Efficient Software Occlusion Culling on Mobile Platform in 'Life After'

Wenhui Tao
Engine Programmer

Agenda

- Motivation
- Lightweight Software Occlusion Culling Algorithm
- Optimized Culling Pipeline
- High-quality Occlusion Mesh Generator
- Conclusion

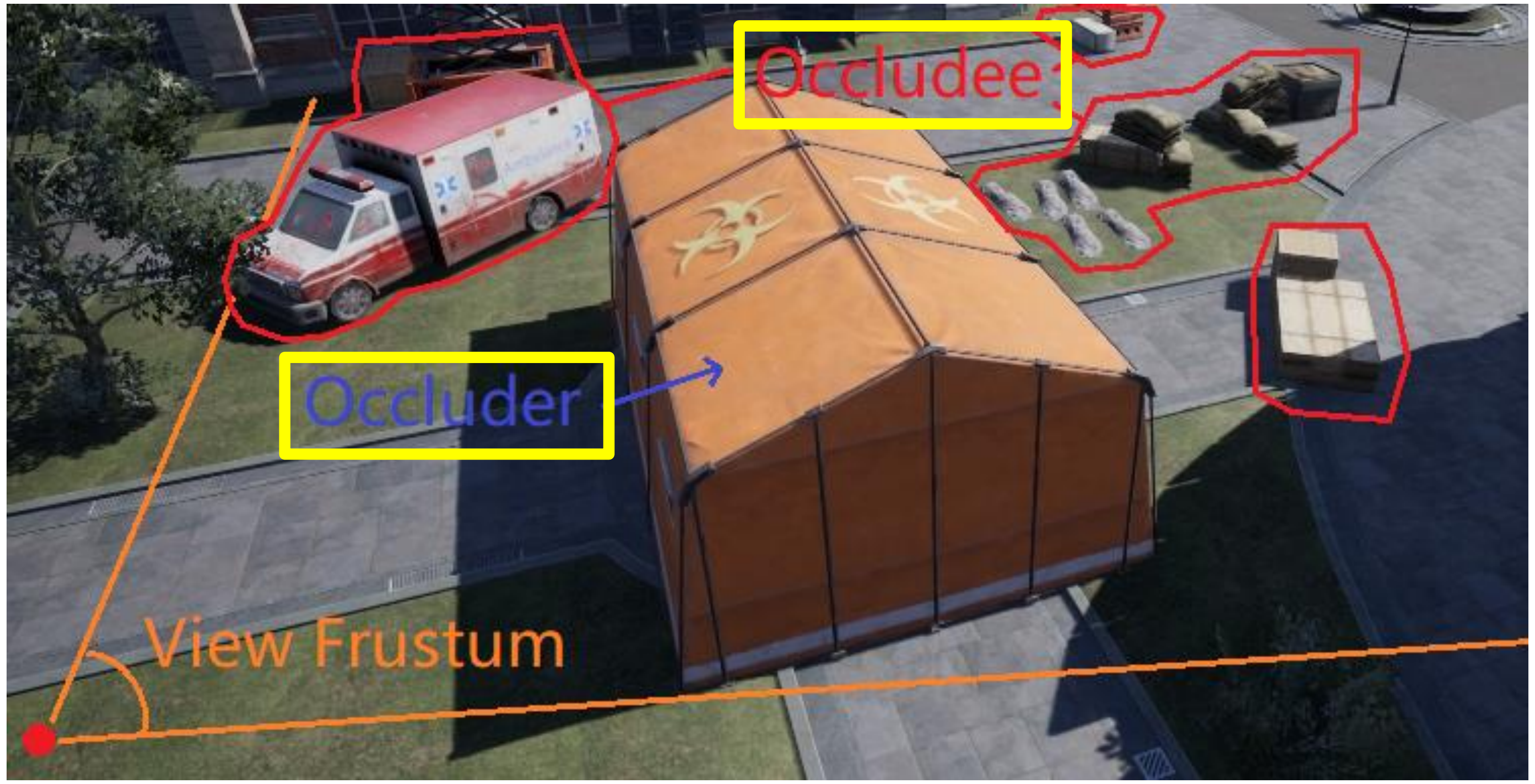
Part 1: Motivation

Background

- Life After

Background

- Life After
- Occlusion Culling
- Occluder
- Occludee
- Culling Rate:
  - Culled Occludees’ Count / Total Occludees’ Count
- False Occlusion
  - A visible occludee be incorrectly culled



Occlusion Culling

- Important for performance
  - 60+% objects can be culled

- Requirements
  - High culling rate
  - NO false occlusion
  - Supporting dynamic objects
  - Small package size
  - Fast

Related Work

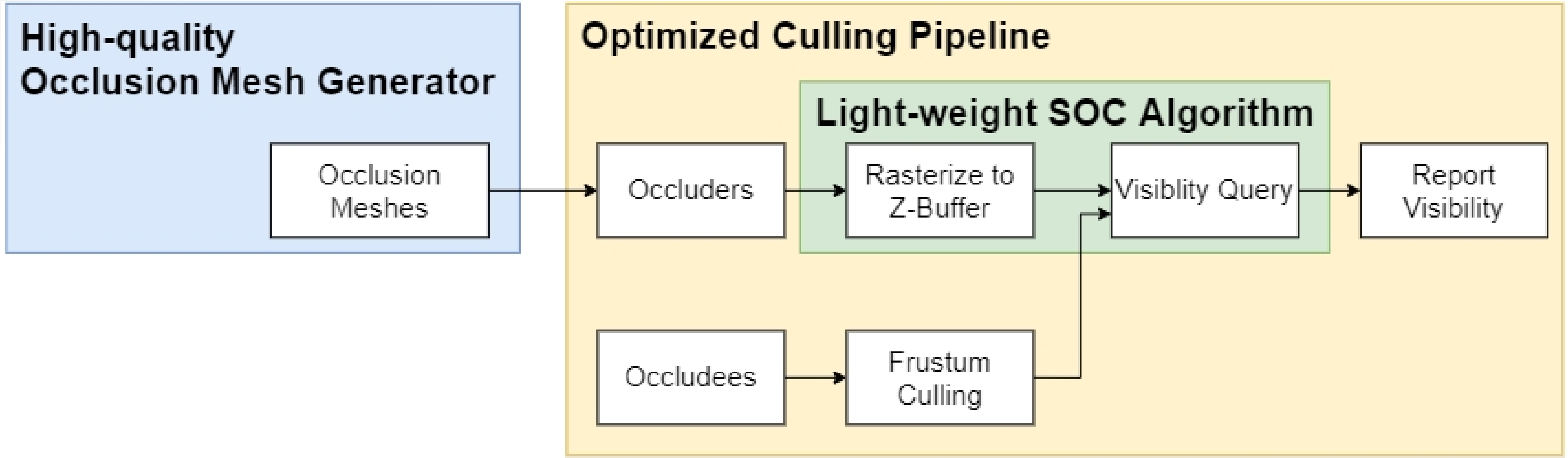
	PVS	Hardware Occlusion Query	SOC - Good Occlusion Meshes	SOC - Bad Occlusion Meshes
No False Occlusion	O	X	O	X
Dynamic Objects	X	O	O	O
Culling Rate	High	Very High	High	Low
Efficiency	Fast	Slow	Slow	Extremely Slow
Package Size	Big	0	Small	Small

Target

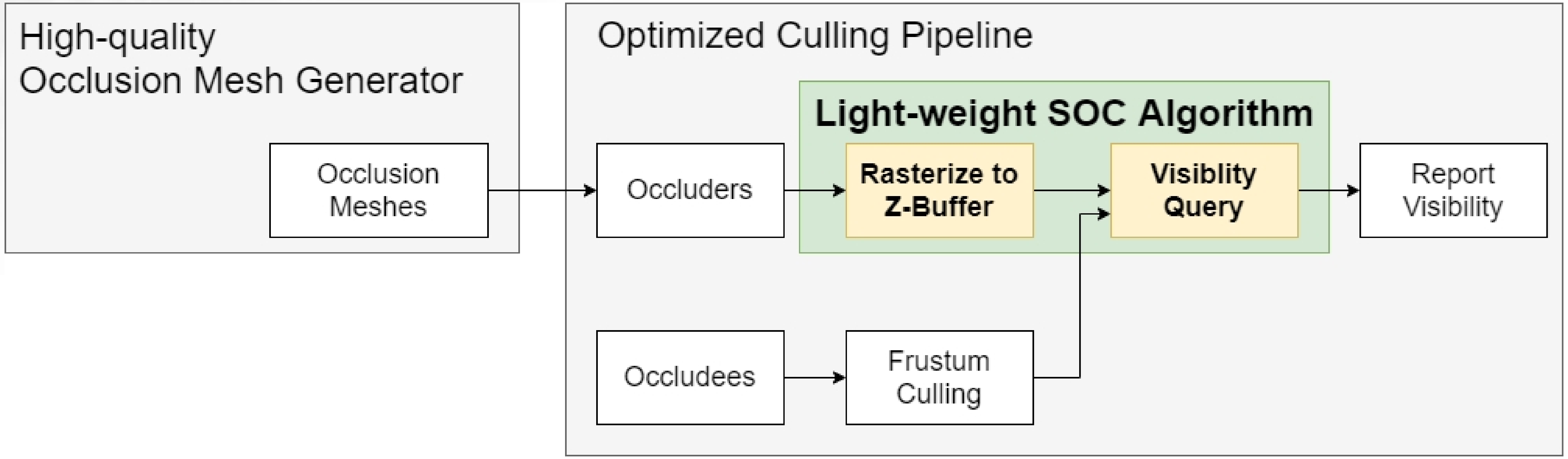
- Do SOC as fast as possible
- Optimize the algorithm for the mobile platform
- Use good occlusion meshes

Our Solution

- A complete solution consisting of 3 parts:
  - Light-weight SOC Algorithm
  - Optimized Culling Pipeline
  - Offline High-quality Occlusion Mesh Generator



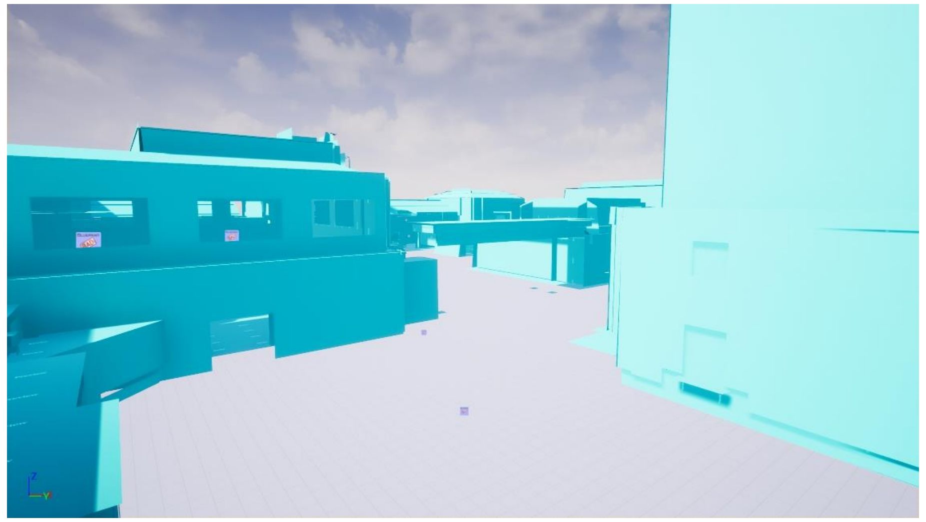
Part 2: Light-weight Software Occlusion Culling Algorithm



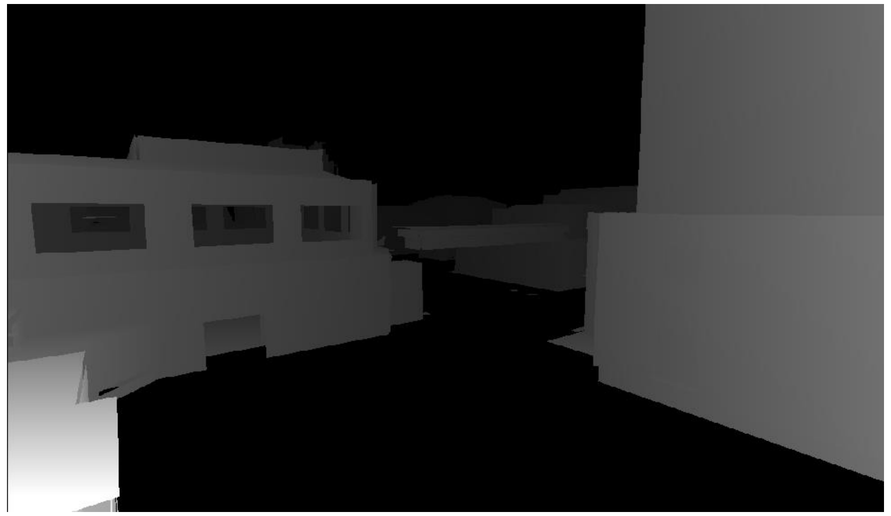
Software Occlusion Culling Algorithm

- Depth test on CPU
  - Generate Depth-Buffer
  - Use Depth-Buffer to identify visibility

Original Scene



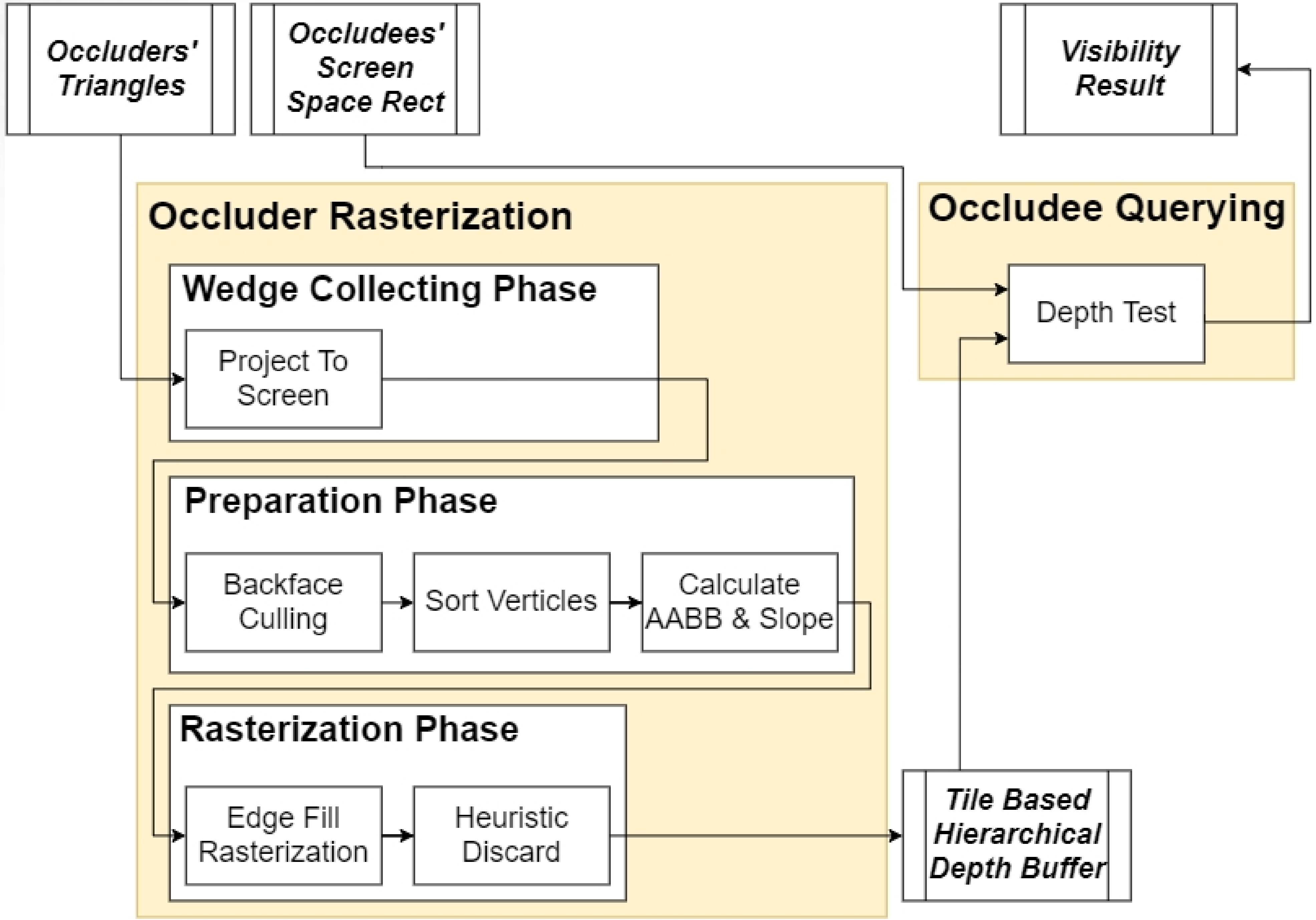
Occluders



Depth Buffer

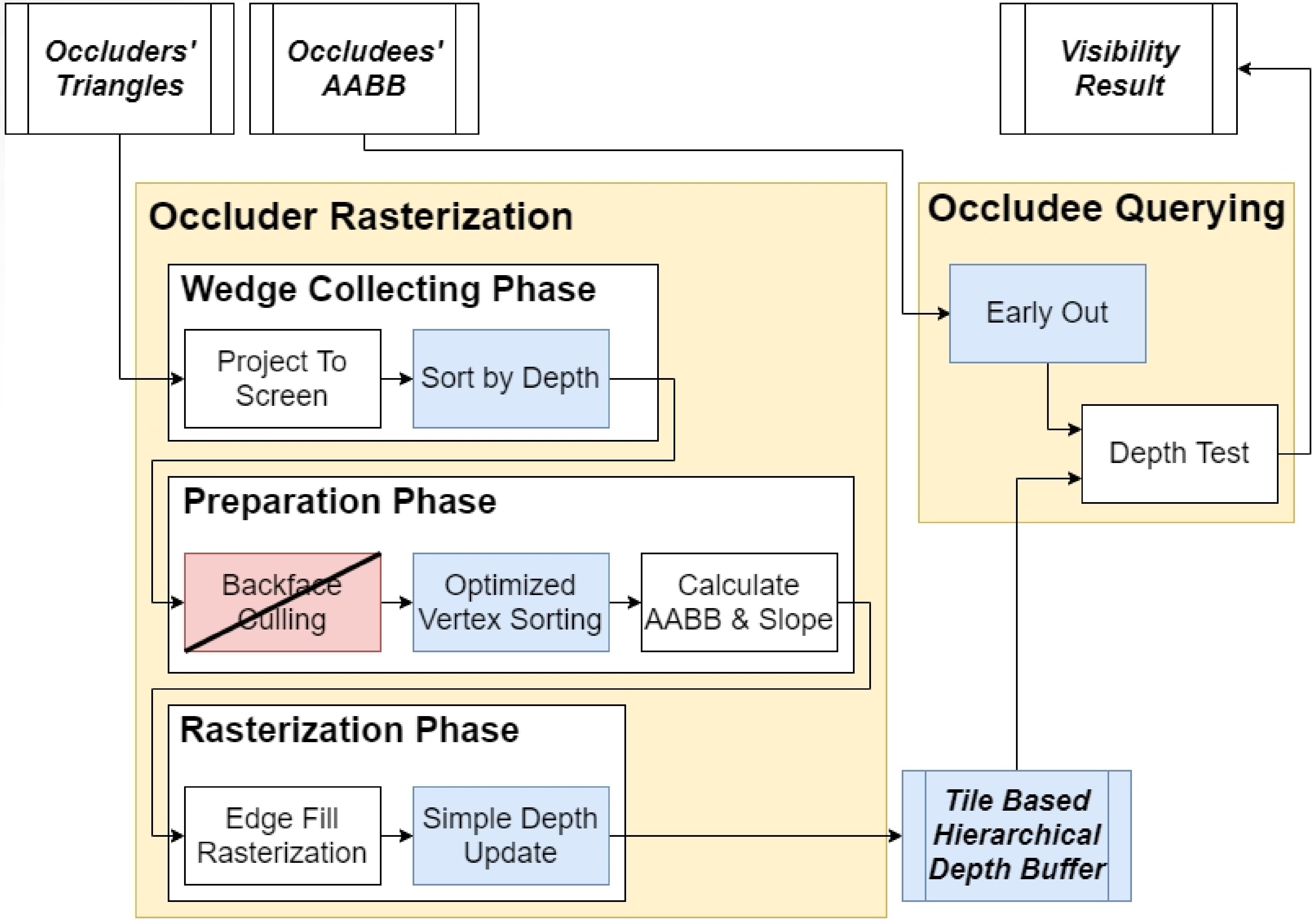
Traditional Masked Software Occlusion Culling

