# Advancements in Tiled-Based Compute Rendering

**Gareth Thomas**
Developer Technology Engineer, AMD

GAME DEVELOPERS CONFERENCE®
MOSCONE CENTER · SAN FRANCISCO, CA
MARCH 2-6, 2015 · EXPO: MARCH 4-6, 2015

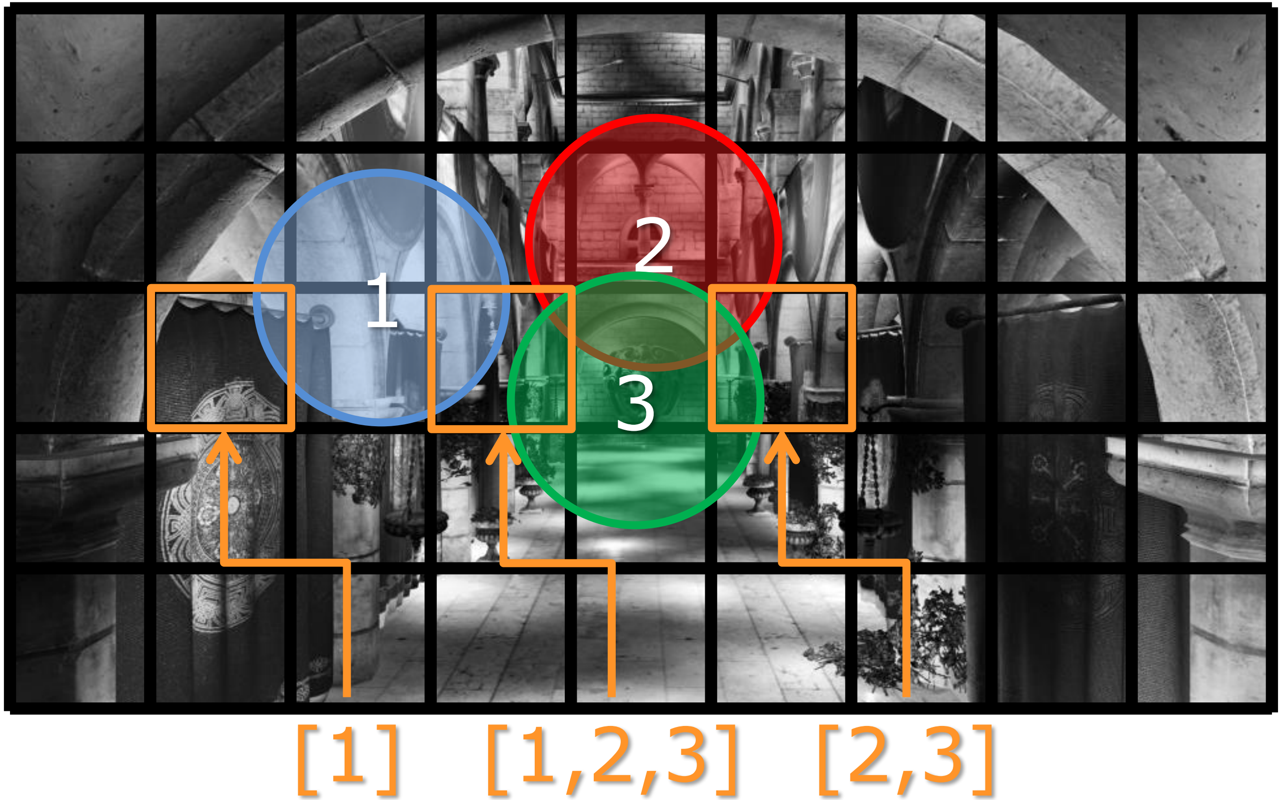
# Agenda

- Current Tech
- Culling Improvements
- Clustered Rendering
- Summary

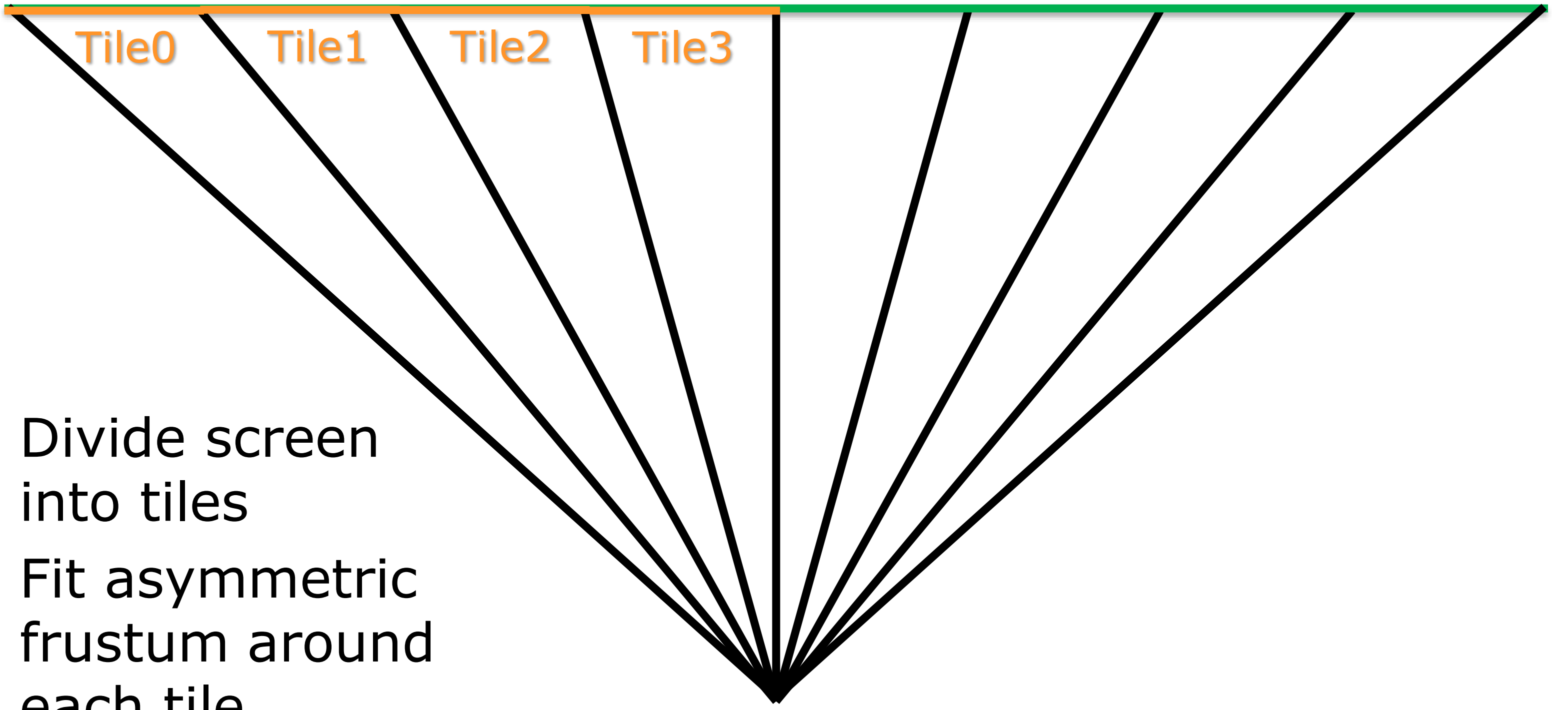
# Proven Tech – Out in the Wild

- Tiled Deferred [Andersson09]
  - Frostbite
  - UE4
  - Ryse
- Forward+ [Harada et al 12]
  - DiRT & GRID Series
  - The Order: 1886
  - Ryse

# Tiled Rendering 101

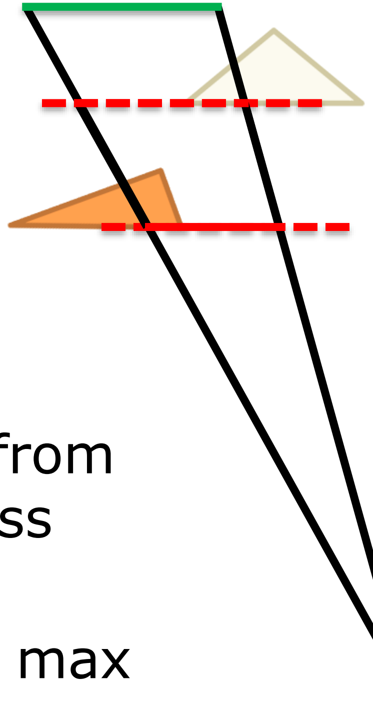


# Tiled Rendering 101




- Divide screen into tiles
- Fit asymmetric frustum around each tile

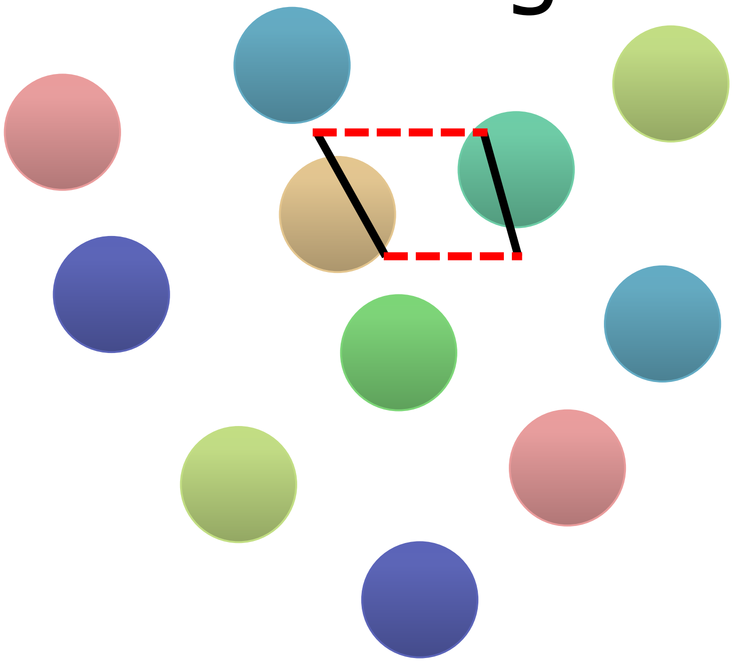
# Tiled Rendering 101



- Use z buffer from depth pre-pass as input
- Find min and max depth per tile

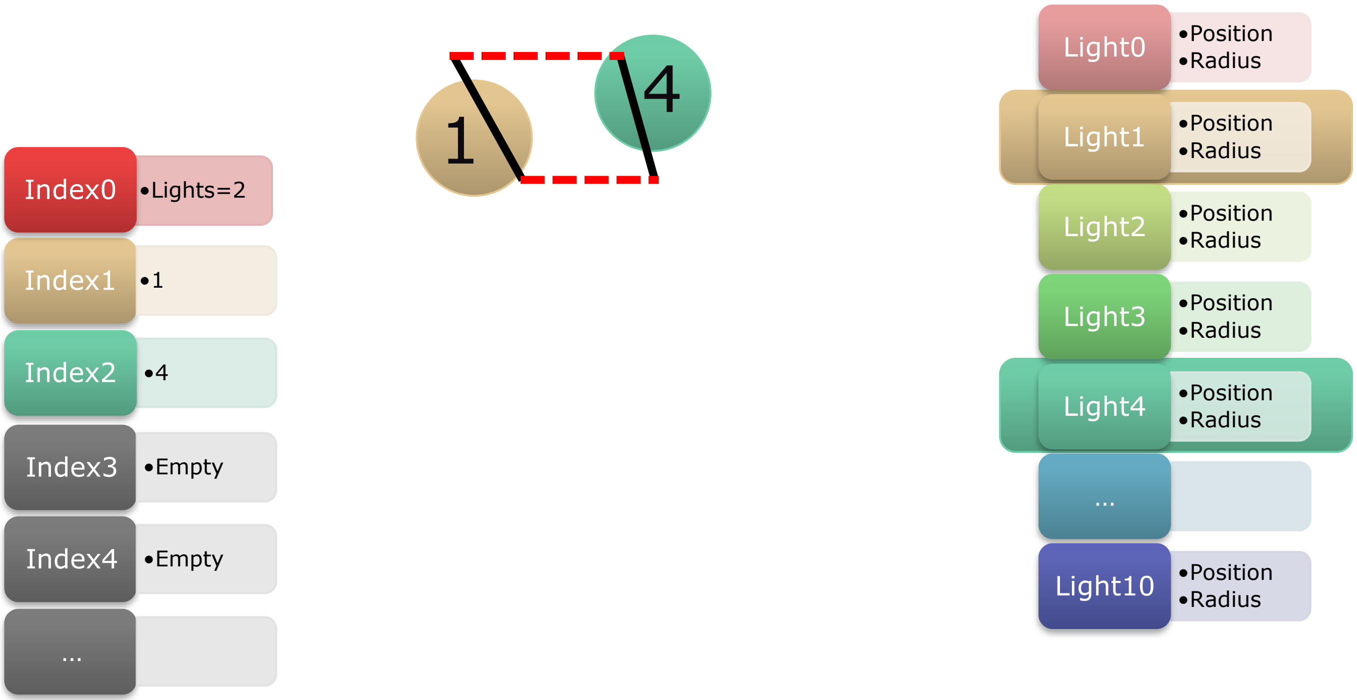
- Use this frustum for intersection testing

# Tiled Rendering 101



| | |
|---|---|
| Light0 | •Position<br>•Radius |
| Light1 | •Position<br>•Radius |
| Light2 | •Position<br>•Radius |
| Light3 | •Position<br>•Radius |
| Light4 | •Position<br>•Radius |
| … | |
| Light10 | •Position<br>•Radius |

# Tiled Rendering 101



# Targets for Improvement

- Z Prepass (on Forward+)
- Depth bounds
- Light Culling
- Color Pass

# Depth Bounds

- Determine min and max bounds of the depth buffer on a per tile basis
- Atomic Min Max [Andersson09]

```
groupshared uint ldsZMin;
groupshared uint ldsZMax;

[numthreads(16, 16, 1)]
void CalculateDepthBoundsCS( uint3 globalIdx : SV_DispatchThreadID, uint3 localIdx : SV_GroupThreadID )
{
    uint localIdxFlattened = localIdx.x + localIdx.y*16;

    if( localIdxFlattened == 0 )
    {
        ldsZMin = 0x7f7fffff;  // FLT_MAX as a uint
        ldsZMax = 0;
    }

    GroupMemoryBarrierWithGroupSync();

    float depth = g_DepthTexture.Load( uint3(globalIdx.x,globalIdx.y,0) ).x;  // read one depth sample per thread

    uint z = asuint( ConvertProjDepthToView( depth ) );  // reinterpret as uint

    if( depth != 0.0 )
    {
        InterlockedMax( ldsZMax, z );  // atomic min & max
        InterlockedMin( ldsZMin, z );
    }

    GroupMemoryBarrierWithGroupSync();

    float maxZ = asfloat( ldsZMax );  // reinterpret back to float
    float minZ = asfloat( ldsZMin );
}
```

