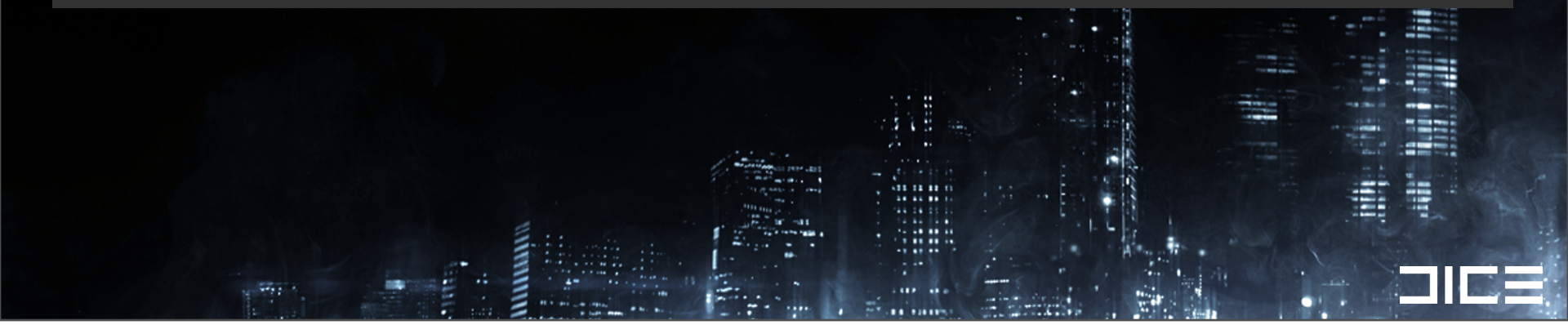
In general we got 20-30% speedup with this.

For super simple lighting with no materials you'll break even or spend maybe an extra 1-2ms per frame adding this test.

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 );
```



Tuesday, March 8, 2011

A Key point here is to solve for the attenuation value from the light to all the pixels before you do any material specific work.

The floatingCompareGreater than instruction fcgt is something we already need for clamping attenuation so the orx and or instructions to collapse the 16 pixel selection masks is the core extra work needed for the branch test.

With large tile depth extents + limited amount of light overlapping in world/screenspace gets huge wins with this optimization.

In general we got 20-30% speedup with this.

For super simple lighting with no materials you'll break even or spend maybe an extra 1-2ms per frame adding this test.

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!
```


A Key point here is to solve for the attenuation value from the light to all the pixels before you do any material specific work.

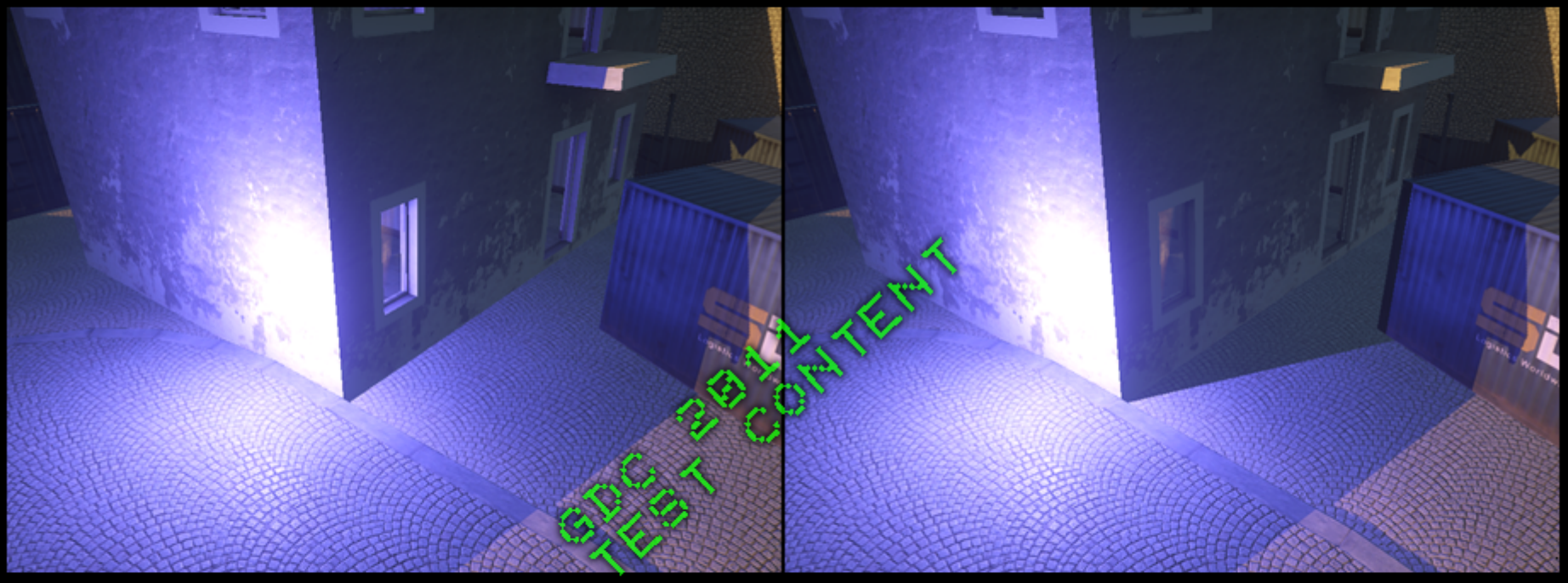
The floatingCompareGreater than instruction fcgt is something we already need for clamping attenuation so the orx and or instructions to collapse the 16 pixel selection masks is the core extra work needed for the branch test.

With large tile depth extents + limited amount of light overlapping in world/screenspace gets huge wins with this optimization.

In general we got 20-30% speedup with this.

For super simple lighting with no materials you'll break even or spend maybe an extra 1-2ms per frame adding this test.

Algorithmic optimization #5



No Light Clipping

Light Clipping Against House Wall

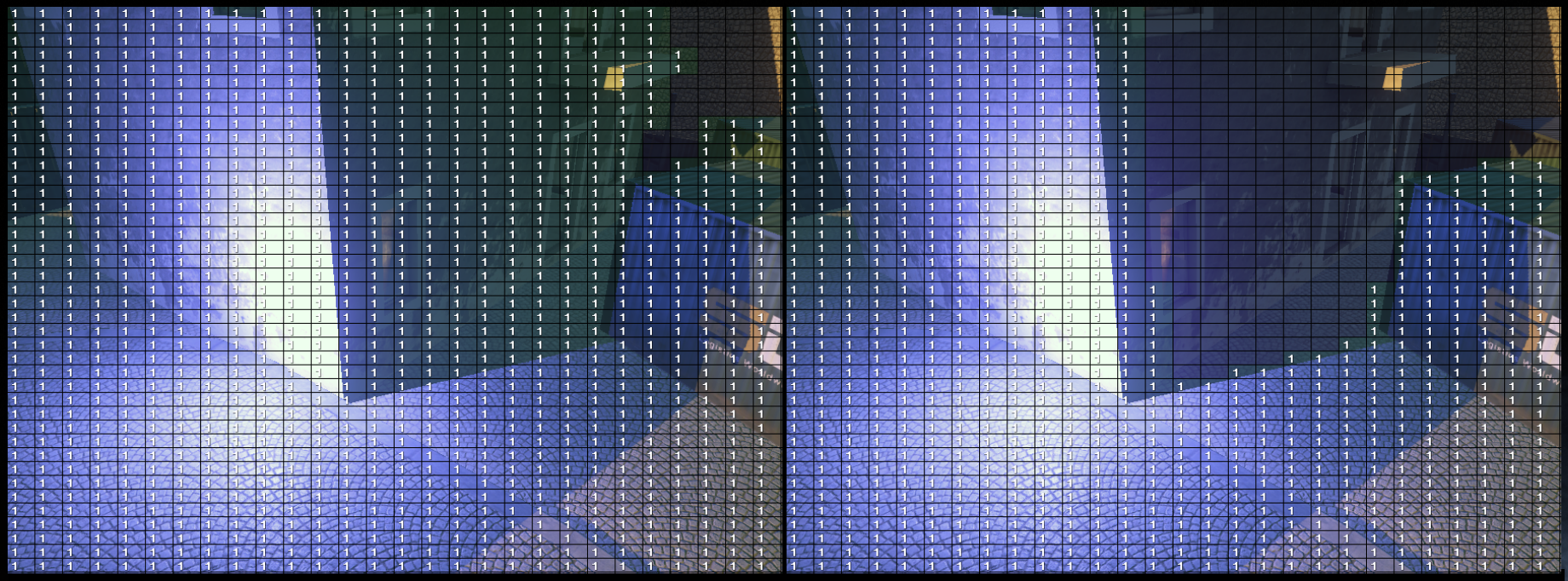
Tuesday, March 8, 2011

Goal: remove light entry from a tile that is in shadowed area marked by multiple clipping planes

This effect is something that is usually done with either stenciling or multiple clipping planes in a GPU shader.

On the SPUs, we can use our knowledge of the zbuffer and subtile depth extents to analytically clip at a higher level and remove lights from specific tiles to improve performance even further and get sharp shadows or masking areas from light to prevent unwanted light bleeding.

Algorithmic optimization #5



Goal: remove light entry from a tile that is in shadowed area marked by multiple clipping planes

This effect is something that is usually done with either stenciling or multiple clipping planes in a GPU shader.

On the SPUs, we can use our knowledge of the zbuffer and subtile depth extents to analytically clip at a higher level and remove lights from specific tiles to improve performance even further and get sharp shadows or masking areas from light to prevent unwanted light bleeding.