- SIMD: optimized for AVX
- Hierarchical depth buffer
- Edge Fill Rasterization
- Heuristic Discard



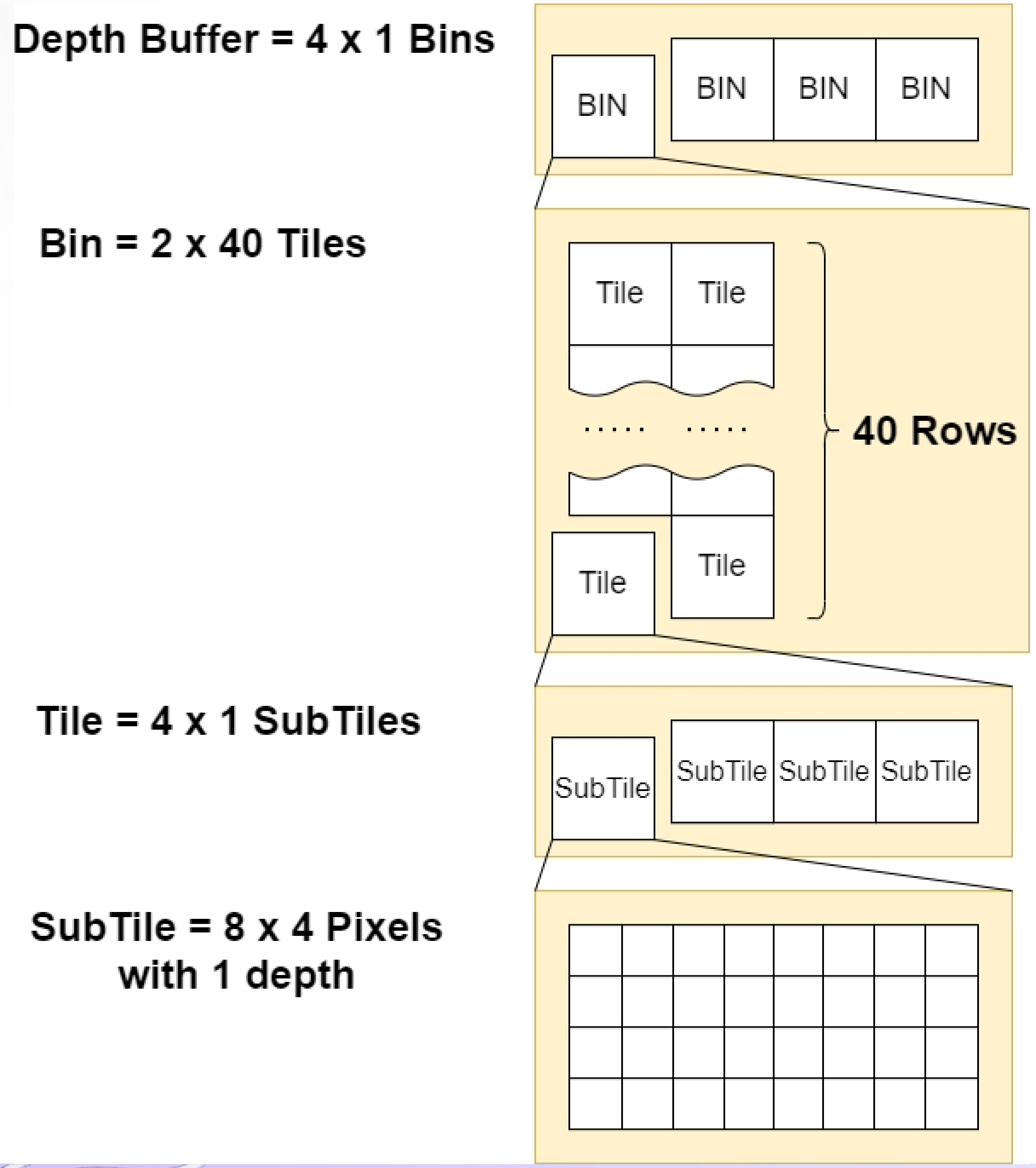
Lightweight Software Occlusion Culling

- SIMD: optimized for Neon
- Hierarchical depth buffer
- Preparation Phase optimized
- Edge Fill Rasterization
- Heuristic Discard → Sorting
- Add 'Early Out' step



Depth Buffer Structure

- Low Resolution: 256x160 Pixels
- Hierarchical Structure
  - Bin – Tile – SubTile - Pixel
- For each SubTile:
  - 8x4 pixels
  - 1 depth value
- For each Tile:
  - 128 pixels → m128i mask
  - 4 depth value → m128



Depth Value

- Relatively correct depth value
  - Reverse-Z → Precision
  - Far plane project to 0 → Convenient to memset

- Simplified projection matrix

$$M_{proj} = \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & c(>0) & 0 \end{bmatrix}$$

Wedge Collection Phase

- Project triangle to clip space
  - Record the most conservative depth
- Sort triangles
  - Sort by depth
  - Front-to-back order

Sorting vs. Heuristic Discard

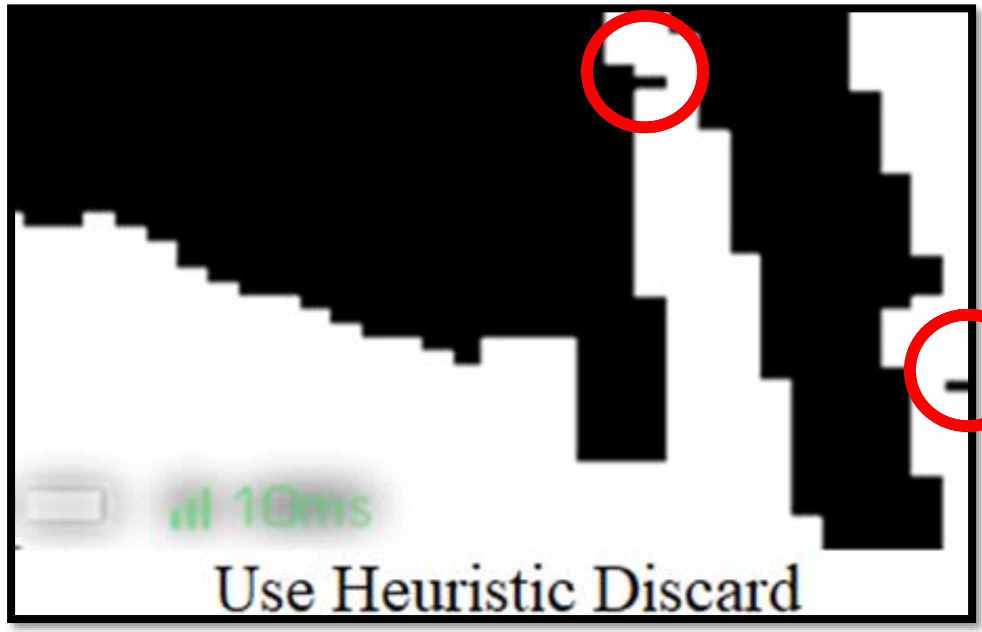
- Ensure the depth buffer is updated correctly

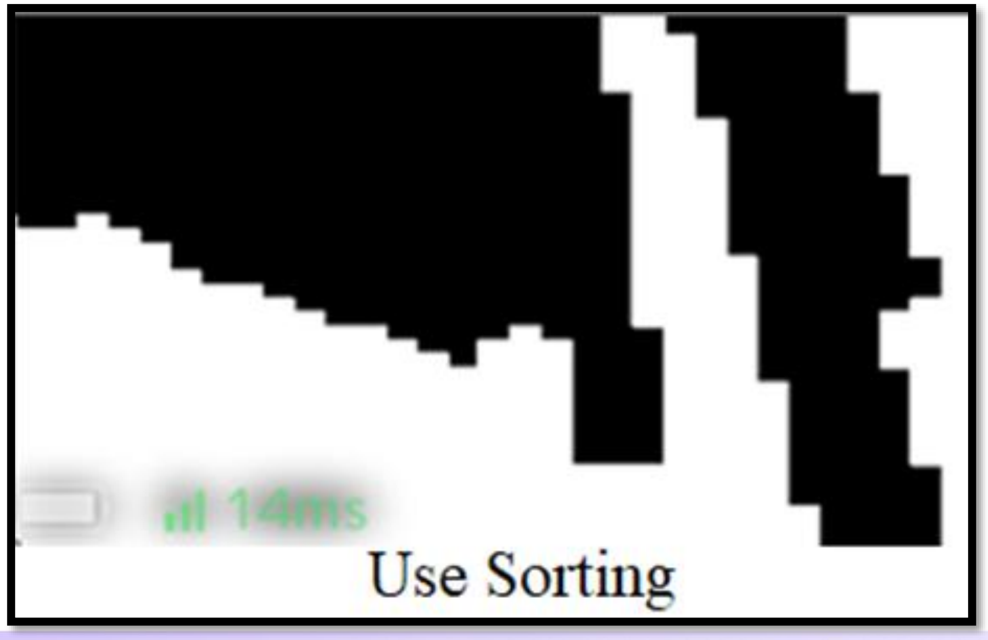
- Contrast
  - Accuracy
  - Speed

Sorting vs. Heuristic Discard

- Accuracy:
  - Sorting is better

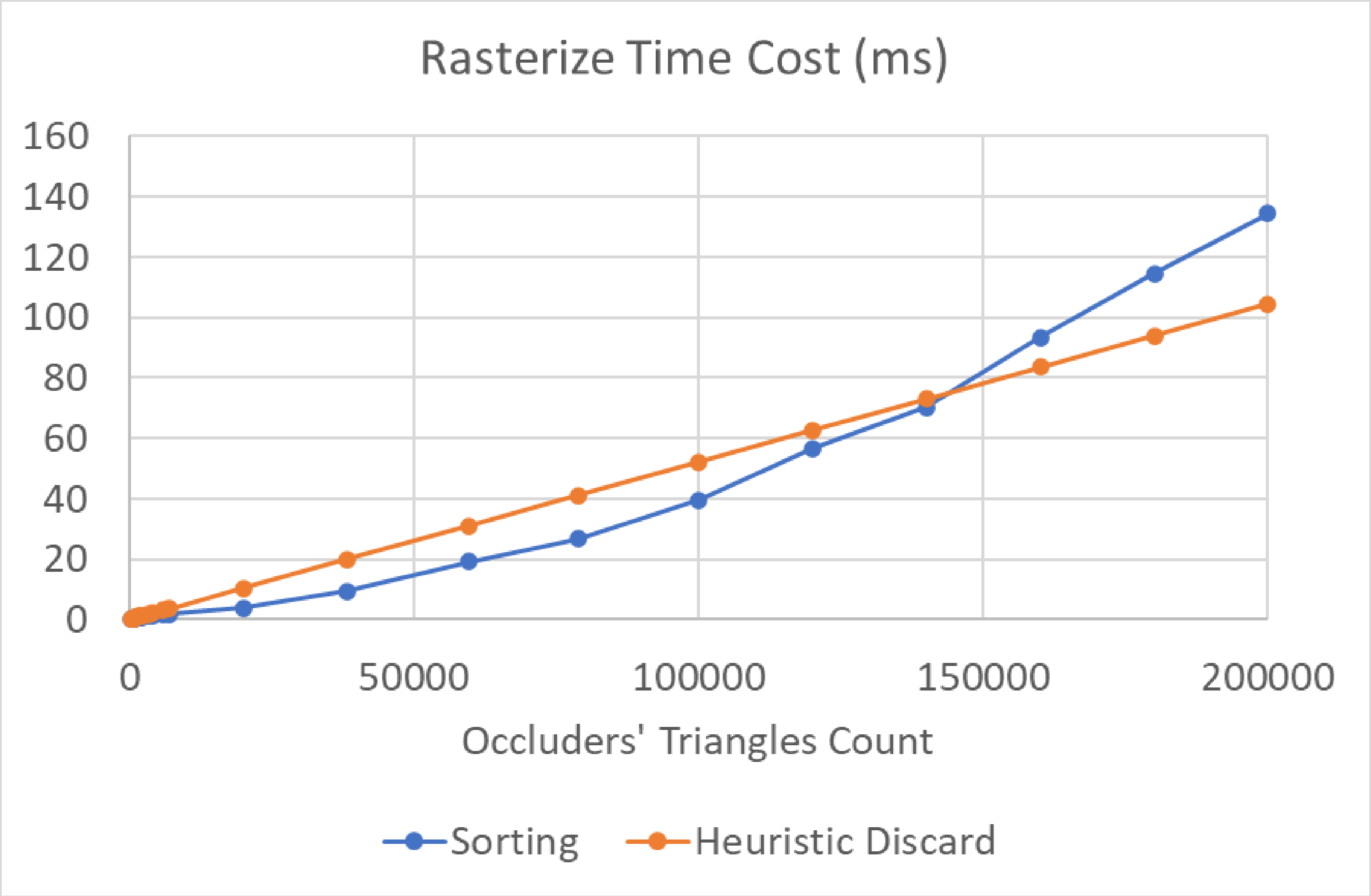
Original Scene





Sorting vs. Heuristic Discard

- Speed:
  - Depend on the occluders’ triangles count



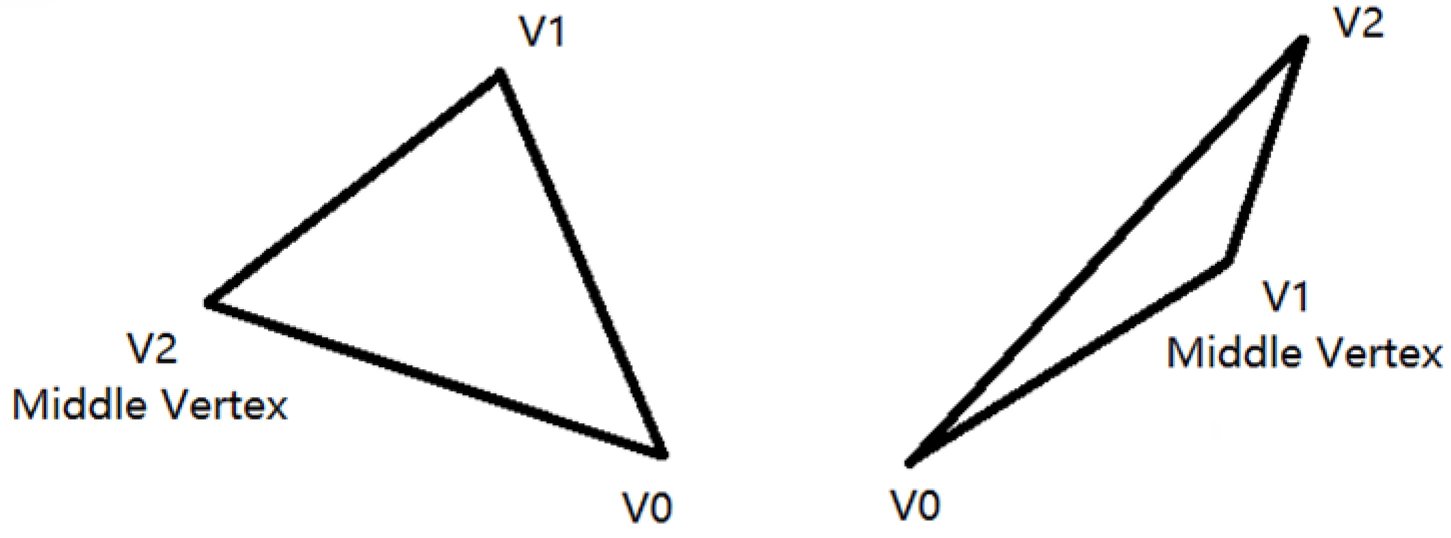
Preparation Phase

- Prepare data for ‘Edge Fill Rasterization’
- Triangles are treat as double-sided
  - No backface culling need
- Data is calculated in batches by SIMD

Vertex Sorting

- Motivation: find information of a specified edge from packed data

- Rotate triangle into specified patterns
  - Counter-clockwise
  - V0 has the smallest y
  - Identity the middle vertex in y



Specified Patterns

Vertex Sorting

- More is less!
  - Extra task: Sort into counter-clockwise
  - No need to preserve the original vertex order

- Accomplish in 22 SIMD Instructions

Vertex Sorting

```
const VectorRegister p0y_p1y_compare = VectorSubtract(vertex_y[0], vertex_y[1]);
const VectorRegister temp_pos1_x = VectorBlend(vertex_x[0], vertex_x[1], p0y_p1y_compare);
const VectorRegister temp_pos1_y = VectorBlend(vertex_y[0], vertex_y[1], p0y_p1y_compare);
vertex_x[0] = VectorBlend(vertex_x[1], vertex_x[0], p0y_p1y_compare);
vertex_y[0] = VectorBlend(vertex_y[1], vertex_y[0], p0y_p1y_compare);

const VectorRegister p0y_p2y_compare = VectorSubtract(vertex_y[0], vertex_y[2]);
const VectorRegister temp_pos2_x = VectorBlend(vertex_x[0], vertex_x[2], p0y_p2y_compare);
const VectorRegister temp_pos2_y = VectorBlend(vertex_y[0], vertex_y[2], p0y_p2y_compare);
vertex_x[0] = VectorBlend(vertex_x[2], vertex_x[0], p0y_p2y_compare);
vertex_y[0] = VectorBlend(vertex_y[2], vertex_y[0], p0y_p2y_compare);
```