# Parallel Reduction

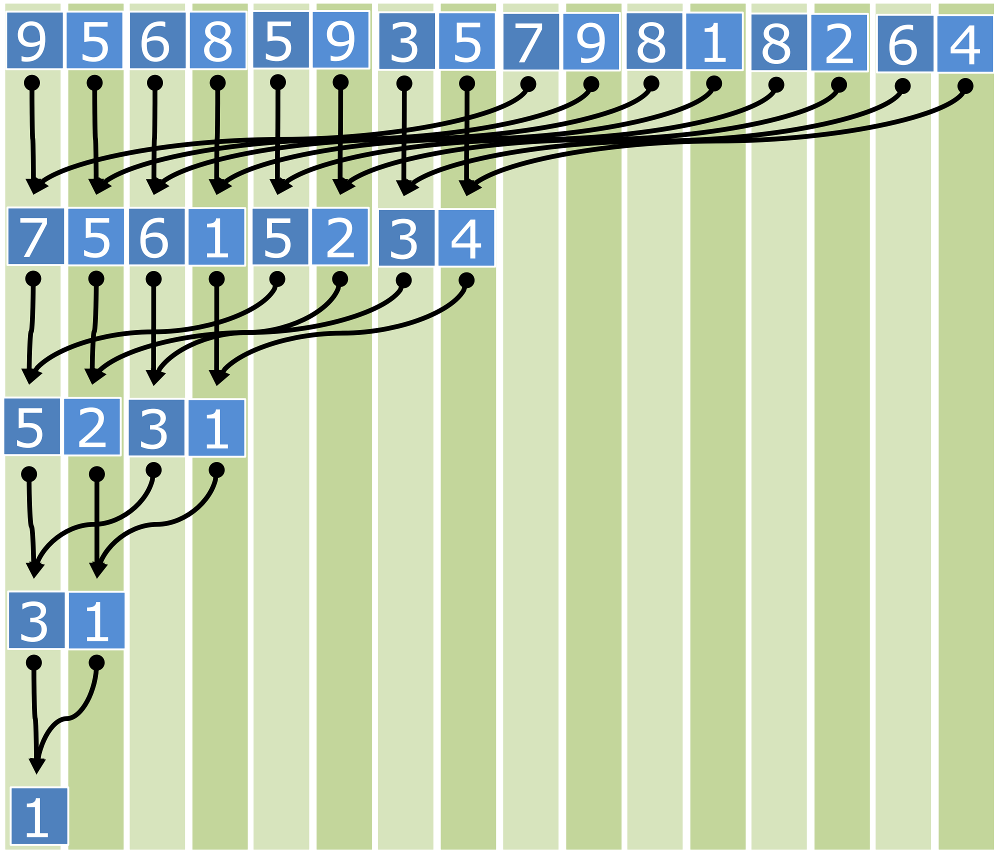
- Atomics are useful but not efficient
- Compute-friendly algorithm
- Great material already available:
  - "Optimizing Parallel Reduction in CUDA" [Harris07]
  - "Compute Shader Optimizations for AMD GPUs: Parallel Reduction" [Engel14]

depth[tid] = min(depth[tid],depth[tid+8])

depth[tid] = min(depth[tid],depth[tid+4])

depth[tid] = min(depth[tid],depth[tid+2])

depth[tid] = min(depth[tid],depth[tid+1])



# Implementation details

- First pass reads 4 depth samples
- Needs to be separate pass
- Write bounds to UAV
    - Maybe useful for other things too

```
groupshared float ldsZMin[64];
groupshared float ldsZMax[64];

[numthreads(8,8,1)]
void CalculateDepthBoundsCS( uint3 globalIdx : SV_DispatchThreadID, uint3 localIdx : SV_GroupThreadID, uint3 groupIdx : SV_GroupID )
{
    uint2 sampleIdx = globalIdx.xy*2;

    float depth00 = g_SceneDepthBuffer.Load(uint3(sampleIdx.x,   sampleIdx.y,   0)).x;        float viewPosZ00 = ConvertProjDepthToView(depth00);
    float depth01 = g_SceneDepthBuffer.Load(uint3(sampleIdx.x,   sampleIdx.y+1,0)).x;        float viewPosZ01 = ConvertProjDepthToView(depth01);
    float depth10 = g_SceneDepthBuffer.Load(uint3(sampleIdx.x+1,sampleIdx.y,   0)).x;        float viewPosZ10 = ConvertProjDepthToView(depth10);
    float depth11 = g_SceneDepthBuffer.Load(uint3(sampleIdx.x+1,sampleIdx.y+1,0)).x;        float viewPosZ11 = ConvertProjDepthToView(depth11);

    float minZ00 = (depth00 != 0.f) ? viewPosZ00 : FLT_MAX;                                 float maxZ00 = (depth00 != 0.f) ? viewPosZ00 : 0.0f;
    float minZ01 = (depth01 != 0.f) ? viewPosZ01 : FLT_MAX;                                 float maxZ01 = (depth01 != 0.f) ? viewPosZ01 : 0.0f;
    float minZ10 = (depth10 != 0.f) ? viewPosZ10 : FLT_MAX;                                 float maxZ10 = (depth10 != 0.f) ? viewPosZ10 : 0.0f;
    float minZ11 = (depth11 != 0.f) ? viewPosZ11 : FLT_MAX;                                 float maxZ11 = (depth11 != 0.f) ? viewPosZ11 : 0.0f;

    uint threadNum = localIdx.x + localIdx.y*8;

    ldsZMin[threadNum] = min(minZ00,min(minZ01,min(minZ10,minZ11)));
    ldsZMax[threadNum] = max(maxZ00,max(maxZ01,max(maxZ10,maxZ11)));

    GroupMemoryBarrierWithGroupSync();

    if (threadNum < 32)
    {
        ldsZMin[threadNum] = min(ldsZMin[threadNum],ldsZMin[threadNum+32]);                 ldsZMax[threadNum] = max(ldsZMax[threadNum],ldsZMax[threadNum+32]);
        ldsZMin[threadNum] = min(ldsZMin[threadNum],ldsZMin[threadNum+16]);                 ldsZMax[threadNum] = max(ldsZMax[threadNum],ldsZMax[threadNum+16]);
        ldsZMin[threadNum] = min(ldsZMin[threadNum],ldsZMin[threadNum+8]);                  ldsZMax[threadNum] = max(ldsZMax[threadNum],ldsZMax[threadNum+8]);
        ldsZMin[threadNum] = min(ldsZMin[threadNum],ldsZMin[threadNum+4]);                  ldsZMax[threadNum] = max(ldsZMax[threadNum],ldsZMax[threadNum+4]);
        ldsZMin[threadNum] = min(ldsZMin[threadNum],ldsZMin[threadNum+2]);                  ldsZMax[threadNum] = max(ldsZMax[threadNum],ldsZMax[threadNum+2]);
        ldsZMin[threadNum] = min(ldsZMin[threadNum],ldsZMin[threadNum+1]);                  ldsZMax[threadNum] = max(ldsZMax[threadNum],ldsZMax[threadNum+1]);
    }

    GroupMemoryBarrierWithGroupSync();

    if(threadNum == 0)
    {
        g_DepthBounds[groupIdx.xy] = float2(ldsZMin[0],ldsZMax[0]);
    }
}
```

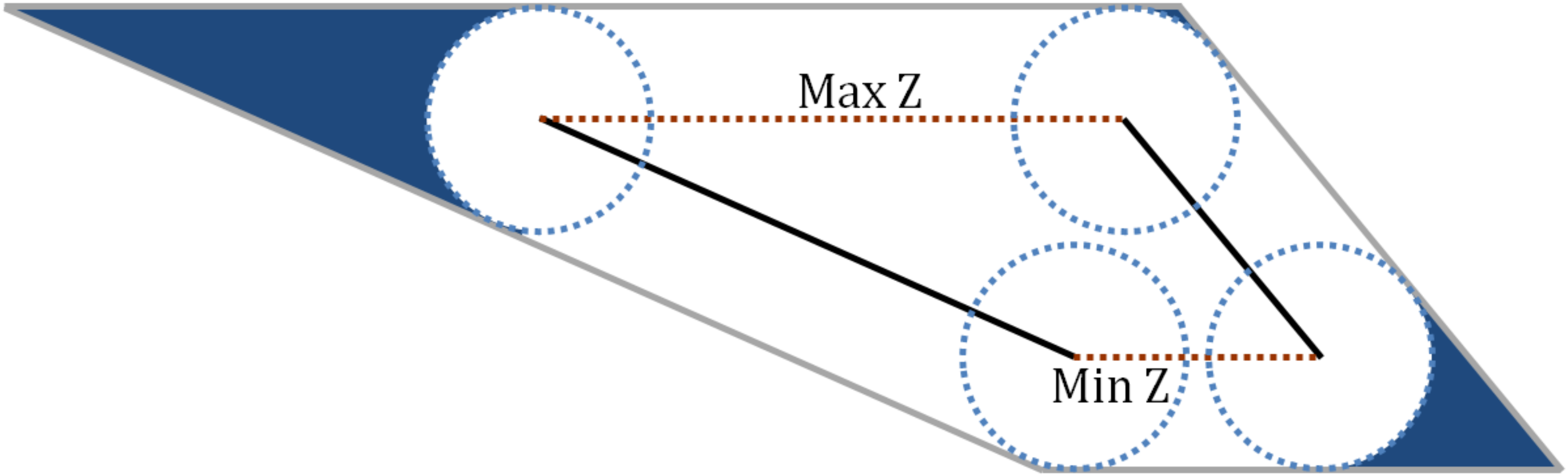
# Parallel Reduction - Performance

| | Atomic Min/Max | Parallel Reduction |
|---|---|---|
| AMD R9 290X | 1.8ms | 1.60ms |
| NVIDIA GTX 980 | 1.8ms | 1.54ms |

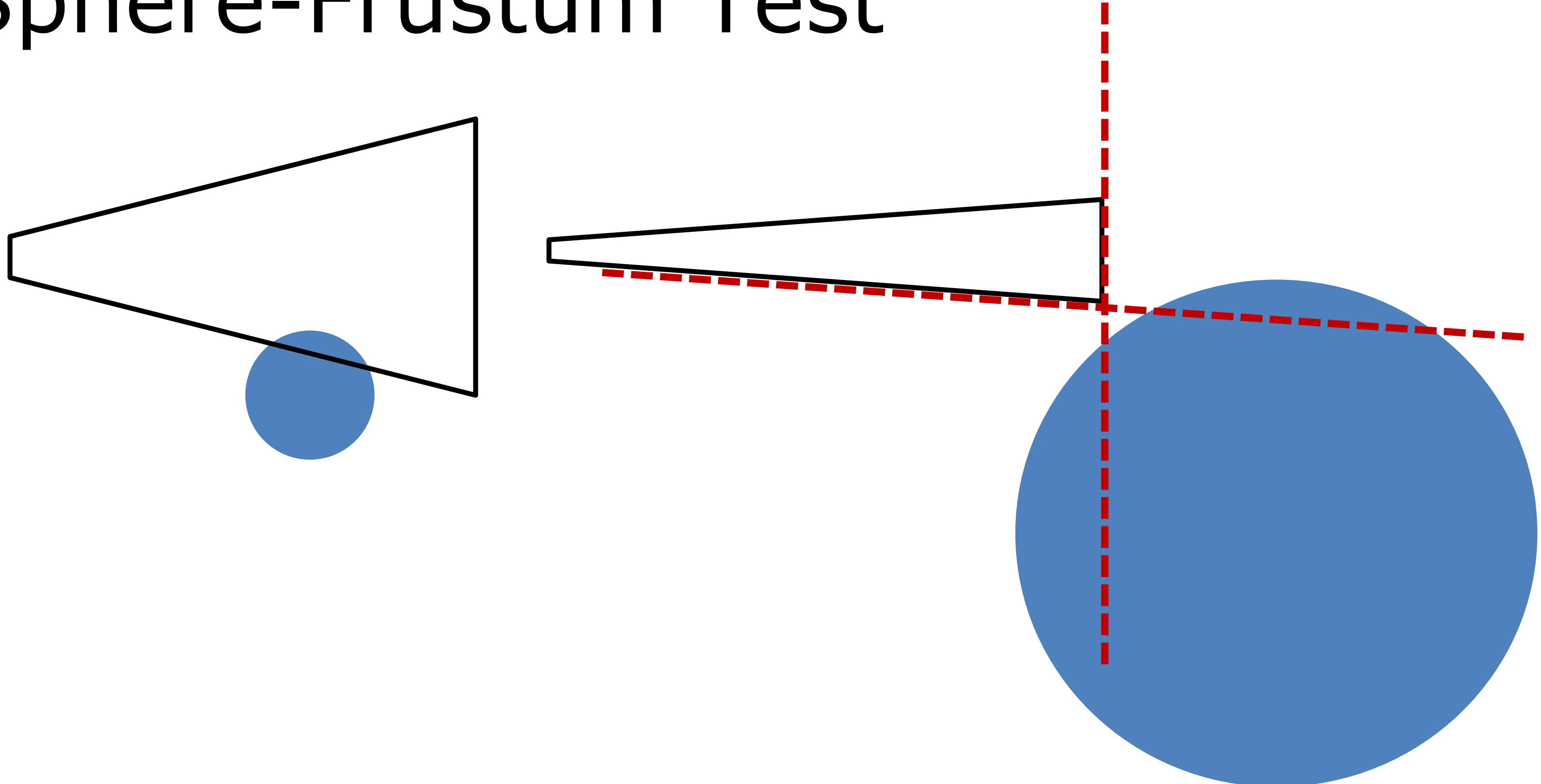
- Combined cost of depth bounds and light culling of 2048 lights at 3840x2160
- Parallel reduction pass takes ~0.35ms
- Faster than Atomic Min/Max on the GPUs tested

