A 2.5D CULLING FOR FORWARD+

AMD
Takahiro Harada

AGENDA

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

FORWARD+

EUROGRAPHICS 2012 / C. Andujar, E. Puppo

Forward+: Bringing Deferred Lighting to the Next Level

Takahiro Harada, Jay McKee, and Jason C.Yang

Advanced Micro Devices, Inc.

Abstract

1. Introduction

Figure 1: A screenshot from the AMD Leo demo using Forward+.

2. Related Work

REAL-TIME SOLUTION COMPARISON

- 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+

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

$$m \leq \tilde{n} \leq n$$

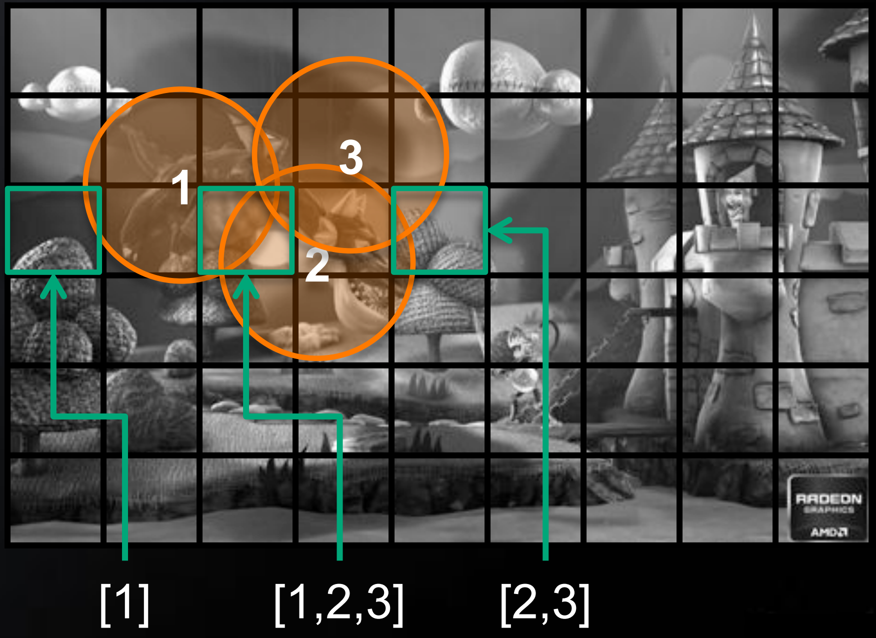
FORWARD RENDERING PIPELINE

SIGGRAPH ASIA 2012

- 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

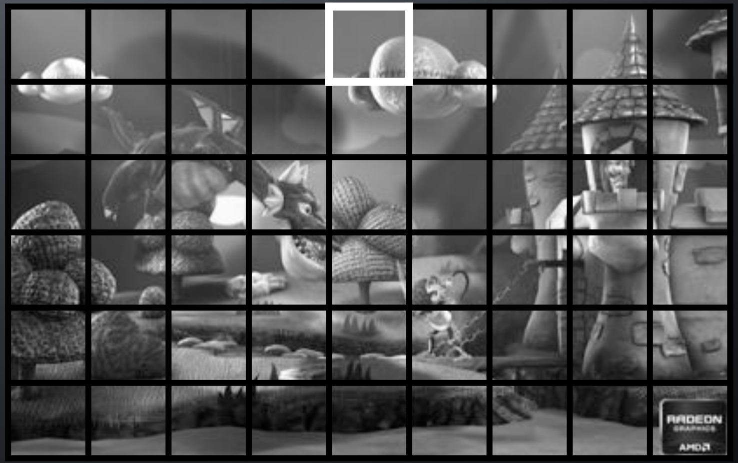
FORWARD+ RENDERING PIPELINE

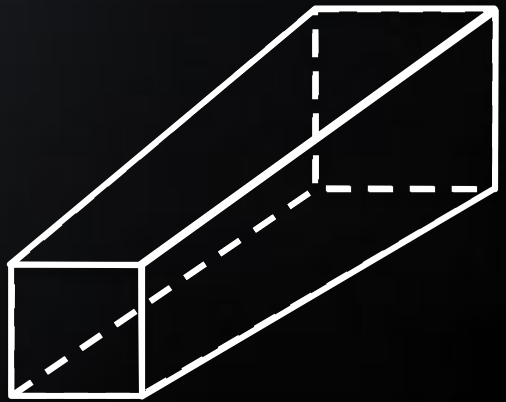
- Depth prepass
  - Fills z buffer
    - Prevent overdraw for shading
    - Used for pixel position reconstruction for light culling
- Light culling
  - Culls light per tile basis
  - Input: z buffer, light buffer
  - Output: light list per tile
- Shading
  - Geometry is rendered
  - Pixel shader
  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