1. Find V_0 By Comparison

```
const VectorRegister p1x_sub_p0x = VectorSubtract(temp_pos1_x, vertex_x[0]);
const VectorRegister p1y_sub_p0y = VectorSubtract(temp_pos1_y, vertex_y[0]);
const VectorRegister p2x_sub_p0x = VectorSubtract(temp_pos2_x, vertex_x[0]);
const VectorRegister p2y_sub_p0y = VectorSubtract(temp_pos2_y, vertex_y[0]);
const VectorRegister cross = VectorSubtract(VectorMultiply(p2x_sub_p0x, p1y_sub_p0y), VectorMultiply(p1x_sub_p0x, p2y_sub_p0y));
```

2. Cross product to judge $\angle V_0$

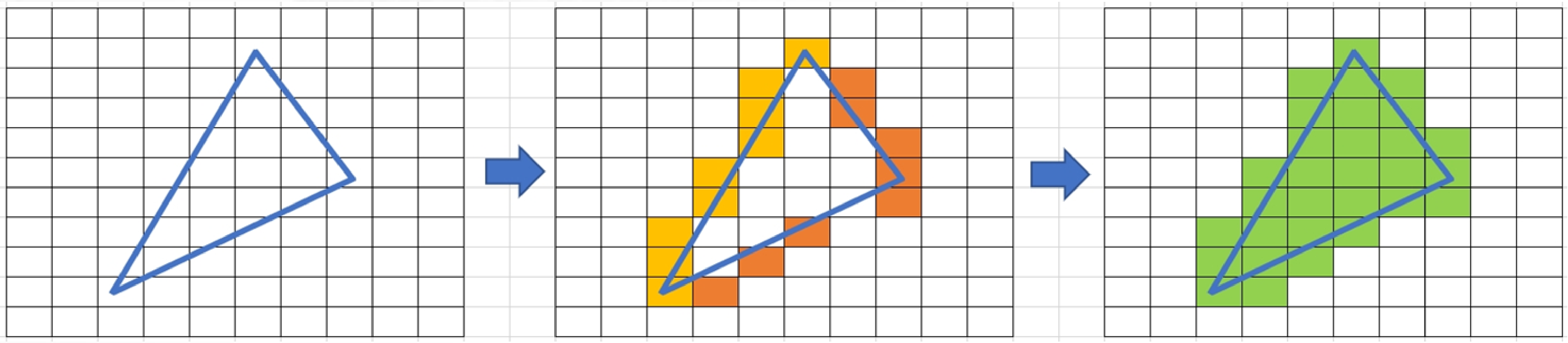
```
vertex_x[2] = VectorBlend(temp_pos1_x, temp_pos2_x, cross);
vertex_y[2] = VectorBlend(temp_pos1_y, temp_pos2_y, cross);
vertex_x[1] = VectorBlend(temp_pos2_x, temp_pos1_x, cross);
vertex_y[1] = VectorBlend(temp_pos2_y, temp_pos1_y, cross);

mid_vertex_mask = ~VectorMaskBits(VectorSubtract(vertex_y[2], vertex_y[1]));
```

3. Arrange V_1 and V_2 in counter-clockwise

Rasterization Phase

- Coverage detection: Edge Fill Rasterization, same as MSOC



1. Find the leftmost and Rightmost pixel each row
2. Fill all the pixels between

- Depth updating: greatly simplified
  - Record current depth when a sub-tile is fully covered

Occludee Visibility Query

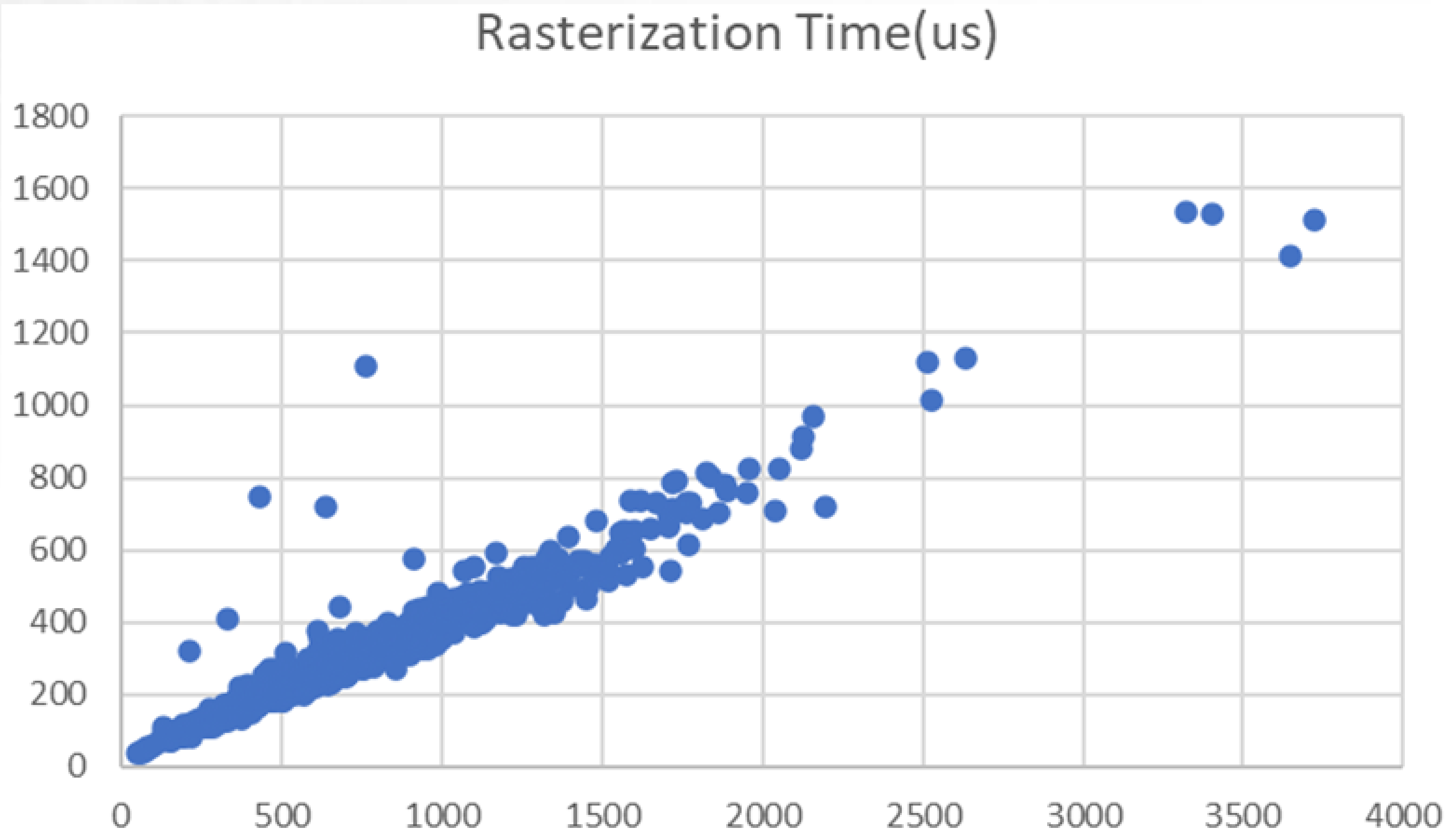
- Project occludee’s AABB to screen space
- Early Out: Fast Pass & Fast Fail
- Depth test

Early Out

- Too small after projection → invisible
- Close to camera → visible
- 50% occludees can be skipped

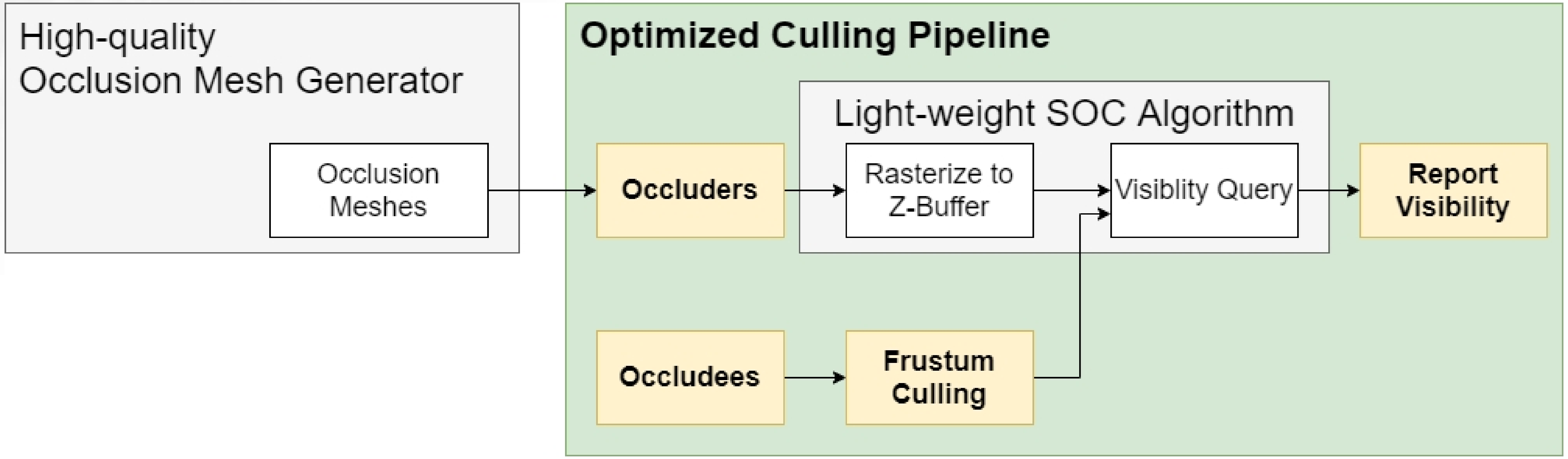
Performance of Algorithm

- Rasterization: 0.40us per triangle
- Query Time: 0.56us per occludee
- Early-out Rate: 49.3%



Occludee Count	Pass 'Early Out' Count	Query Time (us)
199	169	158
341	352	283
522	421	358
639	632	461
821	476	439
1203	695	567
1420	850	688
1814	977	958
2184	933	841
2500	1021	1148

Part 3: Optimized Culling Pipeline



Typical Culling Pipeline

- Sample Code:

```
void TypicalCullingPipeline(OccluderAggregation& occluders, OccludeeAggregation& occludees) {
    for (auto occluder : occluders) {
        RasterizeOccluder(occluder);
    }

    for (auto itor = occludees.begin(); itor != occludees.end(); itor++) {
        Occludee* OccludeePtr = *itor;

        if (OccludeePtr->IsForceInvisible())
            continue;
        if (!FrustumCulling(OccludeePtr))
            continue;
        if (!SoftwareOcclusionQuery(OccludeePtr))
            continue;

        OccludeePtr->ReportVisible();
    }
}
```

1. Organize Occluders & Occludees

2. Arrange Culling Modules

3. Report Result

Problems in Typical Culling Pipeline

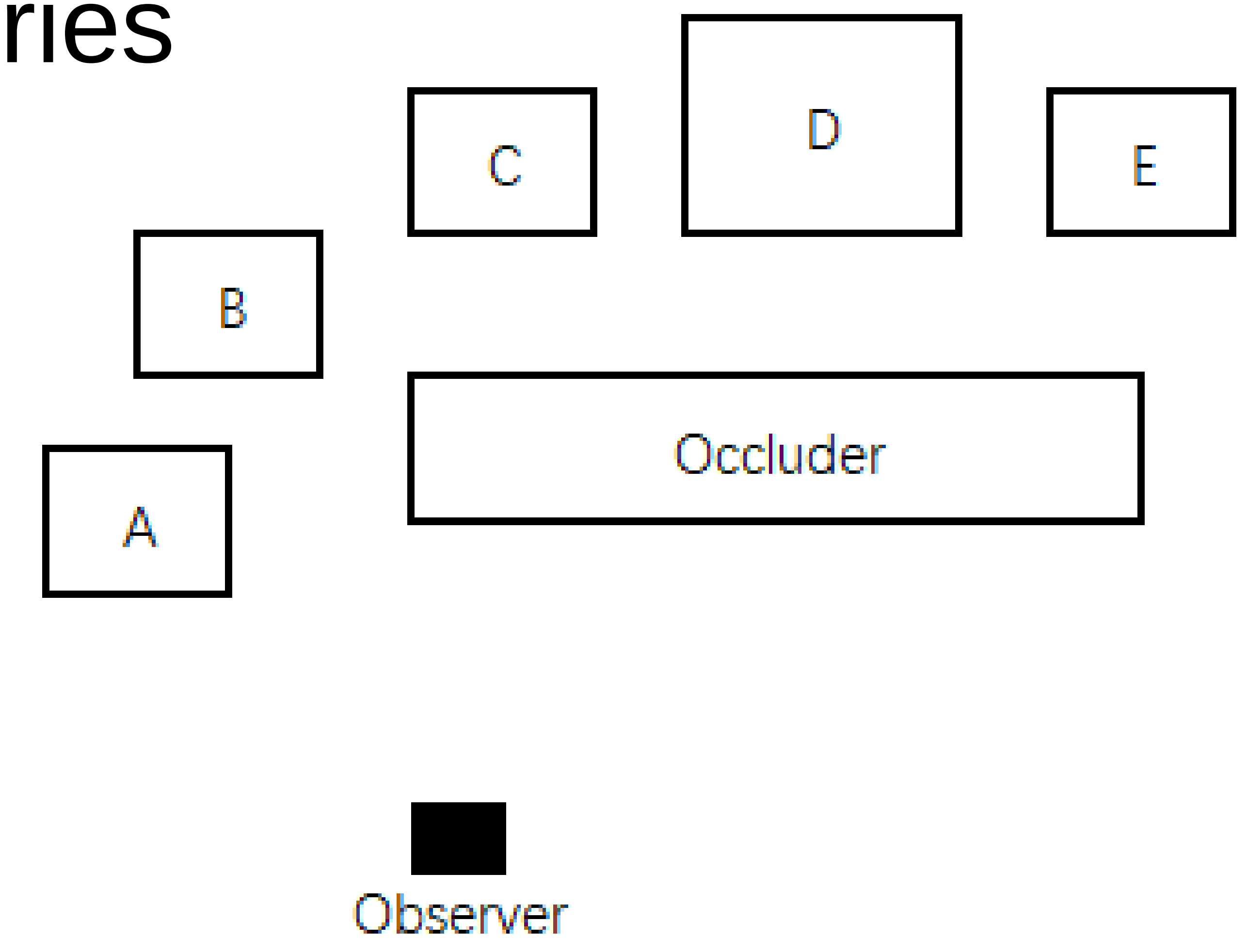
- Performance Issue: 1.5ms for 5000 occludees

- TODO List
  - Reduce cache miss rate in:
    - Data traversal
    - Function call
    - Visibility filter
  - Don’t rasterize triangles with low occlusion power
  - Use multi-threading

Data Organization

- Normal Array
  - Data structure: [A, B, C, D, E]
  - Process: 2 Cache Fetches, 5 Queries

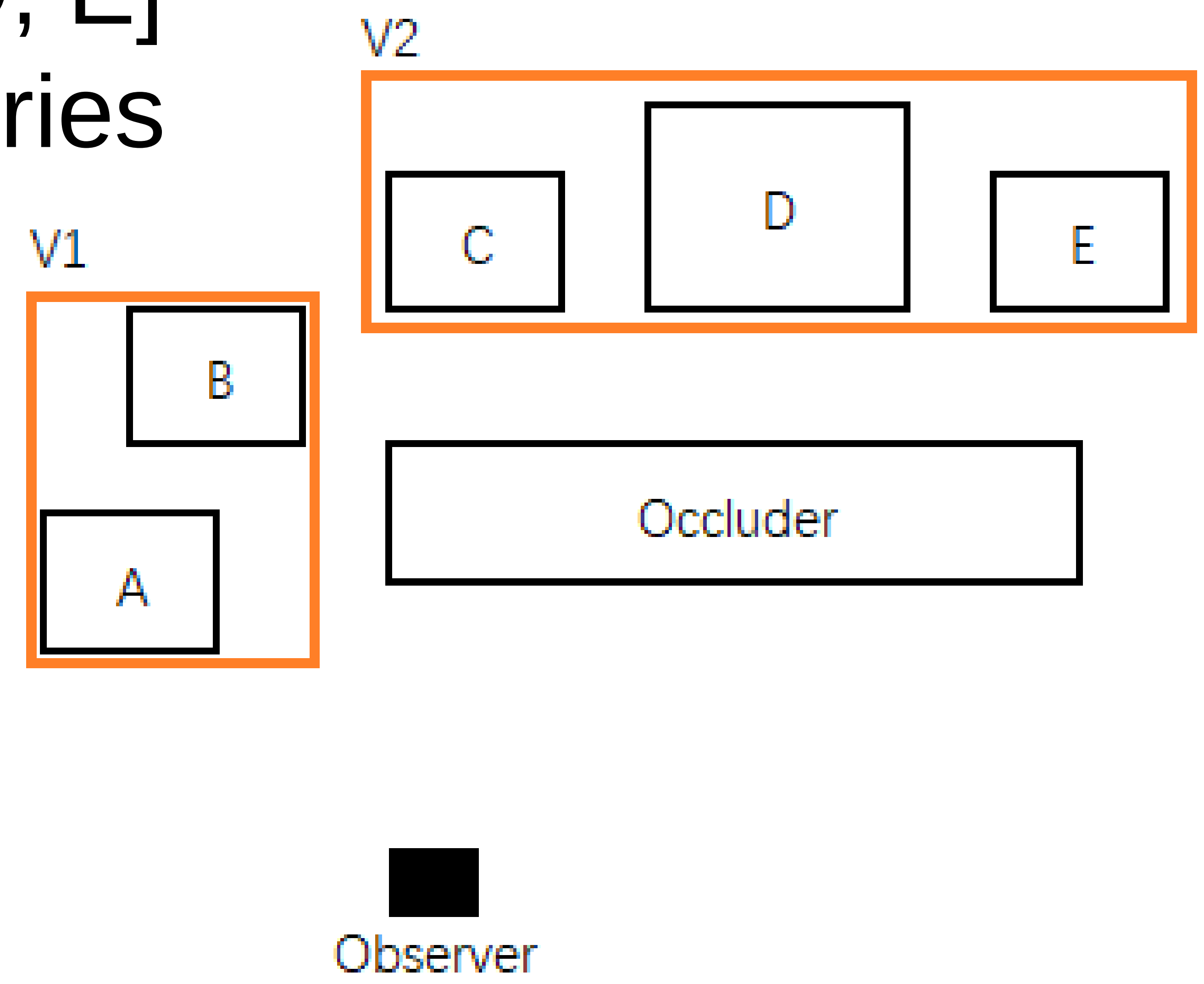
Action	Data
Cache Fetch	A, B, C
Visibility Query	A
Visibility Query	B
Visibility Query	C
Cache Fetch	D, E
Visibility Query	D
Visibility Query	E



Data Organization

- Spatial Acceleration Structures
  - Data structure: [V1, V2, A, B, C, D, E]
  - Process: 3 Cache Fetches, 4 Queries

Action	Data
Cache Fetch	V1, V2, A
Visibility Query	V1
Visibility Query	A
Cache Fetch	B, C, D
Visibility Query	B
Cache Fetch	V1, V2, A
Visibility Query	V2



Data Organization - Conclusion

- Using arrays is better here
  - Visibility queries are very cheap
  - Count of occludees is not large. (about 5k~10k)
  - Almost no cost to maintain the structure

- Use compact data structure
  - Relevant data only
  - Lowest possible precision
  - Use Bit-Field

TODO List

- ☐ Reduce cache miss rate in:
  - ✓ Data traversal: 0.3ms saved
  - ☐ Function call
  - ☐ Visibility filter
- ☐ Don't rasterize triangles with low occlusion power
- ☐ Use multi-threading

Virtual Function Call

- 0.5ms wasted on Virtual Function Call

- More Cache Misses

Normal Function	Virtual Function
Load Function Instruction	Load Vtable
	Load *(vtable+offset)
	Load Function Instruction

- Inefficient L1i-Cache Prefetching
  - The instruction depends on the dynamic type
  - Hard to hide the cache miss costs

Virtual Function Call

- Eliminate all virtual functions in the culling pipeline!