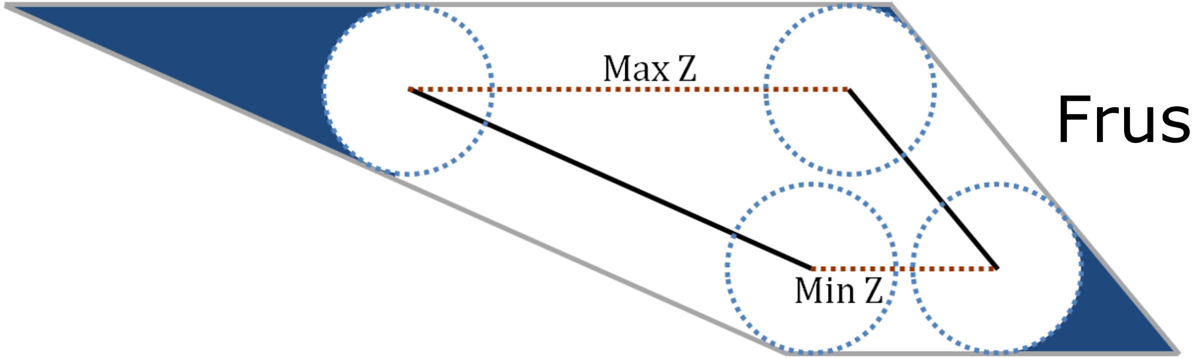
# Light Culling:
# The Intersection Test

# Sphere-Frustum Test



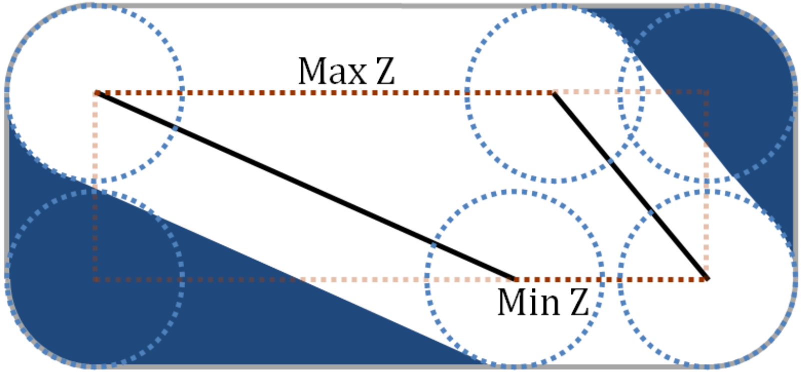
# Sphere-Frustum Test

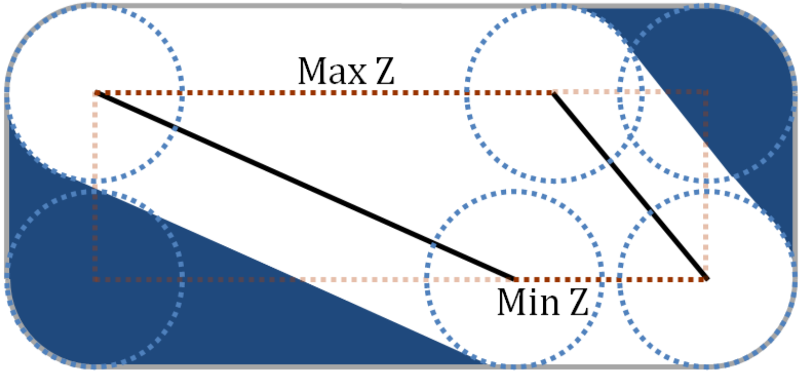




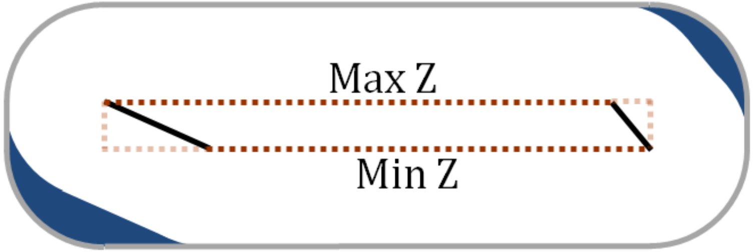

Frustum planes

AABB around Frustum






AABB around long frustum

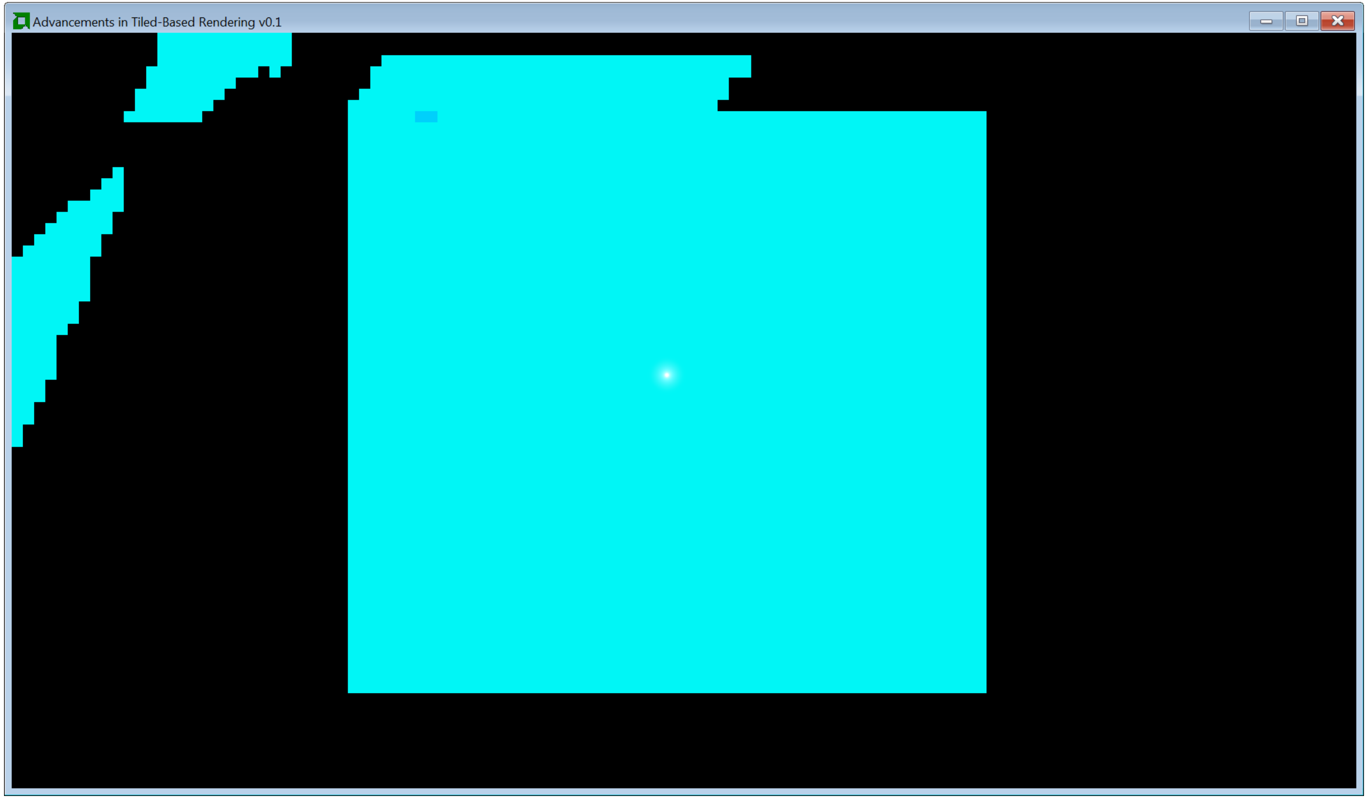



AABB around short frustum

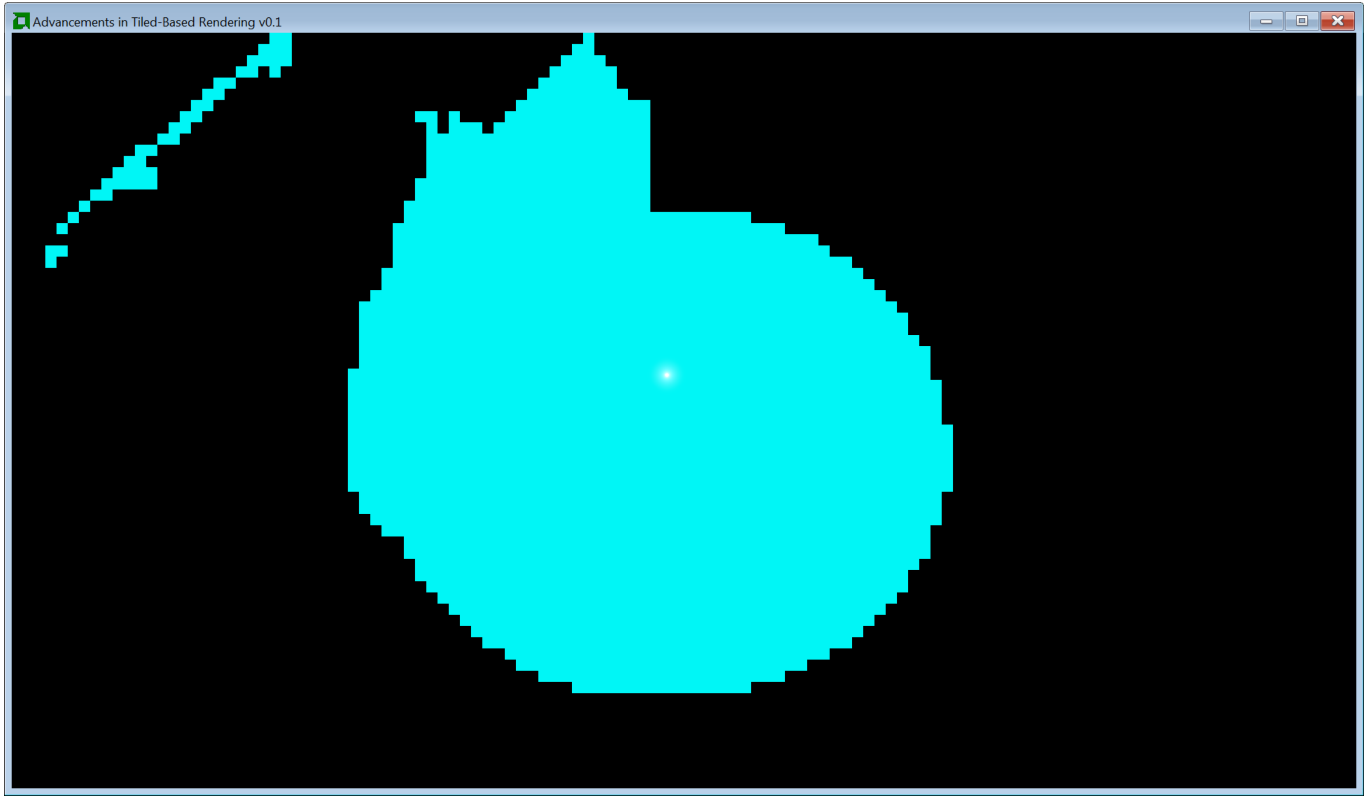
# Arvo Intersection Test [Arvo90]

```
bool TestSphereVsAABB(float3 sphereCenter, float sphereRadius, float3 AABBCenter, float3 AABBHalfSize)
{
    float3 delta = max(0, abs(AABBCenter - sphereCenter) - AABBHalfSize);
    float distSq = dot(delta, delta);
    return distSq <= sphereRadius * sphereRadius;
}
```