CREATING A FRUSTUM FOR A TILE

- 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





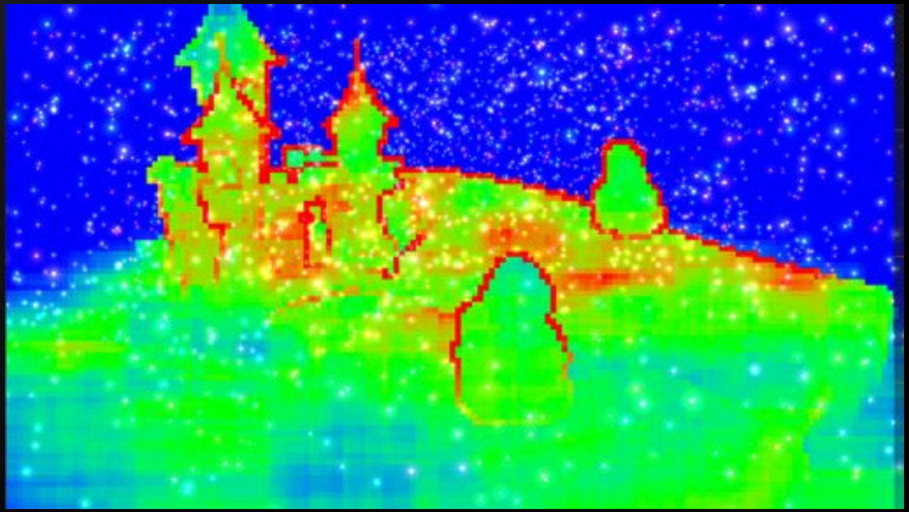
LONG FRUSTUM

- 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

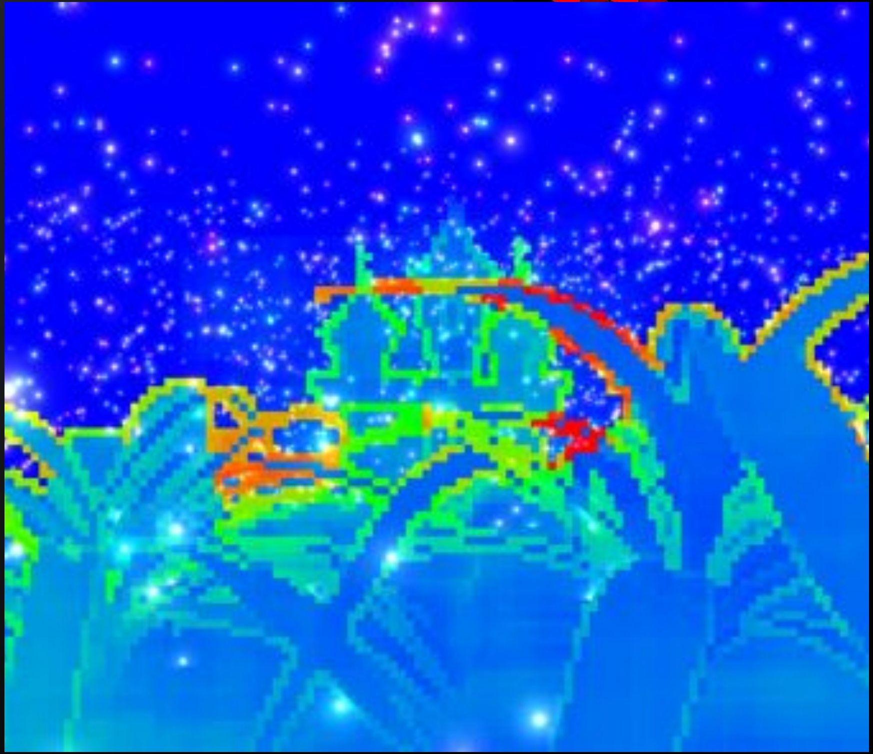
LONG FRUSTUM

- 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



- 0 lights
- 100 lights
- 200 lights

QUESTION