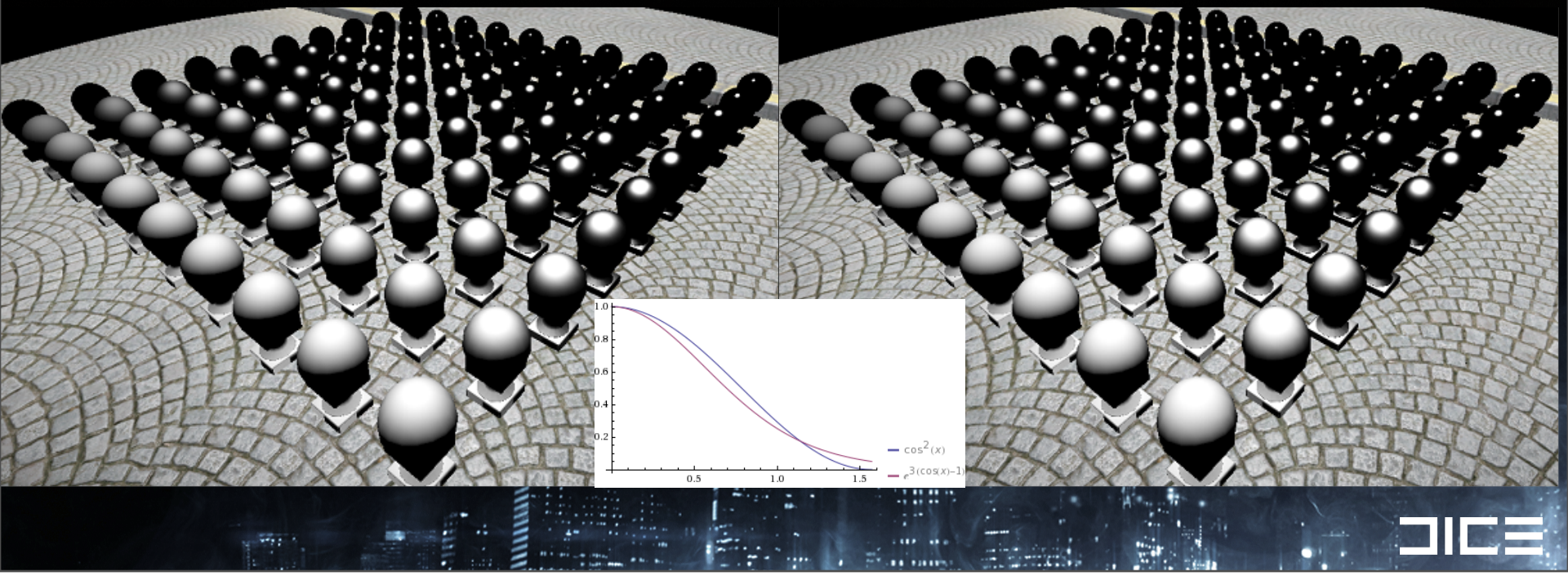
Why so much culling?

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



Tuesday, March 8, 2011

Doing specular shading faster without sacrificing precision, while maintaining the requirements of HDR lighting with an energy conserving physically based model. With the assistance from Matthew Jones of Criterion games.

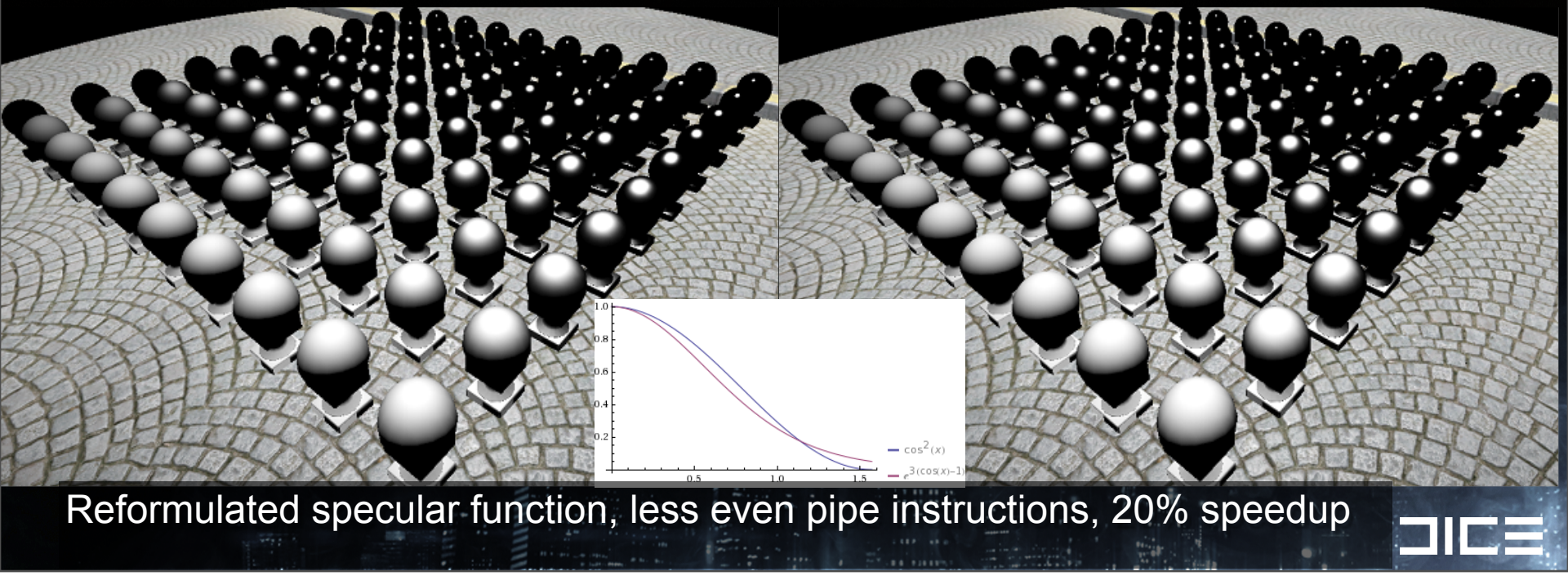
-Our new model has a minor divergence at extremely low specular power, (graphed above) which converges towards the same falloff as original specular model around a specular power of 10 with perceptually similar results.

Reformulated specular calculation, removes pow() which internally is a log() and exp(), and reduces the required math significantly, substituting with exp2().

20% speedup!

Algorithmic Optimization # 6

Spherical Gaussian Based Specular Model



Reformulated specular function, less even pipe instructions, 20% speedup

Tuesday, March 8, 2011

Doing specular shading faster without sacrificing precision, while maintaining the requirements of HDR lighting with an energy conserving physically based model. With the assistance from Matthew Jones of Criterion games.

-Our new model has a minor divergence at extremely low specular power, (graphed above) which converges towards the same falloff as original specular model around a specular power of 10 with perceptually similar results.

Reformulated specular calculation, removes pow() which internally is a log() and exp(), and reduces the required math significantly, substituting with exp2().

20% speedup!