Single Point Light



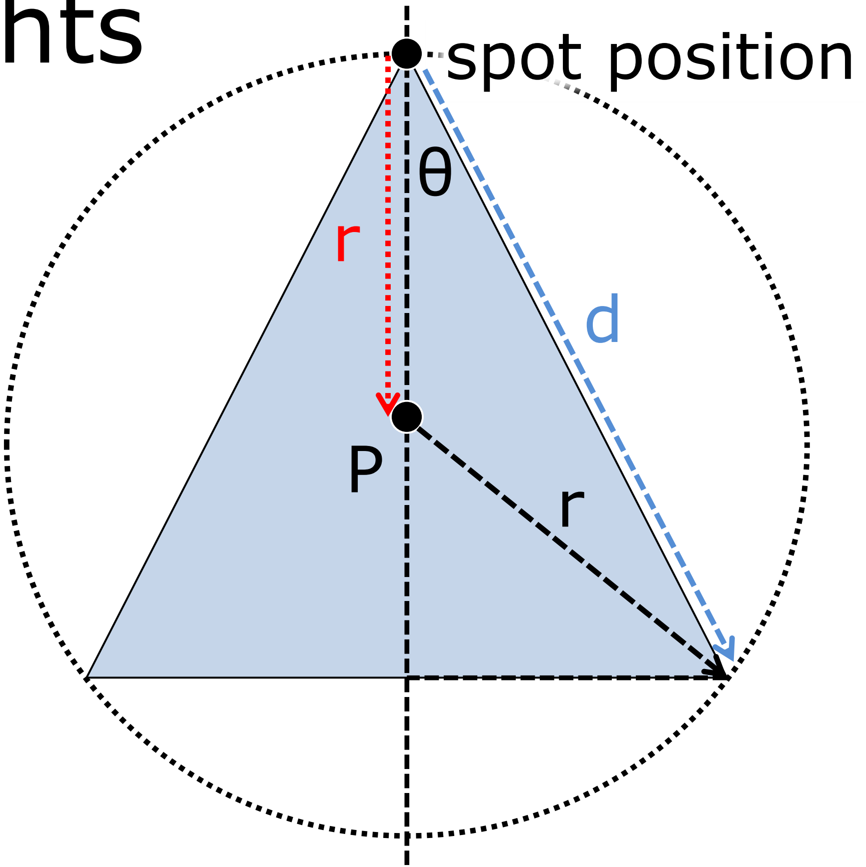

Frustum/Sphere Test



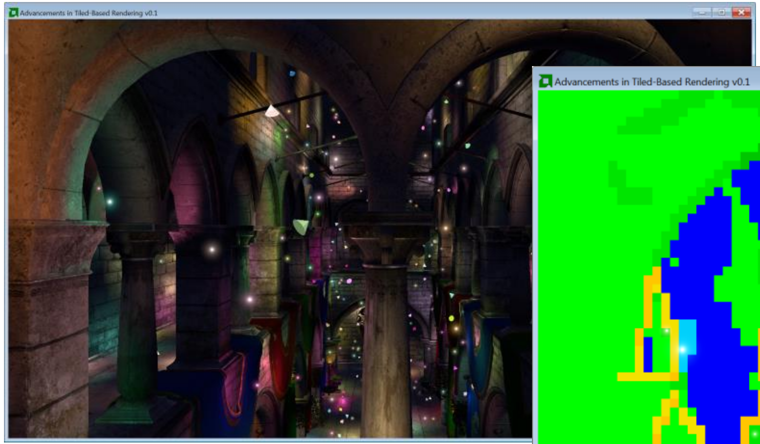

Arvo AABB/Sphere Test

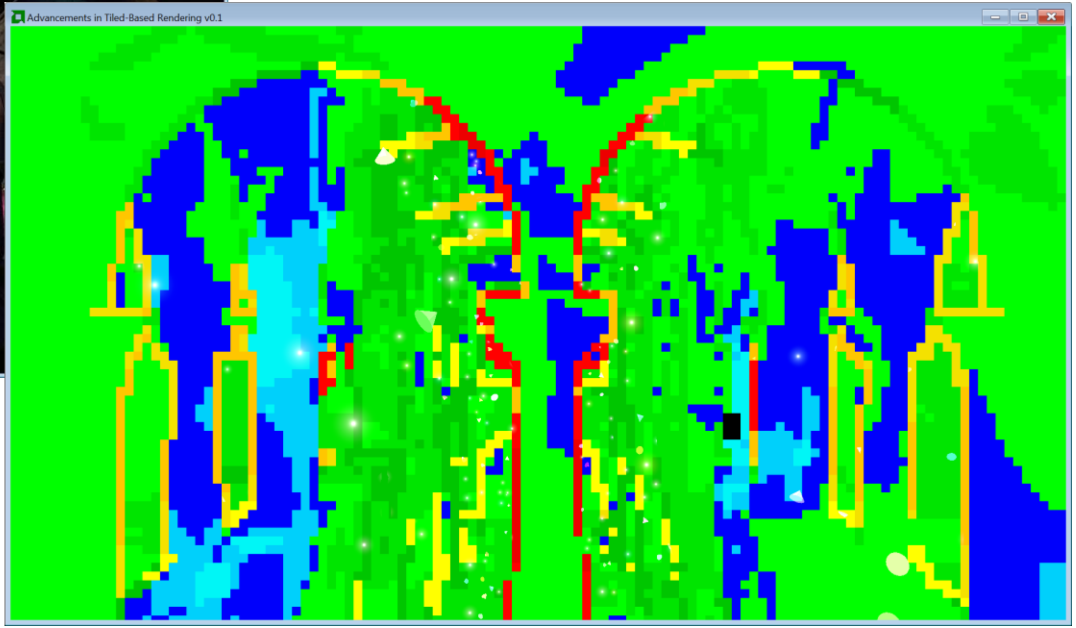
# Culling Spot Lights

- Don’t put bounding sphere around spot light origin
- Tightly bound spot light inside sphere at P with radius r