- 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

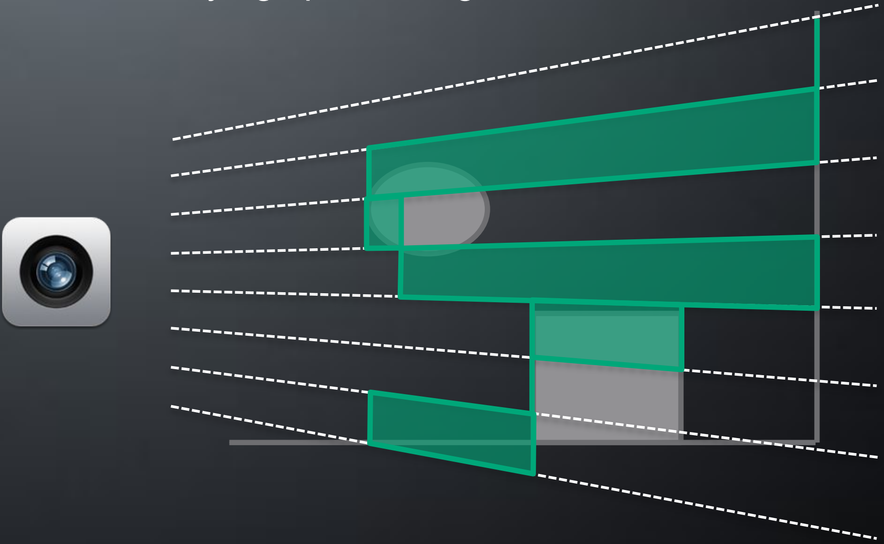
2.5D CULLING

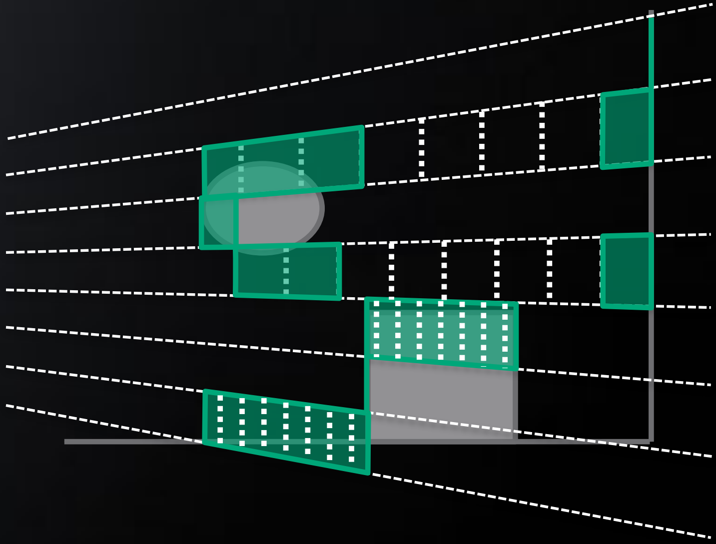
2.5D CULLING

- 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

IDEA

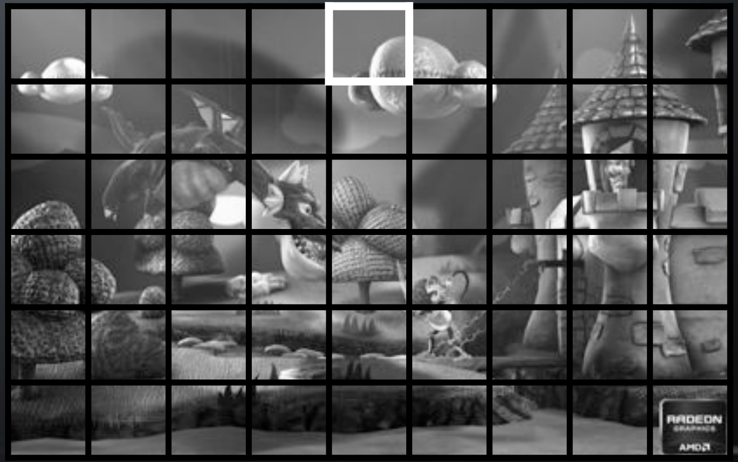
- Split frustum in z direction
  - Uniform split for a frustum
  - Varying split among frustums


(a)

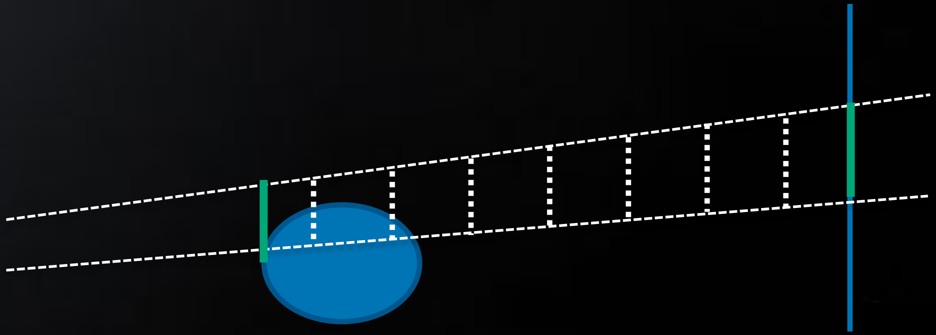

(b)

