



# ***A 2.5D CULLING FOR FORWARD+***

**AMD**  
**Takahiro Harada**



# AGENDA



- Forward+
  - Forward, Deferred, Forward+
  - Problem description
- 2.5D culling
- Results



# FORWARD+

## Forward+: Bringing Deferred Lighting to the Next Level

Takahiro Harada, by McKin, and Aaron C. Yang

Advanced Micro Devices, Inc.

### Abstract

This paper presents *Forward+*, a method of rendering many lights by culling and storing only lights that contribute to the pixel. *Forward+* is an extension to traditional forward rendering. Light culling, implemented using the compact capabilities of the GPU, is added to the pipeline to create lists of lights that are passed to the final rendering shader which can access all information about the lights. Although *Forward+* increases workload to the final shader, it theoretically requires less memory traffic compared to compare-based deferred lighting. Furthermore, it removes the major drawback of deferred techniques, which is a restriction of materials and lighting models. Experiments are performed to compare the performance of *Forward+* and deferred lighting.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Three-Dimensional Graphics and Realism: Color, shading, shadowing, and texture

### 1. Introduction

In recent years, deferred rendering has gained its popularity for rendering in real time, especially in games. The major advantages of deferred techniques are the ability to use many lights, decoupling of lighting from geometry complexity, and manageable shader combinations. However, deferred techniques have drawbacks such as limited material variety, higher memory and bandwidth requirements, handling of transparent objects, and lack of hardware anti-aliasing support (Raytr). Material variety is critical in achieving realistic shading results, which is not a problem for forward rendering. However, forward rendering normally requires writing a small fixed number of lights to limit the potential explosion of shader permutations and needs CPU management of the lights and objects. Also, with expensive dynamic instancing performance on current consoles (e.g., Xbox 360) it is understandable why deferred rendering has become appealing.

The latest GPUs have improved performance, more ALU power and flexibility, and the ability to perform general computation – in contrast to current consoles. Thus, rendering with many lights with forward rendering could be a realistic option, however the naive approach of iterating through every light in a per-pixel shading fashion is impractical.

We present *Forward+* – a method of rendering with many lights by culling and storing only lights that contribute to the pixel. The lights are evaluated one by one in the final shader.



Figure 1: A screenshot from the AMD Last Minute using *Forward+*.

In this manner, we retain all the positive aspects of forward rendering and gain the ability to render with lots of lights.

This paper first presents the pipeline and gives a high-level explanation of the implementation. The theoretical memory traffic of *Forward+* is compared to deferred lighting.

### 2. Related Work

Forward rendering has practical limitations on the number of lights that can be used when shading (AMD08). De-



- Rendering equation

$$L = \sum_i^n \{L_e f(x, w_i, w_o) V(w_o)\}$$

- Forward

$$L_{forward} = \sum_i^m \{L_e f(x, w_i, w_o) V'(w_o)\}$$

- Deferred

$$L_{deferred} = \sum_i^{\tilde{n}} \{L_e V'(w_o)\} f(x, w_i)$$

- Forward+

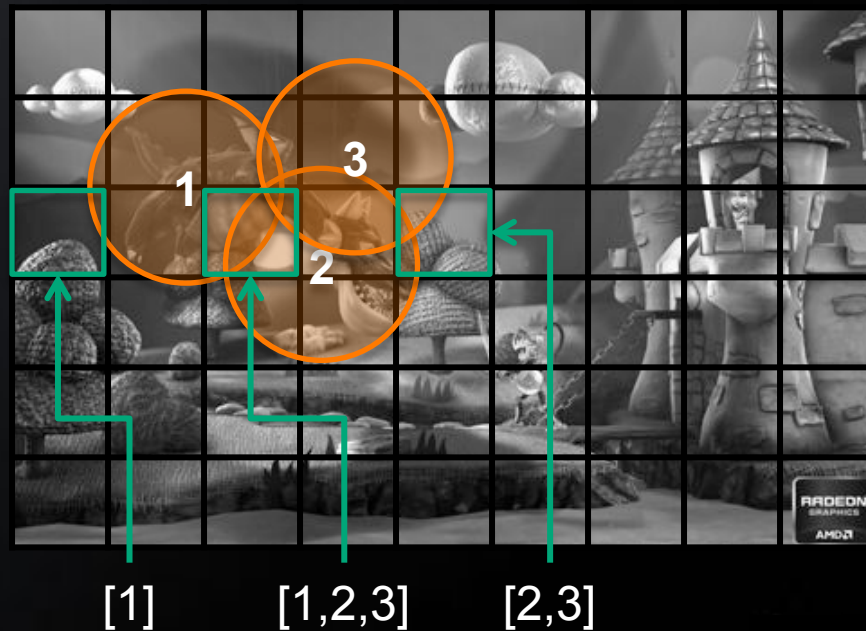
$$L_{forward+} = \sum_i^{\tilde{n}} \{L_e f(x, w_i, w_o) V'(w_o)\}$$
$$m \leq \tilde{n} \leq n$$



- Depth prepass
  - Fills z buffer
    - Prevent overdraw for shading
  
- Shading
  - Geometry is rendered
  - Pixel shader
    - Iterate through light list **set for each object**
    - Evaluates materials for the lights



