



The filtered and culled
Visibility Buffer

Wolfgang Engel
CEO Confetti

Demo



Table of contents

- The Visibility Buffer
- Cluster Culling / Triangle Filtering
- Re-using triangle filtered results for multiple rendering passes

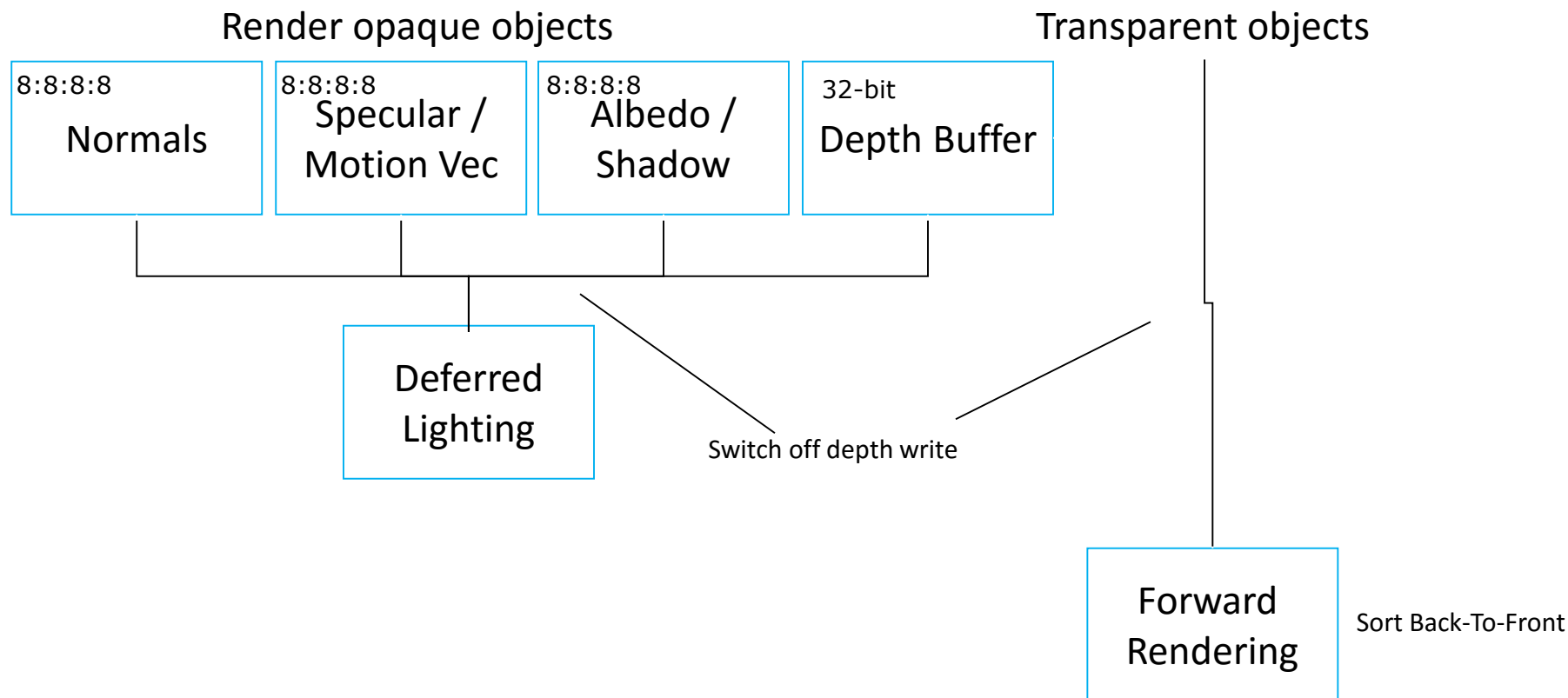
The Visibility Buffer

Motivation

- Forward rendering shades all fragments in triangle-submission order
 - Wastes rendering power on pixels that don't contribute to the final image
- Deferred shading solves this problem in 2 steps:
 - First, surface attributes are stored in screen buffers -> G-Buffer
 - Second, shading is computed for visible fragments only
- However, deferred shading increases memory bandwidth consumption:
 - Screen buffers for: normal, depth, albedo, material ID,...
 - G-Buffer size becomes challenging at high resolutions

High-Level View 2009

G-Buffer (taken from [Engel2009] Killzone 2 layout)



High-Level View 2014

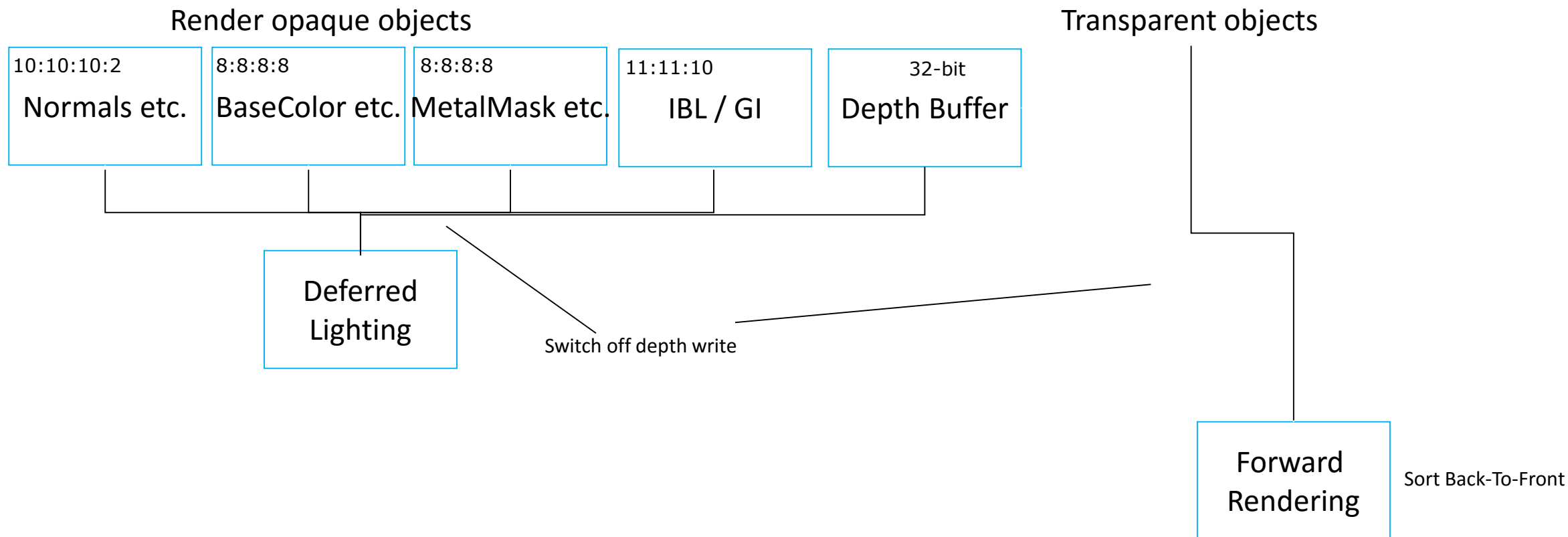
G-Buffer – Frostbite Engine [Lagarde]

	R	G	B	A	
GB0	Normal (10:10)		Smoothness	MaterialId (2)	10:10:10:2
GB1	BaseColor			MatData(5)/Normal(3)	8:8:8:8
GB2	-----	MetalMask	Reflectance	AO	8:8:8:8
GB3	Radiosity/Emissive				11:11:10

Depth - R32G8X24_TYPELESS

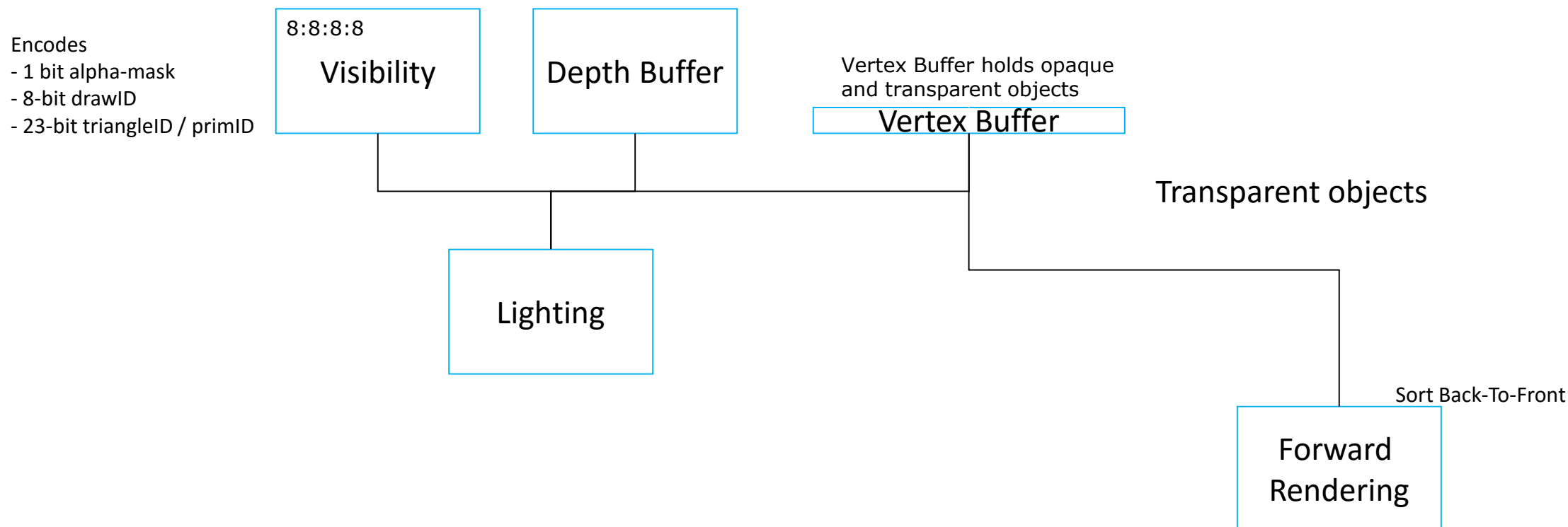
High-Level View 2014

G-Buffer – Frostbite Engine [Lagarde]