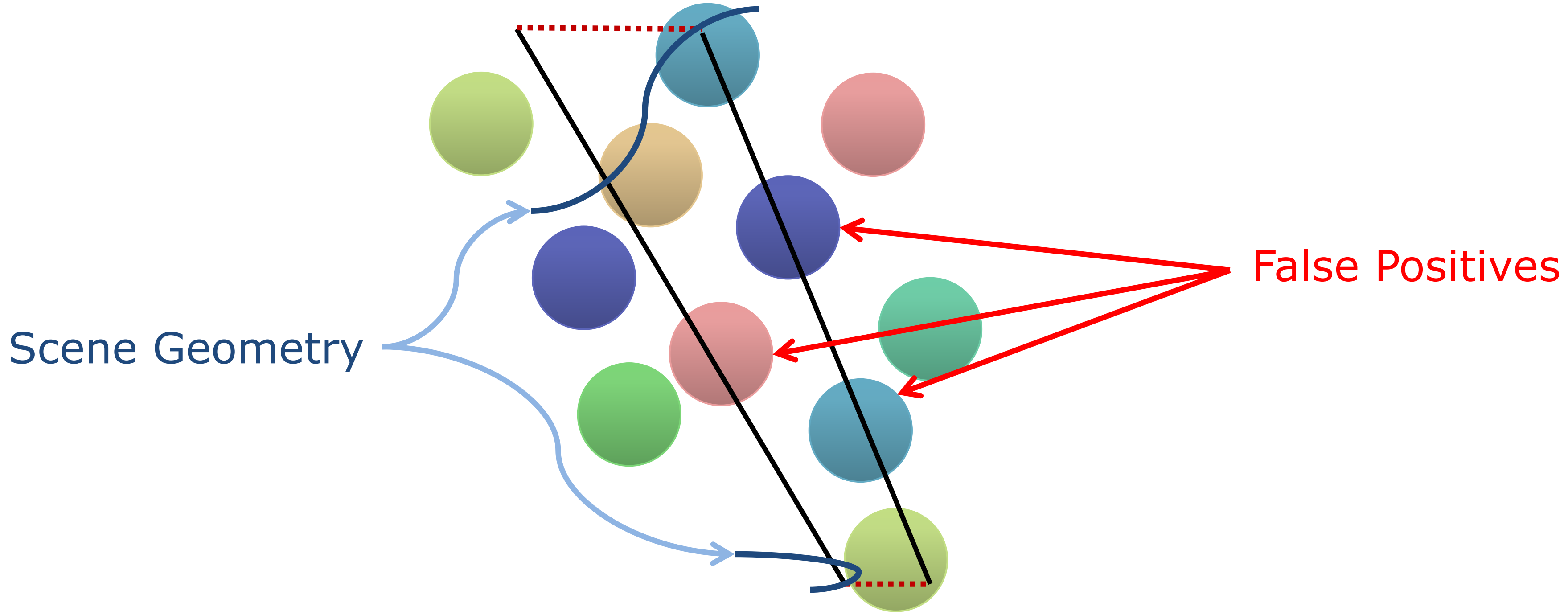


# Depth Discontinuities

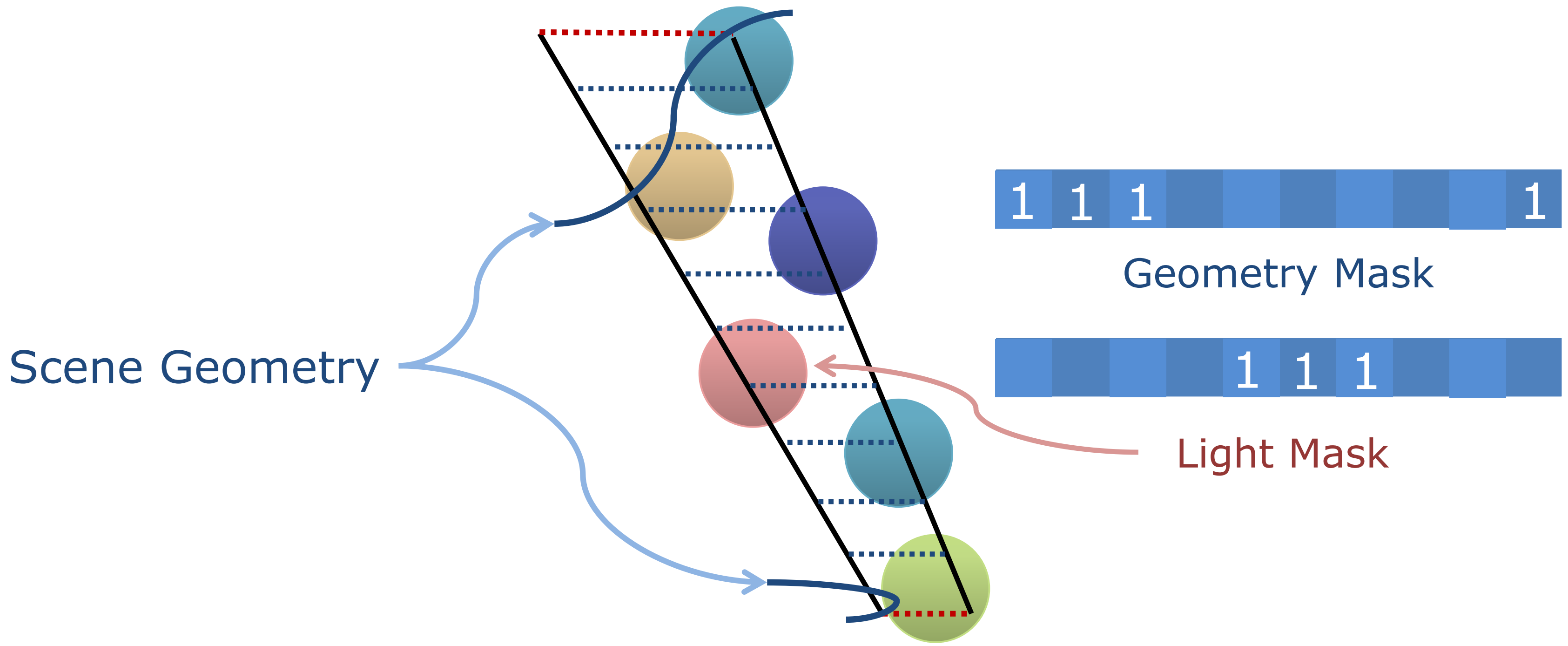


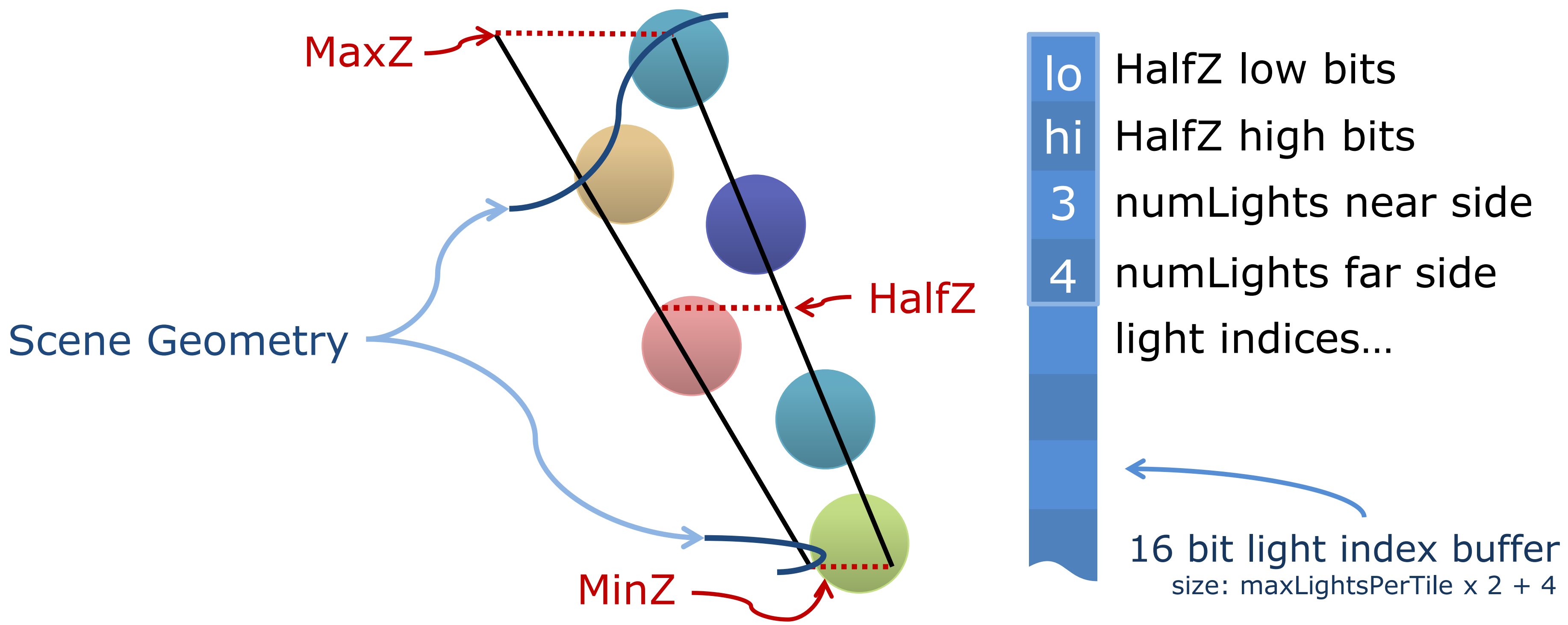
# Depth Discontinuities



# 2.5D Culling [Harada et al 12]

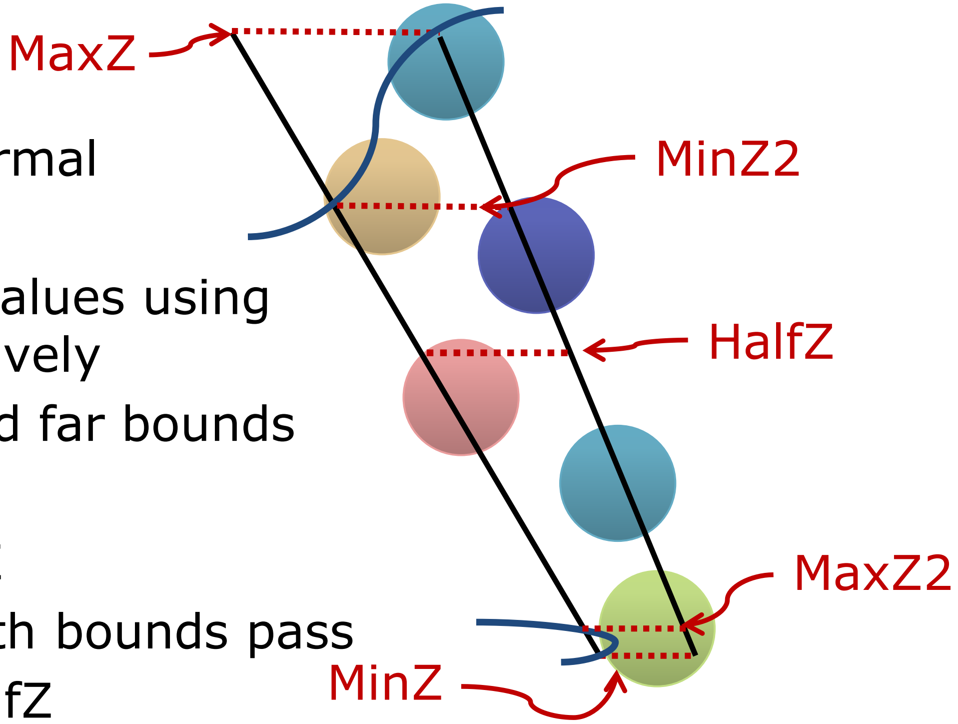


# HalfZ

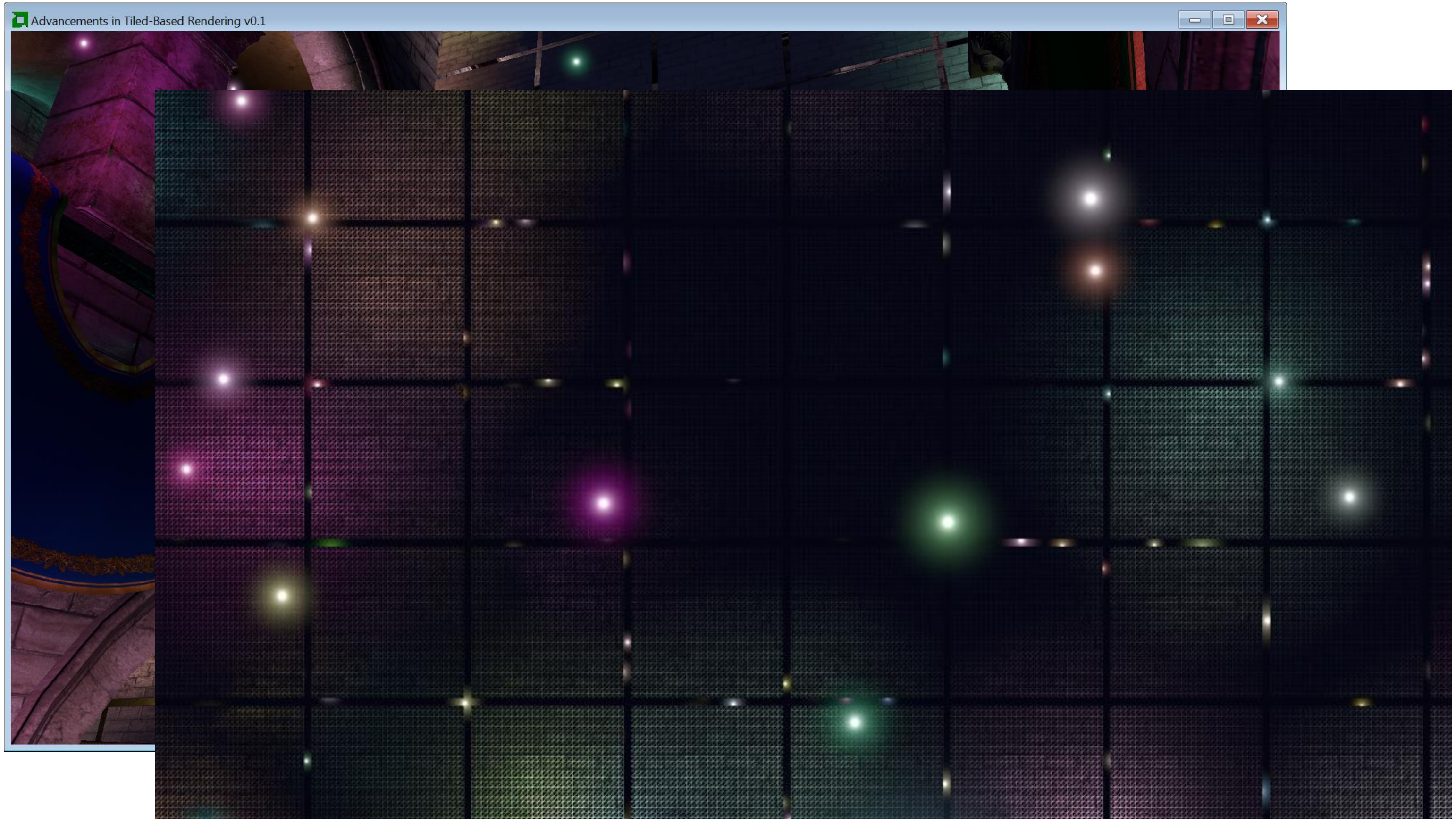

MaxZ
HalfZ
MinZ
Scene Geometry
lo
hi
3
4
HalfZ low bits
HalfZ high bits
numLights near side
numLights far side
light indices…
16 bit light index buffer
size: maxLightsPerTile x 2 + 4

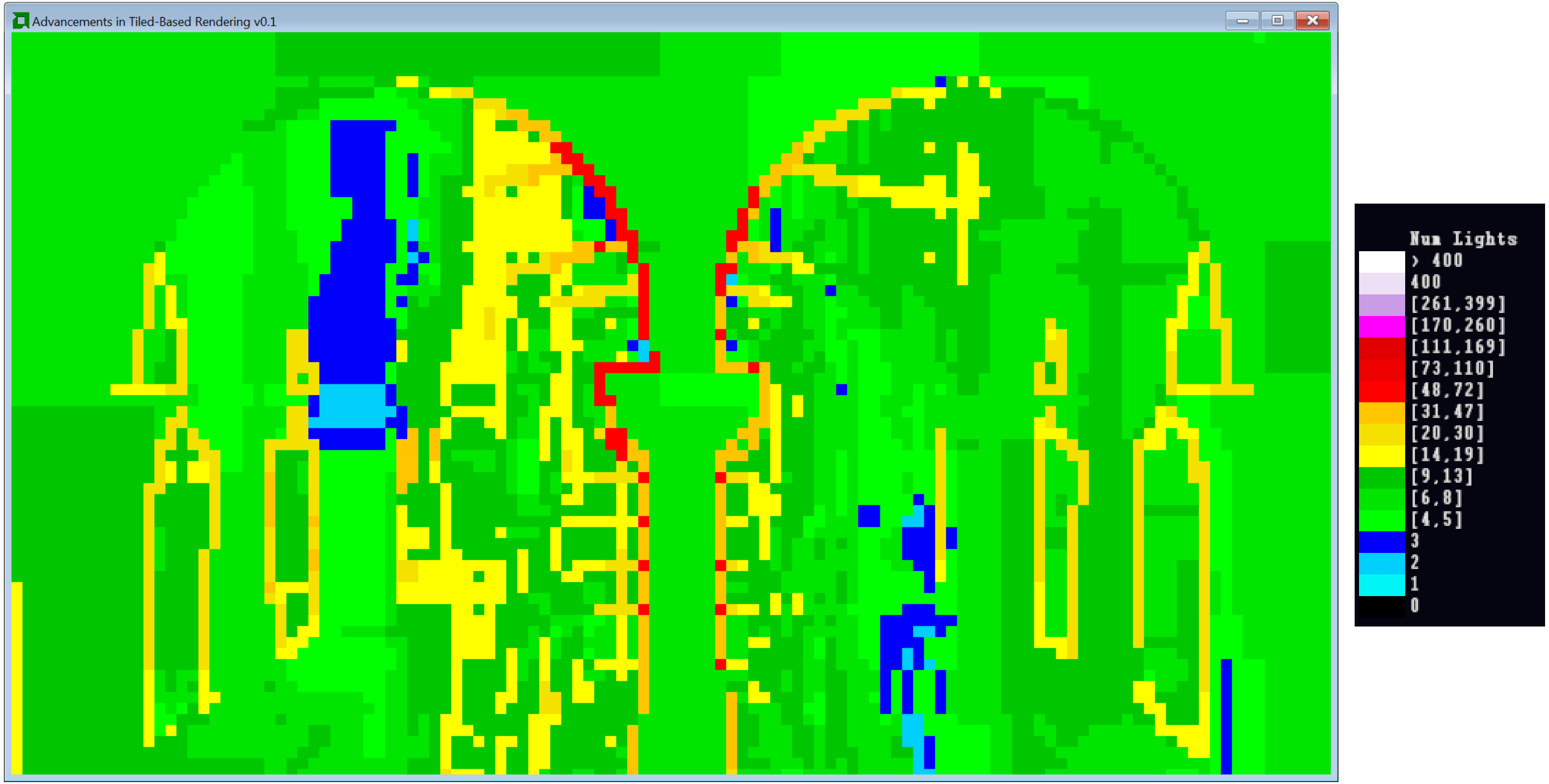
# Modified HalfZ

- Calculate Min & Max Z as normal
- Calculate HalfZ
- Second set of Min and Max values using HalfZ and max & min respectively
- Test against near bounds and far bounds
- Write to either one list
- Or write to two lists *cf.* HalfZ
- Doubles the work in the depth bounds pass
- Worst case converges on HalfZ