- Depth prepass
  - Fills z buffer
    - Prevent overdraw for shading
    - Used for pixel position reconstruction for light culling
- Light culling
  - Culls light per tile basis
  - Input: z buffer, light buffer
  - Output: light list per tile
- Shading
  - Geometry is rendered
  - Pixel shader
    - Iterate through light list **calculated in light culling**
    - Evaluates materials for the lights

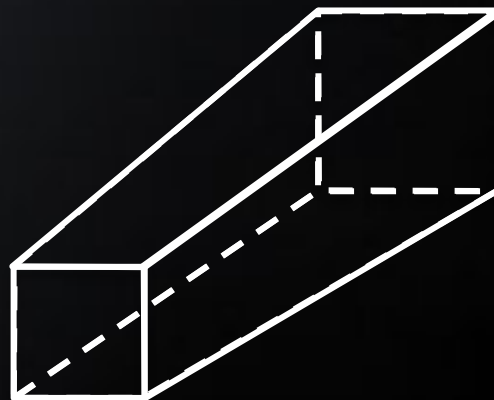
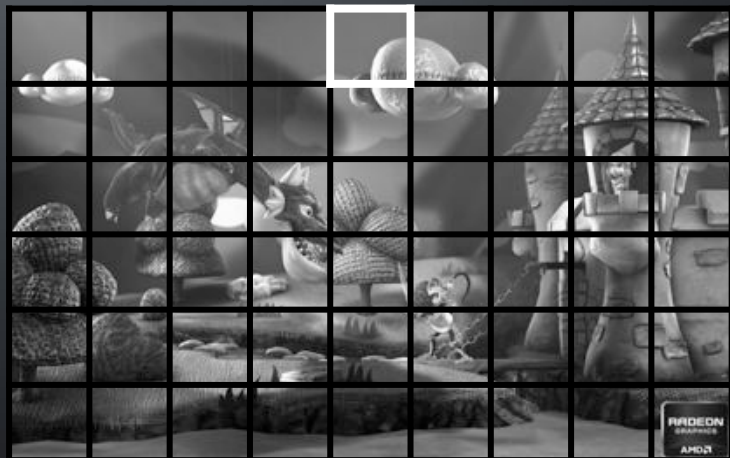


# CREATING A FRUSTUM FOR A TILE



SIGGRAPH  
ASIA 2012

- An edge @SS == A plane @VS
- A tile (4 edges) @SS == 4 planes @VS
  - Open frustum (no bound in Z direction)
- Max and min Z is used to cap



# LONG FRUSTUM

- Screen space culling is not always sufficient
  - Create a frustum from max and min depth values
  - Edge of objects
  - Captures a lot of unnecessary lights



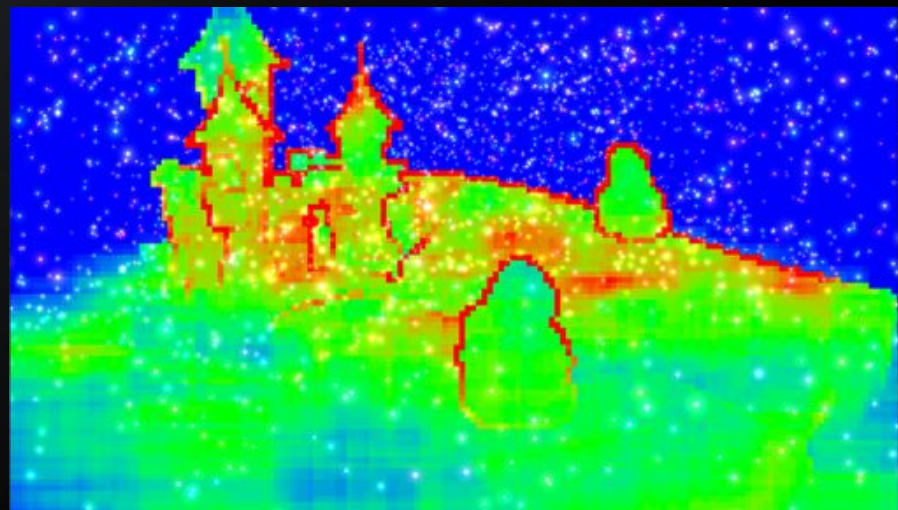
SIGGRAPH  
ASIA2012



# LONG FRUSTUM

- Screen space culling is not always sufficient
  - Create a frustum from max and min depth values
  - Edge of objects
  - Captures a lot of unnecessary lights