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
```

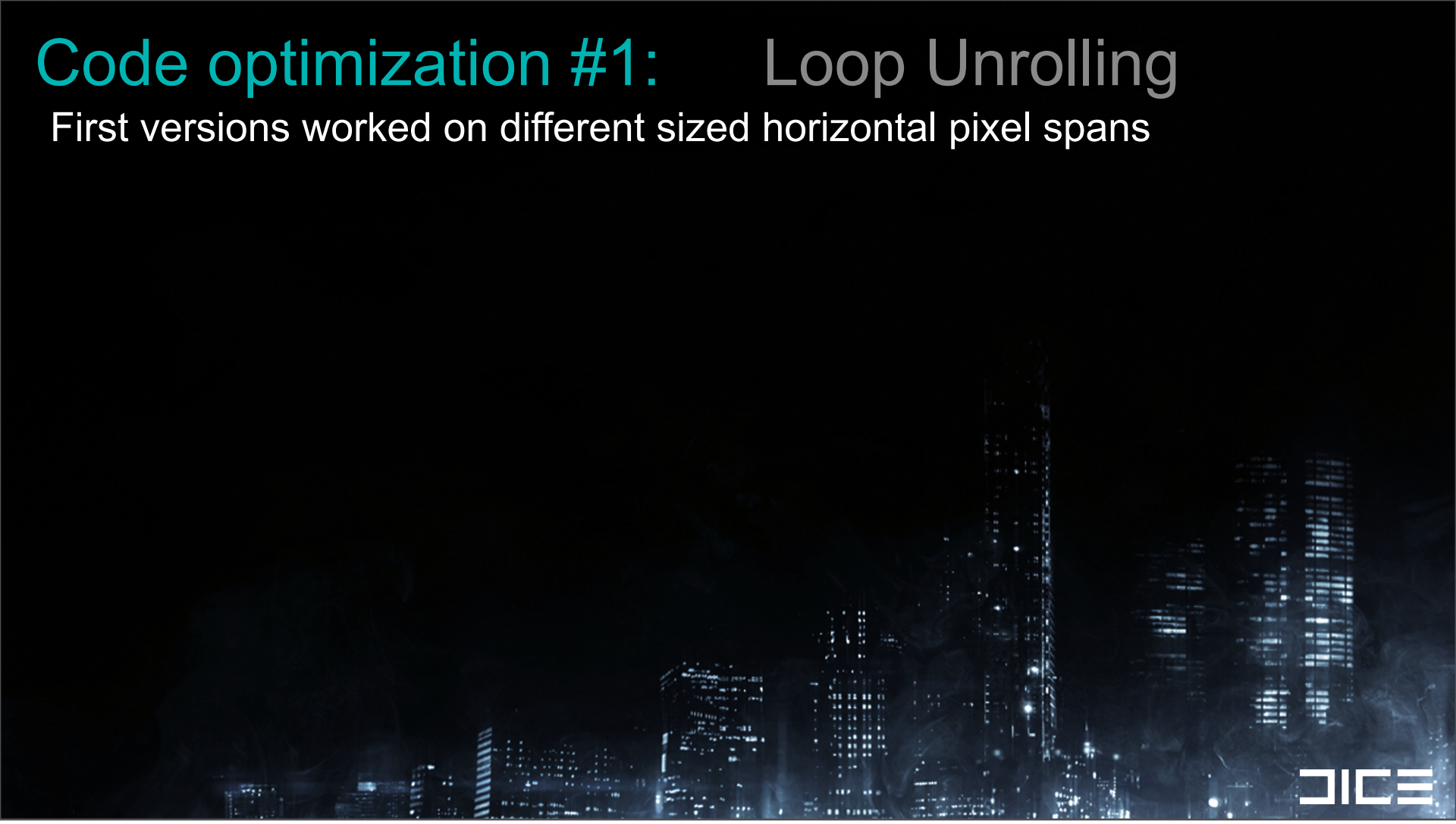
```
X0 X1 x2 x3
Y0 Y1 Y2 Y3
Z0 Z1 Z2 Z3
W0 W1 W2 W3
```


Its ok to SoA your data right before you need to use it, don't make the assumption that you need everything in SoA ahead of time, its not practical for storage reasons to do that. Measure the cost.

An earlier implementation of the spu shading code pretransposed the light data but left me short on LS space and didn't really make things faster

Code optimization #1: Loop Unrolling

First versions worked on different sized horizontal pixel spans



Tuesday, March 8, 2011

-Our first versions worked on different horizontal pixel spans, starting with 4x1 pixel span.

-The register pressure increases with more complex shaders, but the short lifetime of many intermediate calculations tends to avoid register spilling and pushing onto the stack

-There is significant stalling in code until we shaded 16 pixels at a time, we moved to 4x4 pixel quad for good spatial coherency + branching

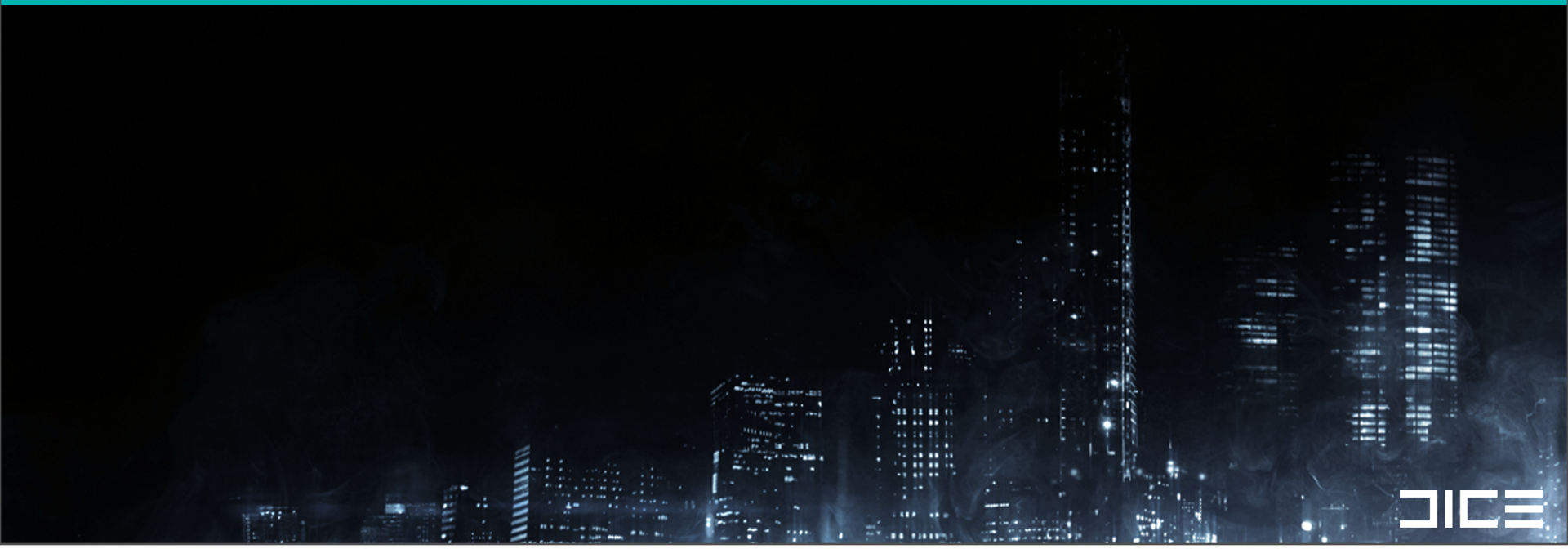
-Unrolling gives performance, but doing it by hand can quickly make the code more difficult to modify quickly.

-Takeaway: Save the unrolling until later after you get a basic 4x1 pixel span with SoA style code up and running with your desired feature set.

Code optimization #1: Loop Unrolling

First versions worked on different sized horizontal pixel spans

4x1 pixels = Minimum SoA implementation



Tuesday, March 8, 2011

-Our first versions worked on different horizontal pixel spans, starting with 4x1 pixel span.

-The register pressure increases with more complex shaders, but the short lifetime of many intermediate calculations tends to avoid register spilling and pushing onto the stack

-There is significant stalling in code until we shaded 16 pixels at a time, we moved to 4x4 pixel quad for good spatial coherency + branching

-Unrolling gives performance, but doing it by hand can quickly make the code more difficult to modify quickly.

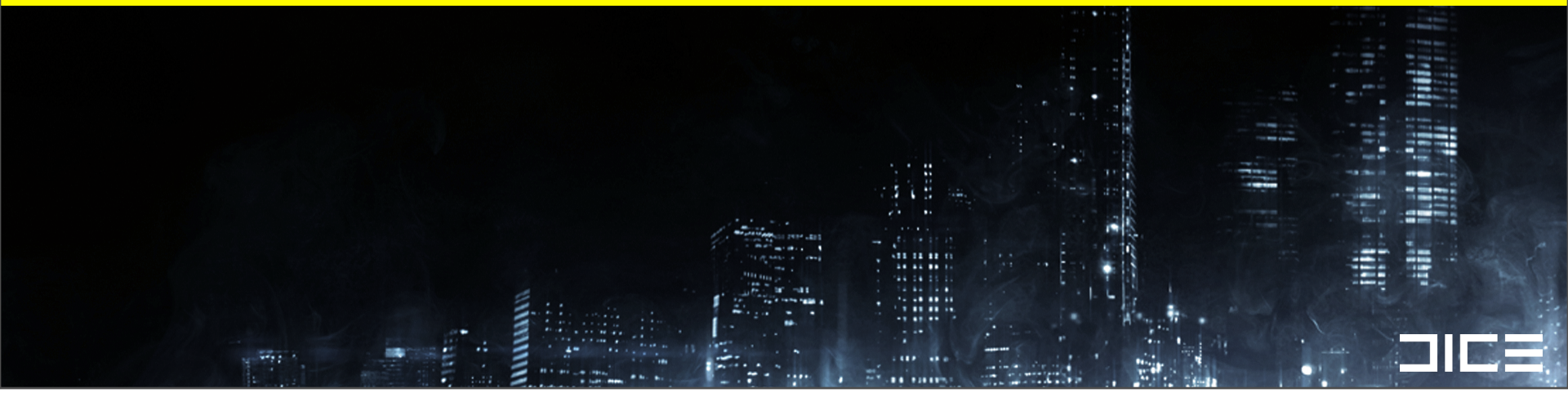