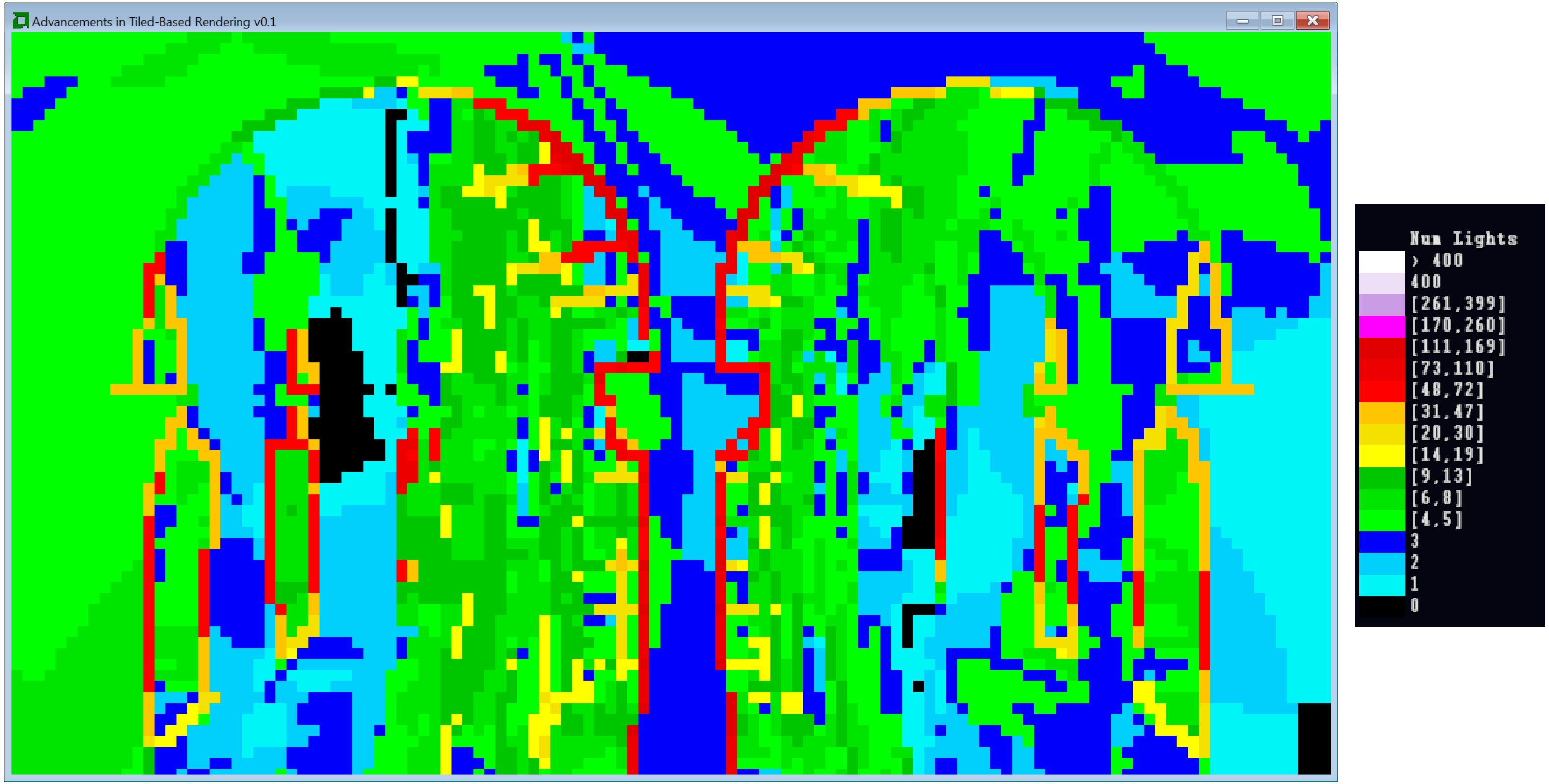



Sponza Atrium + 1 million sub pixel triangles

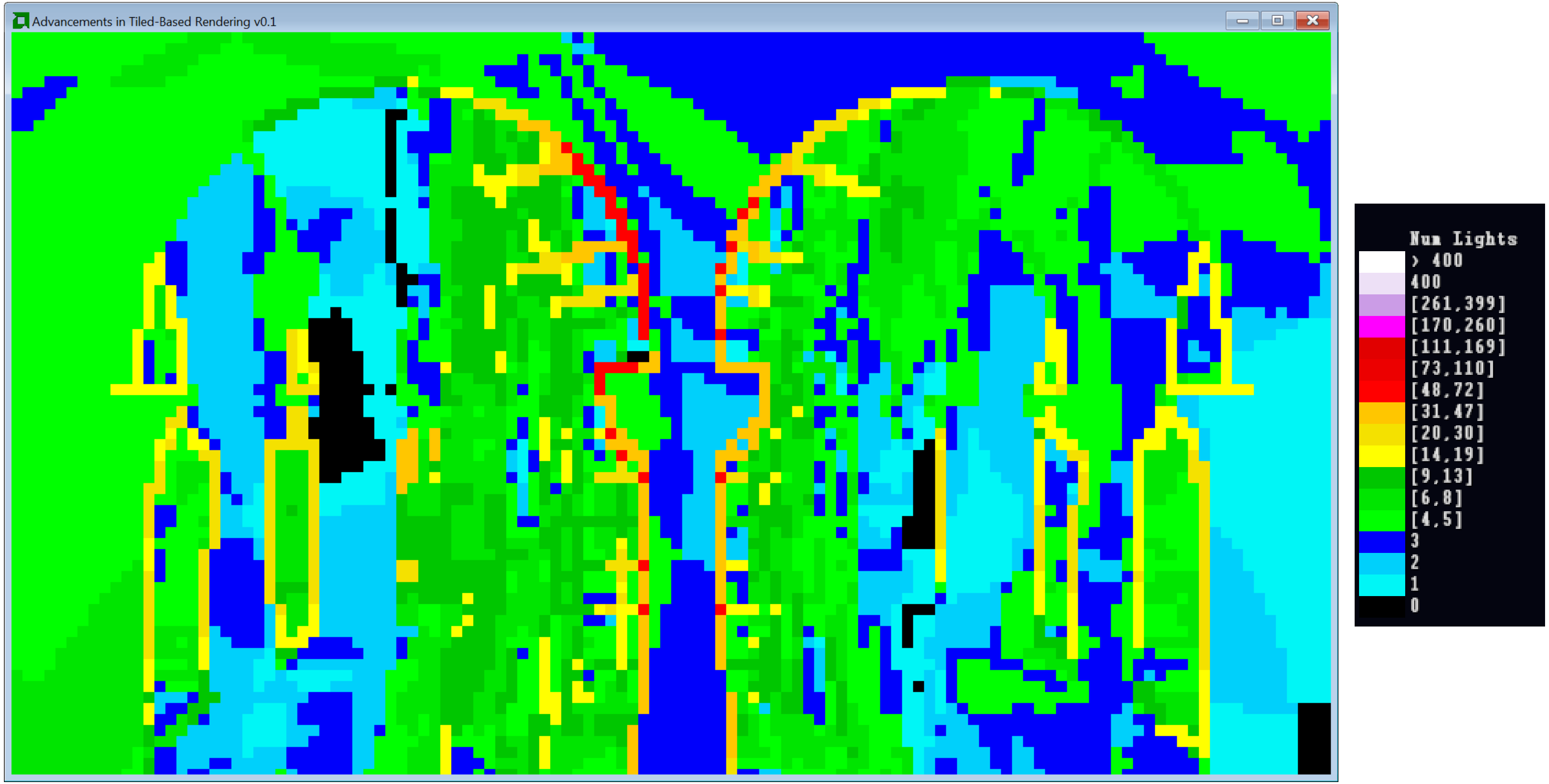

Advancements in Tiled-Based Rendering v0.1



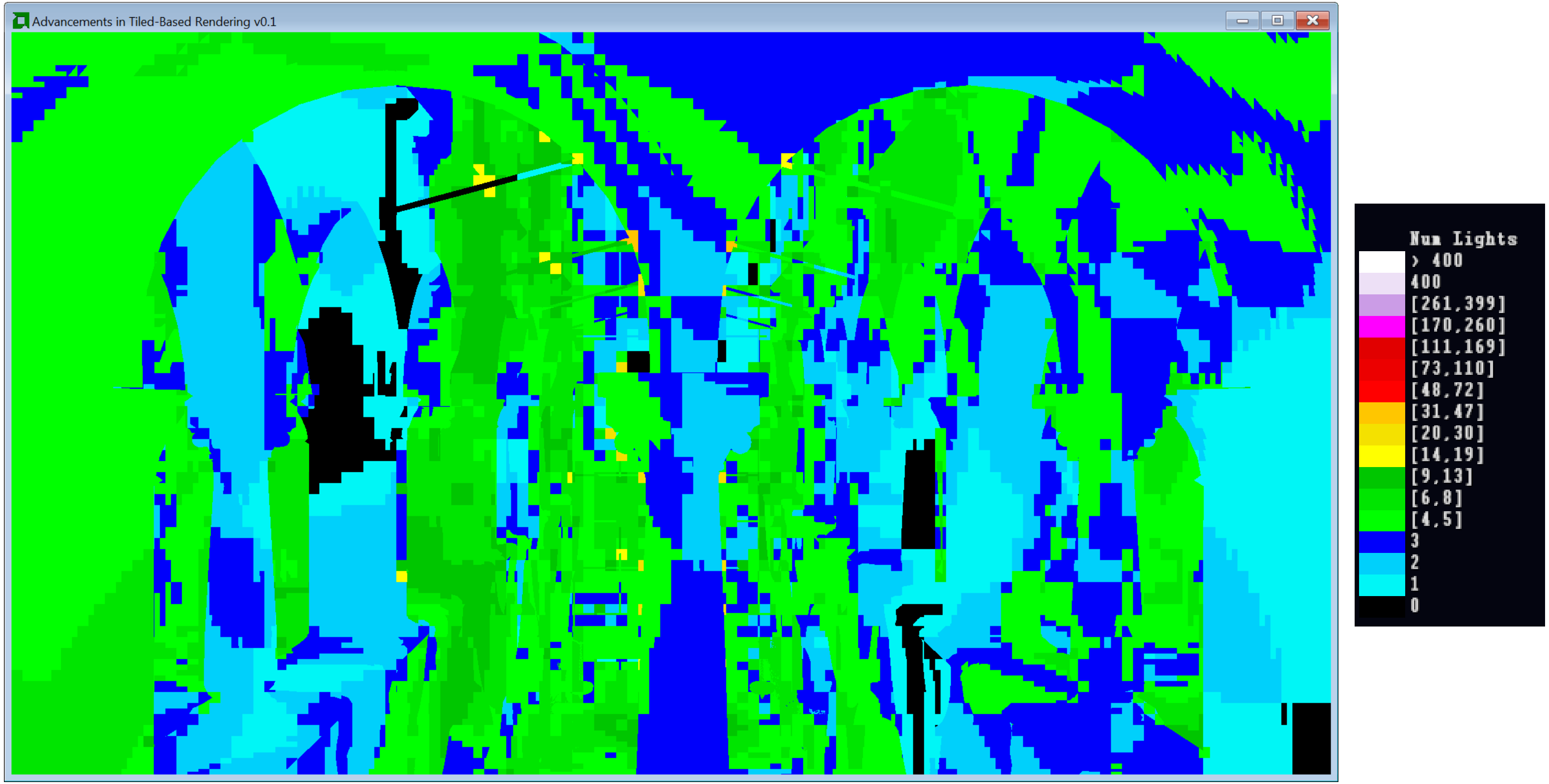

MinMax depth bounds, Frustum culling




MinMax depth bounds, AABB culling




MinMax depth bounds, Hybrid culling (AABB + Frustum sides)