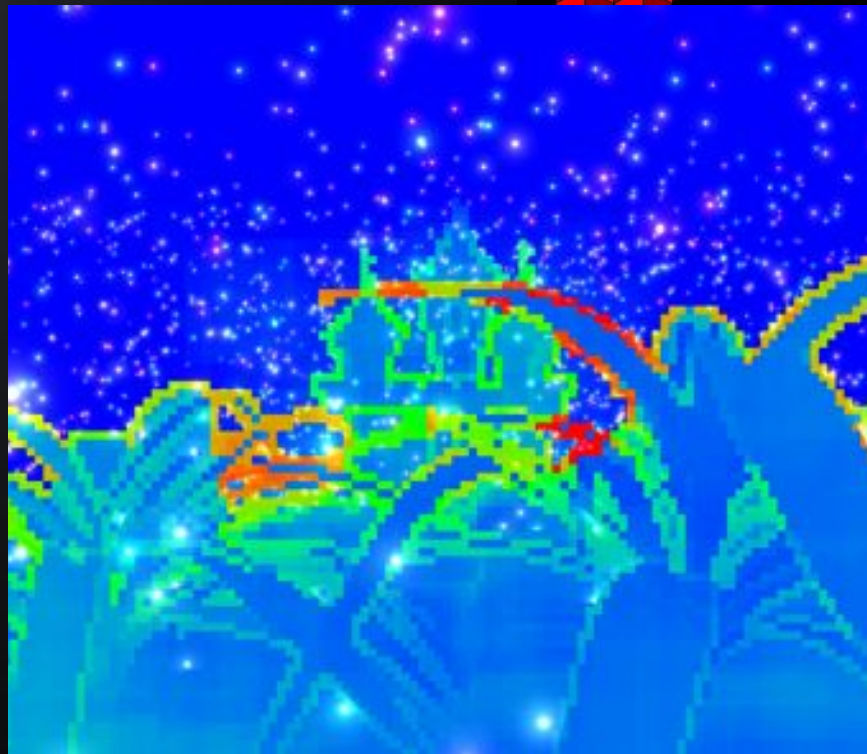
■ 0 lights  
■ 25 lights  
■ 50 lights



## GET WORSE IN A COMPLEX SCENE



SIGGRAPH  
ASIA 2012



■ 0 lights  
■ 100 lights  
■ 200 lights

## QUESTION



SIGGRAPH  
ASIA 2012

- Want to reduce false positives
- Can we improve the culling without adding much overhead?
  - Computation time, memory
  - Culling itself is an optimization
  - Spending a lot of resources for it does not make sense
- Using a 3D grid is a natural extension
  - Uses too much memory



SIGGRAPH  
ASIA2012

## ***2.5D CULLING***

## 2.5D CULLING

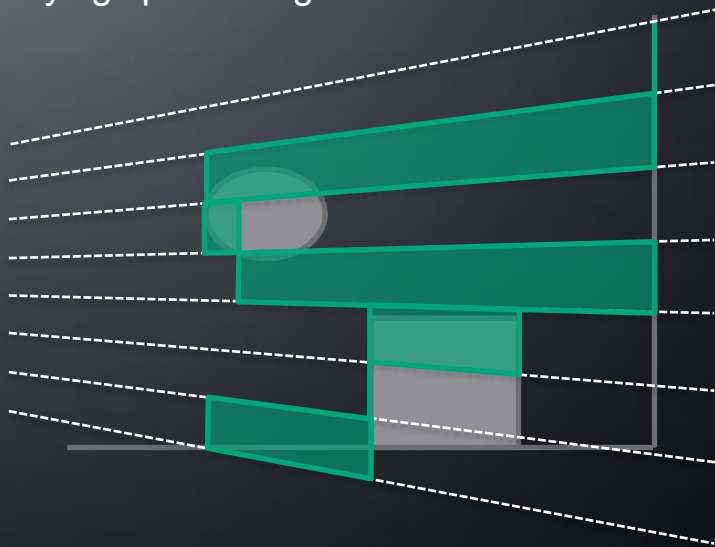


SIGGRAPH  
ASIA 2012

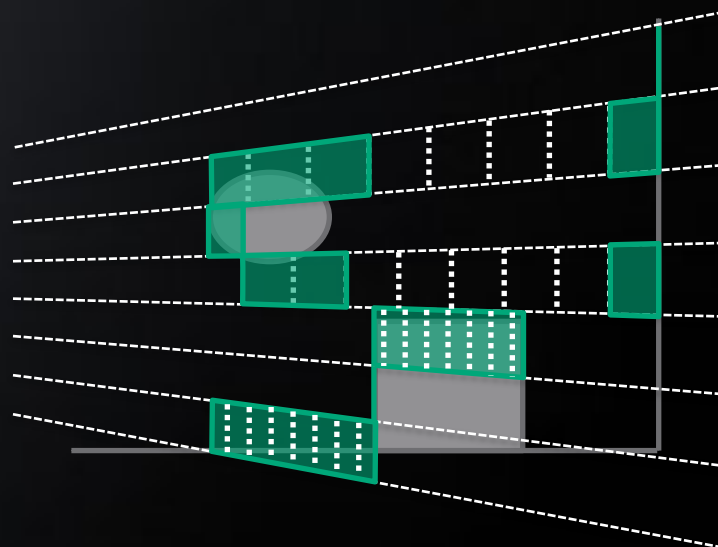
- Additional memory usage
  - 0B global memory
  - 4B local memory per WG (can compress more if you want)
- Additional computation complexity
  - A few bit and arithmetic instructions
  - A few lines of codes for light culling
  - No changes for other stages
- Additional runtime overhead
  - < 10% compared to the original light culling



- Split frustum in z direction
  - Uniform split for a frustum
  - Varying split among frustums



(a)