-Takeaway: Save the unrolling until later after you get a basic 4x1 pixel span with SoA style code up and running with your desired feature set.

Code optimization #1: Loop Unrolling

First versions worked on different sized horizontal pixel spans

4x1 pixels = Minimum SoA implementation

8x1 pixels = 2x unrolled



Tuesday, March 8, 2011

-Our first versions worked on different horizontal pixel spans, starting with 4x1 pixel span.

-The register pressure increases with more complex shaders, but the short lifetime of many intermediate calculations tends to avoid register spilling and pushing onto the stack

-There is significant stalling in code until we shaded 16 pixels at a time, we moved to 4x4 pixel quad for good spatial coherency + branching

-Unrolling gives performance, but doing it by hand can quickly make the code more difficult to modify quickly.

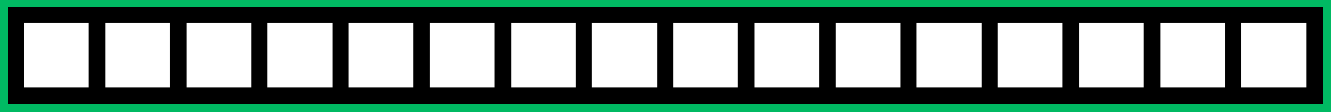
-Takeaway: Save the unrolling until later after you get a basic 4x1 pixel span with SoA style code up and running with your desired feature set.

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

Tuesday, March 8, 2011

-Our first versions worked on different horizontal pixel spans, starting with 4x1 pixel span.

-The register pressure increases with more complex shaders, but the short lifetime of many intermediate calculations tends to avoid register spilling and pushing onto the stack

-There is significant stalling in code until we shaded 16 pixels at a time, we moved to 4x4 pixel quad for good spatial coherency + branching

-Unrolling gives performance, but doing it by hand can quickly make the code more difficult to modify quickly.

-Takeaway: Save the unrolling until later after you get a basic 4x1 pixel span with SoA style code up and running with your desired feature set.

Code optimization #1: Loop Unrolling

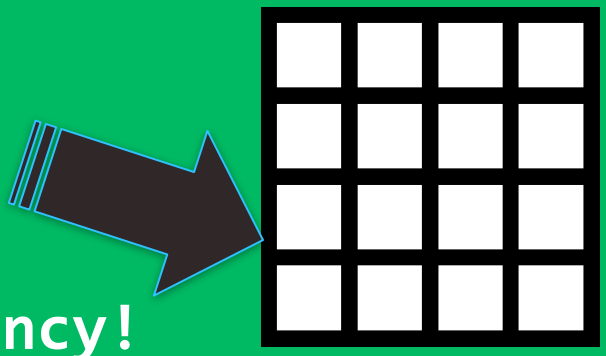
First versions worked on different sized horizontal pixel spans

4x1 pixels = Minimum SoA implementation

8x1 pixels = 2x unrolled

4x4 pixels = 4x unrolled

+ Improved Spatial Coherency!



Tuesday, March 8, 2011

-Our first versions worked on different horizontal pixel spans, starting with 4x1 pixel span.