High-Level View

Visibility Buffer (similar to [Burns][Schied])



Visibility Buffer PC 1080p – Memory

Memory	Description	NoMSAA	2xMSAA	4xMSAA
Visibility Buffer	keeps address of each triangle in 32-bit per pixel 4 bytes * 1920 * 1080	7.9 MB	15.8 MB	31.6* MB
Depth Buffer	4 byte * 1920 * 1080	7.9 MB	15.8 MB	31.6* MB
Hierarchical Z	4 byte * 1920 * 1080 * 1/64	0.12 MB	-	-
Vertex Buffer	28 bytes per vertex + padding float3 position; uint normal; uint tangent; uint texCoord; uint materialID; uint pad; // to align to 128 bits – 32 byte for NVIDIA Does not increase with resolution.			
Textures		21* MB	-	-
	Draw arguments, Uniform, Descriptors etc.	2* MB	-	-
Overall		38.80* MB	54.60* MB	86.20* MB

* rough estimate; driver might increase it

G-Buffer PC 1080p – Memory

Memory	Description	NoMSAA	2xMSAA	4xMSAA
Normals	10:10:10:2 - 4 bytes * 1920 * 1080	7.9 MB	15.8* MB	31.6* MB
PBR	8:8:8:8 - 4 bytes * 1920 * 1080	7.9 MB	15.8* MB	31.6* MB
Albedo	8:8:8:8 - 4 bytes * 1920 * 1080	7.9 MB	15.8* MB	31.6* MB
GI / IBL	11:11:10 - 4 bytes * 1920 * 1080	7.9 MB	15.8* MB	31.6* MB
Depth	8:8:8:8 - 4 bytes * 1920 * 1080	7.9 MB	15.8* MB	31.6* MB
Hierarchical Z	4 bytes * 1920 * 1080 * 1/64	0.49 MB	-	-
Draw arguments. Etc.		2* MB		
Overall		41.99* MB	81.49* MB	160.49* MB

* rough estimate; driver might increase it

Visibility Buffer PC 4k – Memory

Memory	Description	NoMSAA	2xMSAA	4xMSAA
Visibility Buffer	keeps address of each triangle in 32-bit per pixel 4 bytes * 3840 * 2160	31.64 MB	63.28* MB	126.56* MB
Depth Buffer	4 bytes * 3840 * 2160	31.64 MB	63.28* MB	126.56* MB
Hierarchical Z	4 bytes * 3840 * 2160 * 1/64	0.49 MB	-	-
Vertex Buffer	28 bytes per vertex + padding float3 position; uint normal; uint tangent; uint texCoord; uint materialID; uint pad; // to align to 128 bits – 32 byte for NVIDIA Does not increase with resolution.		-	-
Textures		21* MB		
	Draw arguments, Uniform, Descriptors etc.	2* MB		
Overall		86.77 MB	150.05 MB	276.61 MB

* rough estimate; driver might increase it

G-Buffer PC 4k – Memory

Memory	Description	NoMSAA	2xMSAA	4xMSAA
Normals	10:10:10:2 - 4 bytes * 3840 * 2160	31.64 MB	63.28* MB	126.56* MB
PBR	8:8:8:8 - 4 bytes * 3840 * 2160	31.64 MB	63.28* MB	126.56* MB
Albedo	8:8:8:8 - 4 bytes * 3840 * 2160	31.64 MB	63.28* MB	126.56* MB
GI / IBL	11:11:10 - 4 bytes * 3840 * 2160	31.64 MB	63.28* MB	126.56* MB
Depth	8:8:8:8 - 4 bytes * 3840 * 2160	31.64 MB	63.28* MB	126.56* MB
Hierarchical Z	4 bytes * 3840 * 2160 * 1/64	0.49 MB	-	-
Draw arguments. Etc.		2* MB	-	-
Overall		160.69* MB	318.89* MB	635.29* MB

* rough estimate; driver might increase it

Visibility Buffer XOne 1080p – Memory

[illegible]

Visibility Buffer – Filling pseudocode

- Visibility Buffer generation step
- For each pixel in screen:
 - Pack (alpha masked bit, drawID, primitiveID) into 1 32-bit UINT
 - Write that into a screen-sized buffer
- The tuple (alpha masked bit, drawID, primitiveID) will allow a shader to access the triangle data in the shading step

Visibility Buffer – Shading pseudocode

- For each pixel in screen-space we do:
 1. Get drawID/triangleID at pixel pos
 2. Load data for the 3 vertices from the VB
 3. Compute triangle gradients
 4. Interpolate vertex attributes at pixelpos using gradients
(could do triangle / object-space lighting)
 - a) Attribs use w from position to compute perspective correct interpolation
 - b) MVP matrix is applied to position
 5. We have all data ready: shade and calculate final color

Visibility Buffer - Benefits

- Better decouples visibility from shading
 - Calculating derivatives can be done separate from the shading phase (we include it in the moment)
 - We can shade then with different frequency or quality
- Improves memory efficiency
 - Improves cache utilization
 - Memory accesses are highly coherent
→ high cache hit rates
 - A G-Buffer needs to store data per screen-space pixel. Compared to a vertex / index buffer some of this data is redundant
→ we can see 99% L2 cache hits for the Visibility Buffer for textures, vertex and index buffers

Visibility Buffer - Benefits

- Stores less data for complex lighting models like e.g. PBR compared to a G-Buffer
 - PBR data for Visibility Buffer is a struct in constant memory indexed by material id in vertex structure
 - This struct holds indices into a texture array for various PBR textures
 - It also holds per-material descriptions of what is necessary to drive BRDF
 - Any data that changes per-pixel is stored in textures that are referenced by the struct
- Decouples G-buffer footprint from screen resolution
 - Improves performance at high resolutions: 2K, 4K, MSAA ...
 - Improves performance on bandwidth-limited platforms