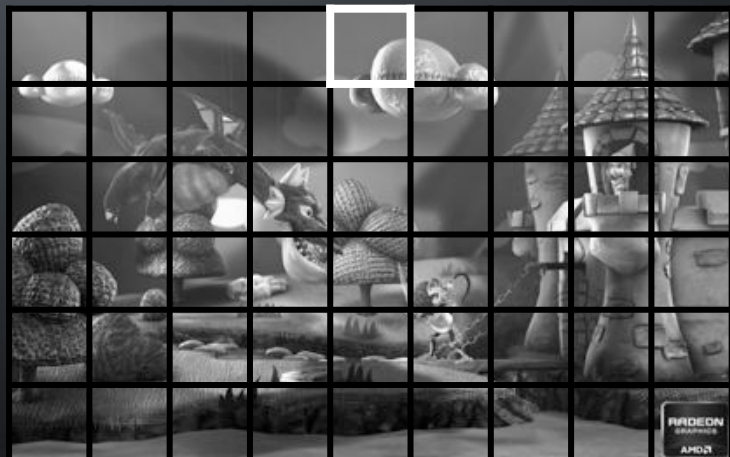


(b)

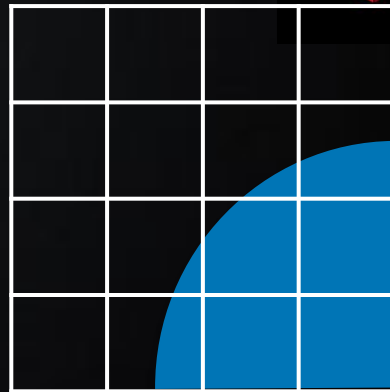


# FRUSTUM CONSTRUCTION

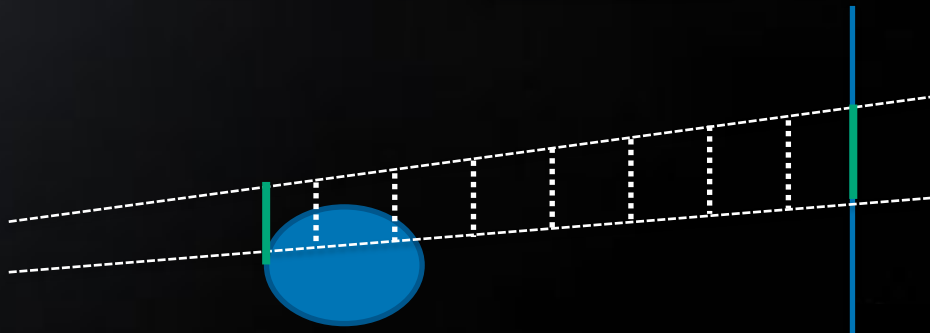
- Calculate depth bound
  - max and min values of depth
- Split depth direction into 32 cells
  - Min value and cell size
- Flag occupied cell
- A 32bit depth mask per work group



A tile

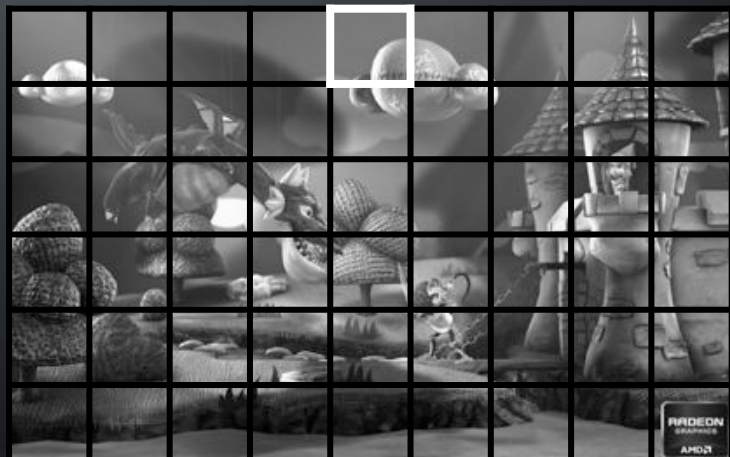


SIGGRAPH  
ASIA 2012



# FRUSTUM CONSTRUCTION

- Calculate depth bound
  - max and min values of depth
- Split depth direction into 32 cells
  - Min value and cell size
- Flag occupied cell
- A 32bit depth mask per work group



SIGGRAPH  
ASIA2012

A tile

7	7	7	7
7	7	7	2
7	7	2	1
7	2	1	0

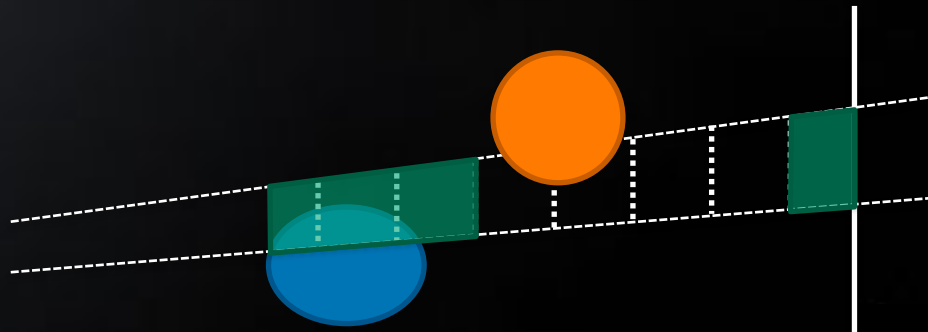


Depth mask = 11100001



- If a light overlaps to the frustum
  - Calculate depth mask for the light
  - Check overlap using the depth mask of the frustum
- Depth mask & Depth mask
  - $11100001 \text{ \& } 00011000 = 00000000$