-The register pressure increases with more complex shaders, but the short lifetime of many intermediate calculations tends to avoid register spilling and pushing onto the stack

-There is significant stalling in code until we shaded 16 pixels at a time, we moved to 4x4 pixel quad for good spatial coherency + branching

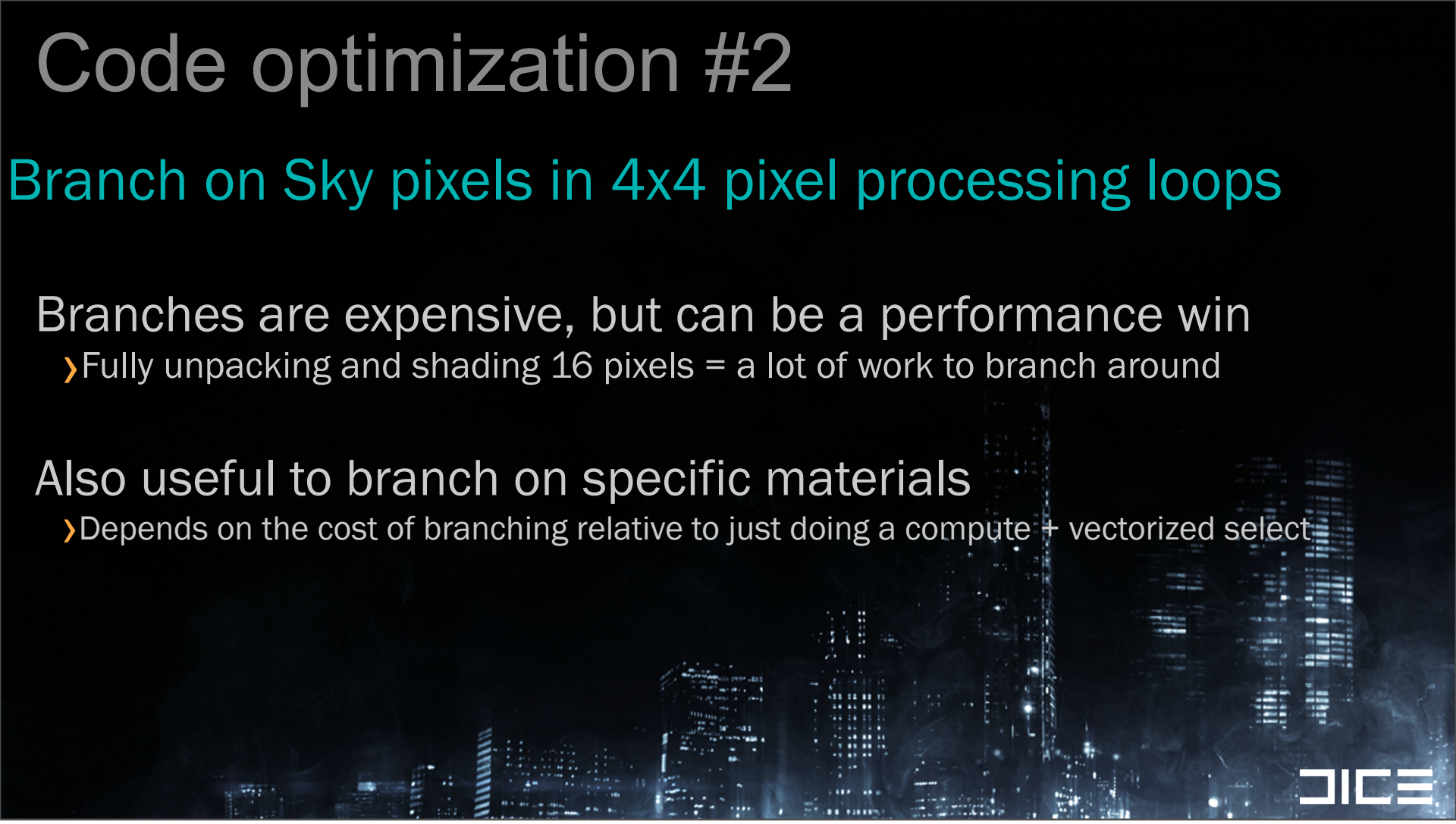
-Unrolling gives performance, but doing it by hand can quickly make the code more difficult to modify quickly.

-Takeaway: Save the unrolling until later after you get a basic 4x1 pixel span with SoA style code up and running with your desired feature set.

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



One important point to note is that the expense of unpacking the data and lighting is significant enough to bring us a performance win

to branch early and skip 4x4 pixel quads that are entirely sky pixels within a tile that contains some amount of sky pixels and normal scene depth pixels.

This case tends to be a common occurence with trees/foliage silhouettes set against the sky.

*(Image of trees against sky, mark 4x4 pixel quads in tiles where we branch as a special color to illustrate)

Code optimization #3

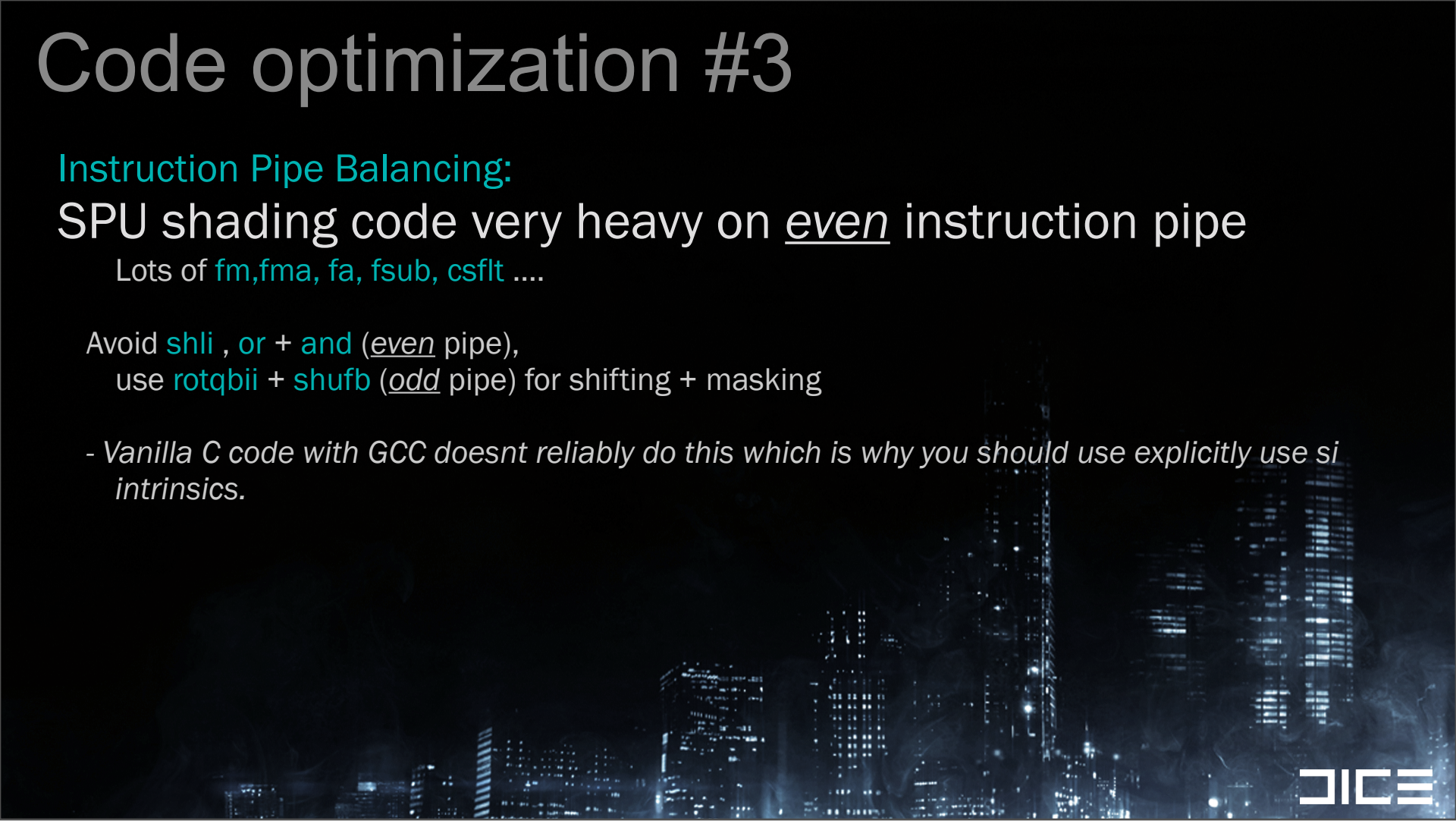
Instruction Pipe Balancing:

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.



Tuesday, March 8, 2011

Spu shading work is by default very heavy on the even instruction set, its important to
Look for opportunities where you can migrate work to off instructions.

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

One thing to keep in mind when doing this type of work on the SPUs, is that it can very heavy on the even instruction pipe due to a lot of arithmetic operations being performed.

There are opportunities to migrate some of this even pipe work to the odd instruction pipe by doing some of our unpacking of 8 bit values to floats via a lookup table.

Example - Migrating the calculation of specular power and renormalization scale to a lookup table.

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)			

Tuesday, March 8, 2011