Visibility Buffer – Triangle Counts

San Miguel Scene

- 8 Million Triangles
- 5 Million Vertices

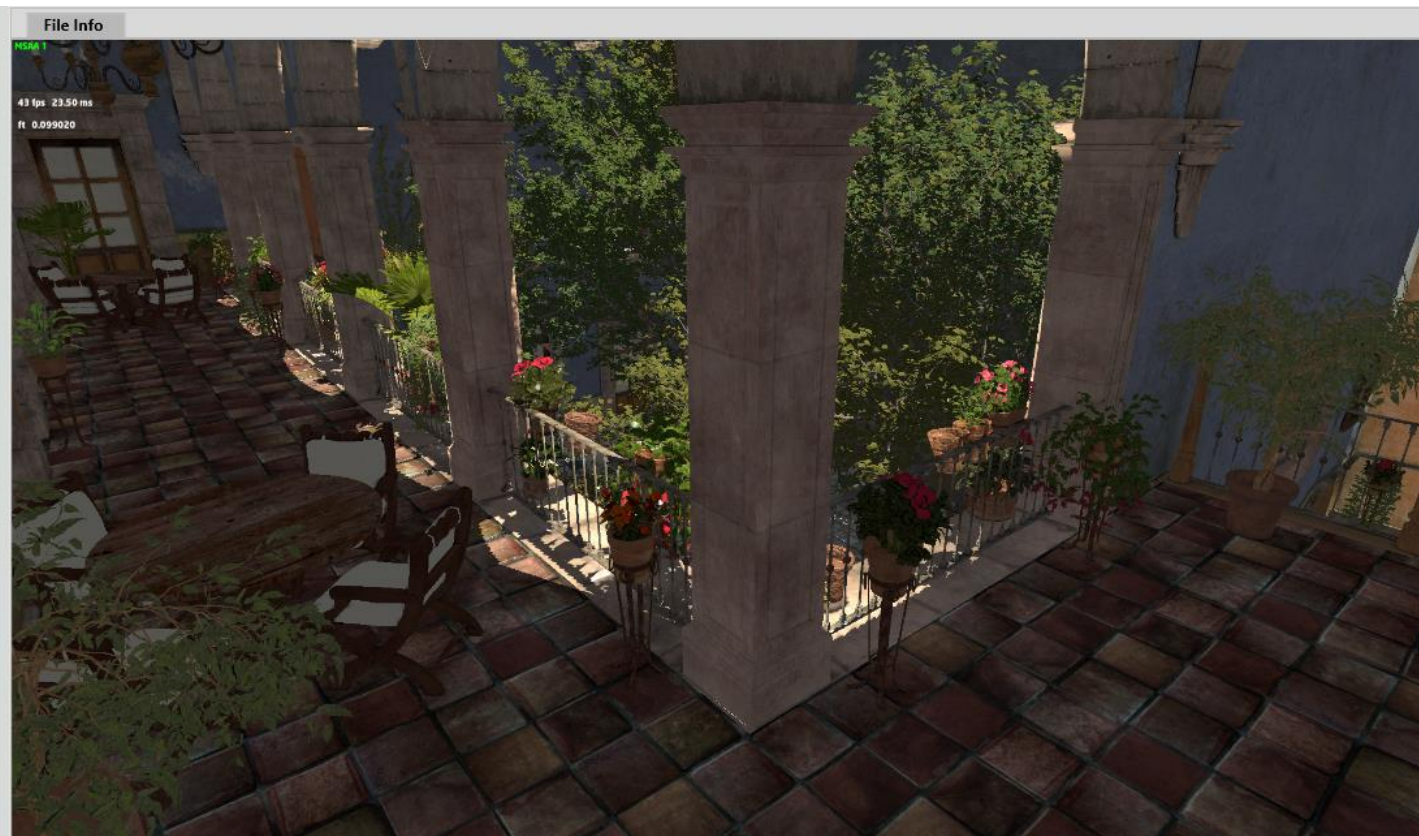
Triangles rendered after culling

- 1.87 Million triangles main view
- 2.40 Million triangles shadow map

Modern Game in 2016

1.8 Million triangles combined in Ultra

Events			
Graphics Filter (Ctrl+E) Counters			
Queue ID	Name	Global ID	PAPrimsIn
0	Signal(obj#1,2898)	0-1	0
4	ResourceBarrier(17,{...})	2-4	0
10	FRAME	5-643	4,277,743
11	CULLING	5-88	0
272	SHADOW MAP	89-364	1,870,187
568	FILL VIZ BUFFER SIMPLIF	365-637	2,407,539
863	RENDER HDAO	638-639	2
873	SHADING VB SIMPLIF	640-642	3
886	UI	643	12
902	DrawInstanced(102,1,0,0) {this->ID3D12GraphicsComman	644	34
908	DrawInstanced(72,1,0,0) {this->ID3D12GraphicsCommand	645	24



How do you do lighting?

- You can pick your favorite lighting architecture -
> Tiled-Deferred <-> Tiled Forward etc.
- Tiled-Forward or Forward+ seems to be a natural fit because it will benefit from the decreased vertex count through culling, filtering and visibility check and offers consistent lighting for opaque and transparent objects
- Triangle or Object-Space lighting is possible

What about Tessellation?

- Following [Wihlidal][Brainerd] this is a pre-step before or in parallel to Triangle filtering

Why didn't we implement this earlier?

- Two recent developments made the Visibility Buffer more attractive compared to a G-Buffer
 - DirectX 12 / Vulkan with multi draw indirect
 - Triangle culling / filtering [EDGE][Chajdas][Wihlidal]
... see the next slide ...

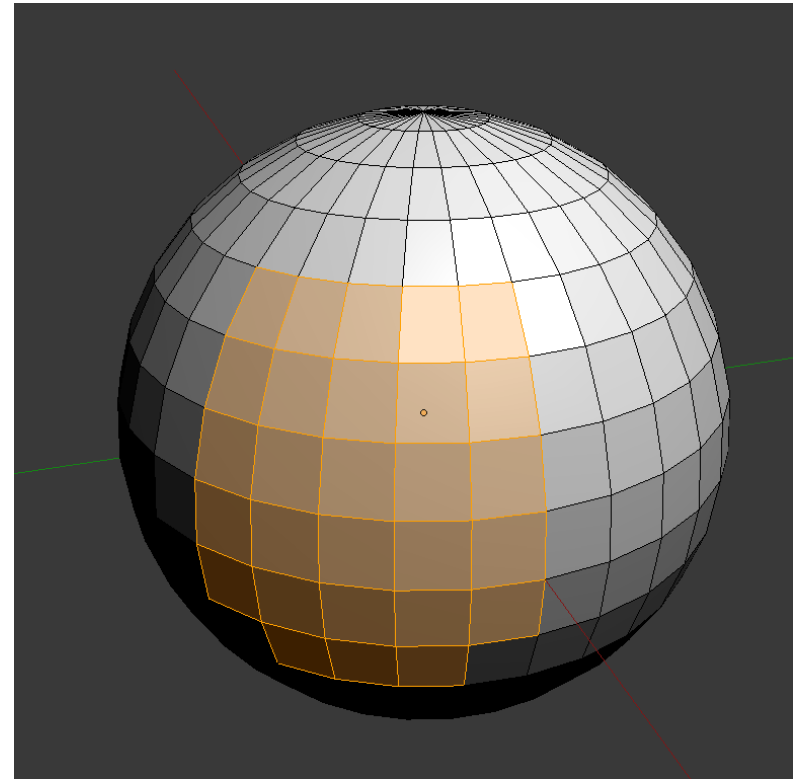
Cluster Culling / Triangle filtering

Motivation

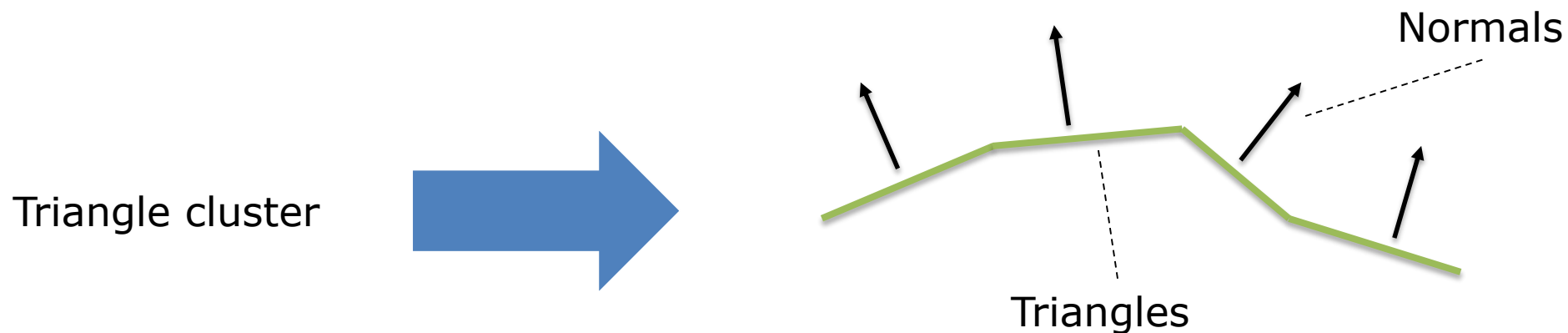
- Polygonal complexity of games increases every year
- Efficient triangle removal is an important aspect
- 2 culling stages
 - Cluster culling : cull groups of triangles before sending them to the GPU (following [Chajdas] on the CPU; [Wihlidal] on GPU)
 - Triangle Filtering: cull individual triangles after being sent to the GPU

Cluster culling

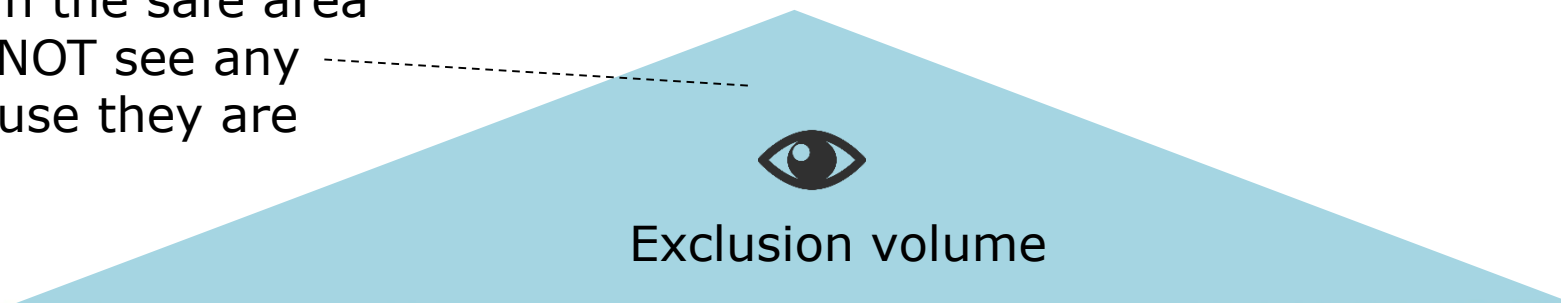
- Groups triangles in small chunks of 256 triangles with similar orientations
- Chunks have a model matrix associated (they can be moved around)
- Each chunk must pass a quick visibility test before being sent to the GPU:
 - Cone test



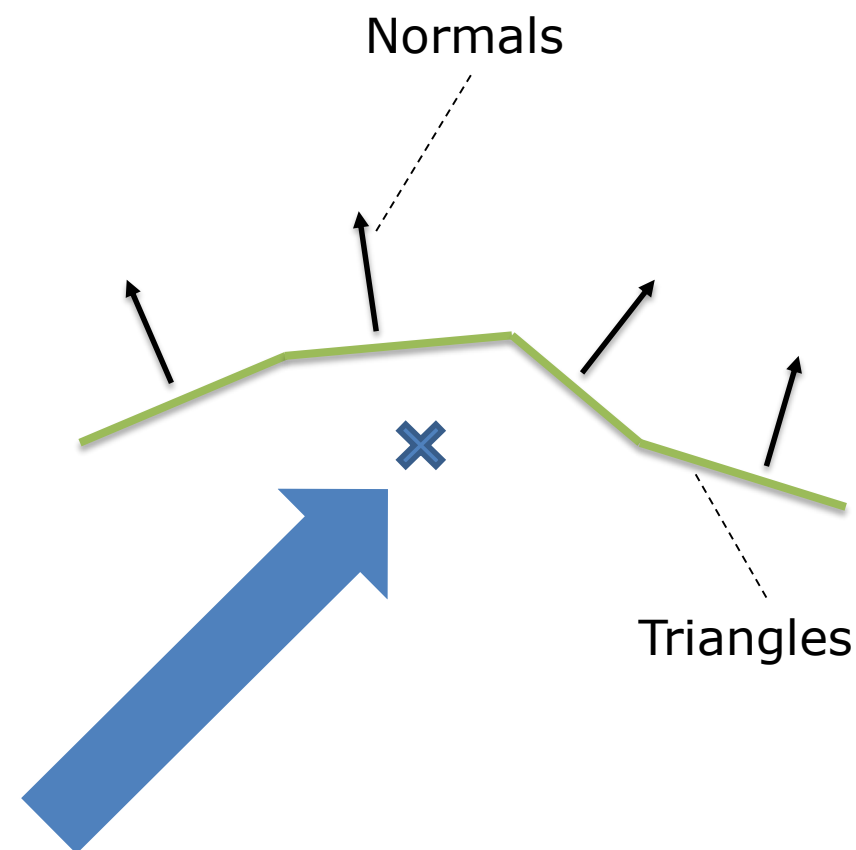
Cone test for fast cluster culling



If the eye is in the safe area
then we can NOT see any
triangle because they are
back-facing