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



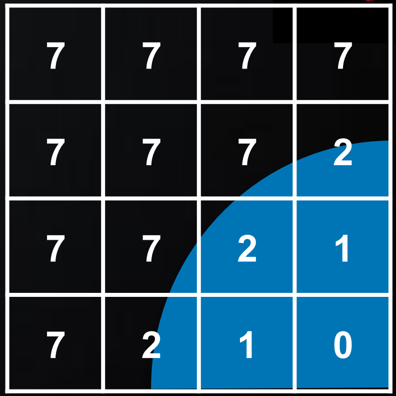
A tile

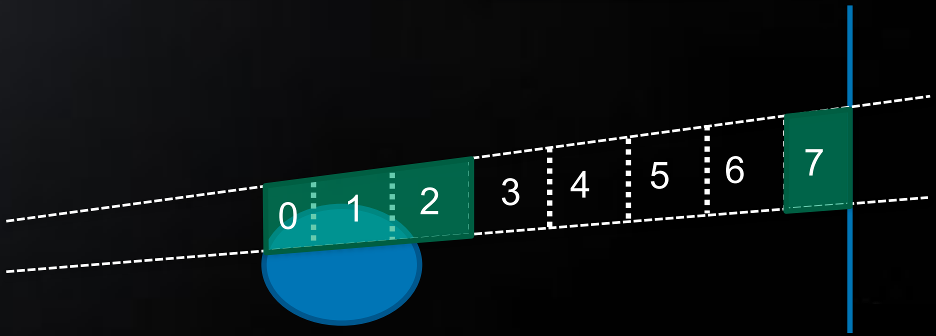


FRUSTUM CONSTRUCTION

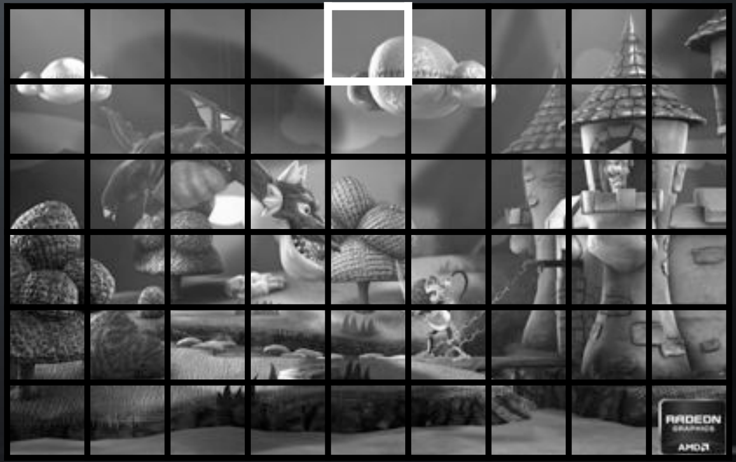
- Calculate depth bound
  - max and min values of depth
- Split depth direction into 32 cells
  - Min value and cell size
- Flag occupied cell
- A 32bit depth mask per work group

A tile





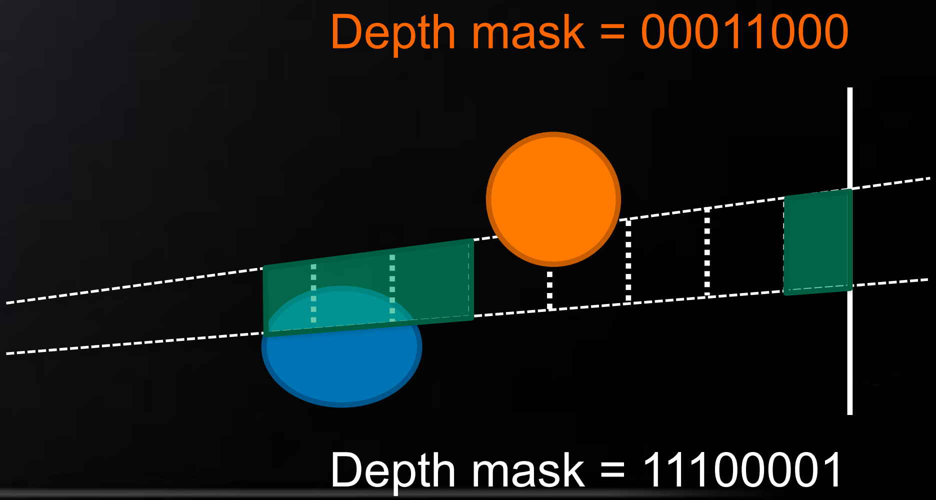
Depth mask = 11100001



LIGHT CULLING

- If a light overlaps to the frustum
  - Calculate depth mask for the light
  - Check overlap using the depth mask of the frustum

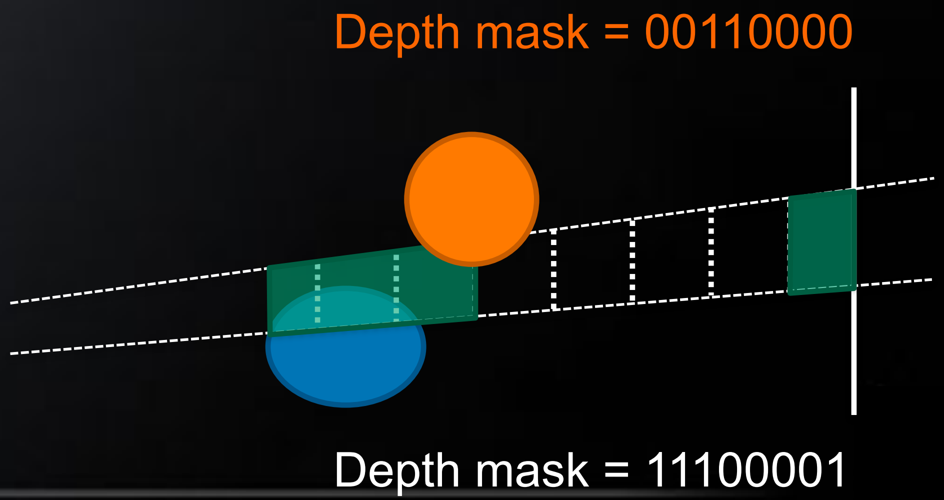
- Depth mask & Depth mask
  - 11100001 & 00011000 = 00000000