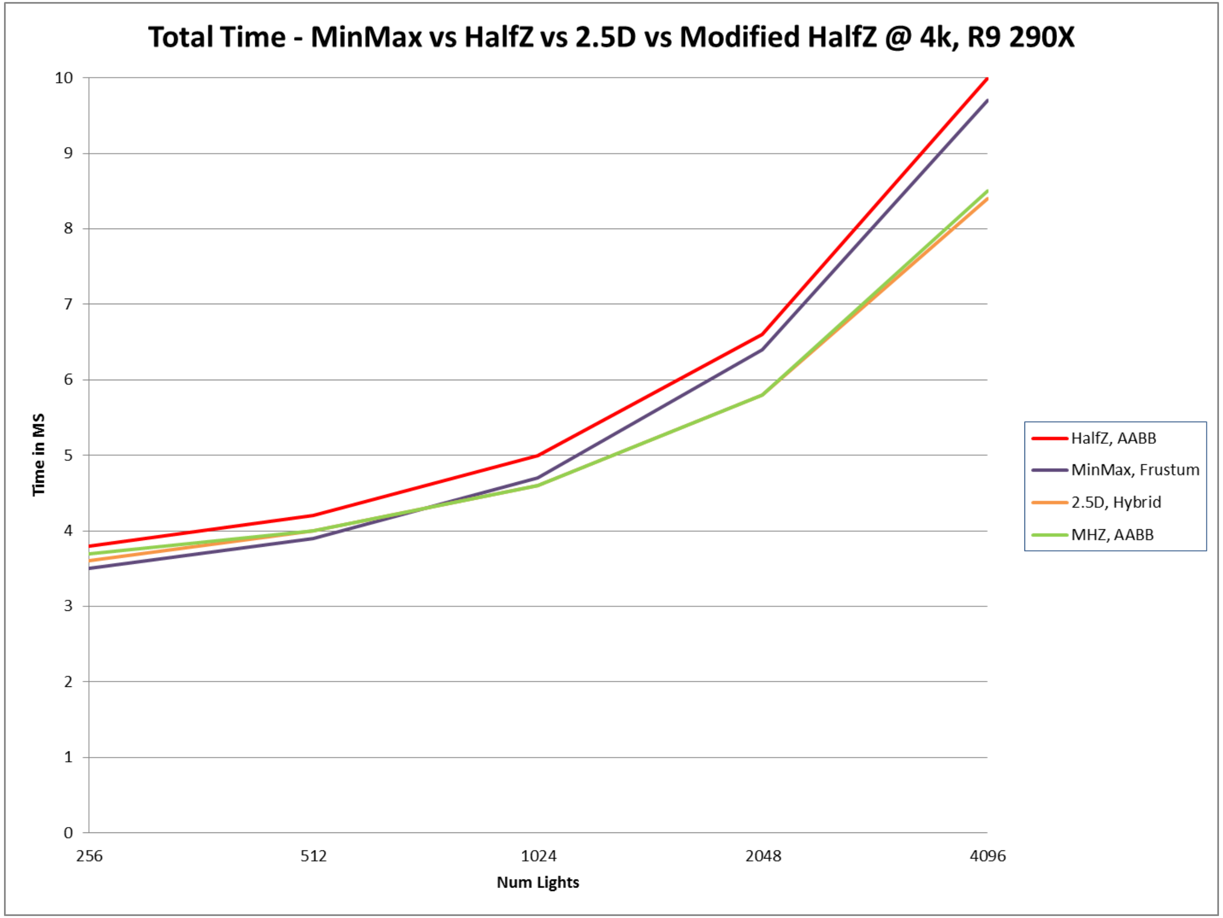


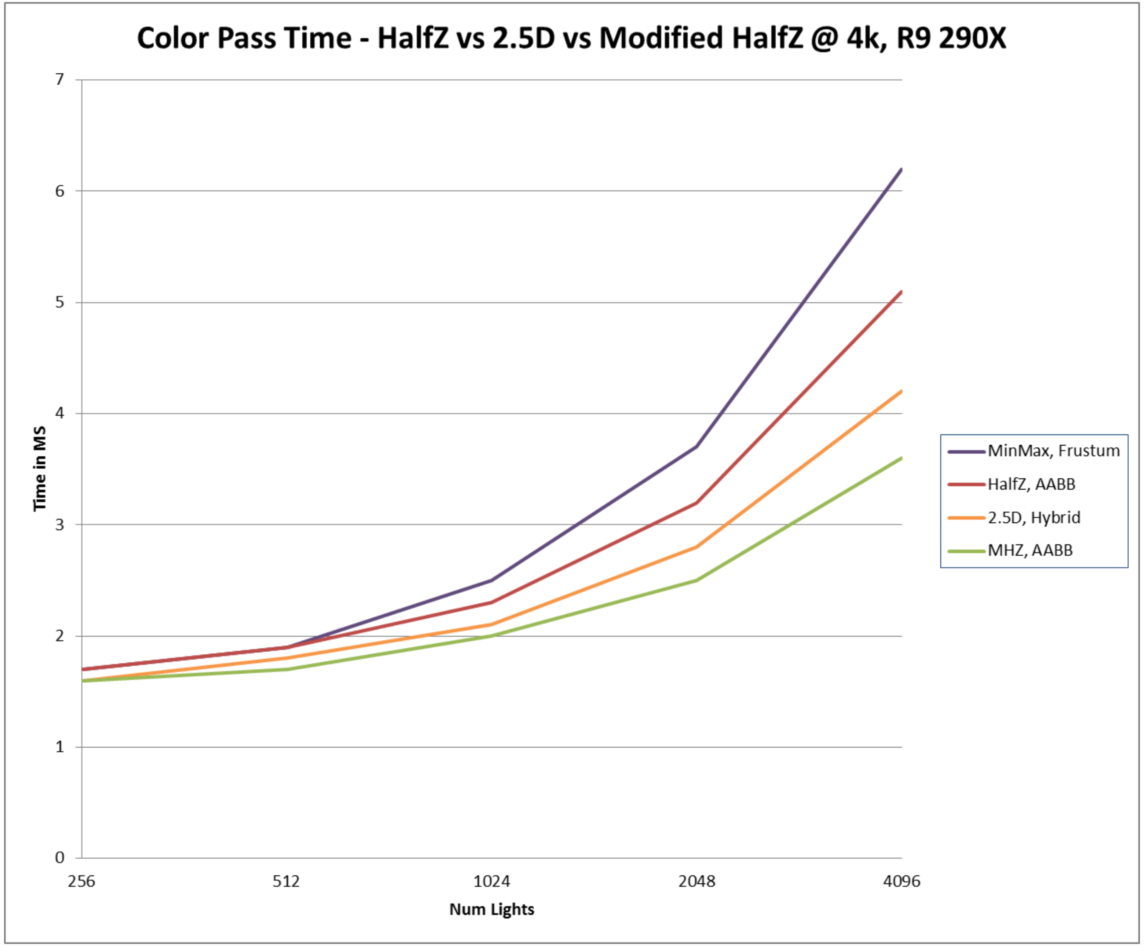

Modified HalfZ depth bounds, AABB culling

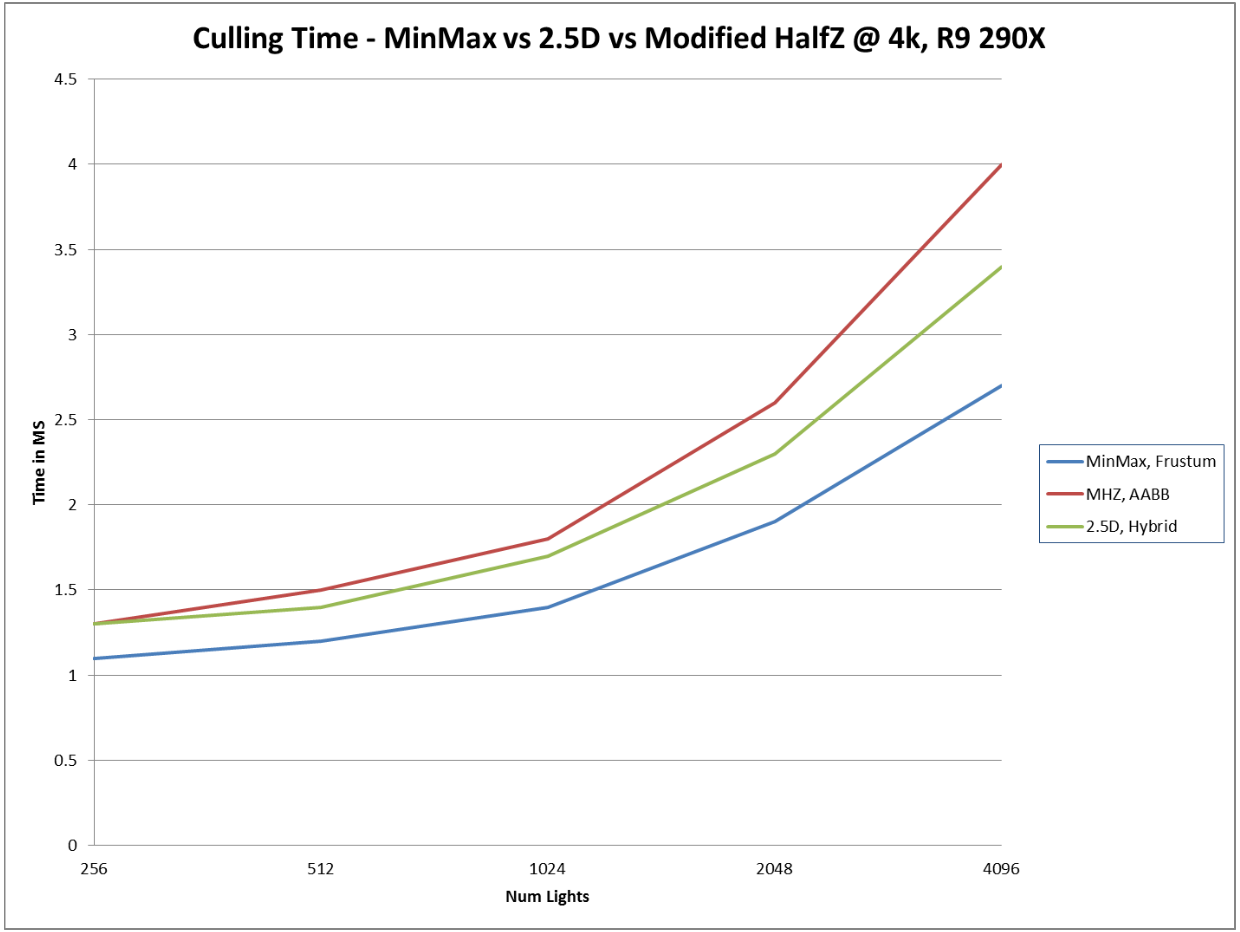
Unreal Engine 4, Infiltrator Demo

MinMax Depth Bounds

Modified HalfZ in one light list

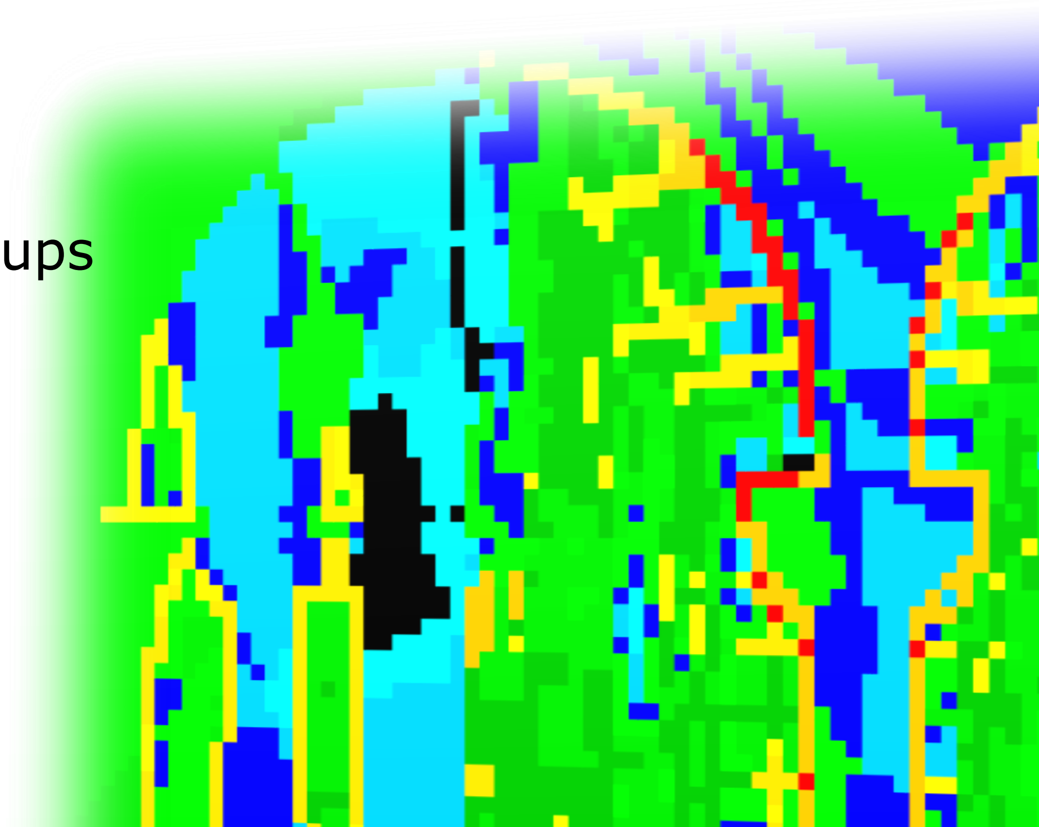

Total Time - MinMax vs HalfZ vs 2.5D vs Modified HalfZ @ 4k, R9 290X
Time in MS
0
1
2
3
4
5
6
7
8
9
10
256
512
1024
2048
4096
Num Lights
HalfZ, AABB
MinMax, Frustum
2.5D, Hybrid
MHZ, AABB

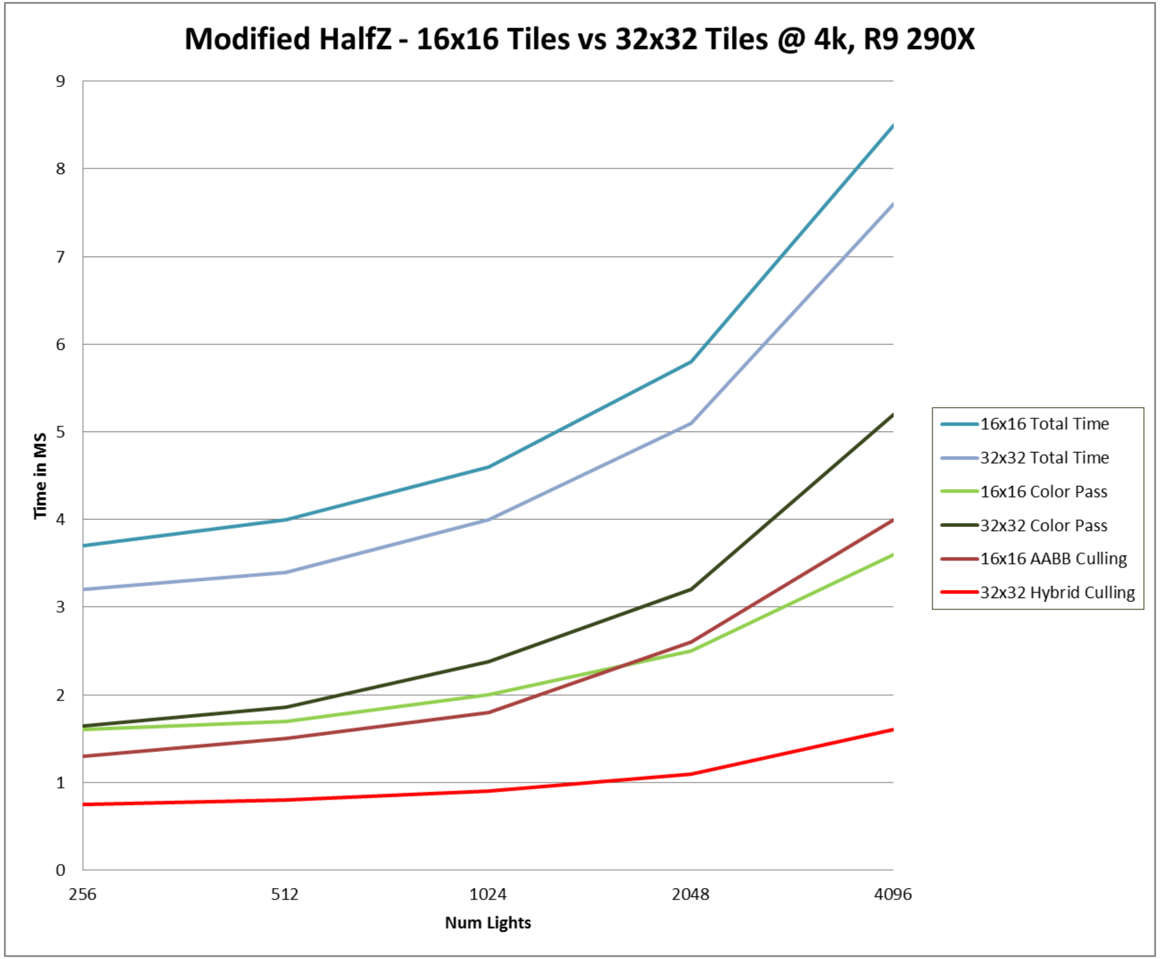

Color Pass Time - HalfZ vs 2.5D vs Modified HalfZ @ 4k, R9 290X
Time in MS
0
1
2
3
4
5
6
7
256
512
1024
2048
4096
Num Lights
MinMax, Frustum
HalfZ, AABB
2.5D, Hybrid
MHZ, AABB


Culling Time - MinMax vs 2.5D vs Modified HalfZ @ 4k, R9 290X
Time in MS
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
256
512
1024
2048
4096
Num Lights
MinMax, Frustum
MHZ, AABB
2.5D, Hybrid

# What happens if we cull 32x32 tiles?

Still using 16x16 thread groups




Modified HalfZ - 16x16 Tiles vs 32x32 Tiles @ 4k, R9 290X
Time in MS
9
8
7
6
5
4
3
2
1
0
256
512
1024
2048
4096
Num Lights
16x16 Total Time
32x32 Total Time
16x16 Color Pass
32x32 Color Pass
16x16 AABB Culling
32x32 Hybrid Culling

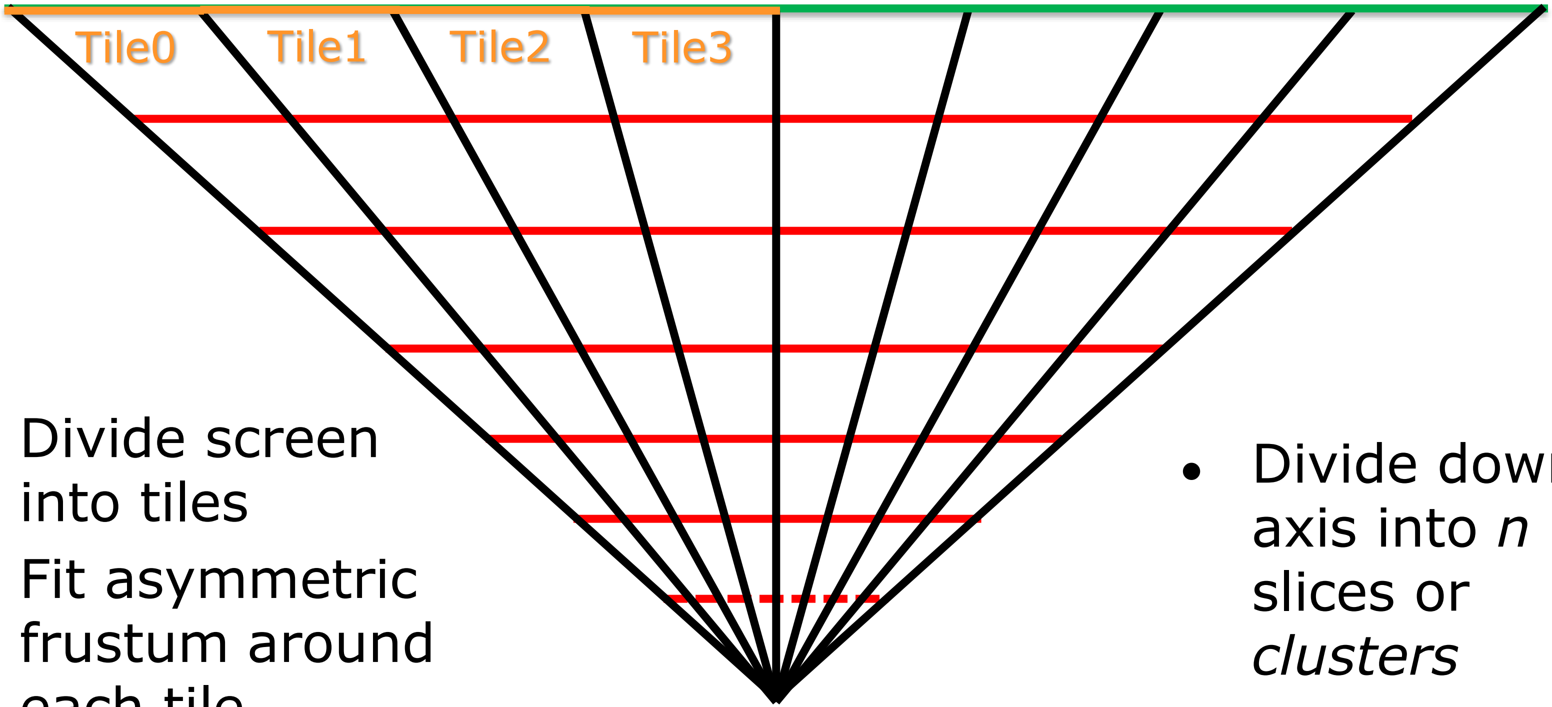
# Culling Conclusion

- Modified HalfZ with AABBs generally works best
  - Even though generating MinZ2 and MaxZ2 adds a little cost
  - Even though culling each light against two AABBs instead of one
- 32x32 tiles saves a good chunk of time in the culling stage
  - …at the cost of color pass efficiency when pushing larger number of lights

# Clustered Rendering [Olsson et al12]

- Production proven in Forza Horizon 2
- Additional benefits on top of 2D culling:
    - No mandatory Z prepass
    - Just works™ for transparencies and volumetric effects
- Can a further reduction in lights per pixel improve performance?