- Modification suggestions:
  - Separate data from its operations
  - Use different arrays to store different types of occludees
  - Call their specified notification functions

TODO List

- ☐ Reduce cache miss rate in:
  - ✓ Data traversal: 0.3ms saved
  - ✓ Function call: 0.5ms saved
  - ☐ Visibility filter
- ☐ Don’t rasterize triangles with low occlusion power
- ☐ Use multi-threading

Cascade of Multiple Visibility Filter

- Occludee need to pass multiple check
  - Logic Setting
  - Frustum Culling
  - Software Occlusion Culling

- Implementation choices:
  - Use a flag to mark visible occludees
  - Use a special array to carry filtered visible occludees

Cascade of Multiple Visibility Filter

- All objects with visible flag: low cache utilization

- Filtered visible objects: data copy overhead

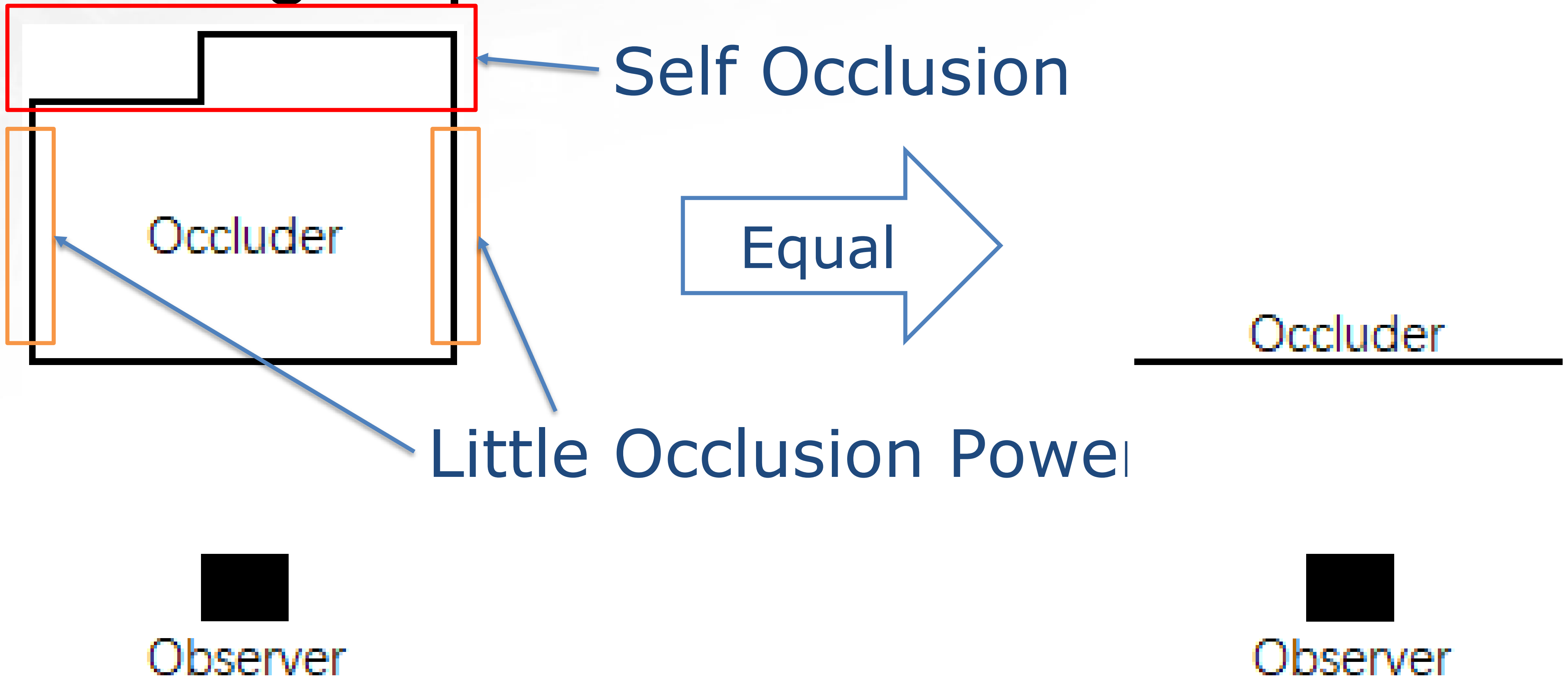
- Conclusion:
  - In ‘Life After’, 80% objects are filtered out
  - ‘Filtered visible objects’ is 0.05ms faster

TODO List

- ✓ Reduce cache miss rate in:
  - ✓ Data traversal: 0.3ms saved
  - ✓ Function call: 0.5ms saved
  - ✓ Visibility filter: 0.05ms saved
- ☐ Don’t rasterize triangles with low occlusion power
- ☐ Use multi-threading

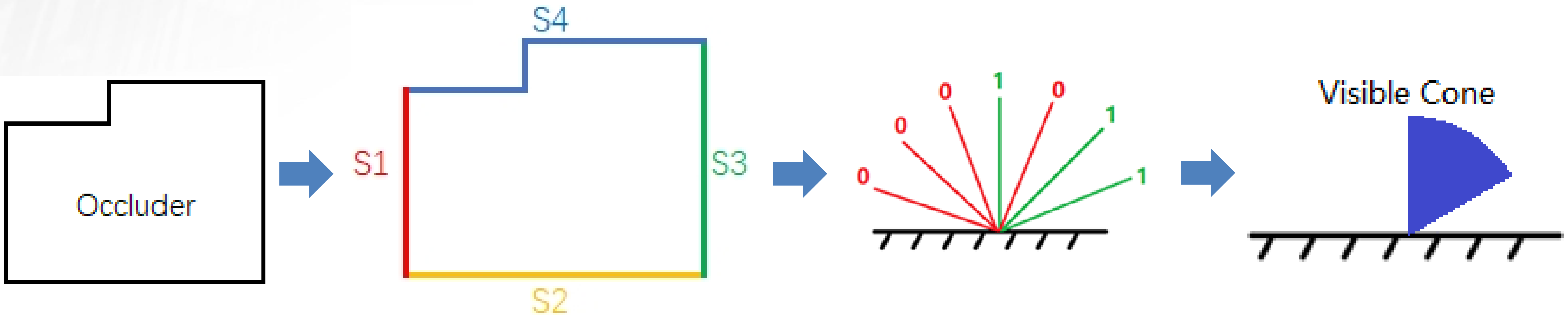
Visible Section

- Motivation
  - Self-Occlusion
  - Triangles parallel to view have little occlusion power



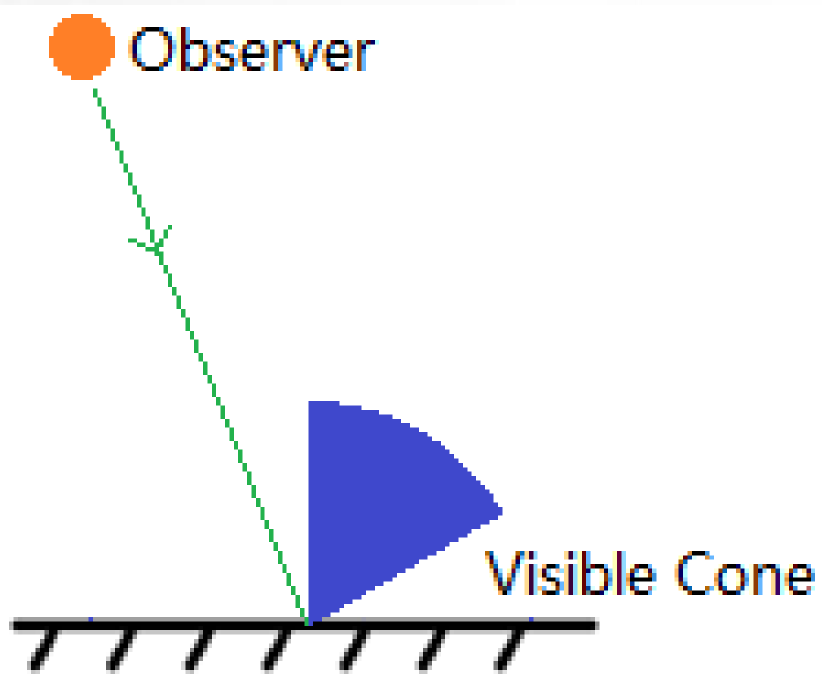
Visible Section

- Bake visible section info
  - Split the mesh into sections
  - For each section
    - Calculate visible size at different viewing angles
    - Get ‘Visible Cone’

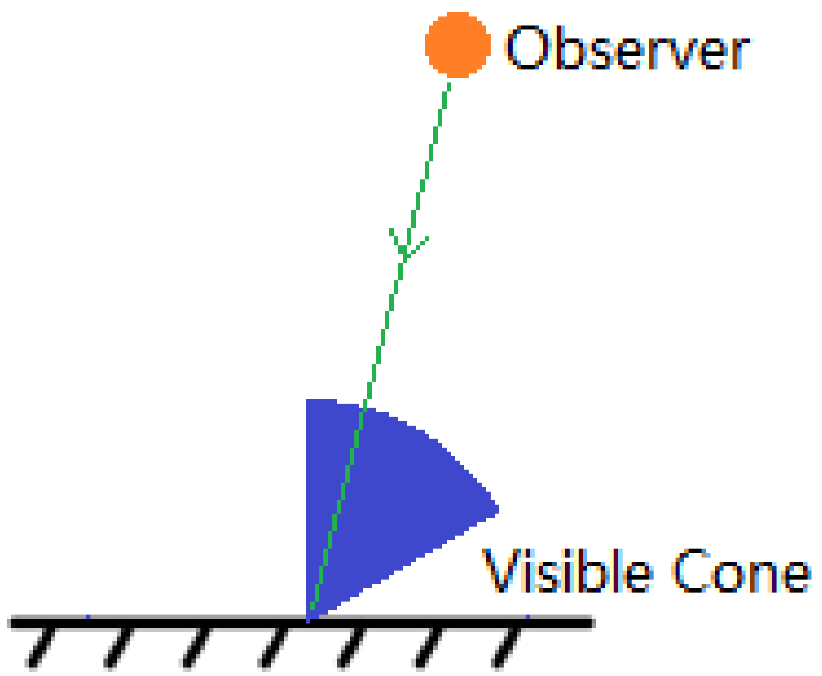


Visible Section

- Runtime identification
  - Check the view angle if it's in the visible cone
  - Only the visible sections are rasterized



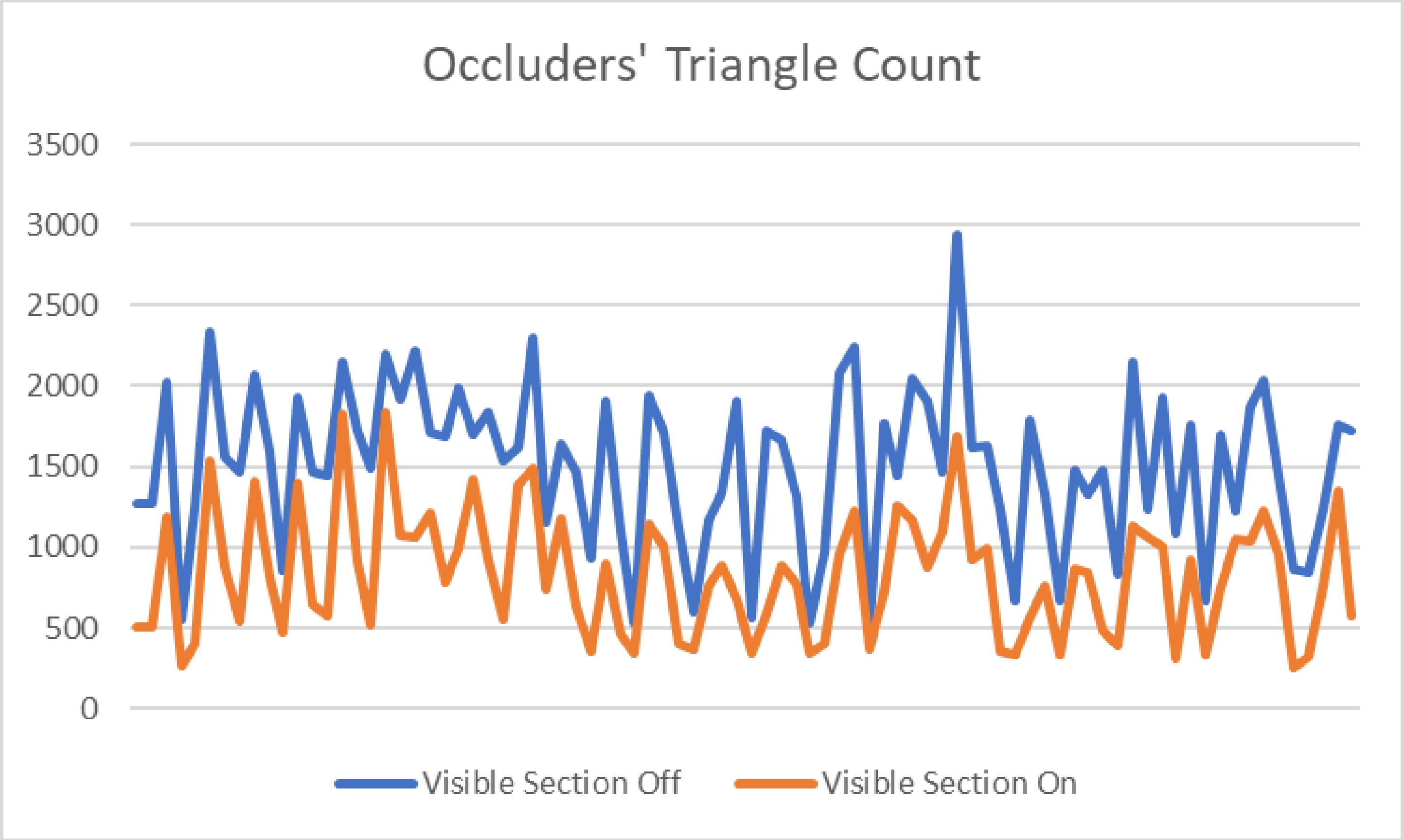
Outside the Cone: Invisible

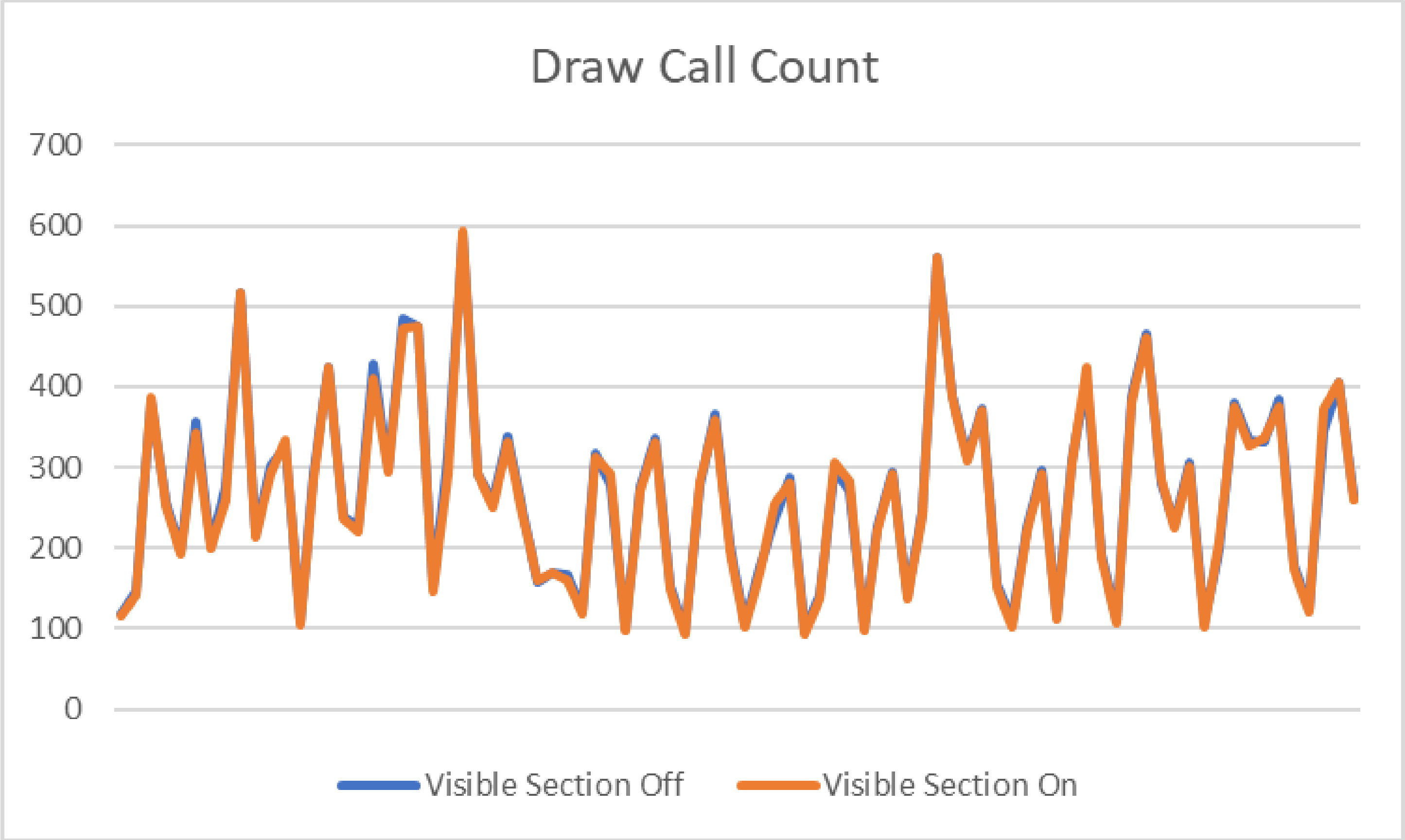


Inside the Cone: Visible

Visible section

- Triangles to rasterize: 1539 → 874
- Draw Call: 290 → 292





TODO List

- ✓ Reduce cache miss rate in:
  - ✓ Data Traversal: 0.3ms saved
  - ✓ Function Call: 0.5ms saved
  - ✓ Visibility Filter: 0.05ms saved
- ✓ Don't rasterize triangles with low occlusion power: rasterization task halved
- ☐ Use multi-threading