LIGHT CULLING

- 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 & 00110000 = 00100000



CODE

Original

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

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

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


        if overlapping then
            appendLight( i )
        end if
    end if
end for
flushLightIndices()
```

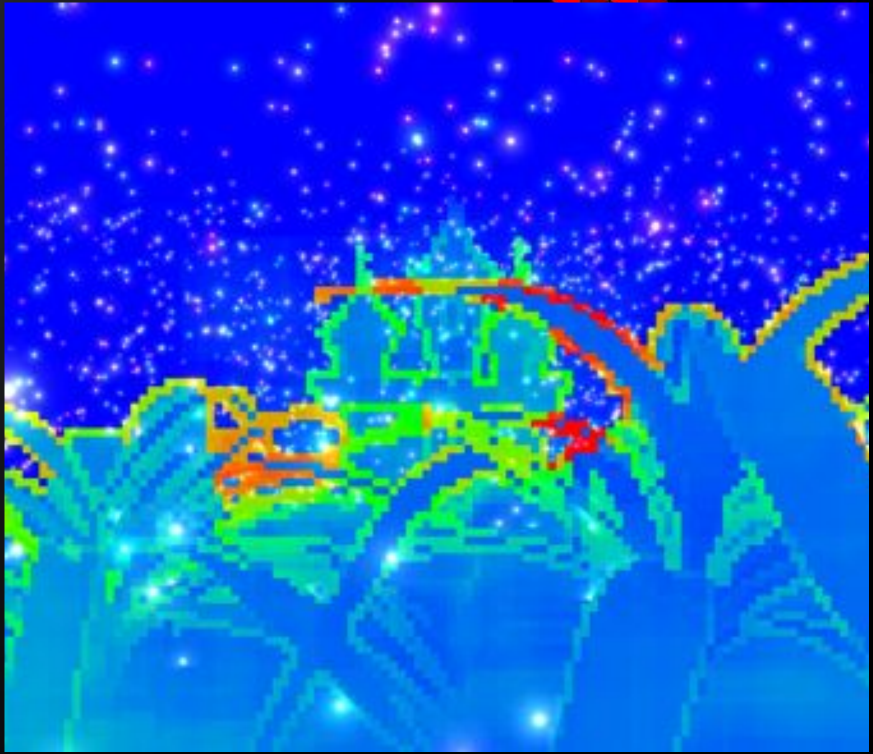
With 2.5D culling

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