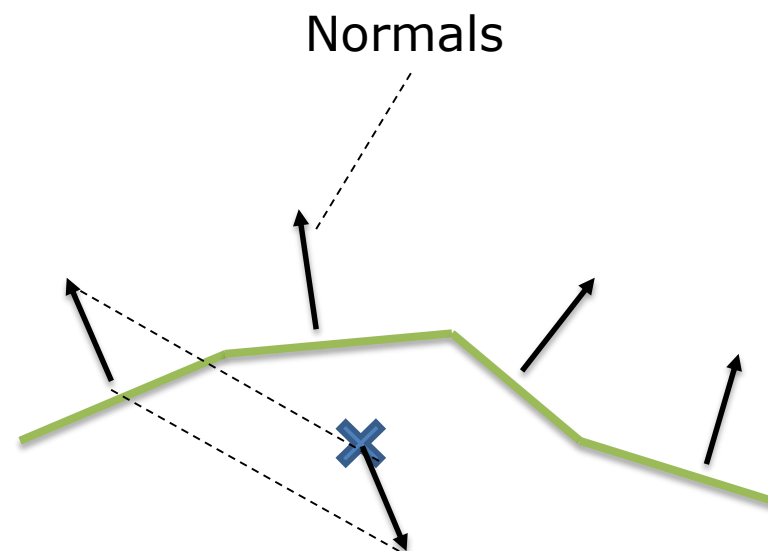
Exclusion volume



First, locate the center
of the cluster

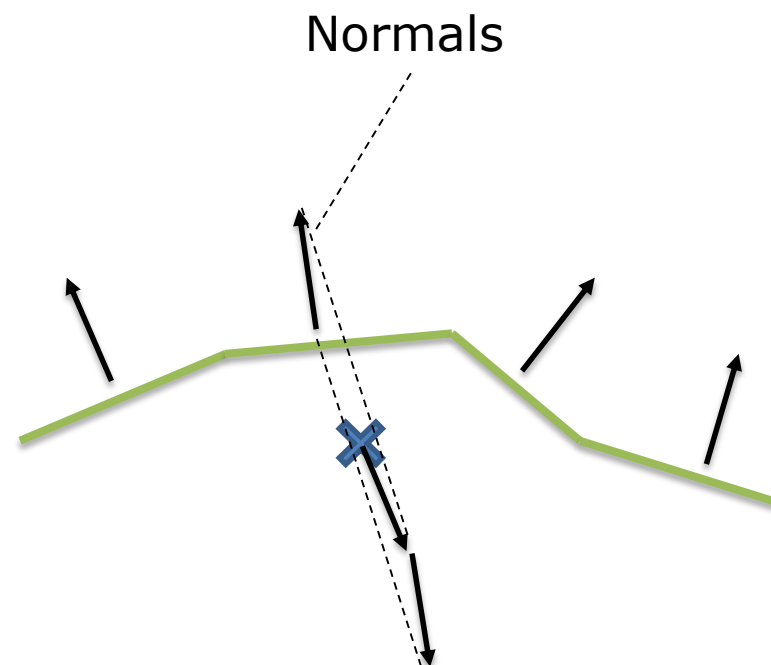
Exclusion volume

Then, negatively
accumulate normal
starting at cluster center



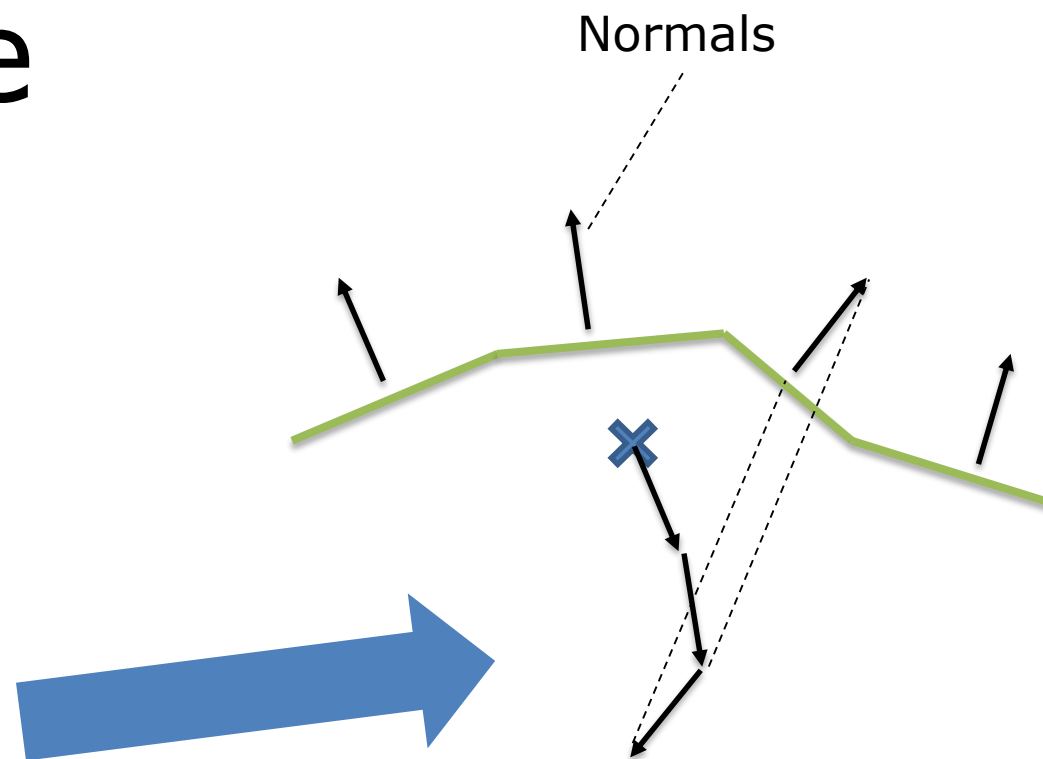
Exclusion volume

Negatively accumulate
second normal after
the first one



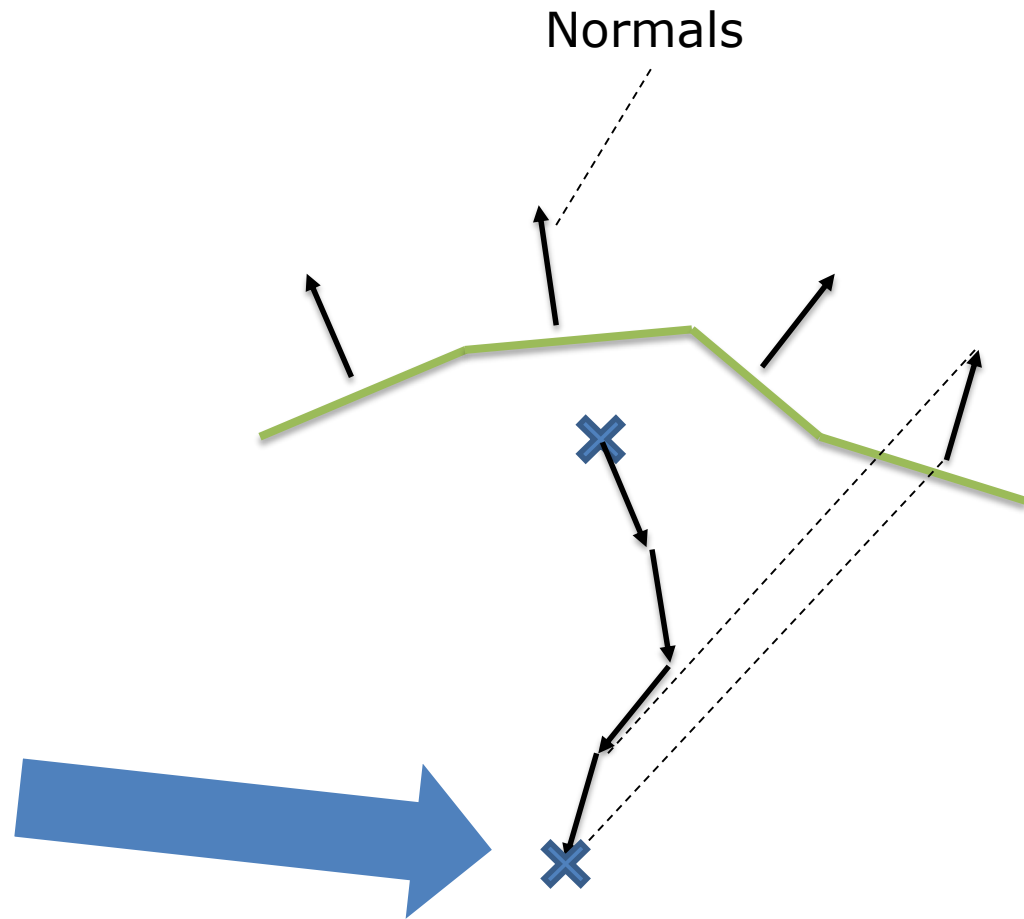
Exclusion volume

Accumulate
next one



Exclusion volume

After accumulating the last one we have the starting point of the exclusion volume and the direction