Multi Threading

- ‘big.LITTLE architecture’ on mobile platform
  - Big Cores: Powerful, but power-hungry
  - Little Cores: Bettery-saving, but much slower
  - Overhead of scheduling

- Task size is critical!
  - Big enough to offset scheduling overhead
  - Much smaller than the task on big cores

- Our Choice
  - ‘visible section picking’ + ‘wedge collecting stage’

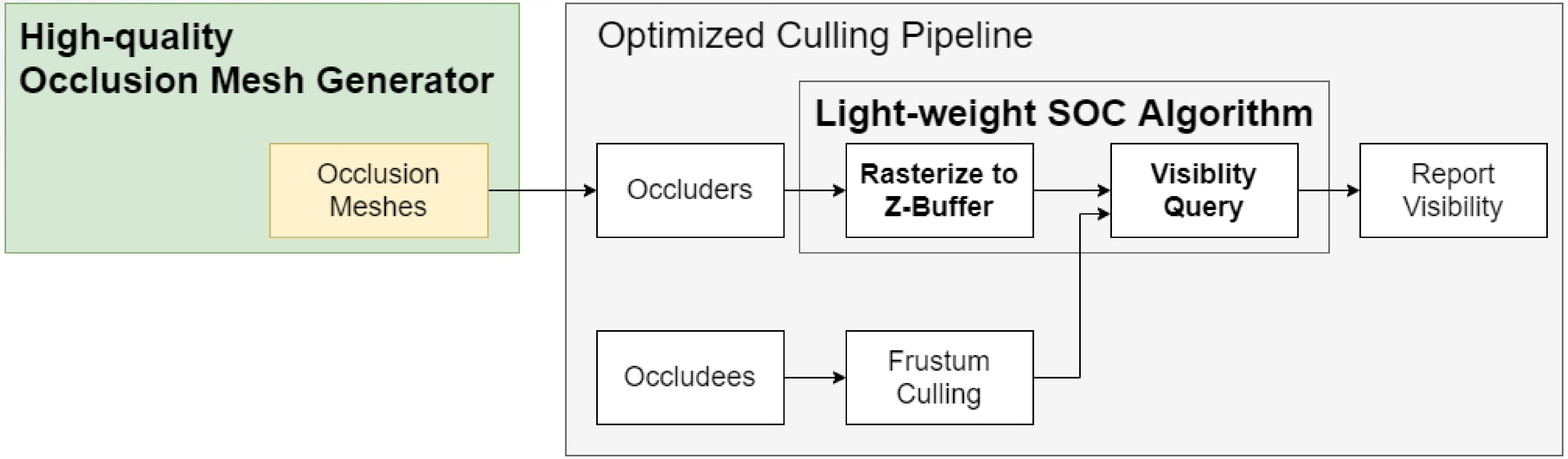
TODO List

- ✓ Reduce cache miss rate in:
  - ✓ Data traversal: 0.3ms saved
  - ✓ Function call: 0.5ms saved
  - ✓ Visibility filter: 0.05ms saved
- ✓ Don't rasterize triangles with low occlusion power: rasterization task halved
- ✓ Use multi-threading: 0.25ms saved

Performance of pipeline

	Pipeline Cost Before (us)	Pipeline Cost After (us)
OPPO R9s	2736.61	763.56
HUAWEI mate8	2985.1	741.82
iPhone 6s	1447.35	412.22
iPhone 7 Plus	799.05	179.04
Samsung S10	1169.75	300.59
iPhone XS	584.54	159.78

Part 4: High-quality Occlusion Mesh Generator

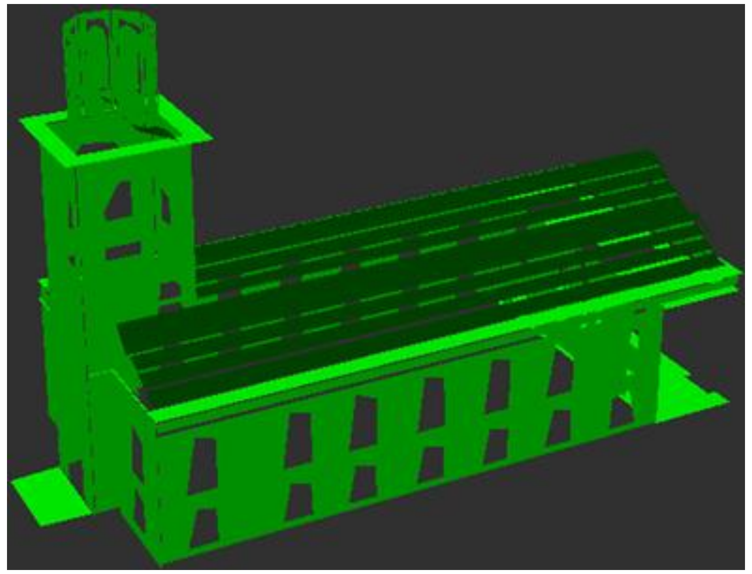


Importance of Occlusion Meshes

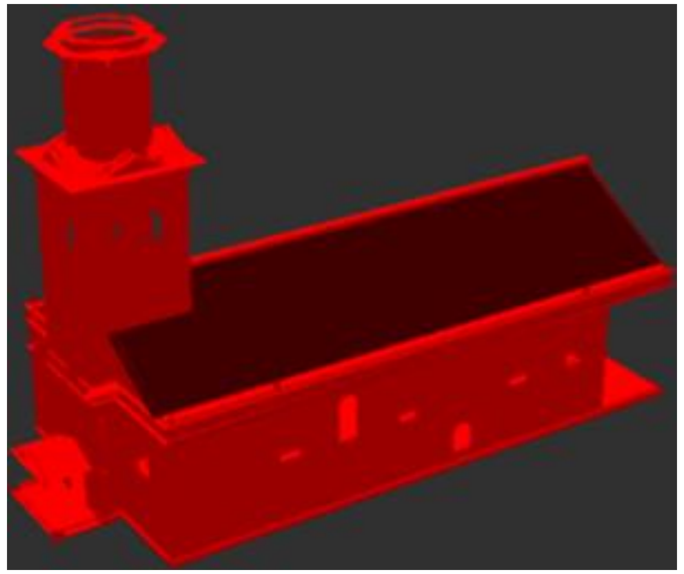
- Budget: 4000 triangles for occluders in total
- Must be high quality

	Medium Quality	High Quality
Culling Rate	35%	65%
False Occlusion	Occasional	No

Original Occluder

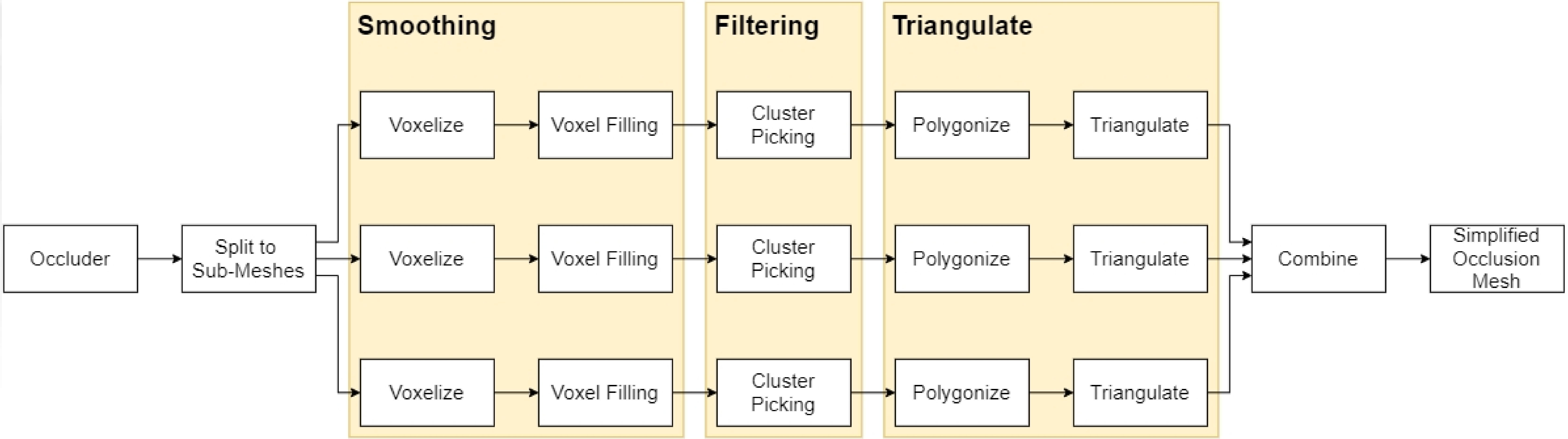


Medium Quality



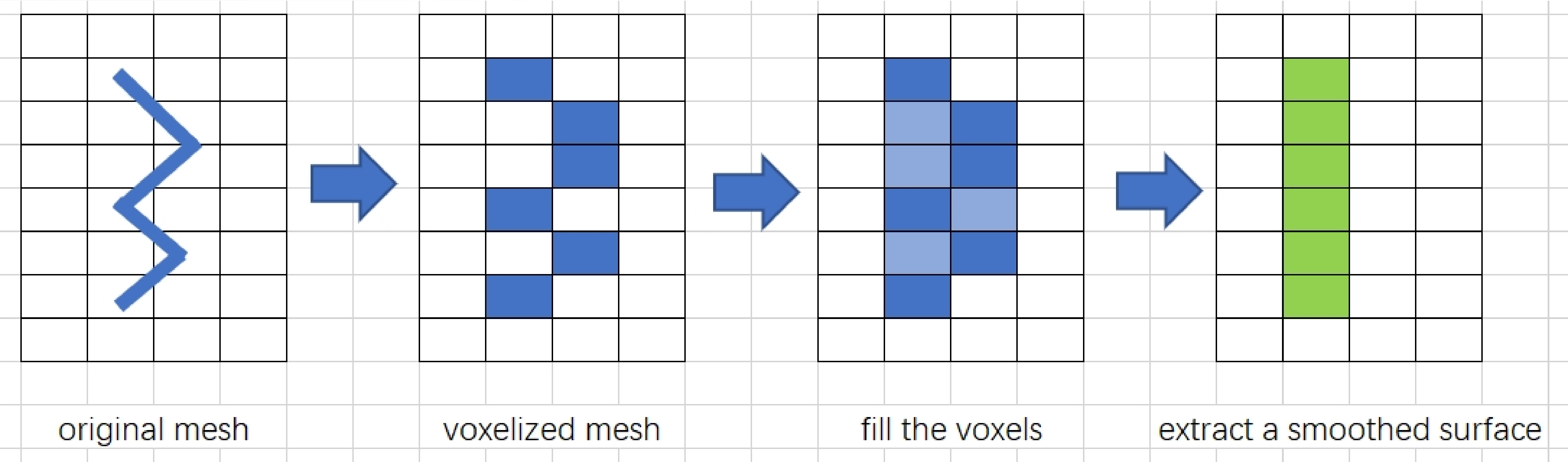
High Quality

Overview of Our Generator



Smooth a 3D Mesh

- Voxelize the mesh
- Fill the voxelized mesh as much as possible



Smooth a 3D Mesh

- The constraint when filling voxels: No False Occlusion

- Mathematical description for 'No False Occlusion':
  - Visibility Function:

$$VF(viewPos, viewDir, occludee) = \begin{cases} False, & At\ viewPos, look\ along\ viewDir, occludee\ is\ invisible \\ True, & At\ viewPos, look\ along\ viewDir, occludee\ is\ visible \end{cases}$$

  - No False Occlusion:

$$\forall occludee, stand\ at\ \forall viewPos, look\ along\ \forall viewDir$$

$$VF_{New\ Occlusion\ Mesh}(viewPos, viewDir, occludee) = False$$

$$\Rightarrow VF_{Original\ Occluder}(viewPos, viewDir, occludee) = False$$

Simplify the Constraint

- $For\ \forall occludee, stand\ at\ \forall \cancel{viewPos}, look\ along\ \forall viewDir$

$$VF_{New}(viewPos, viewDir, occludee) = False \Rightarrow VF_{Original}(viewPos, viewDir, occludee) = False$$

Simplify the Constraint

- $For\ \forall occludee, stand\ at\ \boldsymbol{Near\&Far}, look\ along\ \cancel{\forall viewDir}$

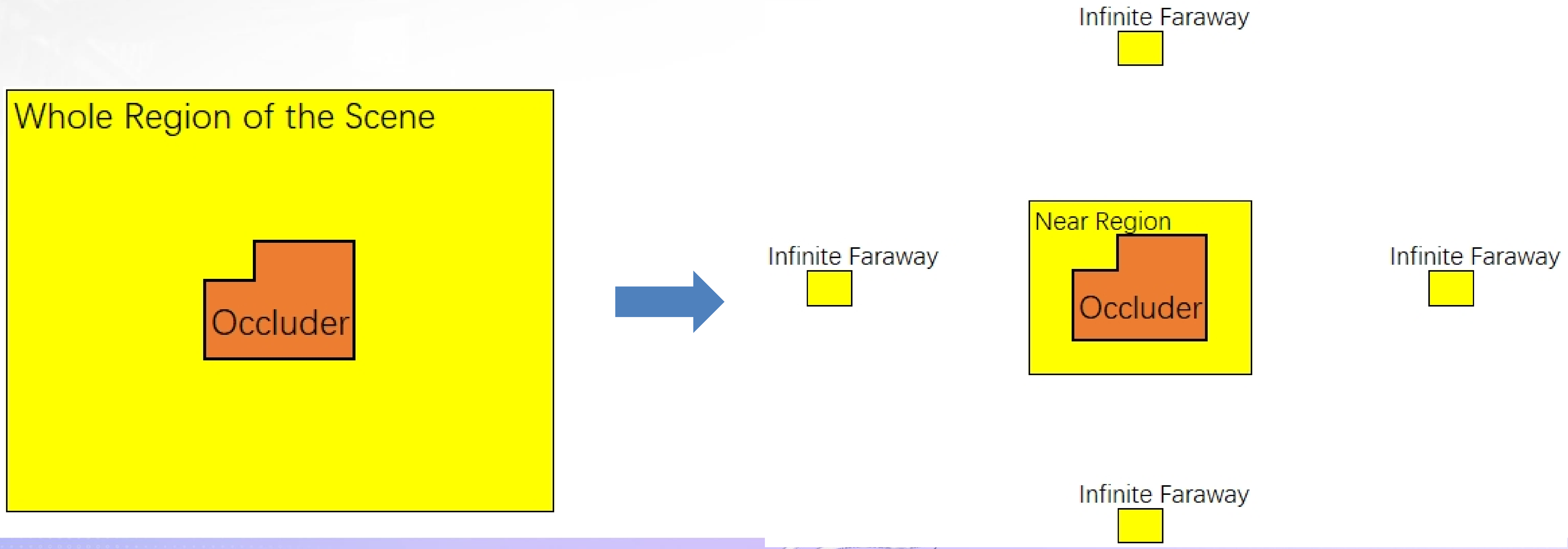
$$VF_{New}(viewPos, viewDir, occludee) = False \Rightarrow VF_{Original}(viewPos, viewDir, occludee) = False$$

- Near: Limited Range
- Far: Infinite faraway

Simplify the Constraint

- $For\ \forall occludee, stand\ at\ \boldsymbol{Near\&Far}, look\ along\ \cancel{\forall viewDir}$

$$VF_{New}(viewPos, viewDir, occludee) = False \Rightarrow VF_{Original}(viewPos, viewDir, occludee) = False$$



Simplify the Constraint

- $For\ \cancel{\forall occludee}, stand\ at\ Near\&Far, look\ along\ \boldsymbol{Specified\ Dirs}$

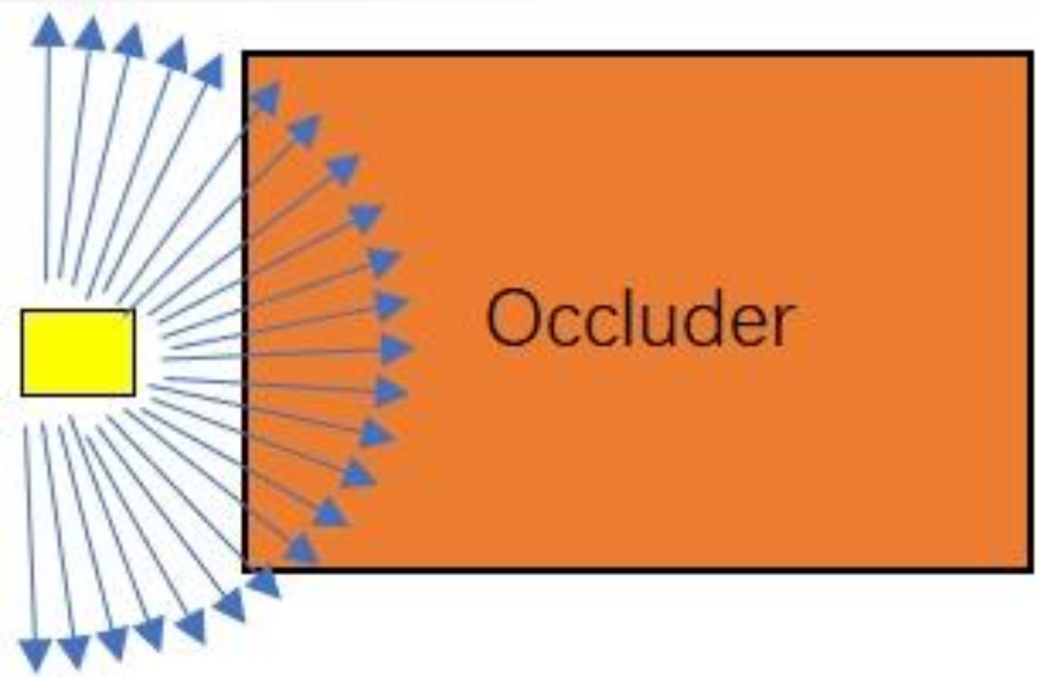
$$VF_{New}(viewPos, viewDir, occludee) = False \Rightarrow VF_{Original}(viewPos, viewDir, occludee) = False$$