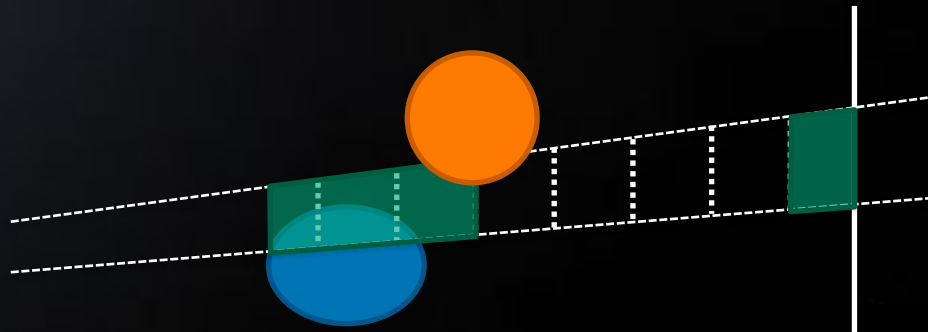
Depth mask = 00011000



Depth mask = 11100001

- If a light overlaps to the frustum
  - Calculate depth mask for the light
  - Check overlap using the depth mask of the frustum
- Depth mask & Depth mask
  - $11100001 \text{ \& } 00110000 = 00100000$

Depth mask = 00110000



Depth mask = 11100001



## Original

**Algorithm 1** Pseudo-code for the 2.5D culling. The difference from frustum culling is highlighted in red.

```

1: frustum[0-4] ← Compute 4 planes at the boundary of a tile
2: z ← Fetch depth value of the pixel
3: ldsMinZ ← atomMin(z)
4: ldsMaxZ ← atomMax(z)
5: frustum[5,6] ← Compute 2 planes using ldsMinZ, ldsMaxZ

7: for all the lights do
8:   iLight ← lights[i]
9:   if overlaps( iLight, frustum ) then

12:     if overlapping then
13:       appendLight( i )
14:     end if
15:   end if
16: end for
17: flushLightIndices()

```

## With 2.5D culling

**Algorithm 1** Pseudo-code for the 2.5D culling. The difference from frustum culling is highlighted in red.

```

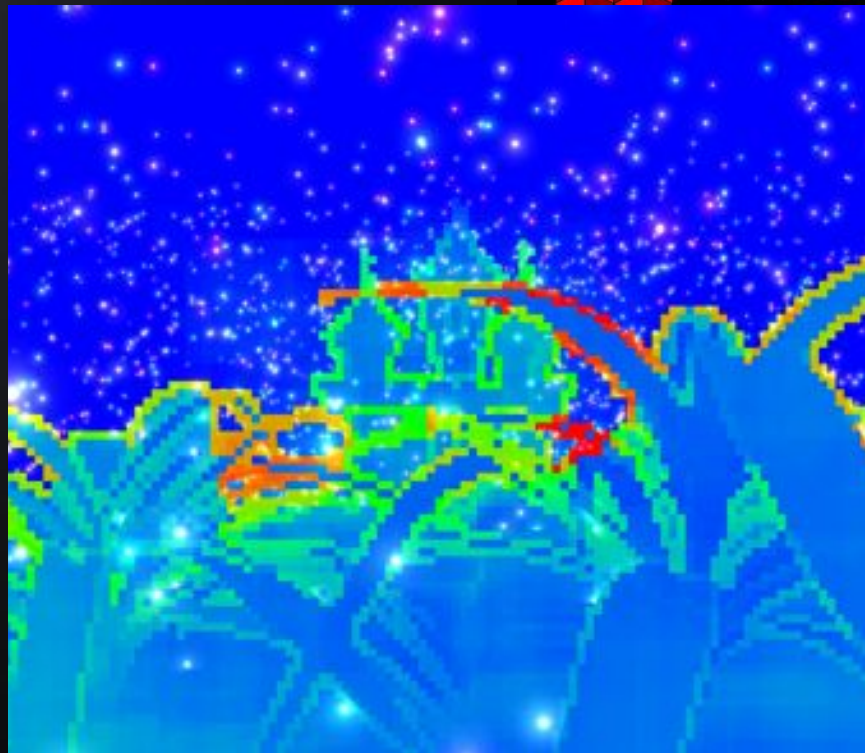
1: frustum[0-4] ← Compute 4 planes at the boundary of a tile
2: z ← Fetch depth value of the pixel
3: ldsMinZ ← atomMin(z)
4: ldsMaxZ ← atomMax(z)
5: frustum[5,6] ← Compute 2 planes using ldsMinZ, ldsMaxZ
6: depthMaskT ← atomOr( 1 << getCellIndex(z) )
7: for all the lights do
8:   iLight ← lights[i]
9:   if overlaps( iLight, frustum ) then
10:     depthMaskL ← Compute mask using light extent
11:     overlapping ← depthMaskT & depthMaskL
12:     if overlapping then
13:       appendLight( i )
14:     end if
15:   end if
16: end for
17: flushLightIndices()