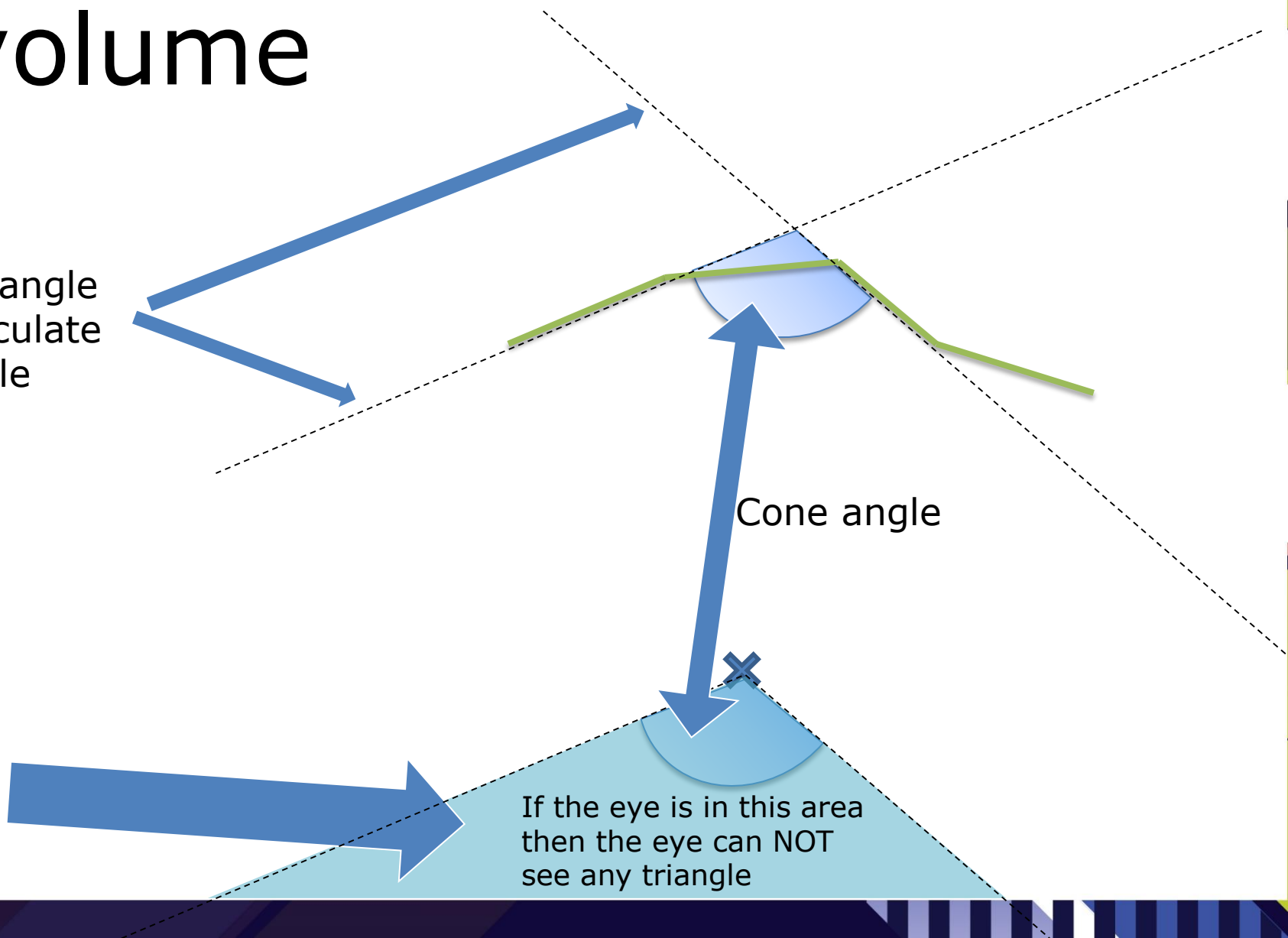
Exclusion volume

Most restrictive triangle planes used to calculate cone open angle

Cone angle

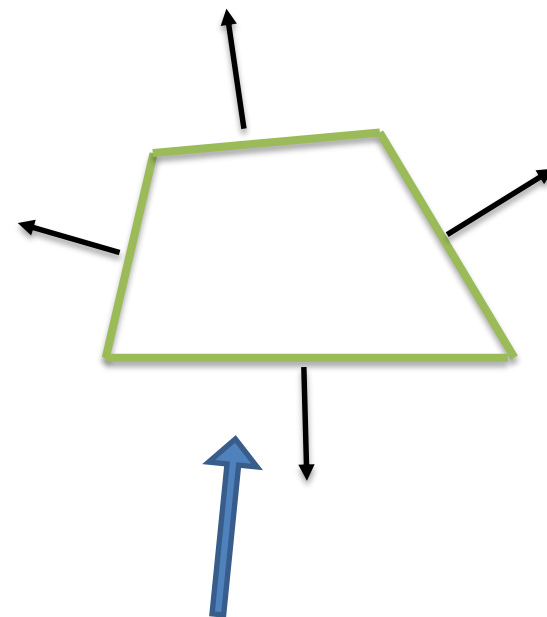
This is the calculated exclusion volume

If the eye is in this area then the eye can NOT see any triangle



Cluster culling efficiency

- Effectivity depends on the orientation of the faces in the cluster
 - The more similar the orientations, the bigger the exclusion / culling volume
- Depending on the triangles the exclusion volume can not be calculated
 - Invalid cluster for cluster culling → just pass it!



Invalid cluster → cluster culling not possible

Compute-based triangle filtering

- Motivation:
 - Cull triangles before they go into the graphics pipeline
 - Use the unused compute units during graphics pipeline execution with async compute
- Compute-based filter → one triangle per thread
 - Degenerate triangle culling
 - Back-face culling
 - Frustum culling
 - Small primitives culling
 - Depth culling (requires coarse depth buffer)(not in this demo)
- Triangle indices that pass these tests are appended to index buffer

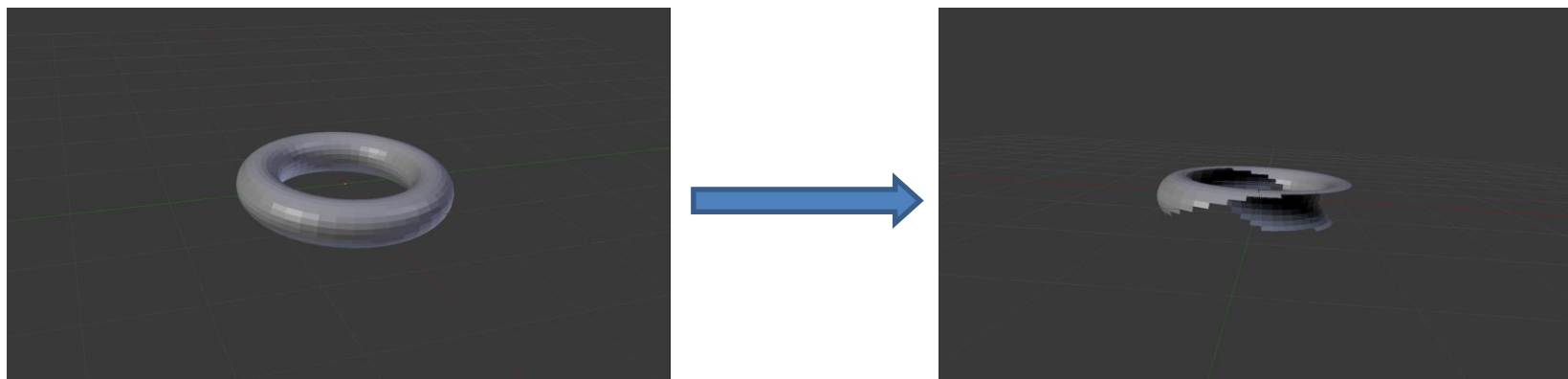
Compute-based triangle filtering

- Degenerate triangle culling
 - Allows to cull invisible zero-area triangles
 - Cost: → quick test (discard if at least two triangle indices are equal)
 - Effectiveness: → low

```
cull = ( indices[0] == indices[1] ||  
         indices[1] == indices[2] ||  
         indices[0] == indices[2] );
```

