A 2.5D CULLING FOR FORWARD+

AMD
Takahiro Harada
AGENDA

- Forward+
  - Forward, Deferred, Forward+
  - Problem description
- 2.5D culling
- Results
FORWARD+
REAL-TIME SOLUTION COMPARISON

- Rendering equation

\[ L = \sum_{i}^{n} \{ L_{ef}(x, w_i, w_o) V(w_o) \} \]

- Forward

\[ L_{\text{forward}} = \sum_{i}^{m} \{ L_{ef}(x, w_i, w_o) V'(w_o) \} \]

- Deferred

\[ L_{\text{deferred}} = \sum_{i}^{\tilde{n}} \{ L_{e} V'(w_o) \} f(x, w_i) \]

- Forward+

\[ L_{\text{forward+}} = \sum_{i}^{\tilde{n}} \{ L_{ef}(x, w_i, w_o) V'(w_o) \} \quad m \leq \tilde{n} \leq n \]
FORWARD RENDERING PIPELINE

- 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
GET WORSE IN A COMPLEX SCENE
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
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
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

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

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
  - $11100001 \& 00110000 = 00100000$

Depth mask = 11100001

Depth mask = 00110000
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()
RESULTS
LIGHT CULLING
LIGHT CULLING + 2.5D CULLING
COMPARISON

With 2.5D culling

Without 2.5D culling

Number of lights (x10)

Number of tiles

220 lights/frustum -> 120 lights/frustum
LIGHT CULLING + 2.5D CULLING
COMPARISON

Number of tiles

Number of lights (x10)

With 2.5D culling
Without 2.5D culling
Performance comparison between Forward+ with frustum culling, Forward+ with 2.5D culling, and Deferred rendering methods. The graph shows the time (ms) on the y-axis and the number of lights on the x-axis. The data indicates a linear increase in time with an increase in the number of lights for all methods. The Deferred method shows the highest time increase, followed by Forward+ with frustum culling, and then Forward+ with 2.5D culling.
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