```
frustum[0-4] ← Compute 4 planes at the boundary of a tile
z ← Fetch depth value of the pixel
ldsMinZ ← atomMin(z)
ldsMaxZ ← atomMax(z)
frustum[5,6] ← Compute 2 planes using ldsMinZ, ldsMaxZ
depthMaskT ← atomOr( 1 ≪ getCellIndex(z) )
for all the lights do
    iLight ← lights[i]
    if overlaps( iLight, frustum ) then
        depthMaskL ← Compute mask using light extent
        overlapping ← depthMaskT ∧ depthMaskL
        if overlapping then
            appendLight( i )
        end if
    end if
end for
flushLightIndices()
```

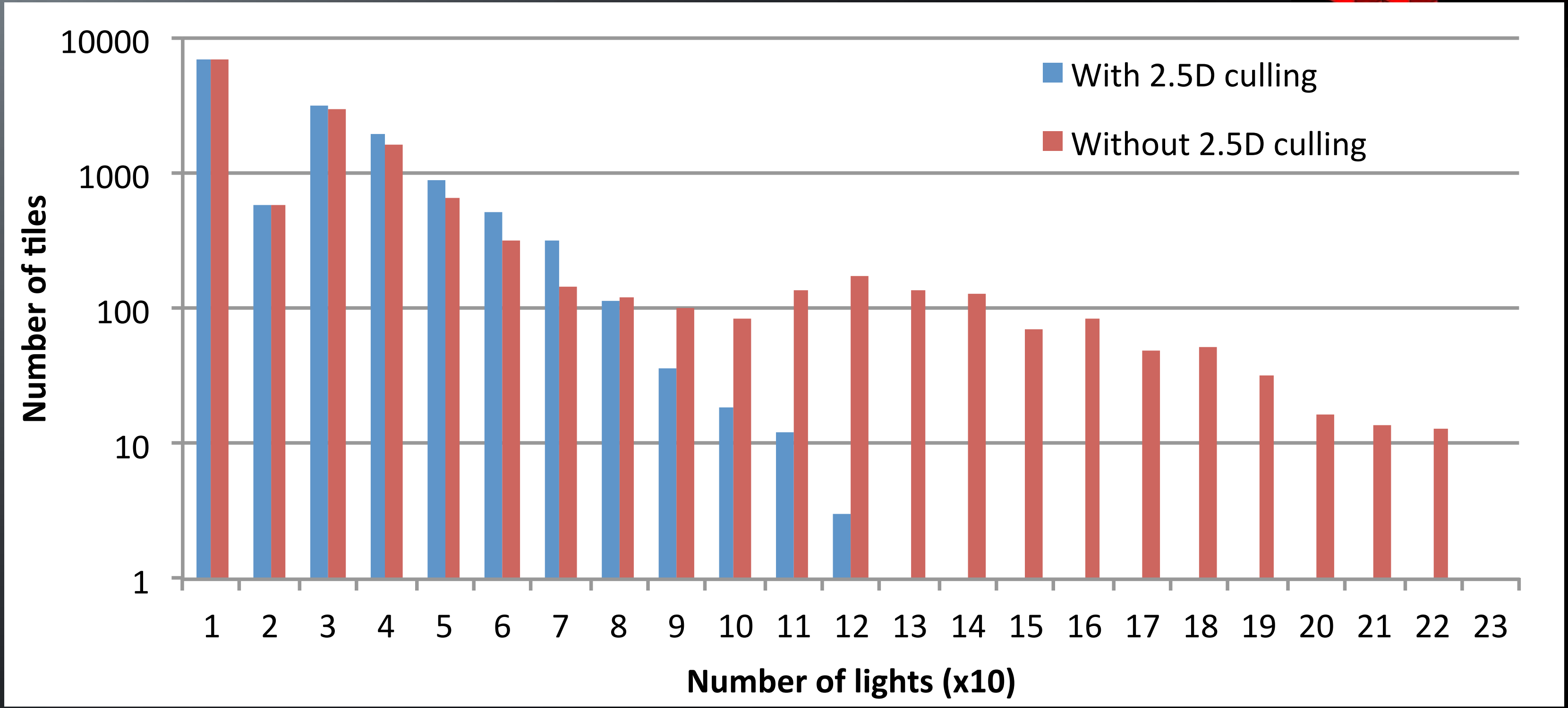
RESULTS

LIGHT CULLING



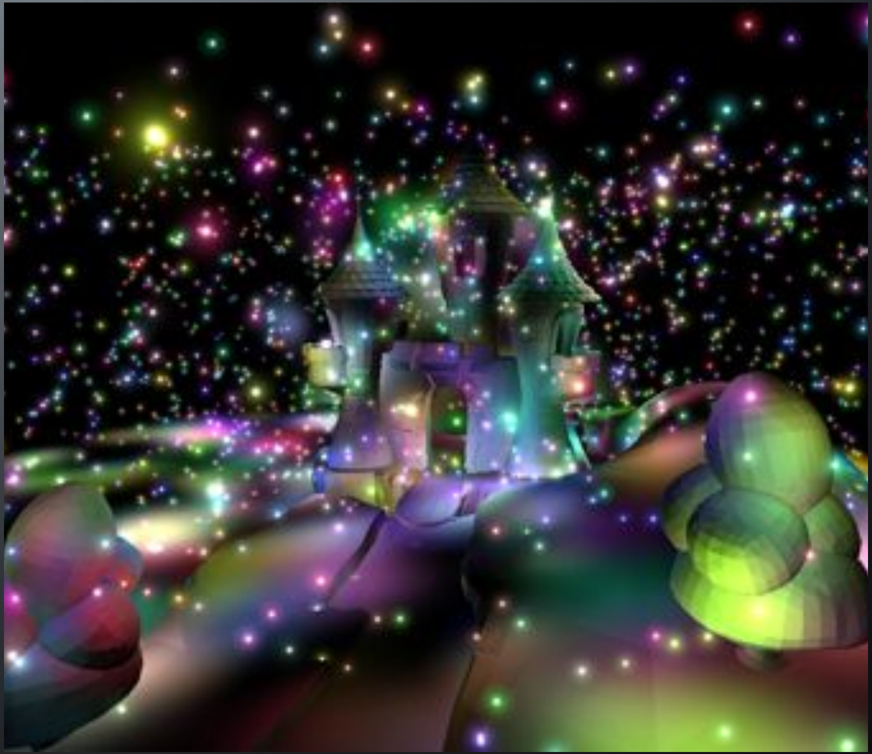
LIGHT CULLING + 2.5D CULLING

COMPARISON

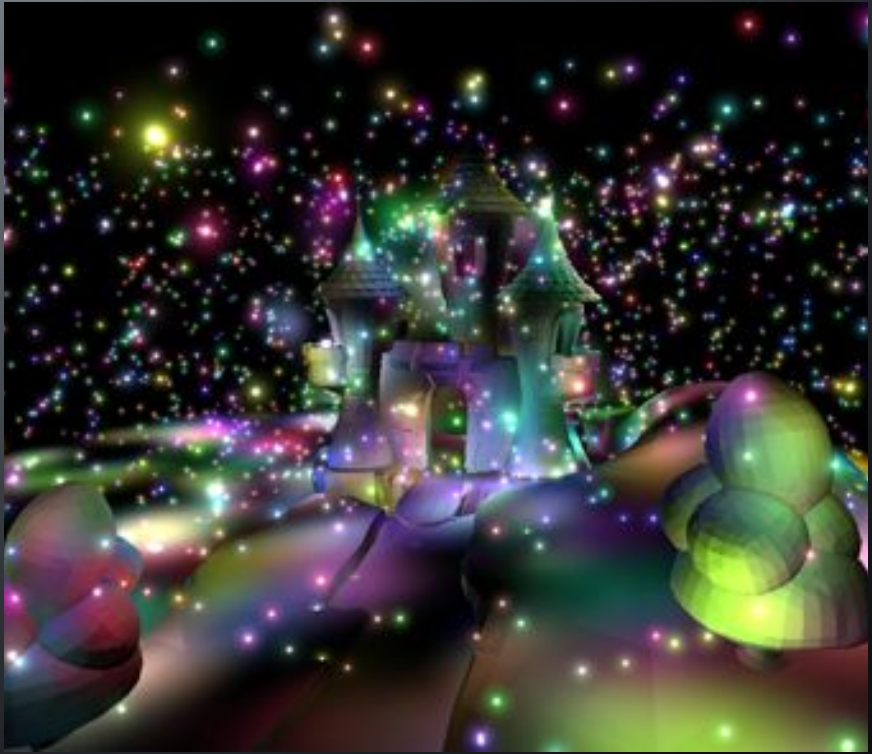


220 lights/frustum -> 120 lights/frustum