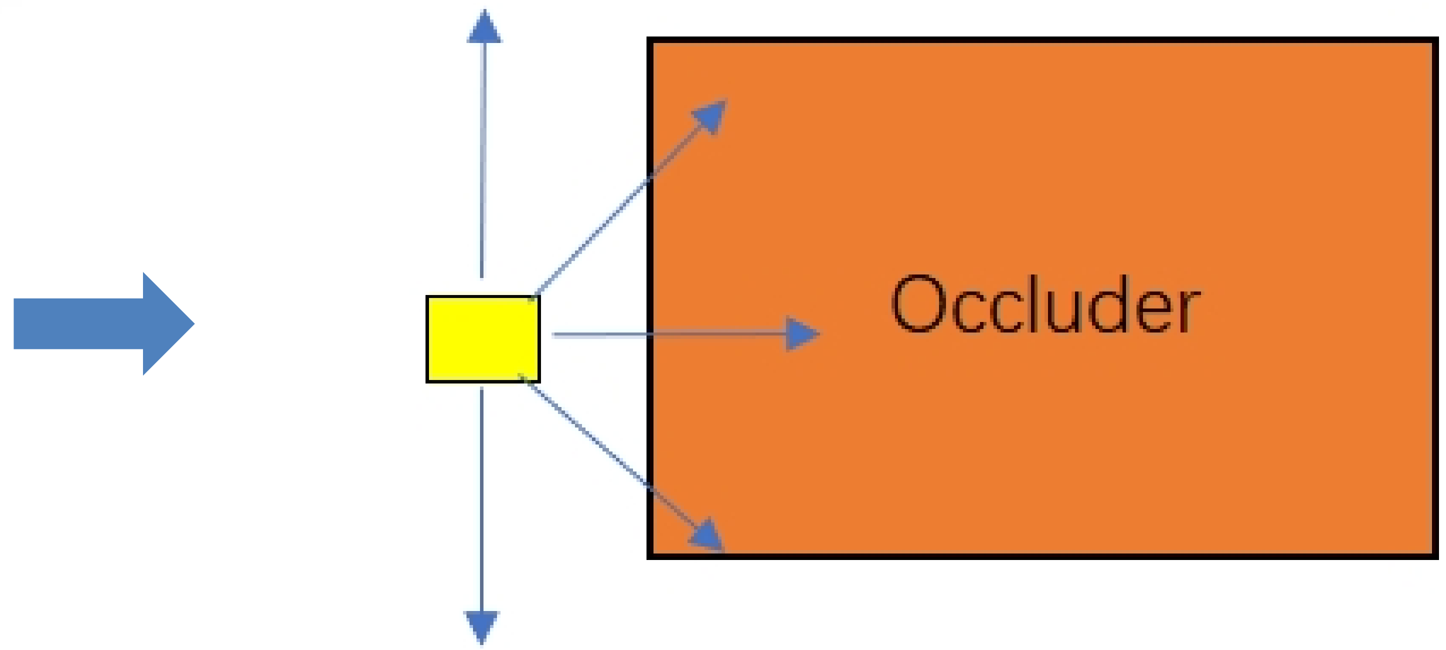
- Chosen viewing directions
  - 3 axes
  - Midline of any 2 axes

Simplify the Constraint

- $For\ \cancel{\forall occludee}, stand\ at\ Near\&Far, look\ along\ \textbf{Specified Dirs}$

$$VF_{New}(viewPos, viewDir, occludee) = False \Rightarrow VF_{Original}(viewPos, viewDir, occludee) = False$$





Simplify the Constraint

- $For\ \mathbf{Important\ Voxels},\ stand\ at\ Near\&Far,\ look\ along\ \mathbf{Specified\ Dirs}$

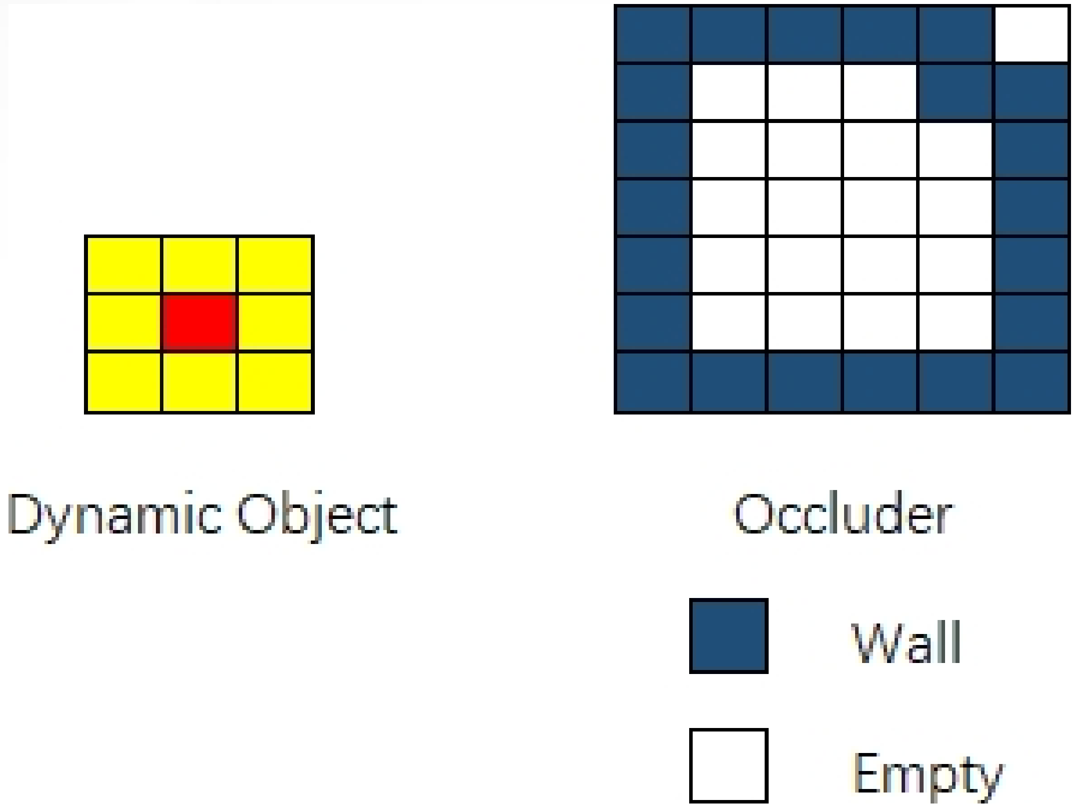
$$VF_{New}(viewPos, viewDir, occludee) = False \Rightarrow VF_{Original}(viewPos, viewDir, occludee) = False$$

- Important Voxels
  - Static occludees already exist
  - The central part of dynamic objects may appear

Simplify the Constraint

- *For **Important Voxels**, stand at Near&Far, look along **Specified Dirs***

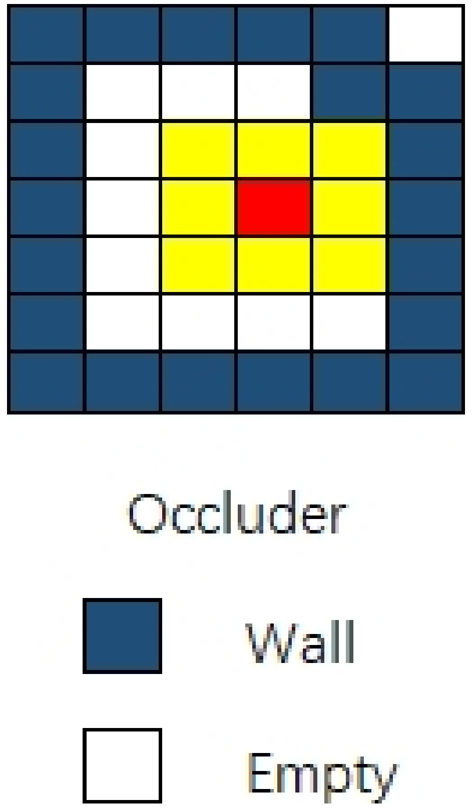
$$VF_{New}(viewPos, viewDir, occludee) = False \Rightarrow VF_{Original}(viewPos, viewDir, occludee) = False$$



Simplify the Constraint

- $For\ \mathbf{Important\ Voxels}, stand\ at\ Near\&Far, look\ along\ \mathbf{Specified\ Dirs}$

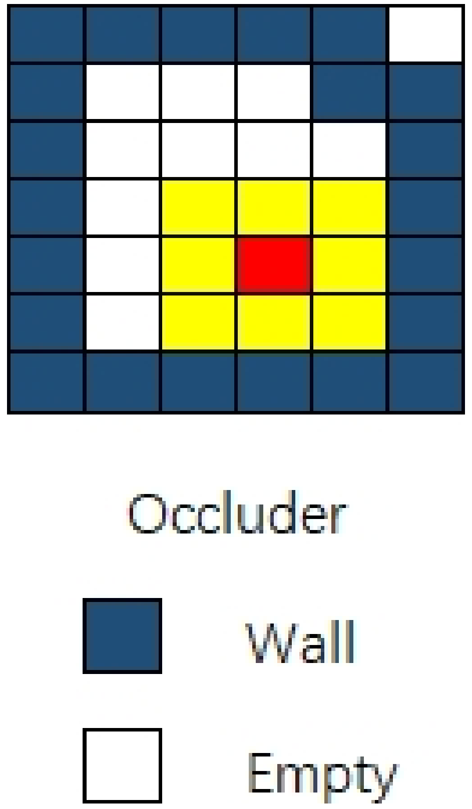
$$VF_{New}(viewPos, viewDir, occludee) = False \Rightarrow VF_{Original}(viewPos, viewDir, occludee) = False$$



Simplify the Constraint

- $For\ \mathbf{Important\ Voxels}, stand\ at\ Near\&Far, look\ along\ \mathbf{Specified\ Dirs}$

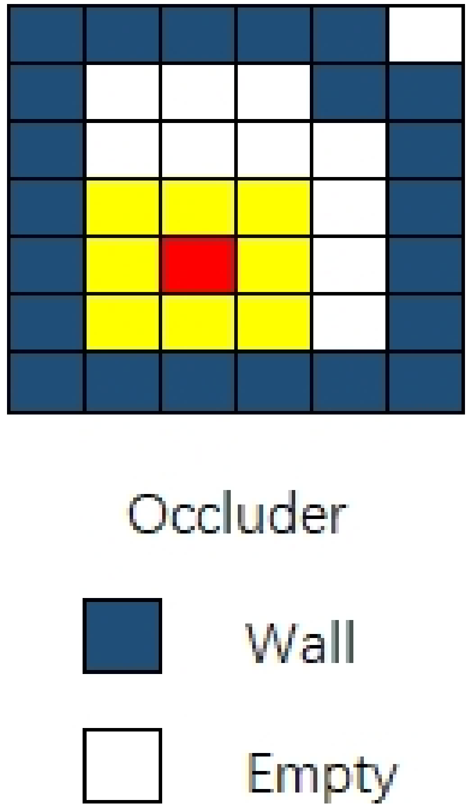
$$VF_{New}(viewPos, viewDir, occludee) = False \Rightarrow VF_{Original}(viewPos, viewDir, occludee) = False$$



Simplify the Constraint

- *For* ***Important Voxels****, stand at* *Near&Far**, look along* ***Specified Dirs***

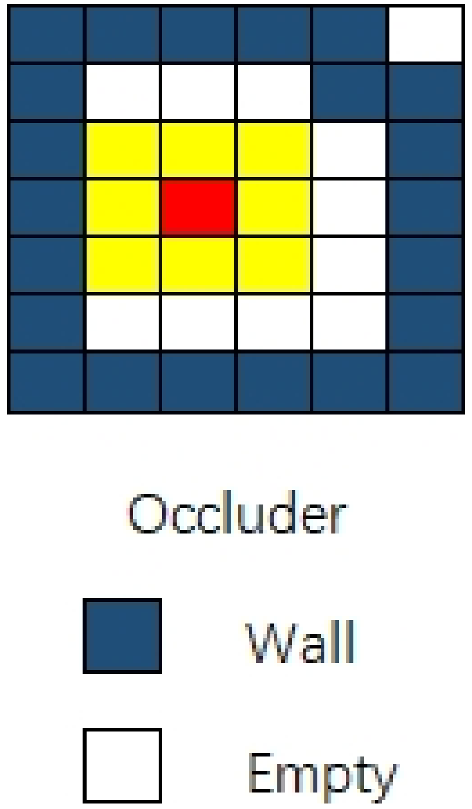
$$VF_{New}(viewPos, viewDir, occludee) = False \Rightarrow VF_{Original}(viewPos, viewDir, occludee) = False$$



Simplify the Constraint

- $For\ \mathbf{Important\ Voxels}, stand\ at\ Near\&Far, look\ along\ \mathbf{Specified\ Dirs}$

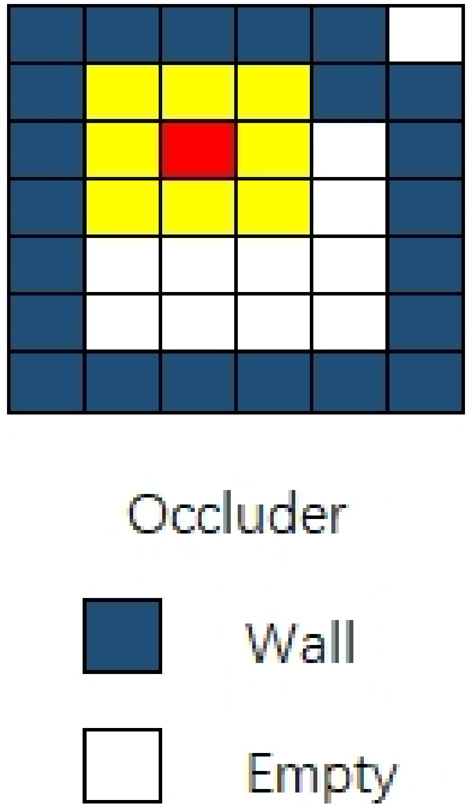
$$VF_{New}(viewPos, viewDir, occludee) = False \Rightarrow VF_{Original}(viewPos, viewDir, occludee) = False$$



Simplify the Constraint

- $For\ \textbf{Important Voxels}, stand\ at\ Near\&Far, look\ along\ \textbf{Specified Dirs}$

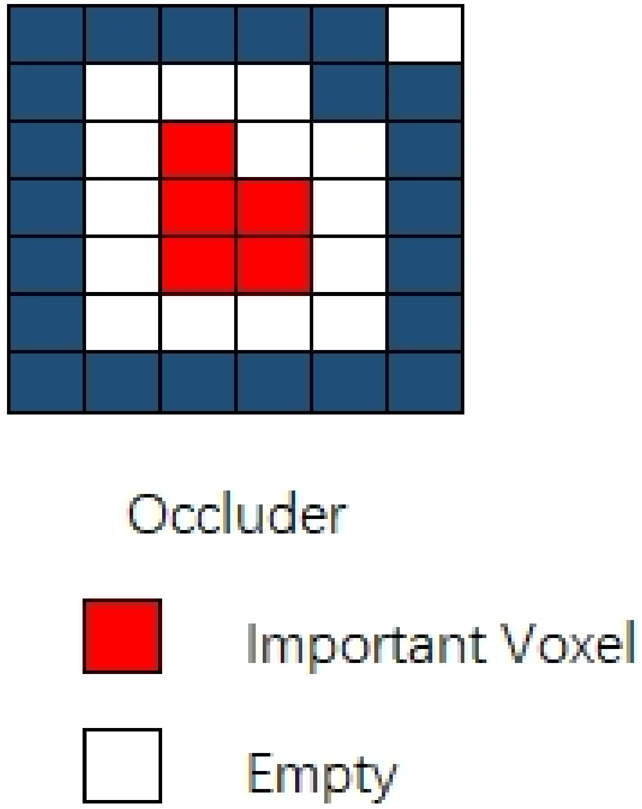
$$VF_{New}(viewPos, viewDir, occludee) = False \Rightarrow VF_{Original}(viewPos, viewDir, occludee) = False$$



Simplify the Constraint

- *For* ***Important Voxels****, stand at* *Near&Far**, look along* ***Specified Dirs***

$$VF_{New}(viewPos, viewDir, occludee) = False \Rightarrow VF_{Original}(viewPos, viewDir, occludee) = False$$