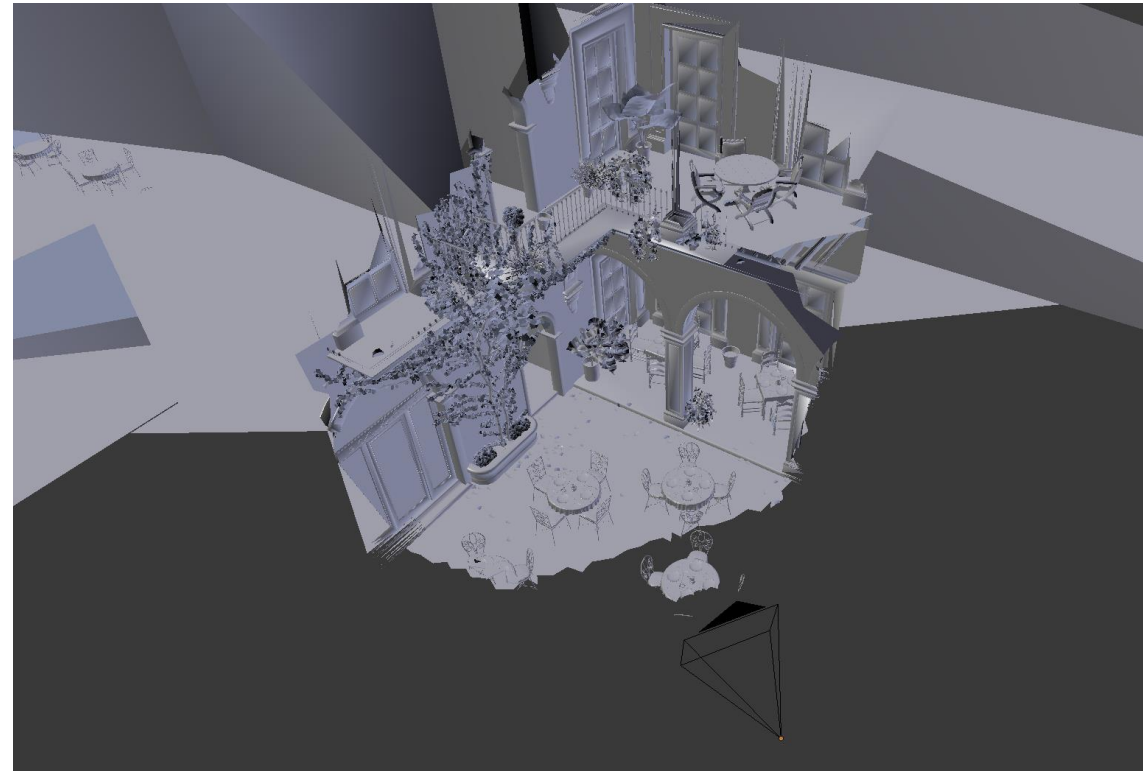
Compute-based triangle filtering

- Back-face culling
 - Allows to cull triangles that face away from the viewer
 - If tessellation is used must take into account max patch height
 - Cost: → calculate the determinant of a 3x3 matrix [Olano] (homogeneous 2D coordinates)
 - Effectiveness: → high (potentially cull 50% of the geometry)



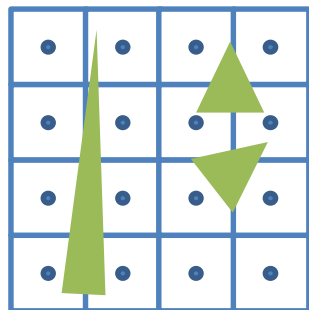
Compute-based triangle filtering

- Frustum culling
 - Allows to cull triangles that are projected outside the clipping cube
 - Takes into account near and far planes
 - Cost: → check if all vertices lie in the negative side of the clip-space cube
 - Effectiveness: → medium-high (depends on the size of the scene and eye pos)



Compute-based triangle filtering

- Small-primitives culling
 - Allows to cull triangles that are too small to be seen
 - Triangles that do not touch any sample point after projection
 - Long and thin triangles that do not touch any sample are culled as well
 - More efficient use of hardware resources
 - Cost: → triangle touches any subpixel samples
 - Effectiveness: → medium (depends on the size of the triangles and screen res)

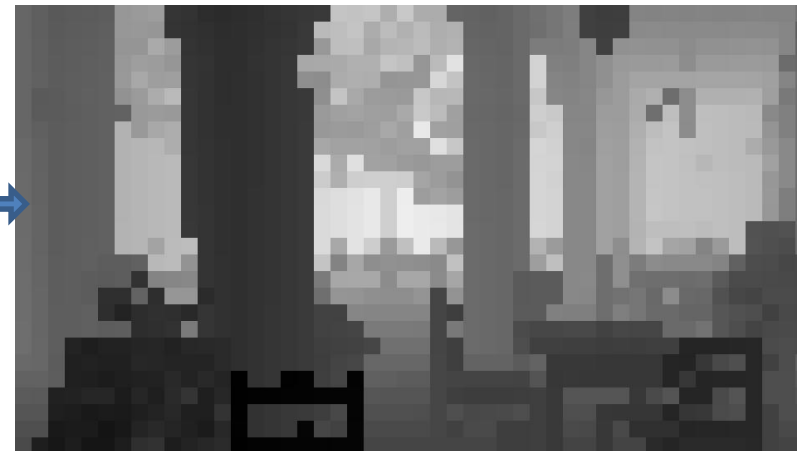


Primitive-rate bound

- only one primitive per cycle per tile can be scanned (see [Wihlidal])
- Very inefficient use of rasterization units

Compute-based triangle filtering

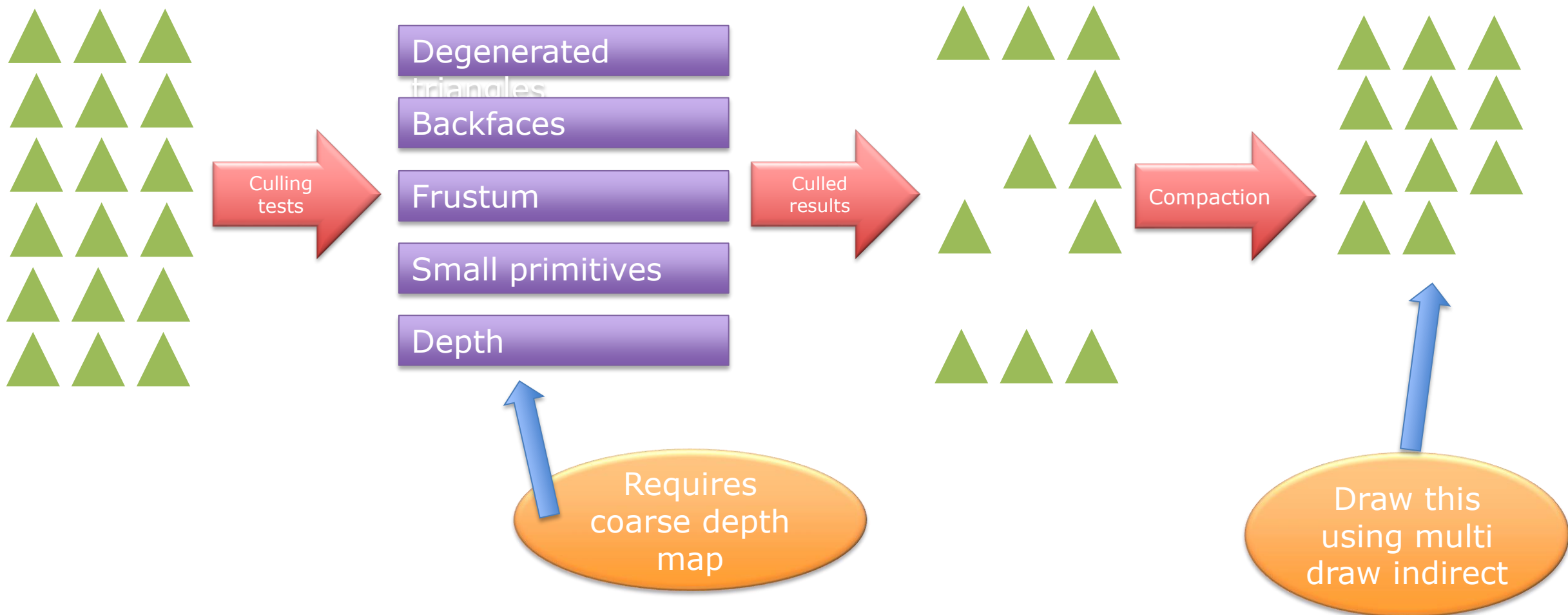
- Depth culling (not used in this demo)
 - Allows to cull triangles that are occluded by the scene
 - This test requires a coarse depth buffer
 - Cost: → load depth values from map and check triangle/BB intersection
 - Effectiveness: → medium-high (depends on scene complexity and the size of the triangles)



Can be generated by

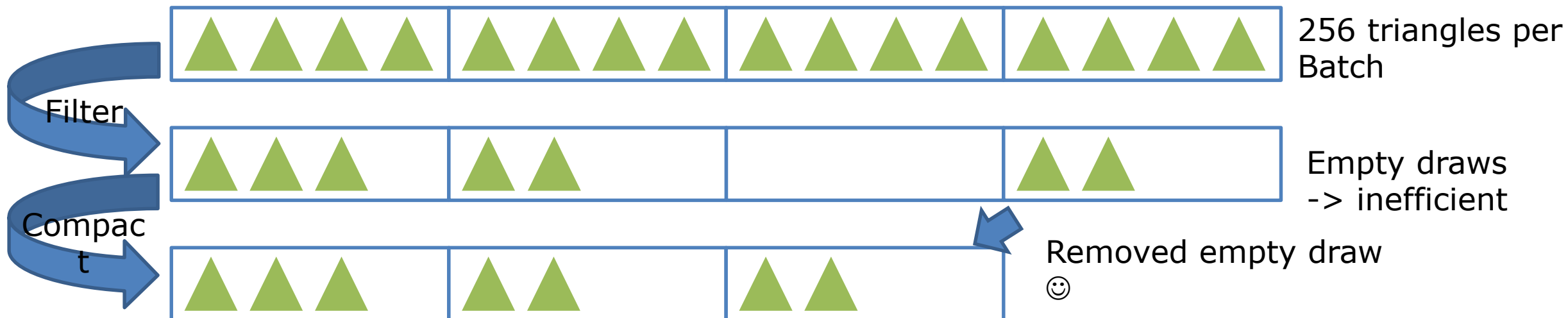
- Downsampling previous z-buffer and reprojecting depths
- Rendering selected LOD geometry at low res

Compute-based triangle filtering



Compute-based triangle filtering

- Triangle filtering is executed on groups of 256 triangles (one batch) -> empty draws
- Draw batch compaction to the rescue
 - Can be run in parallel in a compute shader
 - Eliminates empty draws from the multi indirect draw buffer