Now if you look at our gbuffer format, you will notice that our specular smoothness term is stored in 8 bits, and the only variable term in all the computations above, everything else is constants.

So we can optimize this by migrating this work to a lookup table where we feed in an 8 bit value, and get 2 float values back out per pixel.

As an input, we take the raw quadword of the gbuffer surface that has our specular smoothness value for four pixels and perform 4 shuffles to and zero out all other bytes to do most of the work converting

the values to 4 array offsets.

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

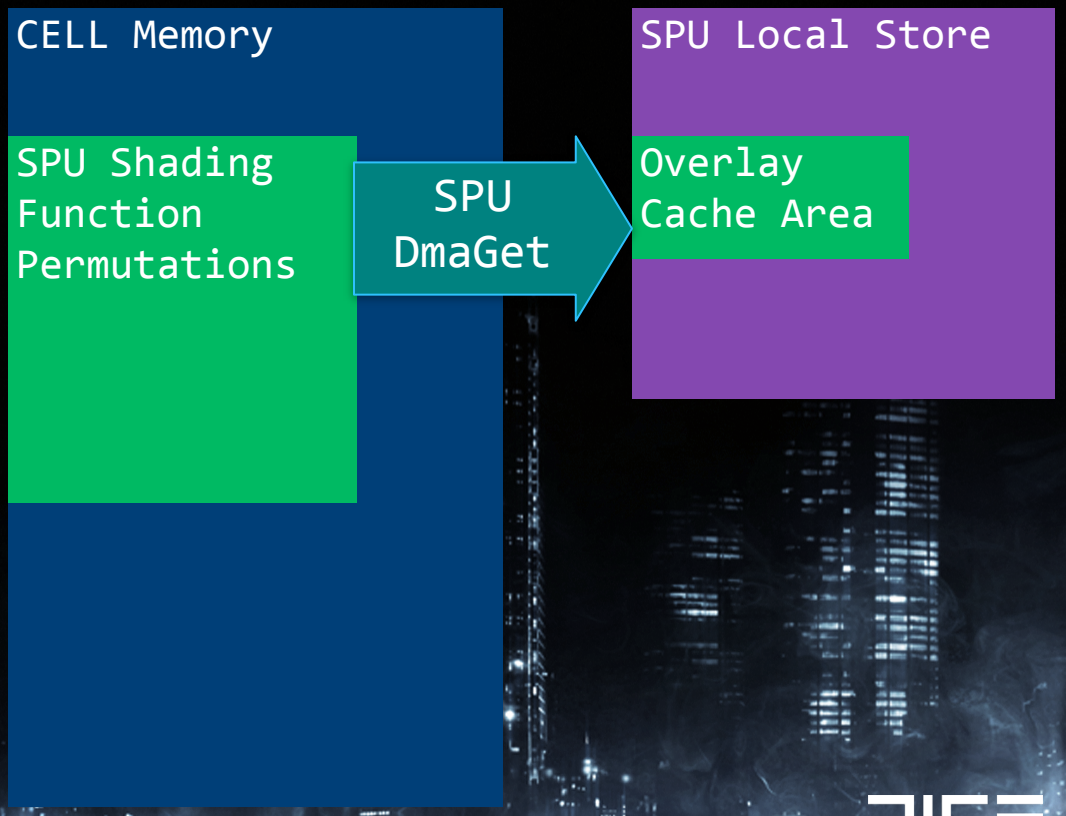
	X	Y	Z	W

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



Tuesday, March 8, 2011

SPU is 'self feeding' which means that it has the means to analyze the arbitrary tile that it is working on and determine which code needs to be run.

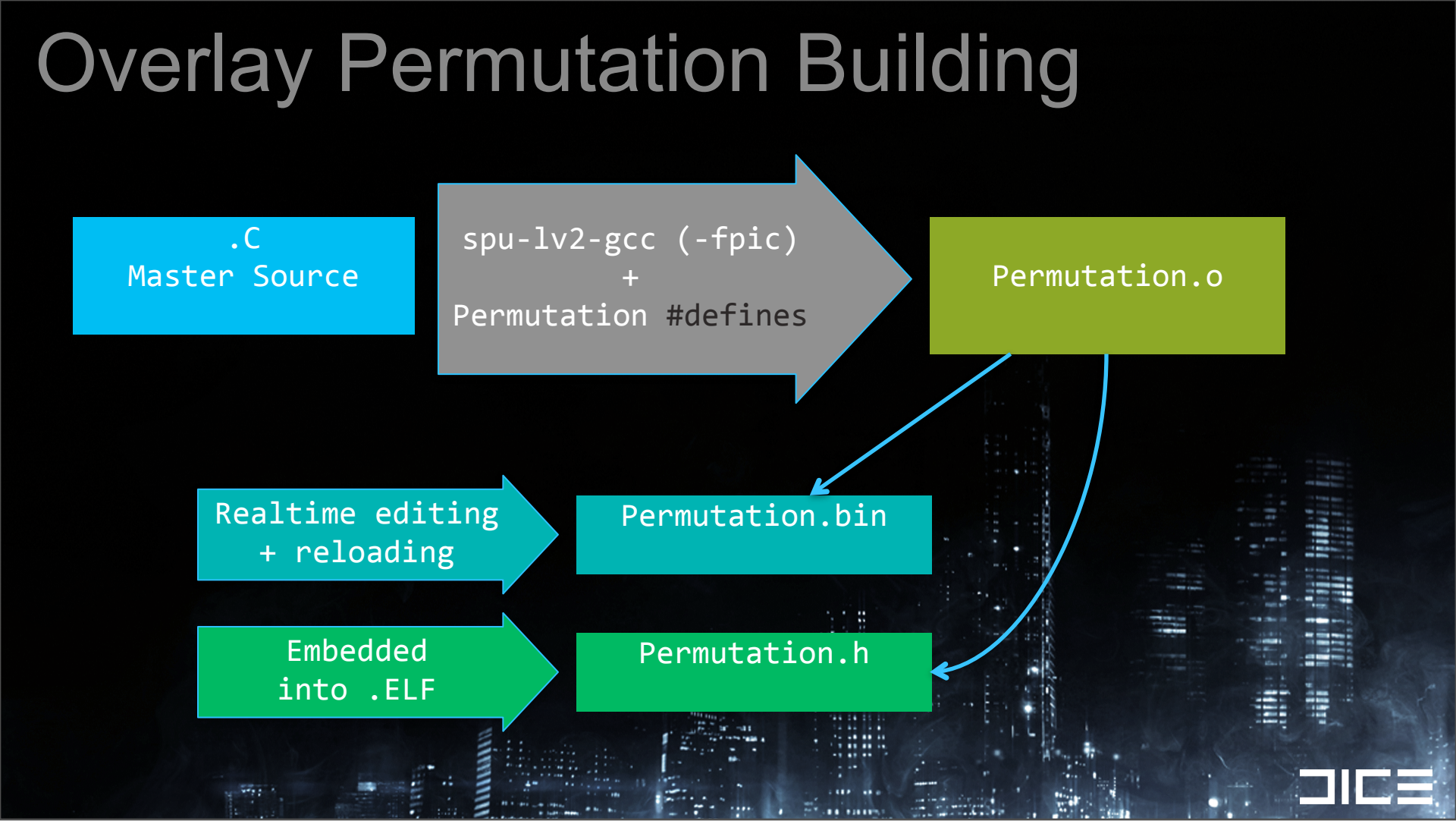
It does this by analyzing the gbuffer data and which lights intersect it to choose the correct and most optimized function permutation.

This makes the SPUs much cooler than GPUs in my opinion! ☺

-Stress that overlays are needed due to BF3 outgrowing the SPU LS as we added permutations

-Material and Lighting feature set

Overlay Permutation Building



Tuesday, March 8, 2011

The .C Master source file is written in a megashader style with a structure that allows us to enable or disable functionality via preprocessor defines to generate optimal code permutations for our needs.

To generate multiple permutations I don't synthesize multiple source files, I simply input the master source file and have several Preprocessor defines specified on the commandline.

Talk about my daily iteration workflow + the use of output .bin files for quick reloading of permutations

Includable .h version of permutations is for certification requirements, created by an awk script courtesy of Insomniac

BEST PRACTICES

Tuesday, March 8, 2011

Best practices : lessons learned in work efficiency and development of this tech for Battlefield 3

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