Simplify the Constraint

- For $\forall occludee \subseteq \{V_{important}\}$,

  $stand\ at\ \forall viewPos \subseteq Pos_{Near} \cup Pos_{Far}$,

  $look\ along\ \forall viewDir \subseteq \{SpecifiedDirections\}$

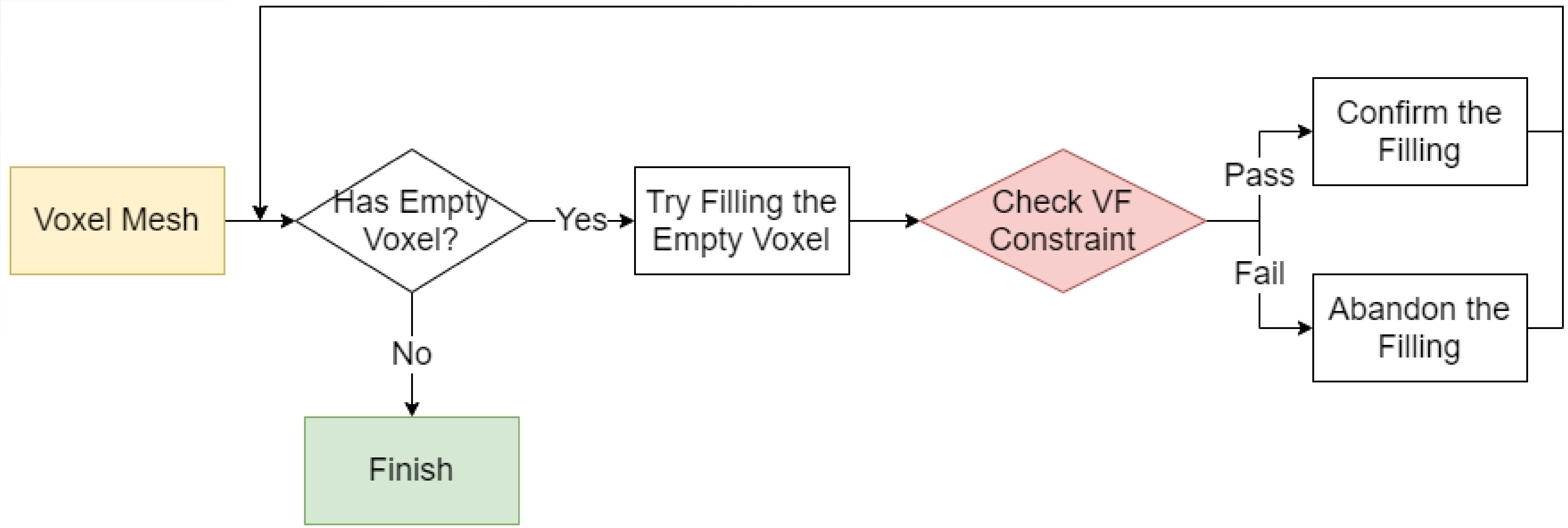
  Satisfy

  $VF_{New\ Occlusion\ Mesh}(viewPos, viewDir, occludee) = False$

  $\Rightarrow VF_{Original\ Occluder}(viewPos, viewDir, occludee) = False$

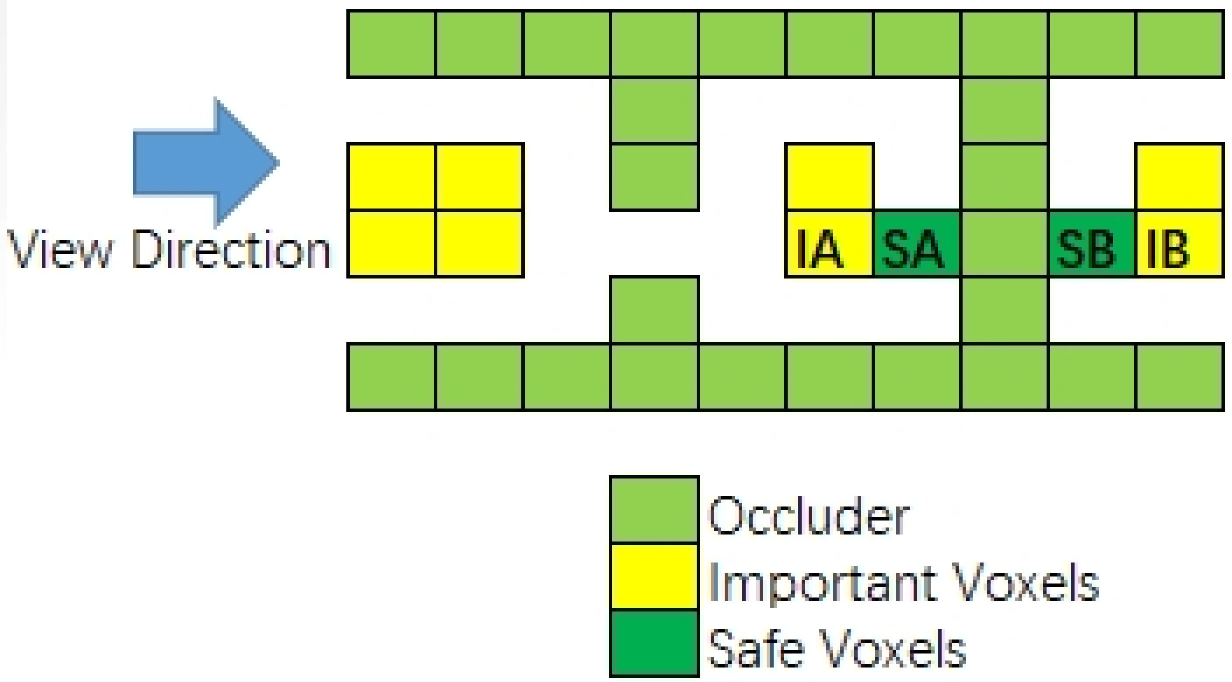
Smooth Mesh with the Constraint

- Use VF(Visibility Function) for smoothing



Smooth Mesh with the Constraint

- Heuristic approach: Deduce safe voxels
  - For a given direction
  - 2 Important voxels are not visible to each other
  - Voxels between are all safe voxels



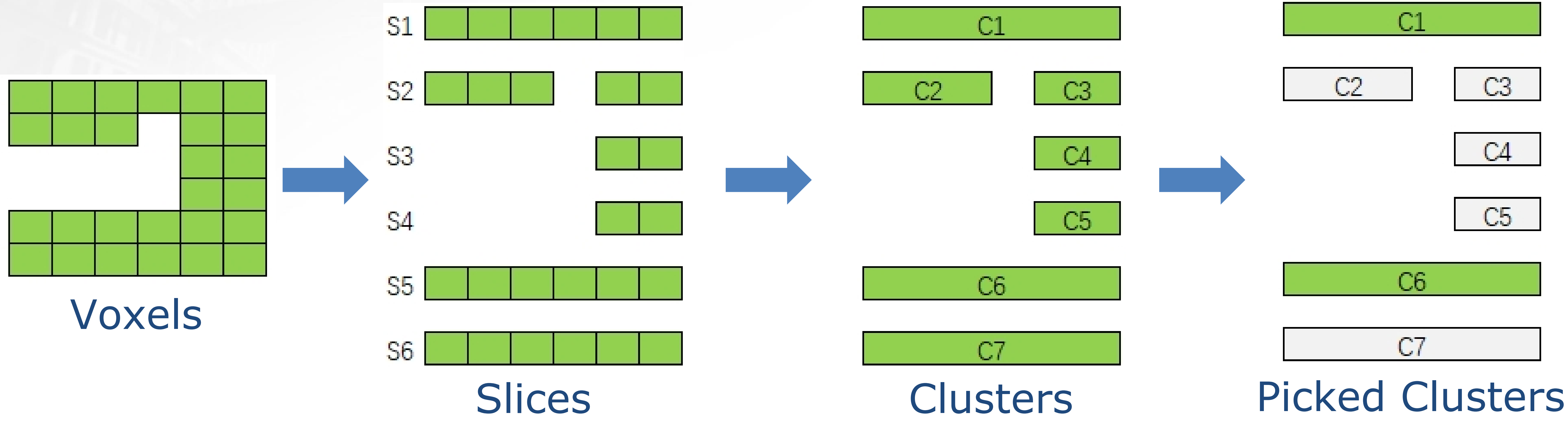
Smooth Mesh with the Constraint

- Heuristic approach: Deduce safe voxels
  - For a given direction
  - 2 Important voxels are not visible to each other
  - Voxels between are all safe voxels

- Use Heuristic approach for smoothing

Filter Clusters

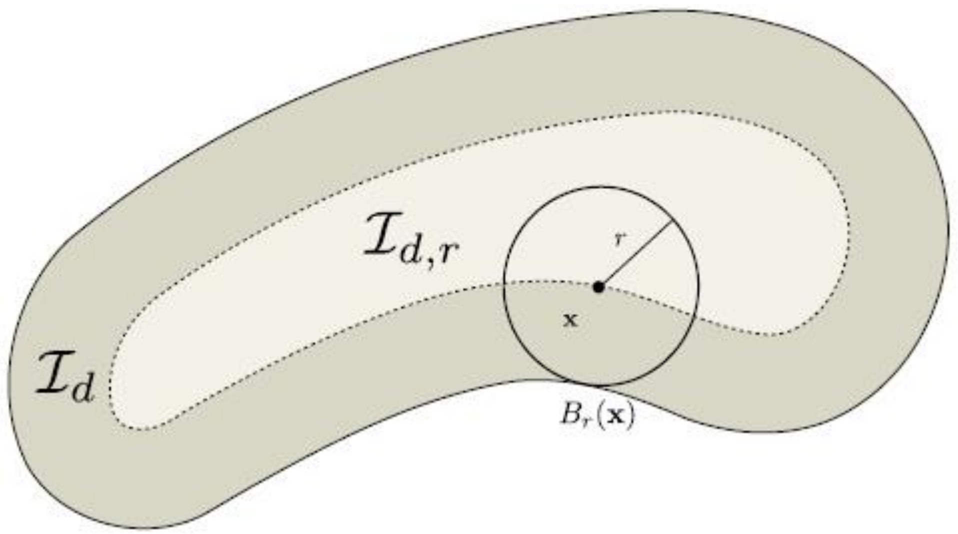
- Slice along a given axis
- Cluster connected voxels
- Pick all clusters with high occlusion power



Filter Clusters

- Calculate Occlusion Power: Ground Truth
  - Run the integration for every cluster
    - D stands for “closest distance to the border of occlusion area ”

$$M(\vec{d}) = \int_{I_{\vec{d}}} D(x)dA \approx \sum_{x \in I_{\vec{d}}} D(x)dA$$



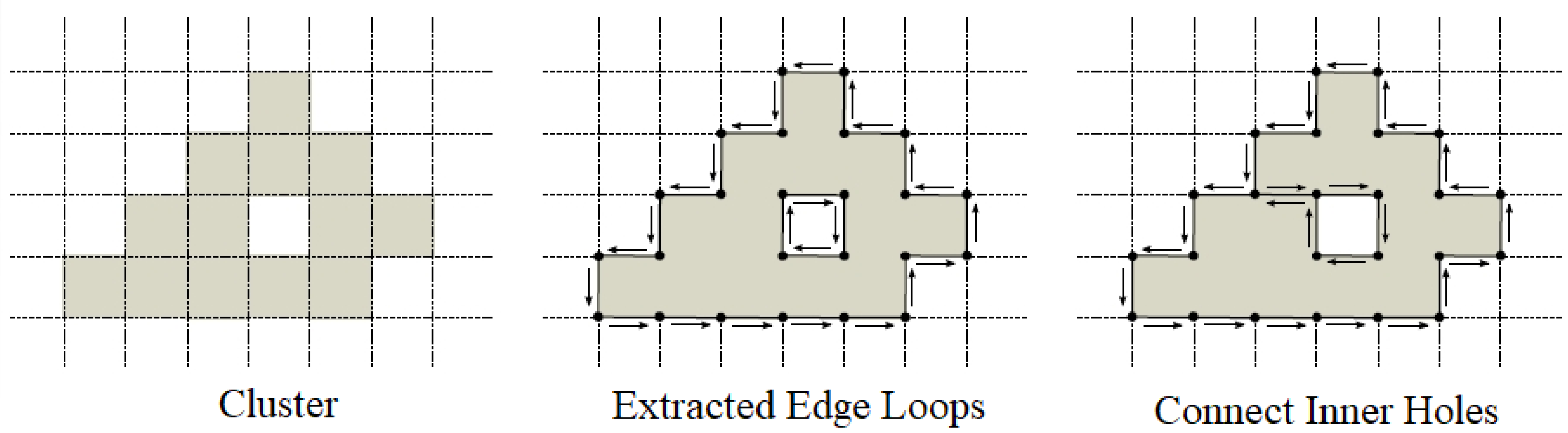
Filter Clusters

- Calculate Occlusion Power: Heuristic approach
  - Find all important voxels that:
    - $VF = False$, when using the original occluder
    - $VF = True$, when using all picked clusters
  - Mark clusters that can occlude these voxels
    - Use the number of marks as relative occlusion power

- 10 times faster than the ground truth calculation

Triangulate

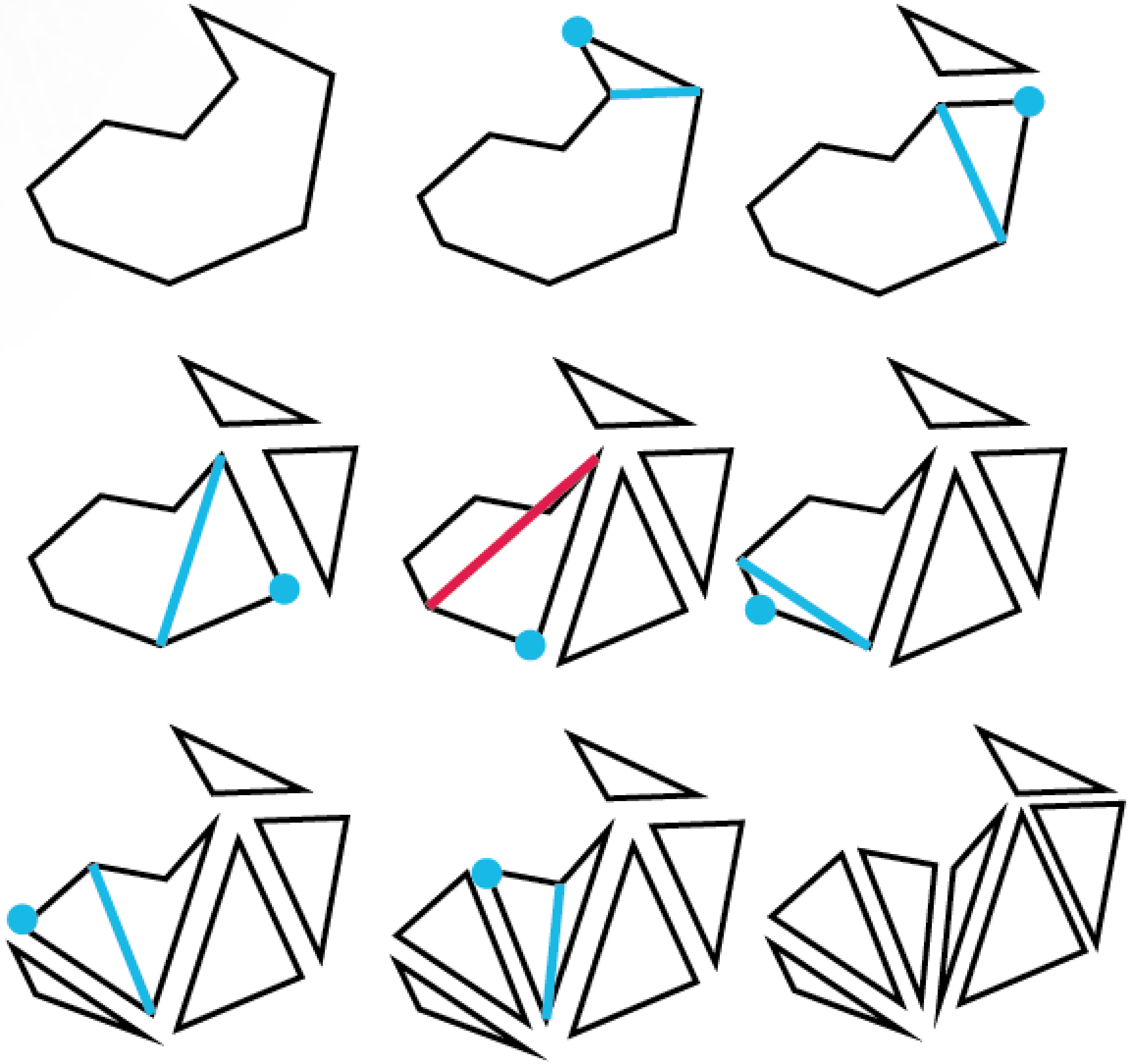
- Polygonization
  - Edge Loop Extraction
  - Edge Loop Simplification
    - Ramer-Douglas-Peucker



Triangulate

- Polygonization
  - Edge Loop Extraction
  - Edge Loop Simplification
    - Ramer-Douglas-Peucker

- Triangulation
  - Ear Clipping



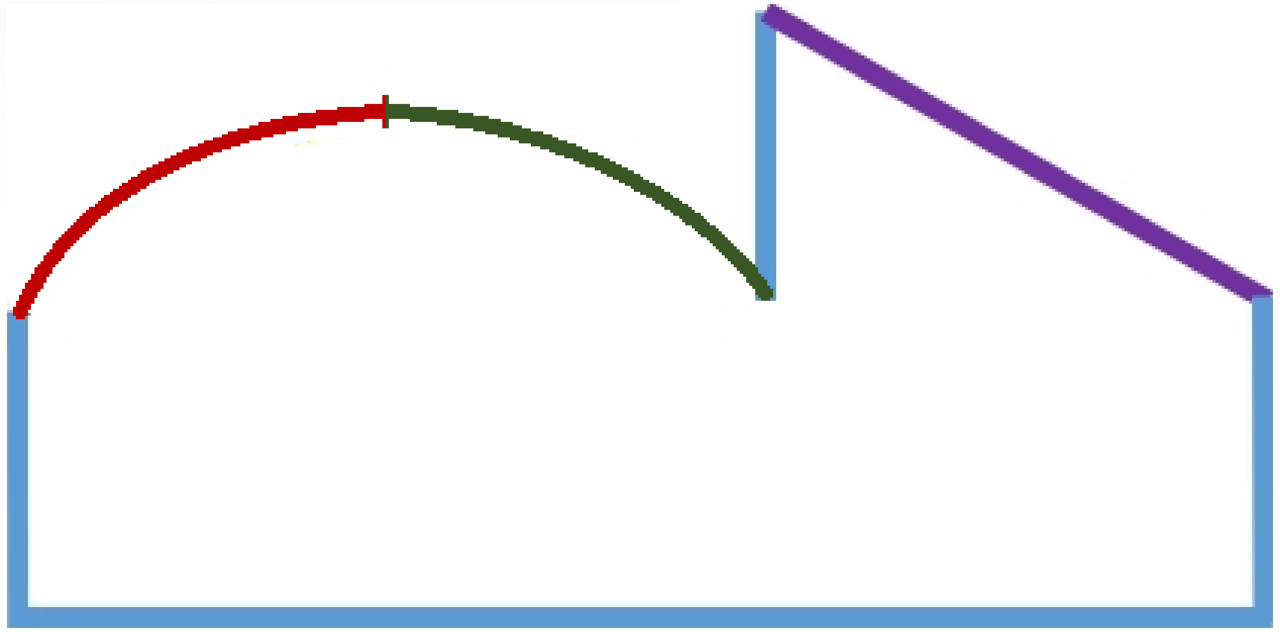
Handle Curved Surfaces

- Strip slopes and curved surfaces into sub-meshes
- Simplify all parts and combine the results