Adding Triangle/cluster filtering - Frame pseudocode

1. [CPU] Early discard invisible geometry using cluster culling
2. [CS] Generate unculled indices and multi draw indirect buffers using triangle filtering (one triangle per thread)
3. Like before

Adding Triangle/cluster filtering – Data management

- For this static scene one large vertex buffer and an index buffer generated by triangle culling and filtering
- Draw batches that hold a block of geometry each for one material
 - > Only two “materials” opaque and alpha masked, transparent objects and other materials would go into the same buffer
- For dynamic objects we would use a dedicated VB/IB pair for each; this is optional

Adding Triangle/cluster filtering – Data management

San Miguel Scene Number of Draw calls (ExecuteIndirect)

- Shadow opaque 214
- Shadow alpha masked 59
- Main view opaque 200
- Main view alpha masked 60
- Dispatch calls for filtering 81

Compute-based triangle filtering - Benefits

- Allows to cull triangles before sending them to the graphics pipeline
 - Avoid overwhelming parts of the graphics pipeline (rasterizer)
- Graphics pipeline is better utilized with the visible triangles (rasterizer efficiency, command processor,...)
- Can make use of async compute to potentially overlap with the graphics pipeline

Re-using triangle filtered results for multiple Views / Rendering Passes

Motivation

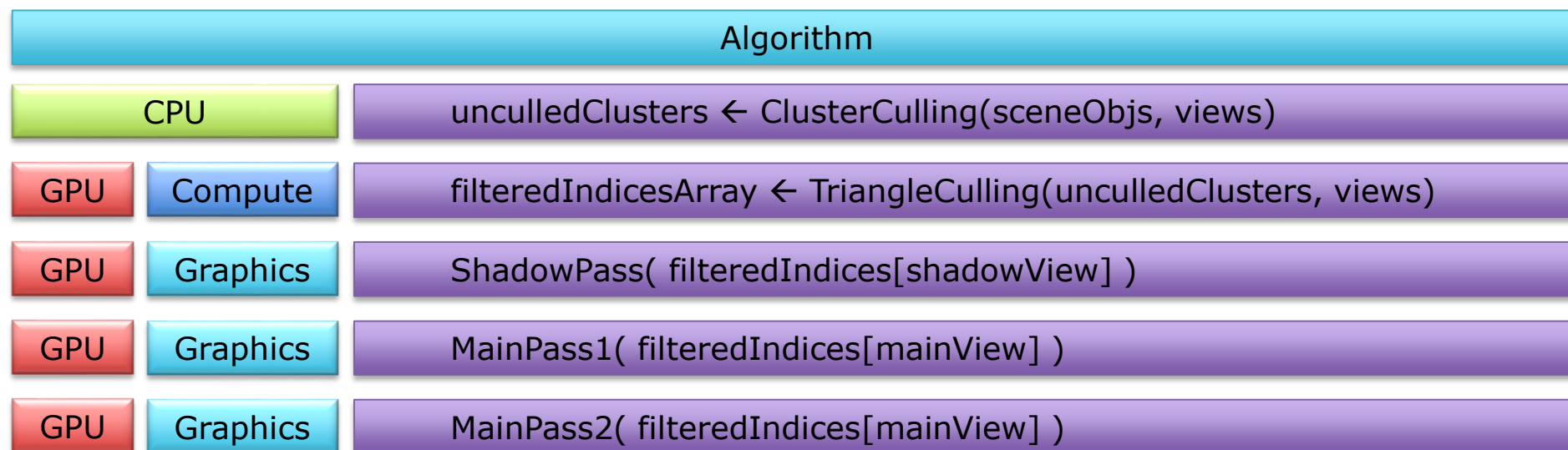
- Compute-based triangle filtering comes at a cost
 - For every triangle:→ load indices and vertices, transform vertices, append (lock) triangle data to index buffer
- We came up with the idea to generate filtered data for several rendering passes like main view, shadows etc.
 - Use filtered data to cull the same triangle set from different views
 - Load indices/vertices, transform vertices only once for all views

Motivation

- Reduces the effectivity of cluster culling / triangle filtering
 - harder to cull cluster for N views
 - however, it was worth it: 😊

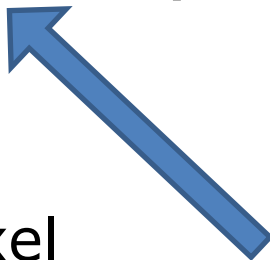
Use filtered data to cull triangles from different views

- The algorithm is generalized to test against different N-views
 - Load indices / vertices once, transform vertices for every view



Adding re-usage of triangle filtered data - Frame pseudocode

1. [CPU] Early discard geometry not visible from any view using cluster culling
2. [CS] Generate N index and N multi draw indirect buffers using triangle filtering testing against the N views (one triangle per thread)
3. For each *i* view use (*ith* index buffer and *ith* MDI buffer):
 1. [Gfx] Clear visibility and depth buffers
 2. [VS,PS] Visibility buffer pass
[PS] Output triangle / instance IDs
 3. [PS] Interpolate attributes from gradients and shade pixel



* Using a dedicated Visibility Buffer for shadow pass is overkill, but you can still use the filtered data for it.

San Miguel Scene average for main view

Results

- ▶ 8 Million triangles
- ▶ 5 Million vertices

Total triangles	Rendered	Culled		
8,010,146	851,517 (10.6%)	7,158,629 (89.4%)	3,185,203 (39.8%)	Back-face
			5,244,787 (65.5%)	Frustum
			1,950,030 (24.4%)	Small primitives

Visibility Buffer 1080p

GPU	Culling	Shadow Map	Fill VB	HDAO	Shade VB	Resolve MSAA	UI	Overall
Visibility Buffer 1080p – No MSAA								
AMD RADEON R9 380	2.59	1.42	2.79	0.66	1.02	-	0.02	8.57
NVIDIA GeForce GTX 970	3.29	1.13	1.69	0.47	1.01	-	0.02	7.68
Visibility Buffer 1080p – No MSAA No Culling								
AMD RADEON R9 380	-	5.17	7.58	0.66	1.02	-	0.02	14.52
NVIDIA GeForce GTX 970	-	3.61	4.21	0.47	1.00	-	0.02	9.44
Visibility Buffer 1080p – 2x MSAA								
AMD RADEON R9 380	2.60	1.42	3.41	0.96	2.33	0.59	0.03	11.44
NVIDIA GeForce GTX 970	3.39	1.14	2.13	0.92	1.81	0.09	0.02	9.63
Visibility Buffer 1080p – 4x MSAA								
AMD RADEON R9 380	2.66	1.45	4.27	1.44	4.40	0.94	0.03	15.27
NVIDIA GeForce GTX 970	3.20	1.14	3.02	1.32	3.44	0.23	0.02	12.47

Deferred Shading 1080p

GPU	Culling	Shadow Map	Fill Buffer	HDAO	Shade Buffer	Resolve MSAA	UI	Overall
Deferred Shading 1080p – No MSAA								
AMD RADEON R9 380	2.60	1.44	4.56	0.66	0.38	-	0.02	9.75
NVIDIA GeForce GTX 970	3.34	1.09	3.37	0.47	0.33	-	0.02	8.72
Deferred Shading 1080p – No MSAA No Culling								
AMD RADEON R9 380	-	5.17	7.86	0.66	0.38	-	0.02	14.16
NVIDIA GeForce GTX 970	-	3.67	5.74	0.47	0.33	-	0.02	10.30
Deferred Shading 1080p – 2x MSAA								
AMD RADEON R9 380	2.63	1.44	7.60	0.96	2.82	0.59	0.02	16.16
NVIDIA GeForce GTX 970	3.25	1.14	5.36	0.92	0.64	0.09	0.02	11.58
Deferred Shading 1080p – 4x MSAA								
AMD RADEON R9 380	2.70	1.48	12.95	1.45	5.27	0.94	0.02	24.90
NVIDIA GeForce GTX 970	3.39	1.14	9.39	1.31	1.41	0.23	0.02	17.00

Visibility Buffer 1440p

GPU	Culling	Shadow Map	Fill VB	HDAO	Shade VB	Resolve MSAA	UI	Overall
Visibility Buffer 1440p – No MSAA								
AMD RADEON R9 380	2.72	1.52	3.47	1.17	1.73	-	0.03	10.72
NVIDIA GeForce GTX 970	3.27	1.09	2.00	0.78	1.55	-	0.02	8.83
Visibility Buffer 1440p – No MSAA No Culling								
AMD RADEON R9 380	-	5.17	7.87	1.10	1.63	-	0.02	15.86
NVIDIA GeForce GTX 970	-	3.74	4.39	0.78	1.56	-	0.02	10.64
Visibility Buffer 1440p – 2x MSAA								
AMD RADEON R9 380	2.85	1.64	4.78	1.82	4.17	1.00	0.02	16.38
NVIDIA GeForce GTX 970	3.36	1.10	2.96	1.54	2.89	0.15	0.02	12.21
Visibility Buffer 1440p – 4x MSAA								
AMD RADEON R9 380	2.66	1.43	5.50	2.42	7.13	1.57	0.03	20.82
NVIDIA GeForce GTX 970	3.39	1.09	5.61	2.46	5.61	0.38	0.02	17.47