```



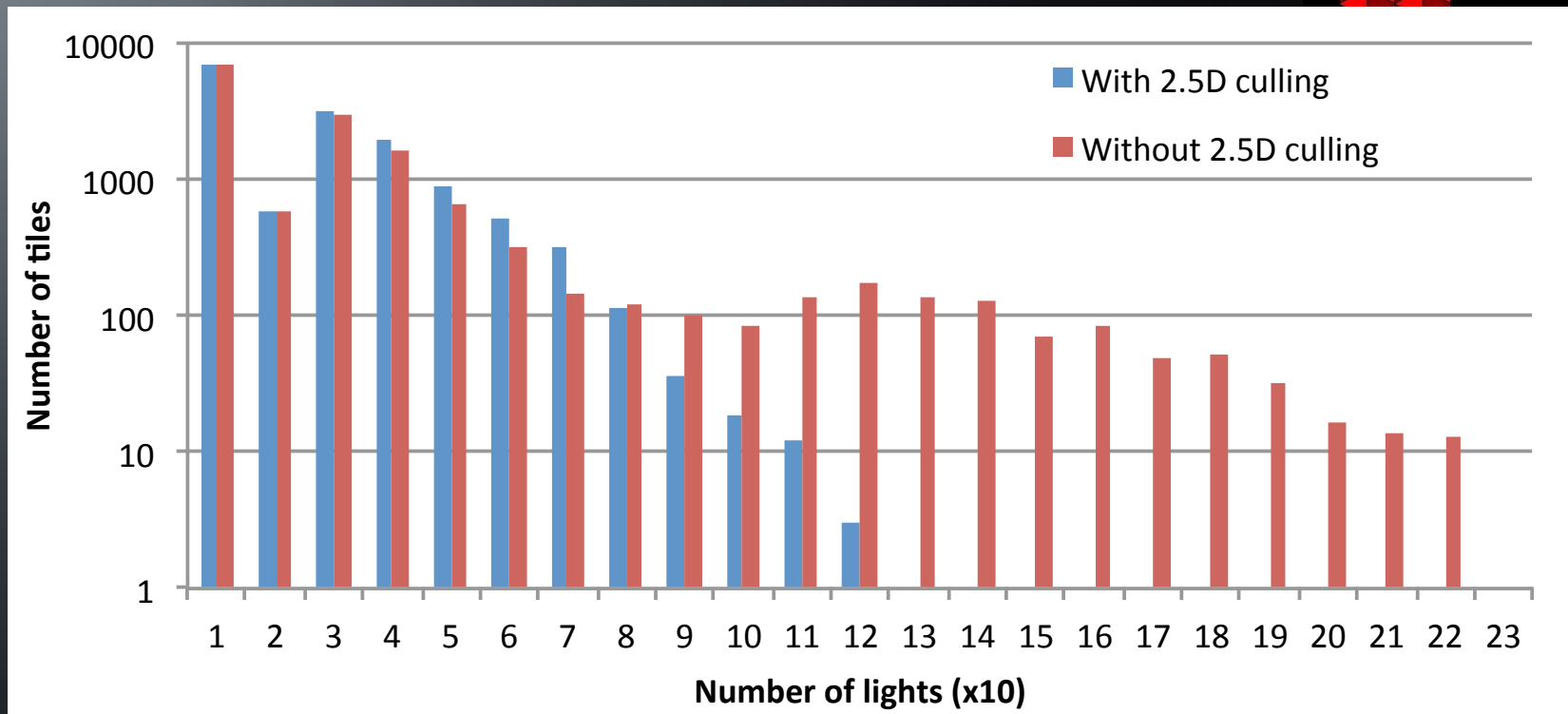
SIGGRAPH  
ASIA2012

## ***RESULTS***

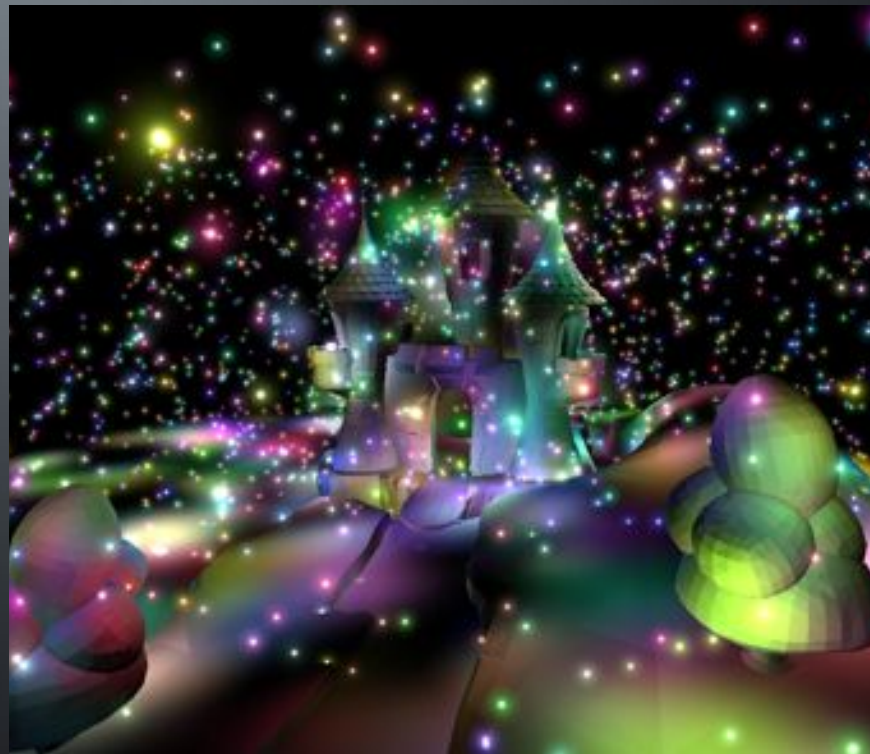




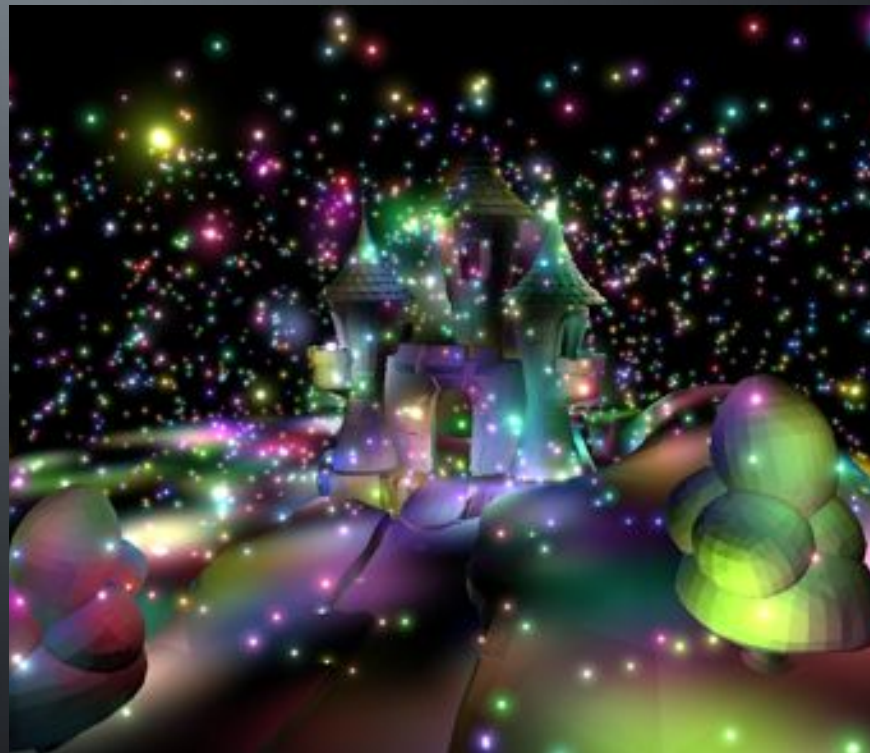


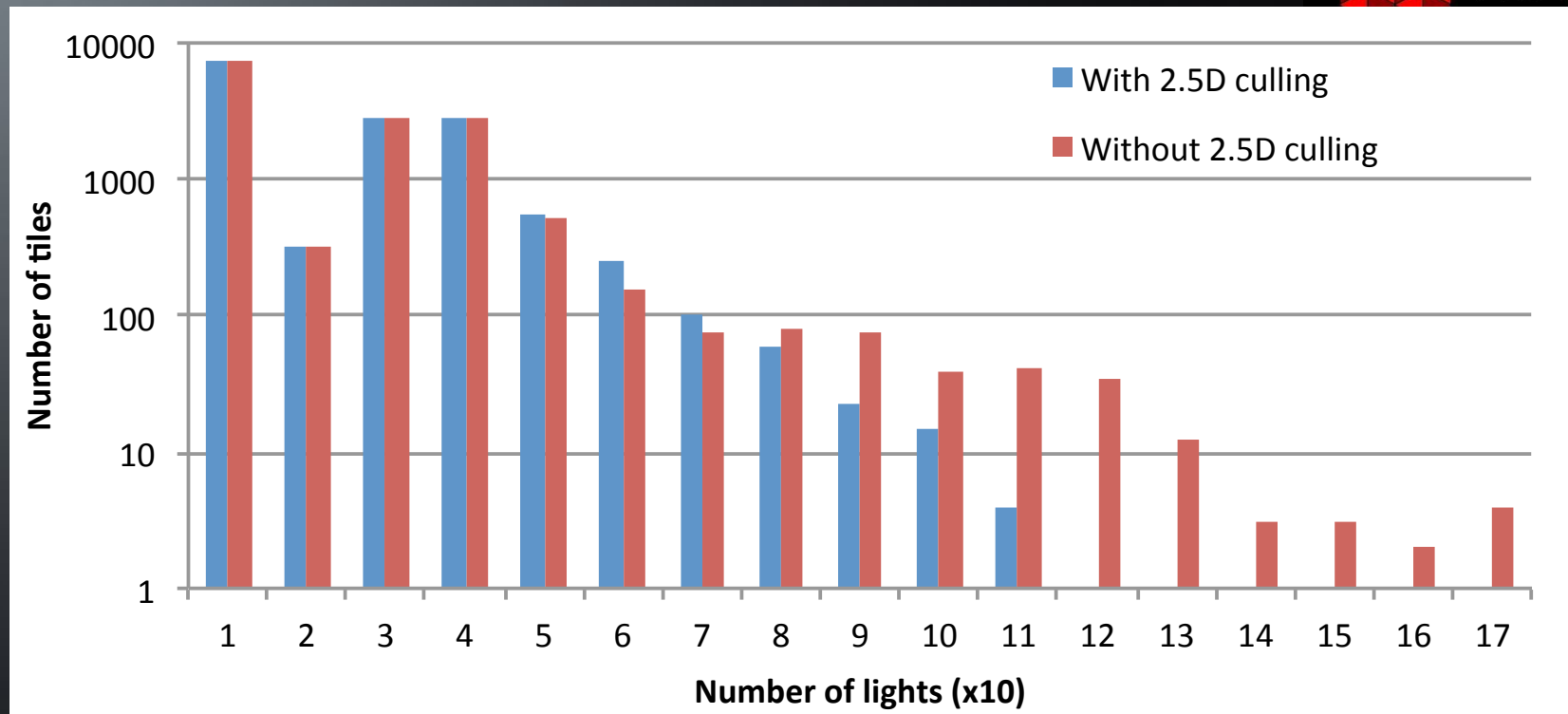


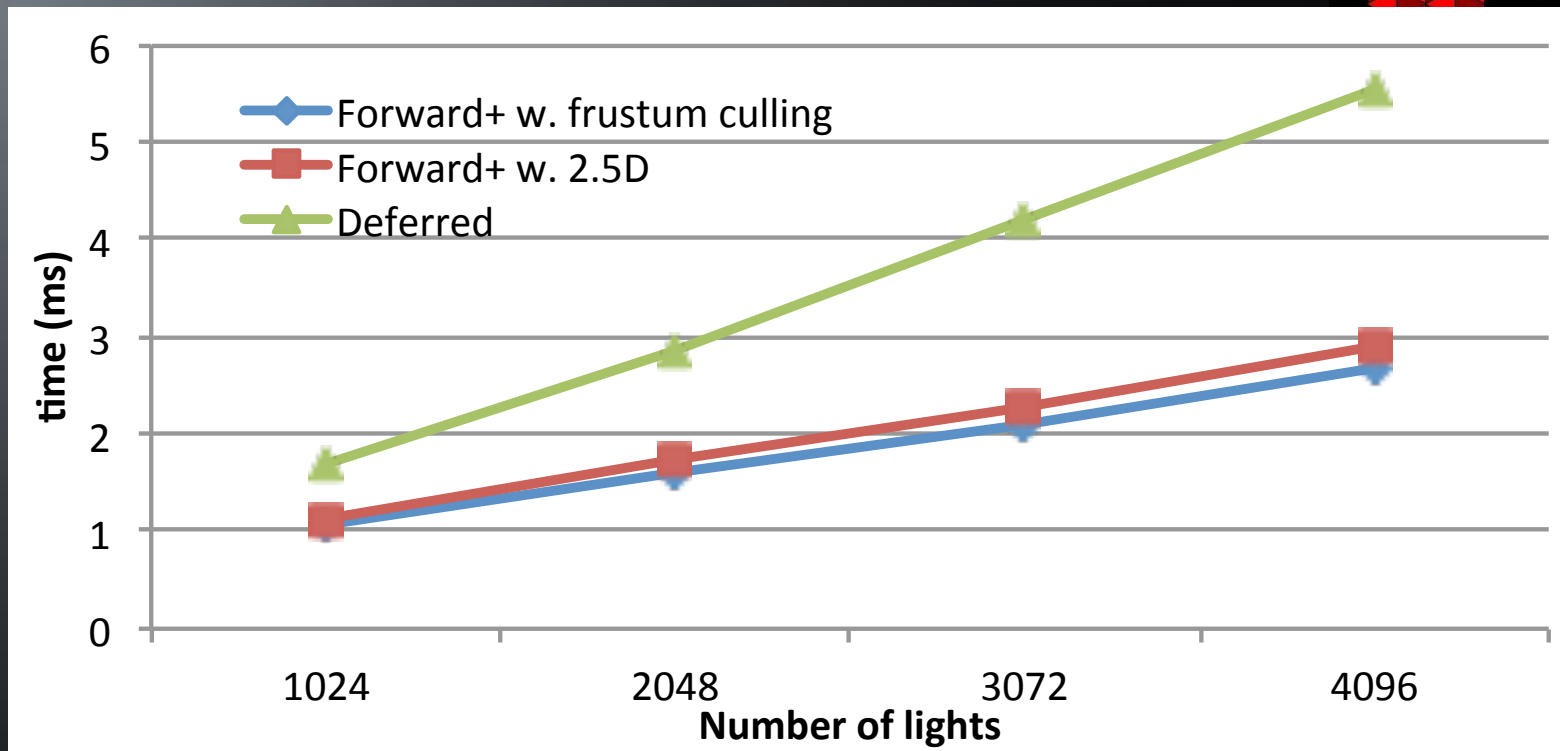
220 lights/frustum -> 120 lights/frustum











# CONCLUSION



SIGGRAPH  
ASIA 2012

- Proposed 2.5D culling which
  - Additional memory usage
    - **0B** global memory
    - **4B** local memory per WG (can compress more if you want)
  - Additional compute complexity
    - **3 lines** of pseudo codes for light culling
    - No changes for other stages
  - Additional runtime overhead
    - **< 10%** compared to the original light culling
- Showed that 2.5D culling reduces false positives