What permutations we use, and what code has branches versus compute+select code is highly dependent on the amount of ALU type work you need to do to support different materials.

Translucency is one of our more expensive effects, so its something we do a branch around, branches are expensive, but we are branching on 16 pixels at a time, which softens the cost of doing this.

Code Permutations Best Practices

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)

Good way to check if you have bugs, switching to your single ubershader should of course not create any visual changes

Managing code size is important, code unrolling

’SPA’ (SPU ASSEMBLER)

Tuesday, March 8, 2011

The little * on the ’Improving Performance’ means that your gains versus hand unrolled GCC may get you enough of a performance boost that SPA doesn’t give you huge wins, but it depends on your functions and their complexity.

'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

Tuesday, March 8, 2011

The little * on the 'Improving Performance' means that your gains versus hand unrolled GCC may get you enough of a performance boost that SPA doesn't give you huge wins, but it depends on your functions and their complexity.

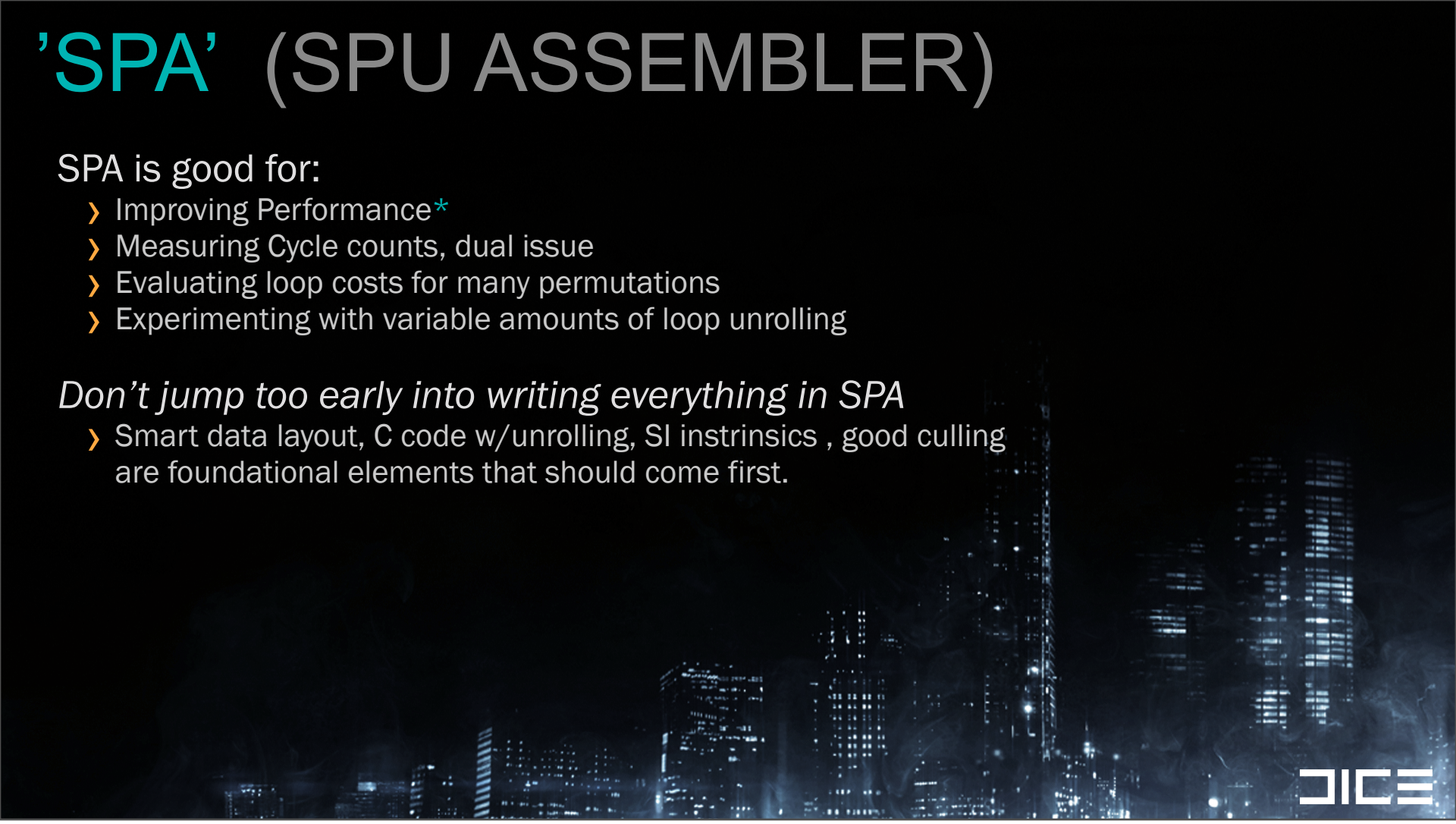
'SPA' (SPU ASSEMBLER)

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.



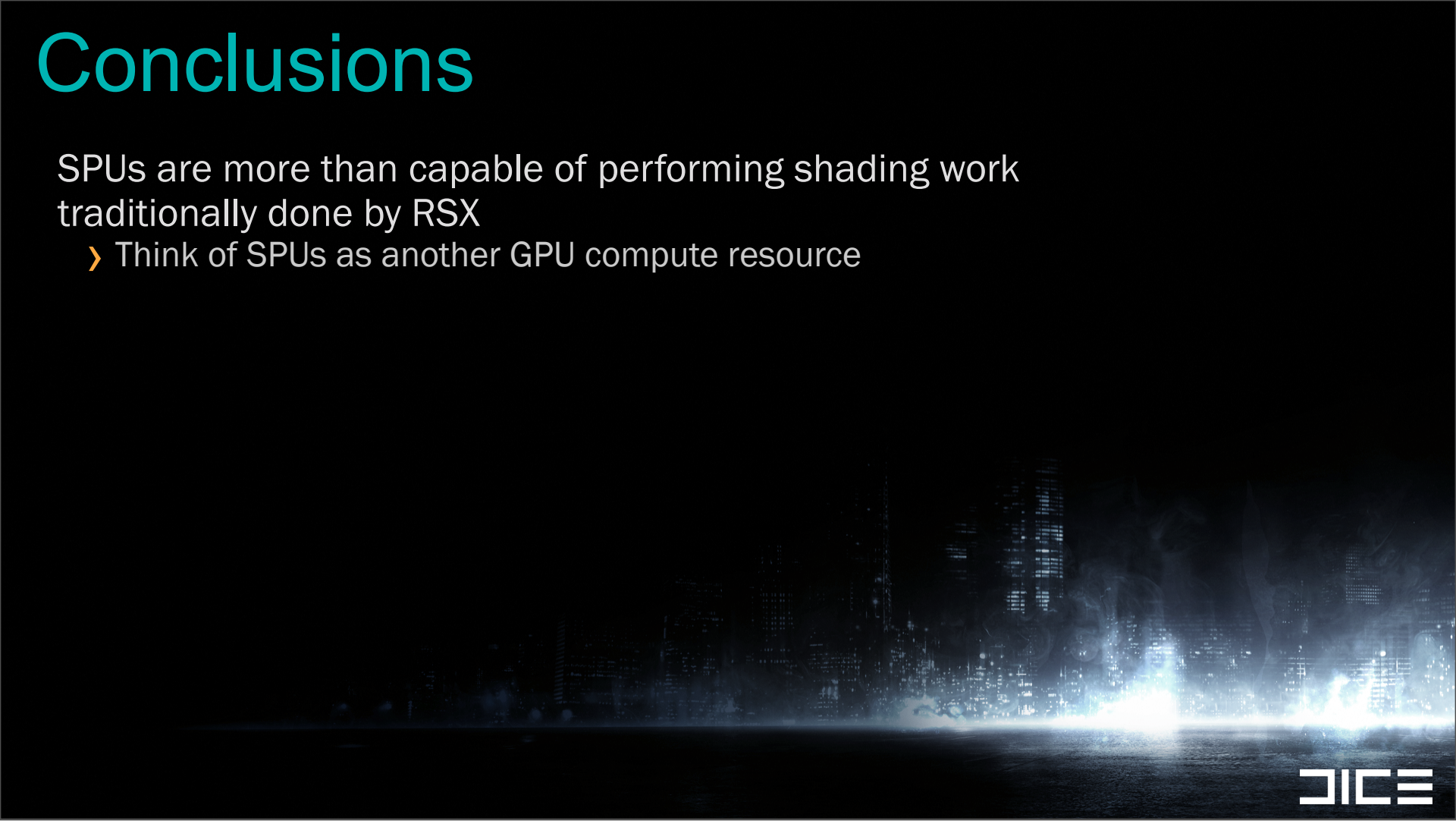
Tuesday, March 8, 2011

The little * on the 'Improving Performance' means that your gains versus hand unrolled GCC may get you enough of a performance boost that SPA doesn't give you huge wins, but it depends on your functions and their complexity.

Conclusions

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

- Think of SPUs as another GPU compute resource



Tuesday, March 8, 2011

*(I am intentionally mixing GPU and RSX terms to clarify points)

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

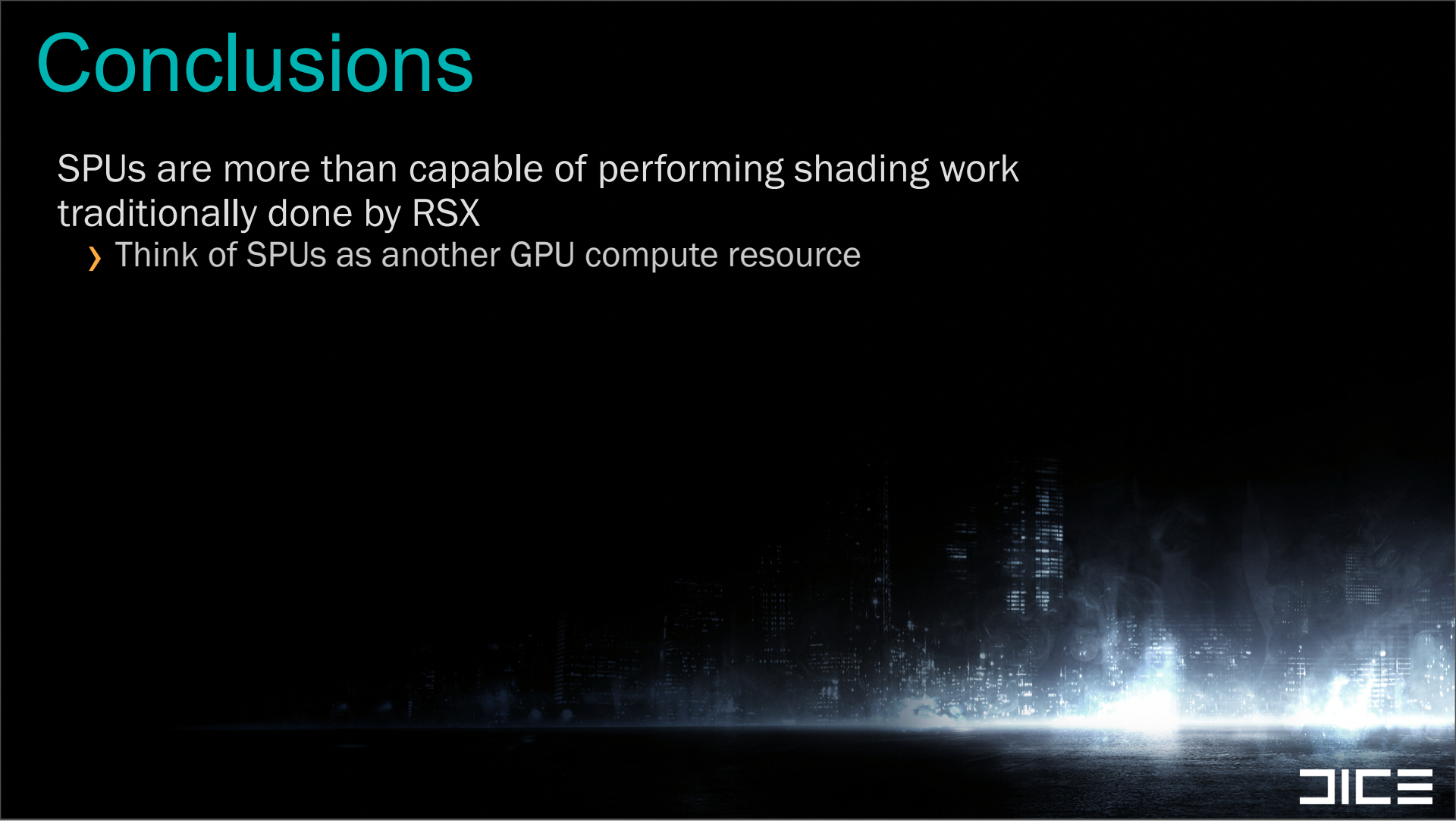
(Z-cull / S-Cull / DepthBounds)

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

Conclusions

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

- Think of SPUs as another GPU compute resource



Tuesday, March 8, 2011

*(I am intentionally mixing GPU and RSX terms to clarify points)