Deferred Shading 1440p

GPU	Culling	Shadow Map	Fill Buffer	HDAO	Shade Buffer	Resolve MSAA	UI	Overall
Deferred Shading 1440p – No MSAA								
AMD RADEON R9 380	2.61	1.42	6.44	1.10	0.63	-	0.02	12.30
NVIDIA GeForce GTX 970	3.47	1.09	4.65	0.78	0.55	-	0.02	10.67
Deferred Shading 1440p – No MSAA No Culling								
AMD RADEON R9 380	-	5.17	9.67	1.10	0.63	-	0.02	16.66
NVIDIA GeForce GTX 970	-	3.63	6.89	0.78	0.54	-	0.02	11.91
Deferred Shading 1440p – 2x MSAA								
AMD RADEON R9 380	2.67	1.46	11.49	1.62	4.73	1.00	0.03	23.09
NVIDIA GeForce GTX 970	3.32	1.10	7.86	1.54	1.08	0.15	0.02	15.23
Deferred Shading 1440p – 4x MSAA								
AMD RADEON R9 380	2.67	1.42	19.42	2.43	8.75	1.57	0.03	36.37
NVIDIA GeForce GTX 970	3.35	1.10	14.98	2.46	2.36	0.38	0.02	24.76

Visibility Buffer 3840 x2160

GPU	Culling	Shadow Map	Fill VB	HDAO	Shade VB	Resolve MSAA	UI	Overall
Visibility Buffer 4k – No MSAA								
AMD RADEON R9 380	2.66	1.42	5.07	2.51	3.43	-	0.02	15.19
NVIDIA GeForce GTX 970	3.46	1.12	3.36	1.82	3.29	-	0.02	13.52
Visibility Buffer 4k – No MSAA No Culling								
AMD RADEON R9 380	-	5.18	9.25	2.51	3.43	-	0.02	20.45
NVIDIA GeForce GTX 970	-	3.74	5.31	1.79	3.25	-	0.02	14.39
Visibility Buffer 4k – 2x MSAA								
AMD RADEON R9 380	2.72	1.42	8.27	3.65	8.27	2.29	0.03	25.87
NVIDIA GeForce GTX 970	3.34	1.09	6.17	3.58	6.17	0.34	0.02	19.87
Visibility Buffer 4k – 4x MSAA								
AMD RADEON R9 380	2.70	1.43	8.73	5.70	15.58	3.61	0.03	37.86
NVIDIA GeForce GTX 970	3.37	1.11	7.86	6.86	12.35	0.87	0.02	32.68

Deferred Shading 3840 x2160

GPU	Culling	Shadow Map	Fill Buffer	HDAO	Shade Buffer	Resolve MSAA	UI	Overall
Deferred Shading 4k – No MSAA								
AMD RADEON R9 380	2.67	1.42	12.19	2.51	1.29	-	0.03	20.19
NVIDIA GeForce GTX 970	3.36	1.20	9.04	1.79	1.21	-	0.02	16.82
Deferred Shading 4k – No MSAA No Culling								
AMD RADEON R9 380	-	5.18	15.00	2.51	1.29	-	0.03	24.06
NVIDIA GeForce GTX 970	-	3.75	10.44	1.79	1.22	-	0.02	17.49
Deferred Shading 4k – 2x MSAA								
AMD RADEON R9 380	2.70	1.42	21.65	3.65	10.86	2.29	0.03	42.68
NVIDIA GeForce GTX 970	3.35	1.10	15.36	3.59	2.41	0.34	0.02	26.27
Deferred Shading 4k – 4x MSAA								
AMD RADEON R9 380	2.72	1.44	35.88	5.74	20.13	3.60	0.02	69.64
NVIDIA GeForce GTX 970	3.40	1.18	30.29	6.87	5.39	0.87	0.02	48.12

Visibility Buffer 1080p

GPU	Culling	Shadow Map	Fill VB	HDAO	Shade VB	Resolve MSAA	UI	Overall
	Visibility Buffer 1080p – No MSAA							
Xbox One	7.19	3.32	3.90	1.73	3.90	-	0.02	19.78
	Visibility Buffer 1080p – No MSAA No Culling							
Xbox One	-	9.16	9.09	1.73	3.89	-	0.02	23.98
	Visibility Buffer 1080p – 2x MSAA							
Xbox One	7.28	3.20	7.57	5.17	7.57	0.46	0.02	28.00

Deferred Shading 1080p

GPU	Culling	Shadow Map	Fill Buffer	HDAO	Shade Buffer	Resolve MSAA	UI	Overall
	Deferred Shading 1080p – No MSAA							
Xbox One	7.19	3.14	11.18	1.74	1.38	-	0.02	24.77
	Deferred Shading 1080p – No MSAA No Culling							
Xbox One	-	9.09	15.07	1.73	1.39	-	0.02	27.45
	Deferred Shading 1080p – 2x MSAA							
Xbox One	7.21	3.18	21.85	5.18	8.46	0.47	0.02	46.41

Summary

Deferred Shading

GPU AMD RADEON R9 380	1080p	1440p	2160p
No MSAA	9.75	12.30	20.19
No MSAA – No Culling	14.16	16.66	24.06
2x MSAA	16.16	23.09	42.68
4x MSAA	24.90	36.37	69.64

NVIDIA GeForce GTX 970	1080p	1440p	2160p
No MSAA	8.72	10.67	16.82
No MSAA – No Culling	10.30	11.91	17.49
2x MSAA	11.58	15.23	26.27
4x MSAA	17.00	24.76	48.12

Xbox One	1080p	1440p	2160p
No MSAA	24.77	-	-
No MSAA – No Culling	27.45	-	-
2x MSAA	46.41	-	-

Visibility Buffer

GPU AMD RADEON R9 380	1080p	1440p	2160p
No MSAA	8.57	10.72	15.19
No MSAA – No Culling	14.52	15.86	20.45
2x MSAA	11.44	16.38	25.87
4x MSAA	15.27	20.82	37.86

NVIDIA GeForce GTX 970	1080p	1440p	2160p
No MSAA	7.68	8.83	13.52
No MSAA – No Culling	9.44	10.64	14.39
2x MSAA	9.63	12.21	19.87
4x MSAA	12.47	17.47	32.68

Xbox One	1080p	1440p	2160p
No MSAA	19.78	-	-
No MSAA – No Culling	23.98	-	-
2x MSAA	28.00	-	-

How about VR?

- We are working on the StarVR SDK. StarVR uses a large field of view with very high resolution
- The Visibility Buffer will help substantially with performance here
 - We can cull and prepare the data for all views and the shadow map views in one go
 - We render forward+ and therefore do not have to deal with transparency issues

Executive Summary

We built a rendering system that

- Cluster culls and filters triangles for different views like main view, shadow view, reflection view, GI view etc.
- The optimized triangles are used to fill a screen-space Visibility Buffer or more Visibility Buffers for more views
- We then render lights, shadows, bounce lights with the optimized geometry based on visibility
- We can differ between visibility of geometry and shading frequency
- We can light per triangle or in so called object space

Future work

- Re-use culled triangles over several frames
- Use intrinsics for several parts of the pipeline [Chajdas Compaction]
- Better improve asynchronous scheduling
 - Async compute is powerful
- Add a Forward+ or Tiled-Forward lighting system
- @SebAaltonen 's trick for using MSAA to reduce bandwidth on XBox One

Source Code

- Free source code in two weeks time (after vacation of some of the guys who worked on this 😊) -> clean-up
- Probably on github if we can fit everything on there 😊 (1 GB limit)
- If you want the code now, send me an e-mail ...

Credits

- Christoph Schied – wrote implementation of his paper with the OpenGL 4.5 run-time at our office
- Confetti People
 - Marijn Tamis – wrote the initial OpenGL 4.5 run-time
 - Leroy Sikkes – wrote the initial DirectX 12 run-time and added hardware performance counters
 - Max Oomen (intern) added linear lighting and fixed many bugs
 - Jesús Gumbau – added triangle filtering, came up with the idea and implemented re-usage of filtered triangle data and made it cross-platform running on NVIDIA and AMD GPUs and then brought it to DirectX 12
 - Jordan Logan – brought it to XBOX One and optimized for this console

Acknowledgements

- Graham Wihlidal DICE Frostbite
- Nicolas Thibieroz, Gareth Thomas, Matthaeus Chajdas, Steven Tovey AMD
- Mike Acton Insomniac
- James McLaren, Q-Games
- Remi Arnaud, Starbreeze / StarVR
- Kev Gee Microsoft

References

- [Burns] Christopher A. Burns, Warren A. Hunt "The Visibility Buffer: A Cache-Friendly Approach to Deferred Shading" Journal of Computer Graphics Techniques (JCGT) 2:2 (2013), 55- 69. Available online at <http://jcgt.org/published/0002/02/04>
- [Chajdas] Matthaeus Chajdas "GeometryFX" <http://gpuopen.com/gaming-product/geometryfx/>
- [Chajdas Compaction] Matthaeus Chajdas "Fast compaction with mbcnt", <http://gpuopen.com/fast-compaction-with-mbcnt/>
- [Edge] Edge Library, PS3 SDK
- [Engel2009] Wolfgang Engel, "Light Pre-Pass", "Advances in Real-Time Rendering in 3D Graphics and Games", SIGGRAPH 2009, http://halo.bungie.net/news/content.aspx?link=Siggraph_09
- [Lagarde] Sebastien Lagarde, Charles de Rousiers, "Moving Frostbite to Physically Based Rendering", Course notes SIGGRAPH 2014
- [Olano] Marc Olano, <http://www.cs.unc.edu/~olano/papers/2dh-tri/2dh-tri.pdf>
- [Schied] Christoph Schied, Carten Dachsbacher "Deferred Attribute Interpolation Shading", GPU Pro 7, CRC Press / shorter and free version: <http://cg.ivd.kit.edu/publications/2015/dais/DAIS.pdf>
- [Wihlidal] Graham Wihlidal, "Optimizing the Graphics Pipeline with Compute", GDC 2016, <http://www.frostbite.com/2016/03/optimizing-the-graphics-pipeline-with-compute/>

wolf@conffx.com