Original Mesh

Handle Curved Surfaces

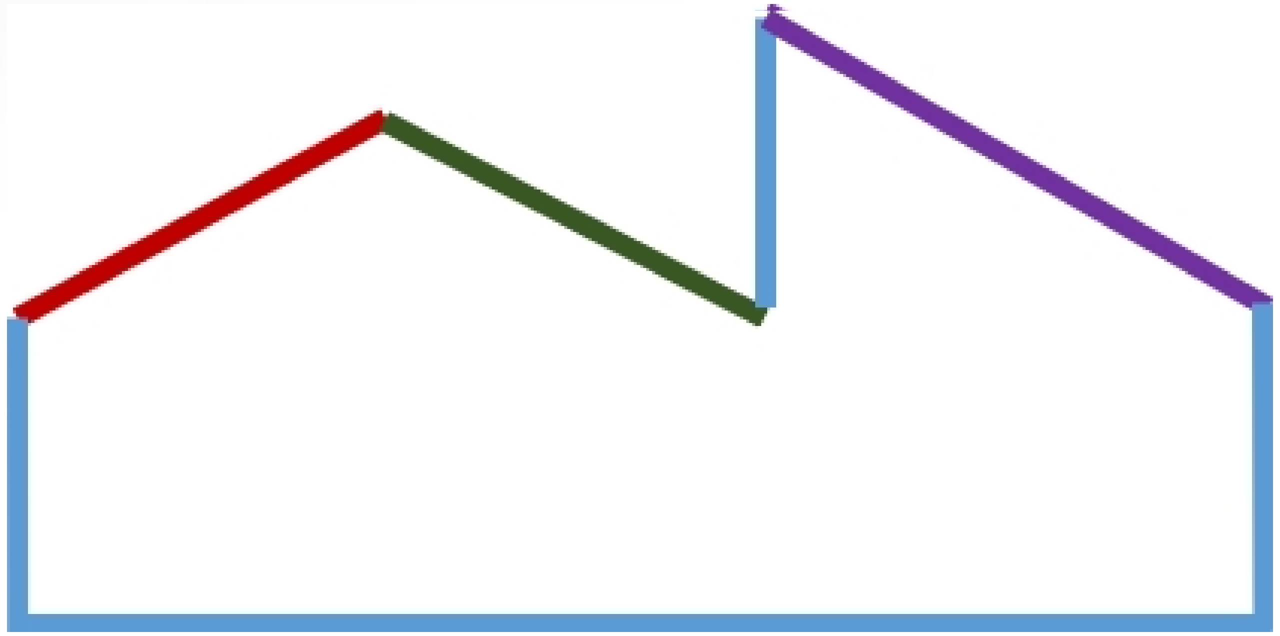
- Strip slopes and curved surfaces into sub-meshes
- Simplify all parts and combine the results



Split Curved Surfaces into sub-meshes

Handle Curved Surfaces

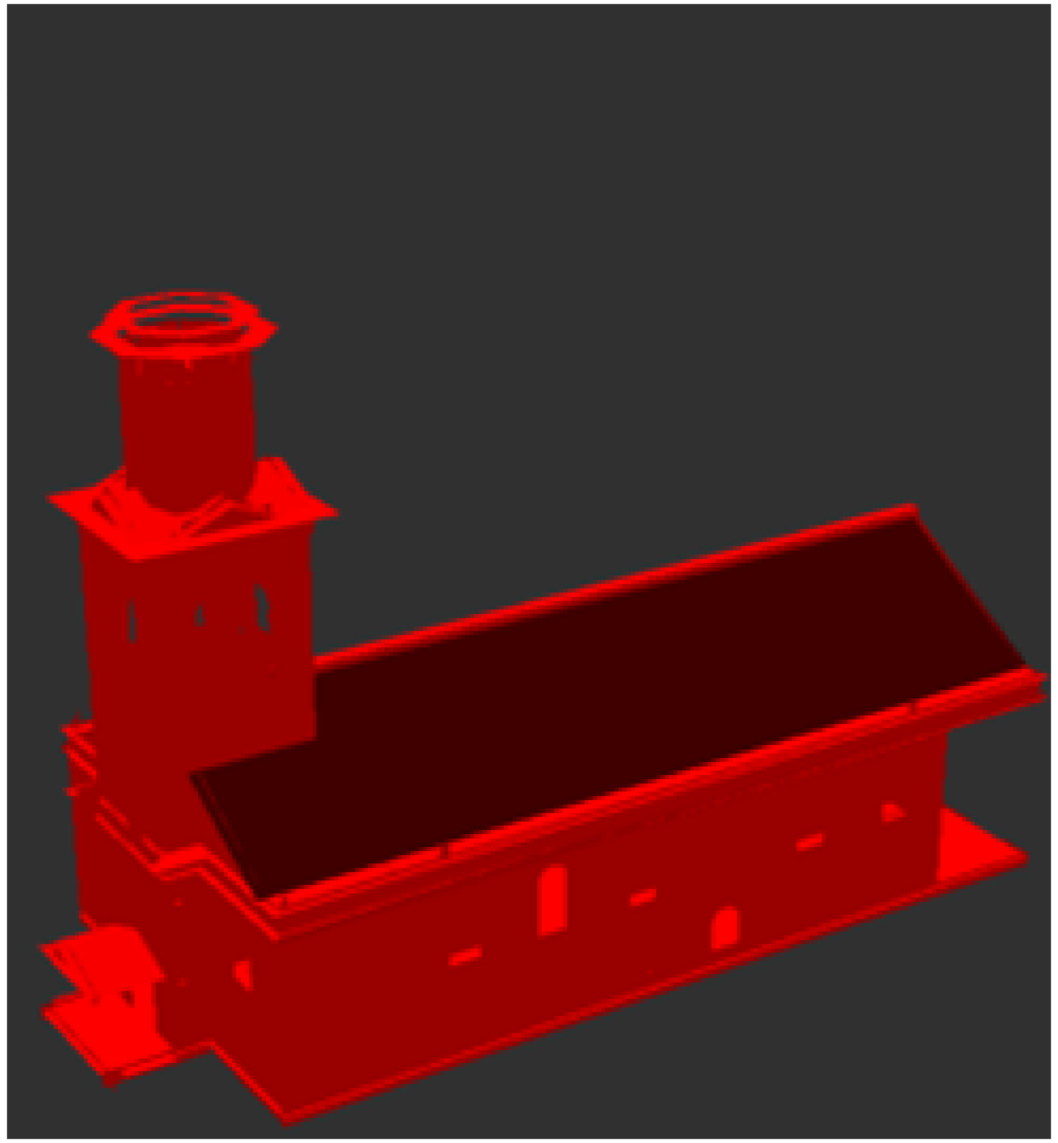
- Strip slopes and curved surfaces into sub-meshes
- Simplify all parts and combine the results



Simplify each and combine the results

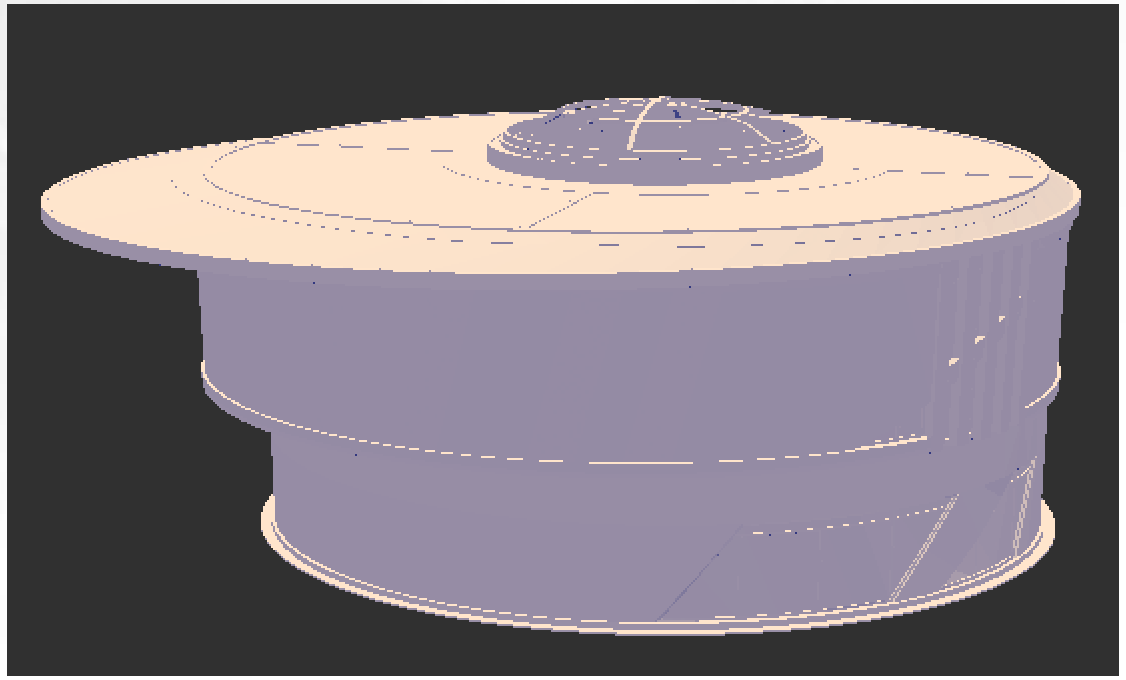
Demonstration of Our Generator

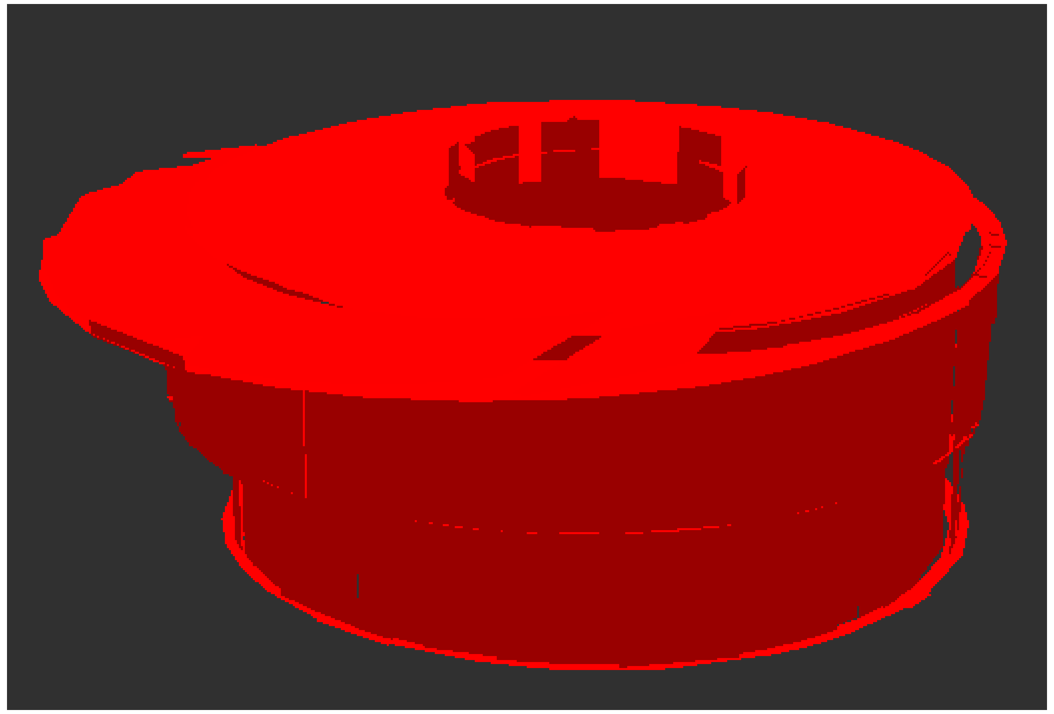
Original Occluder
75240 Triangles



Simplified Occlusion Mesh
896 Triangles

Demonstration of Our Generator

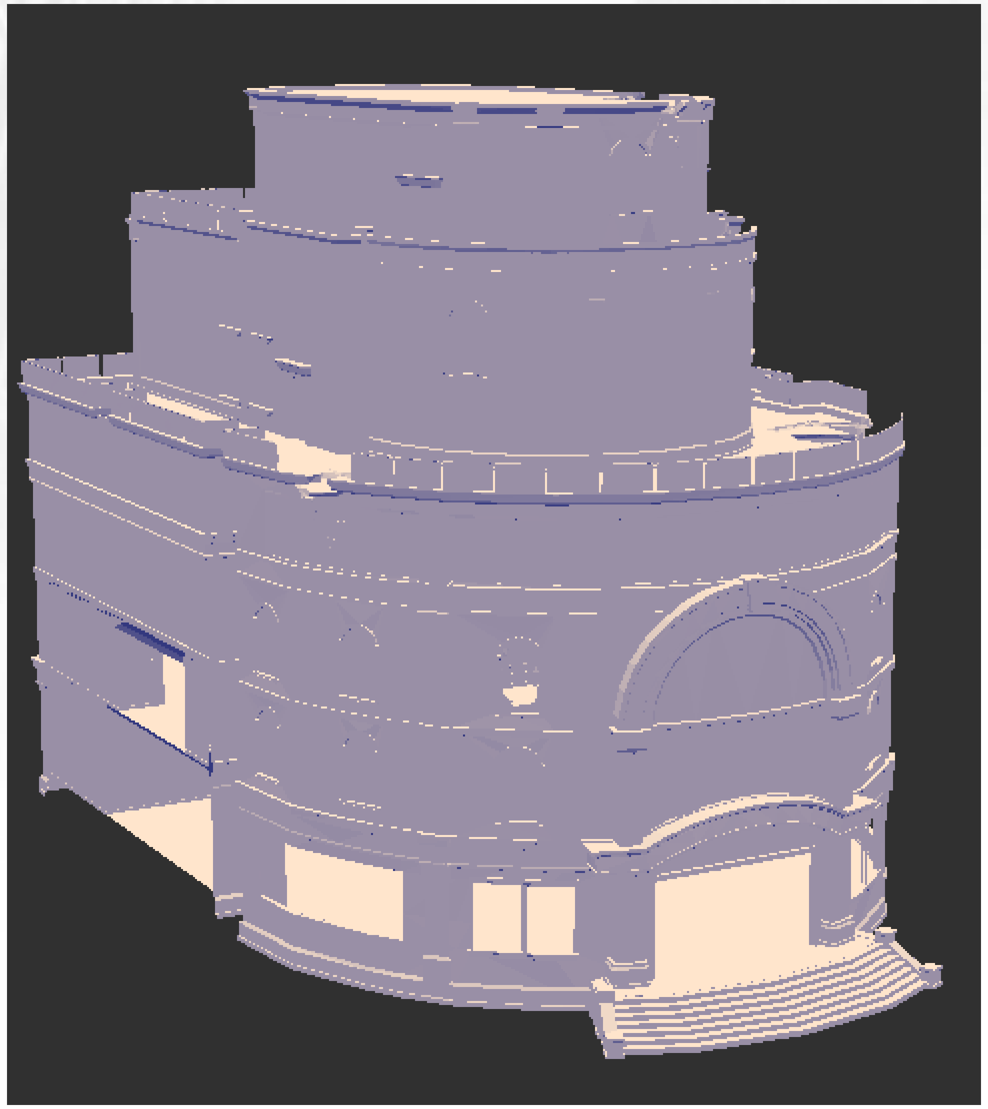




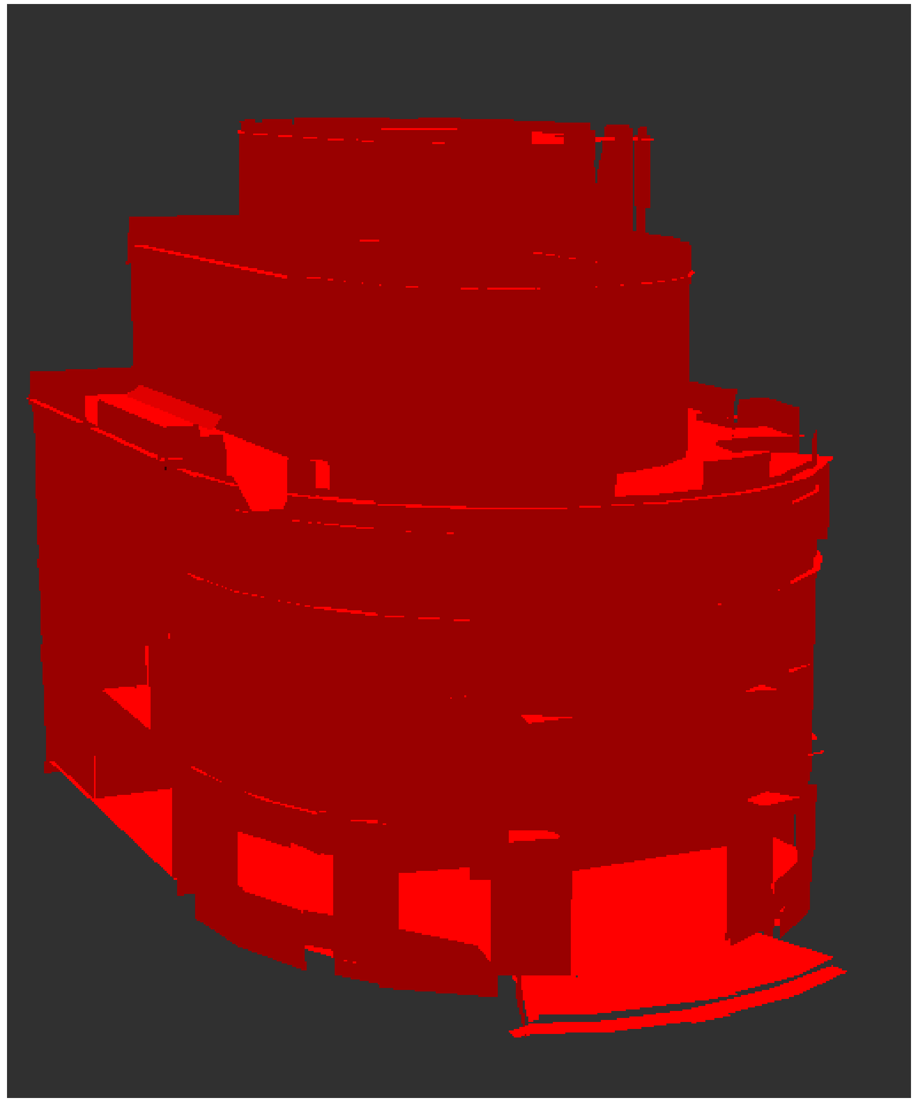
Original Occluder
10716 Triangles

Simplified Occlusion Mesh
916 Triangles

Demonstration of Our Generator



Original Occluder
30589 Triangles



Simplified Occlusion Mesh
688 Triangles

Part 5: Conclusion

Overall Performance

	SOC Cost (us)	Total Culling Cost (us)	SOC Culling Rate	Total Culling Rate
OPPO R9s	1107.28	1547.52	65%	91%
HUAWEI mate8	1052.76	1716.09	65%	91%
iPhone 6s	1074.05	1486.27	65%	91%
iPhone 7 Plus	460.5	627.14	65%	91%
Samsung S10	535.11	929.46	64%	90%
iPhone XS	286.78	437.58	64%	90%

Conclusion

- To get an efficient software occlusion culling solution for mobile platform, one should optimize every part of the solution.
    - Lightweight the SOC algorithm to make it suitable for mobile platform
    - Build a cache-friendly and multi-threading culling pipeline
    - Create high-quality occlusion mesh generator based on visibility function

Thank You!

- Mengyun Yi
- Kaiyuan Zhao
- Qianming Chen
- Wenxiang Tu
- Yili Chen
- Yingxie Gao

Q & A

GDC × NetEase Games
Passion of gamers

March 20-24,2023
San Francisco,CA

#GDC23
#NetEase Games