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

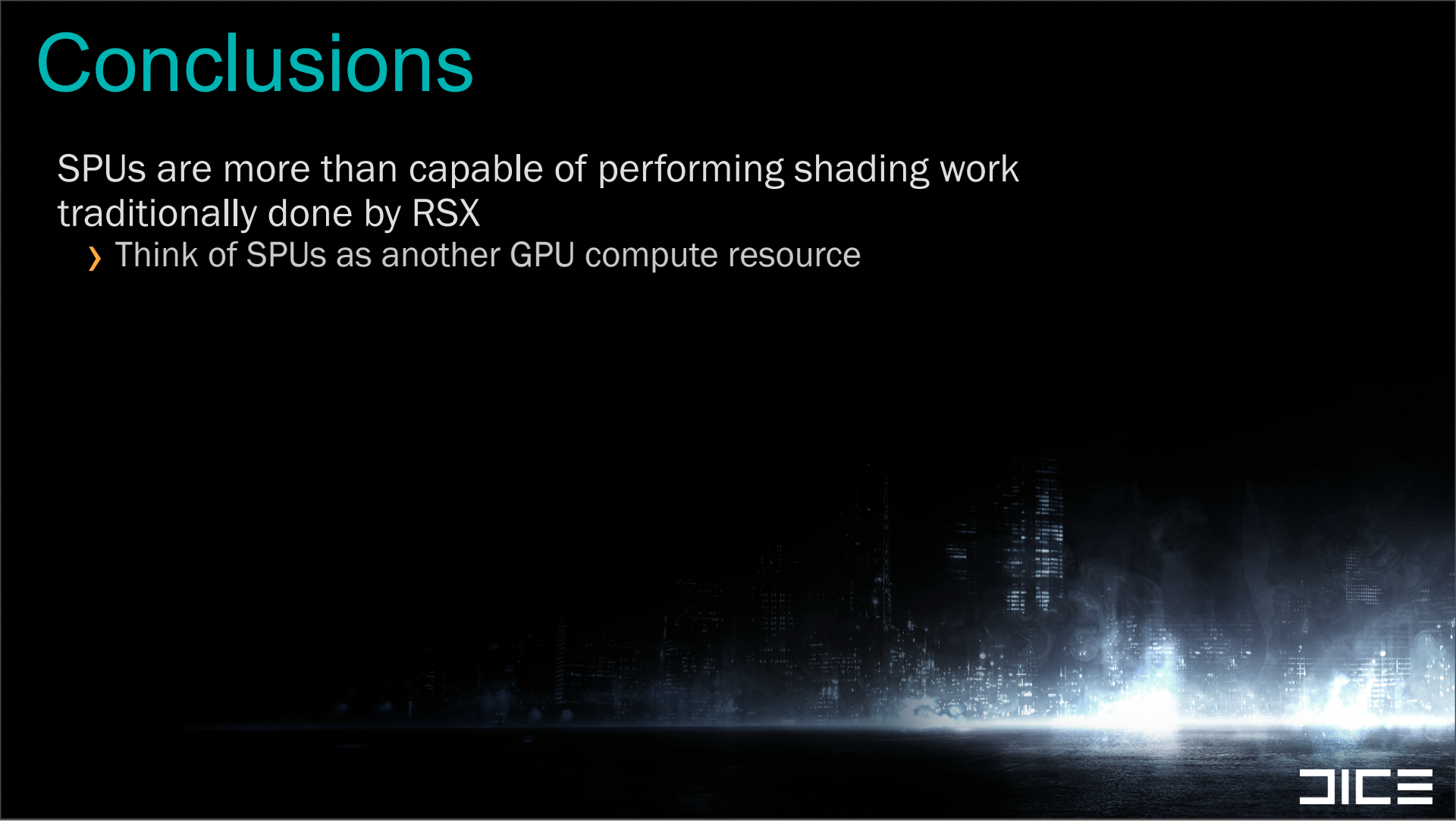
(Z-cull / S-Cull / DepthBounds)

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

Conclusions

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

- Think of SPUs as another GPU compute resource



Tuesday, March 8, 2011

*(I am intentionally mixing GPU and RSX terms to clarify points)

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

(Z-cull / S-Cull / DepthBounds)

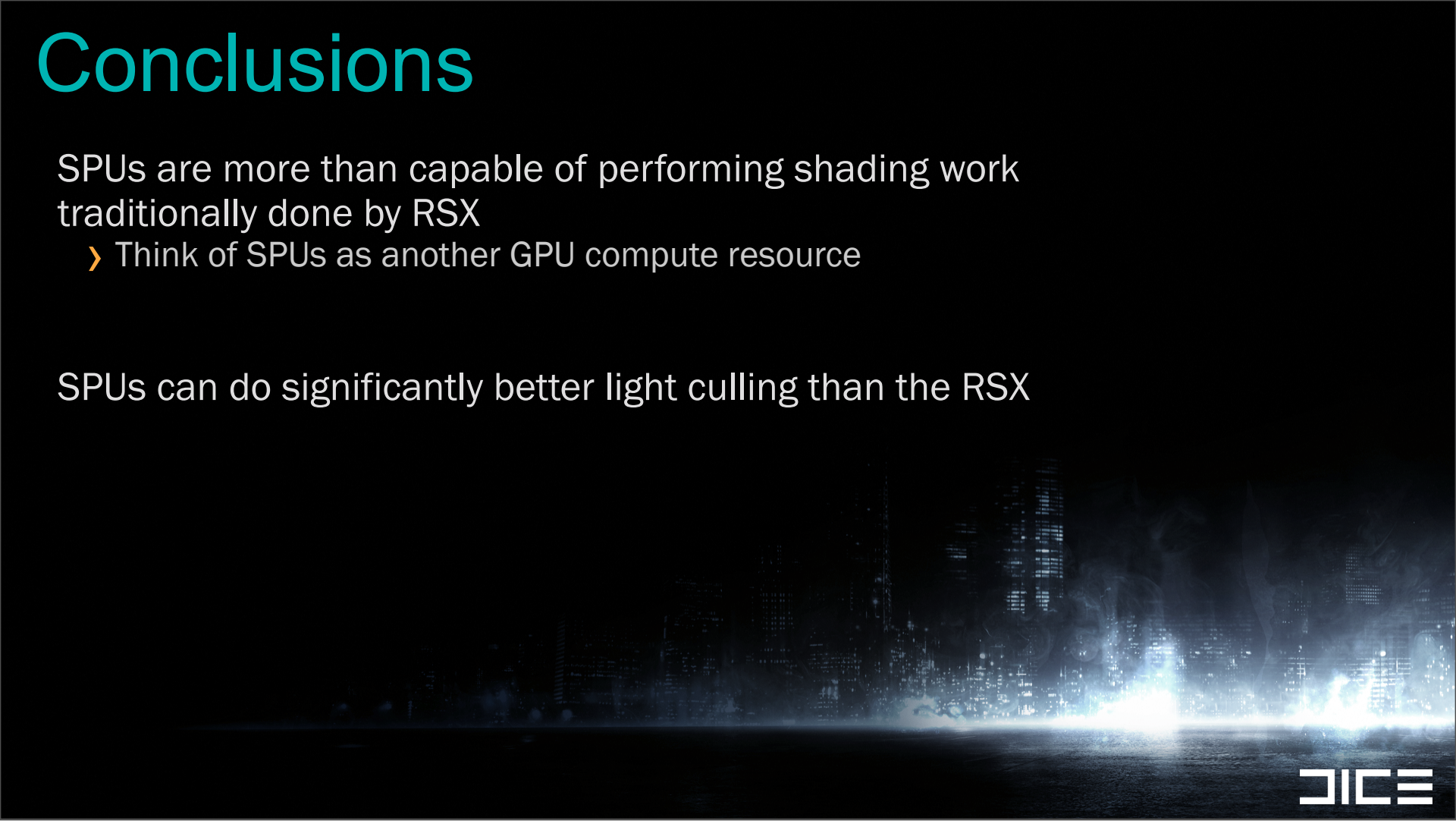
RSX+SPU combined shading creates a great opportunity to raise the bar on graphics!

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



Tuesday, March 8, 2011

*(I am intentionally mixing GPU and RSX terms to clarify points)

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

(Z-cull / S-Cull / DepthBounds)

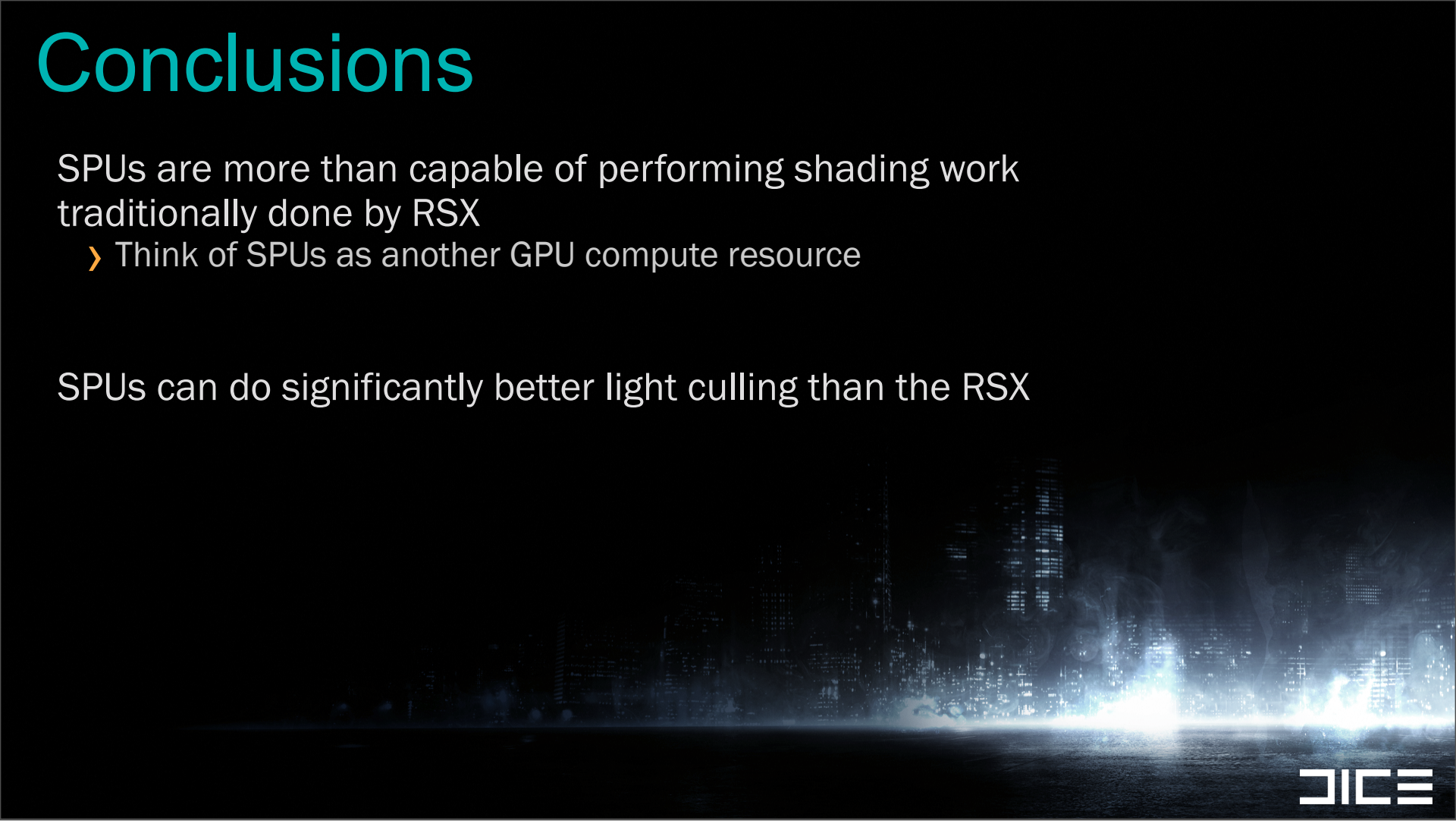
RSX+SPU combined shading creates a great opportunity to raise the bar on graphics!

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



Tuesday, March 8, 2011

*(I am intentionally mixing GPU and RSX terms to clarify points)

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

(Z-cull / S-Cull / DepthBounds)

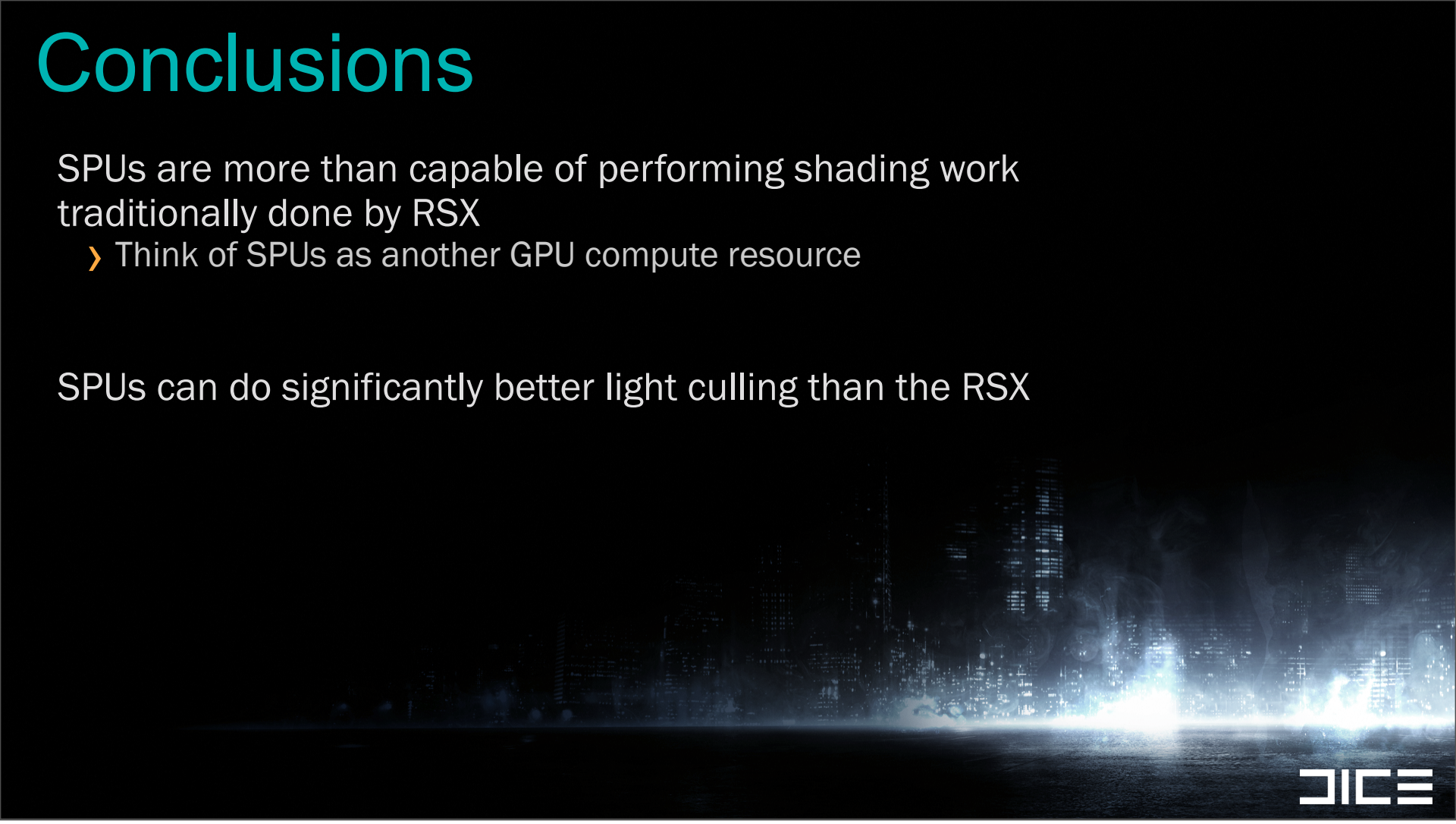
RSX+SPU combined shading creates a great opportunity to raise the bar on graphics!

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



Tuesday, March 8, 2011

*(I am intentionally mixing GPU and RSX terms to clarify points)

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

(Z-cull / S-Cull / DepthBounds)

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

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!

*(I am intentionally mixing GPU and RSX terms to clarify points)

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

Think of SPUs as another GPU compute resource

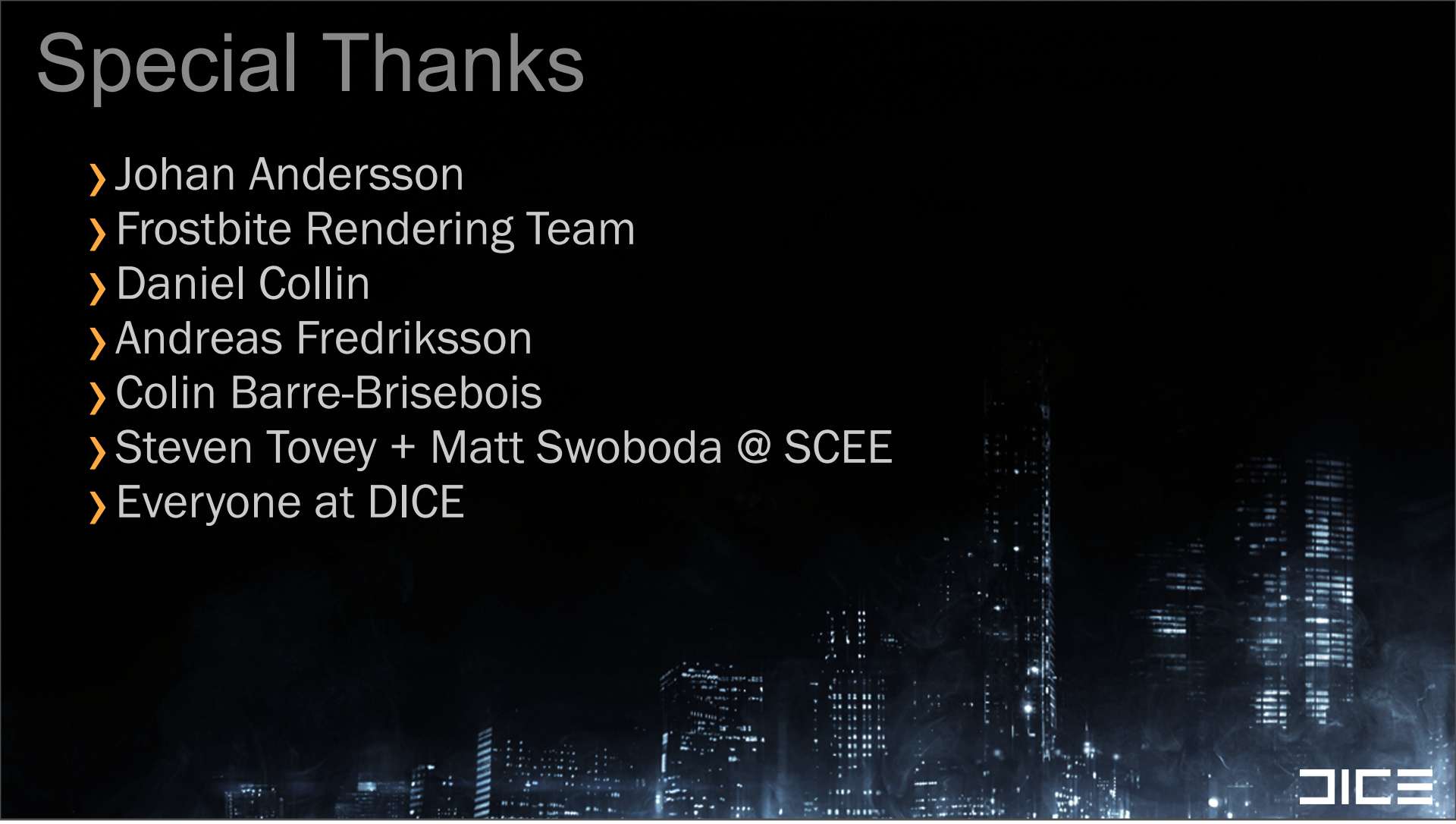
SPUs can do significantly better light culling than the RSX

(Z-cull / S-Cull / DepthBounds)

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

GDC

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

DICE

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