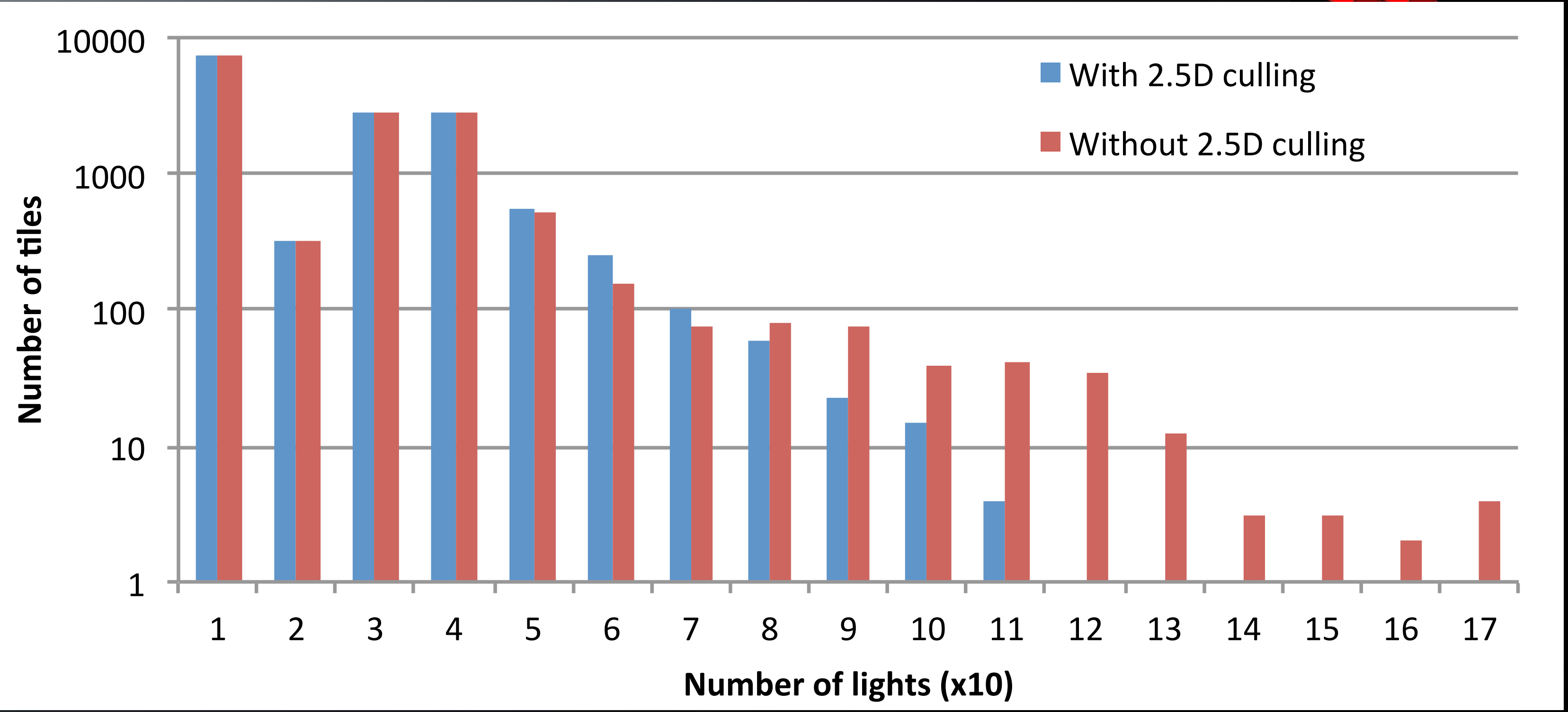
LIGHT CULLING



LIGHT CULLING + 2.5D CULLING

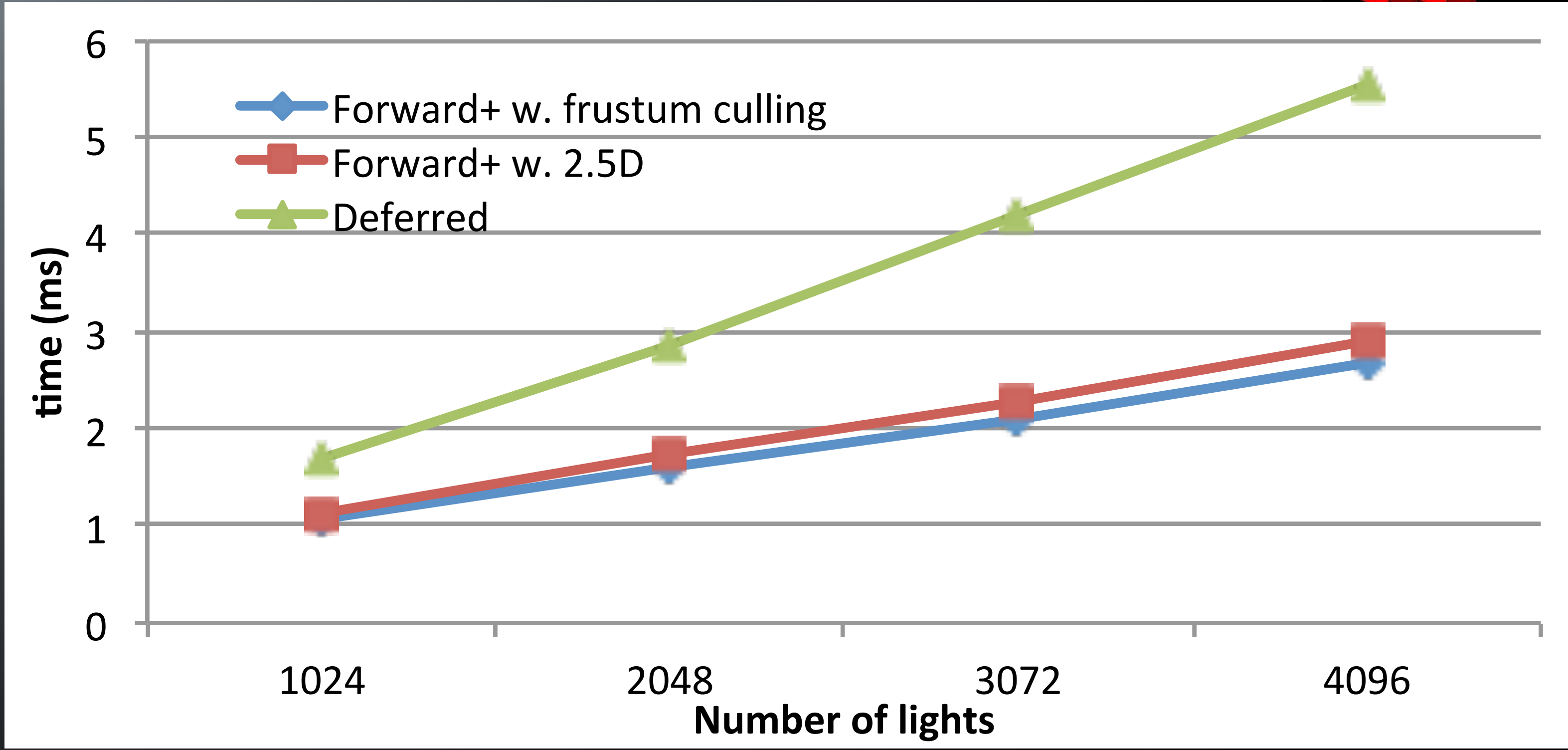


COMPARISON



With 2.5D culling
Without 2.5D culling
Number of tiles
10000
1000
100
10
1
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
Number of lights (x10)

PERFORMANCE



CONCLUSION

- 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