# Clustered Rendering 101

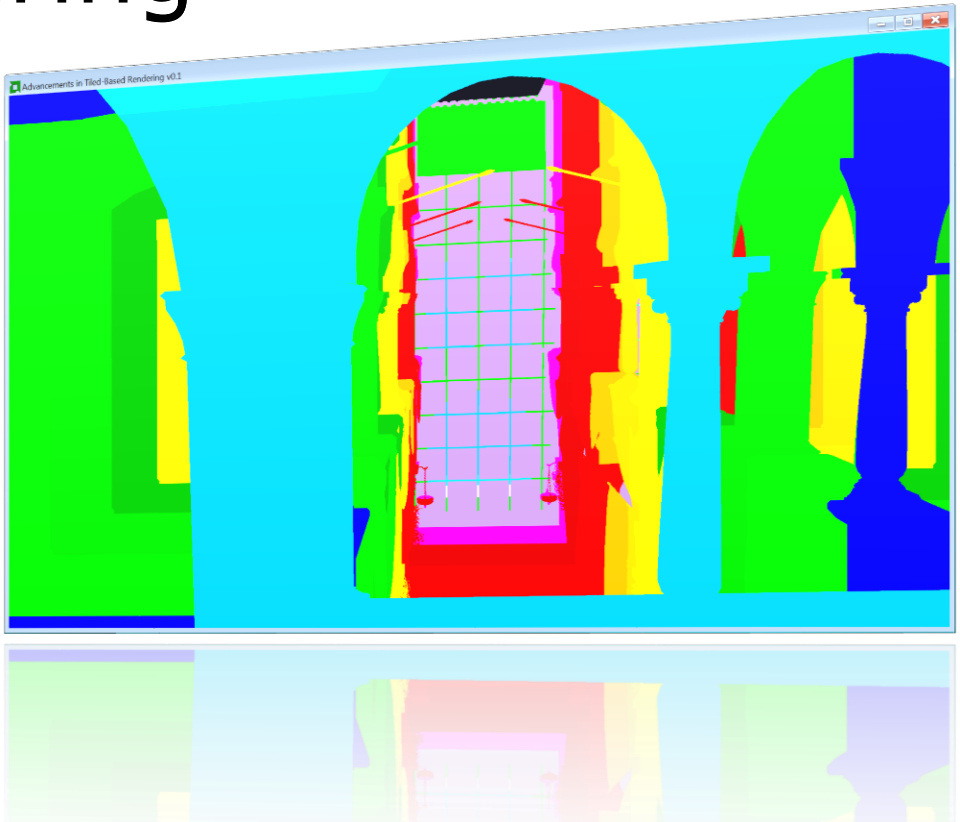



- Divide screen into tiles
- Fit asymmetric frustum around each tile

- Divide down Z axis into *n* slices or *clusters*

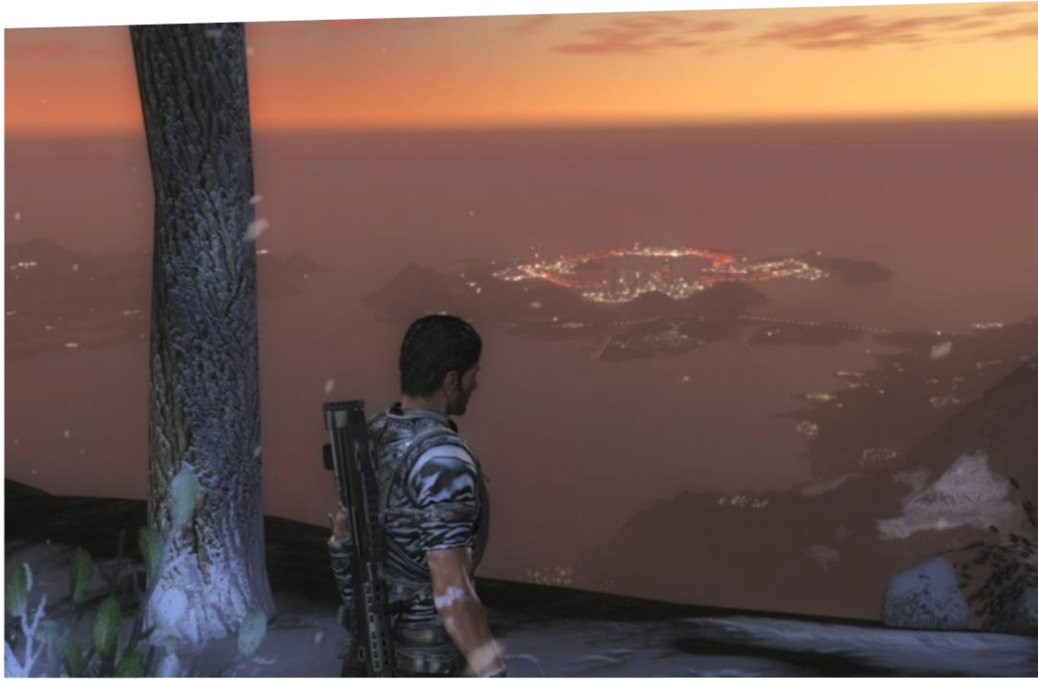
# Clustered Rendering

- Divide up Z axis exponentially
- Start at some sensible near slice
- Cap at some sensible value



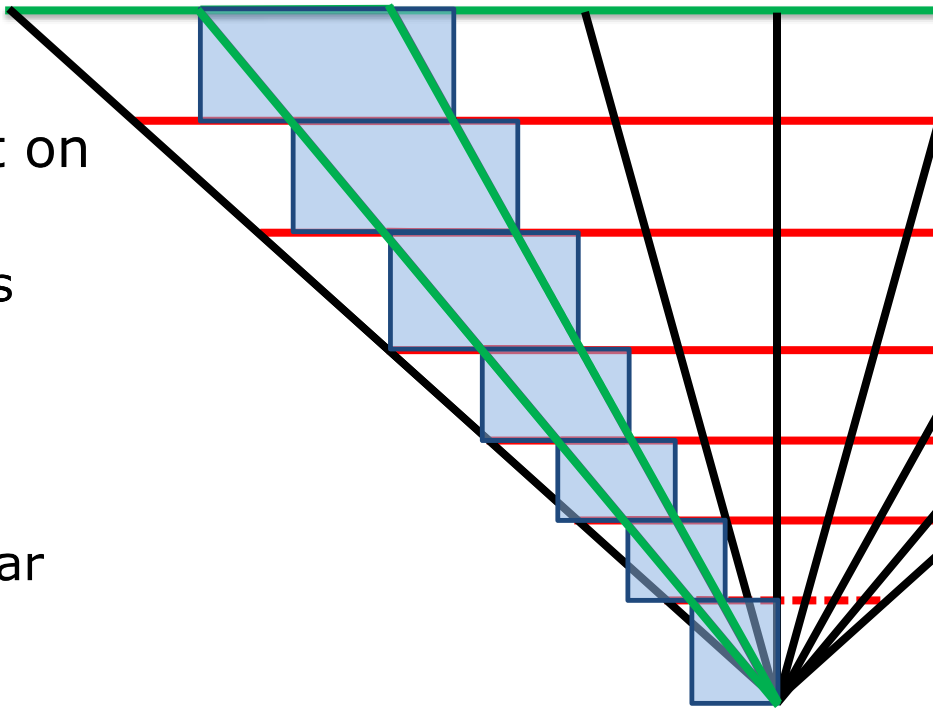
# Provision for far lights

- Fade them out
- Drop back to glares
- Prebake



# Light Culling

- View space AABBs worked best on 2D grid
  - Bad when running say 16 slices
- View space frustum planes are better
  - Calculate per tile planes
  - Then test each slice near and far
  - Optionally, *then* test AABBs



# VRAM Usage

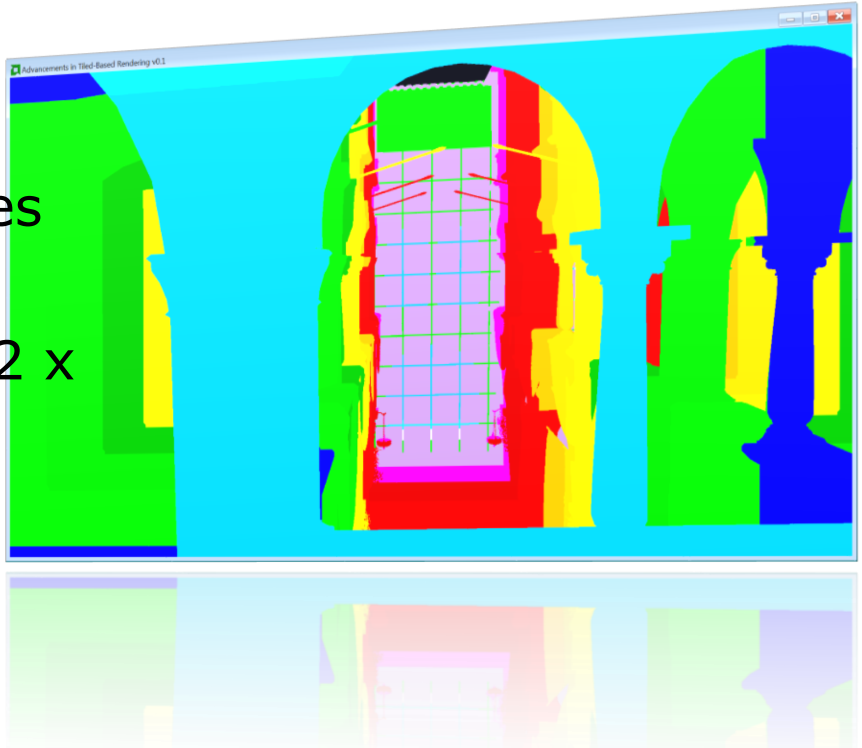
- 16x16 pixel 2D grid requires numTilesX x numTilesY x maxLights
  - 1080p: 120x68x512xuint16 = 8MB
  - 4k: 240x135x512xuint16 = 32MB
- List for each light type (points & spots): 64MB
- So 32 slices: 1GB for point lights only ☹
- Either use coarser grid
- Or use a compacted list

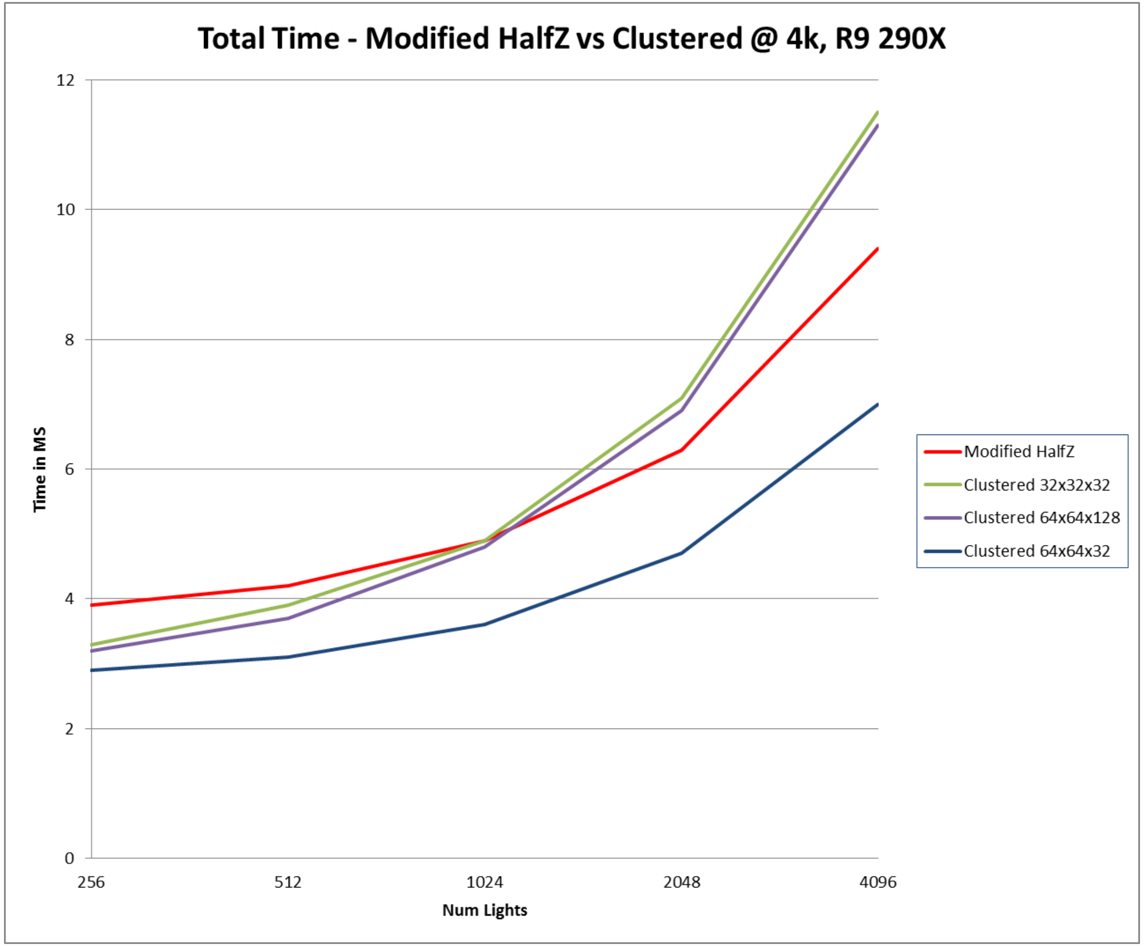
# Compacted List

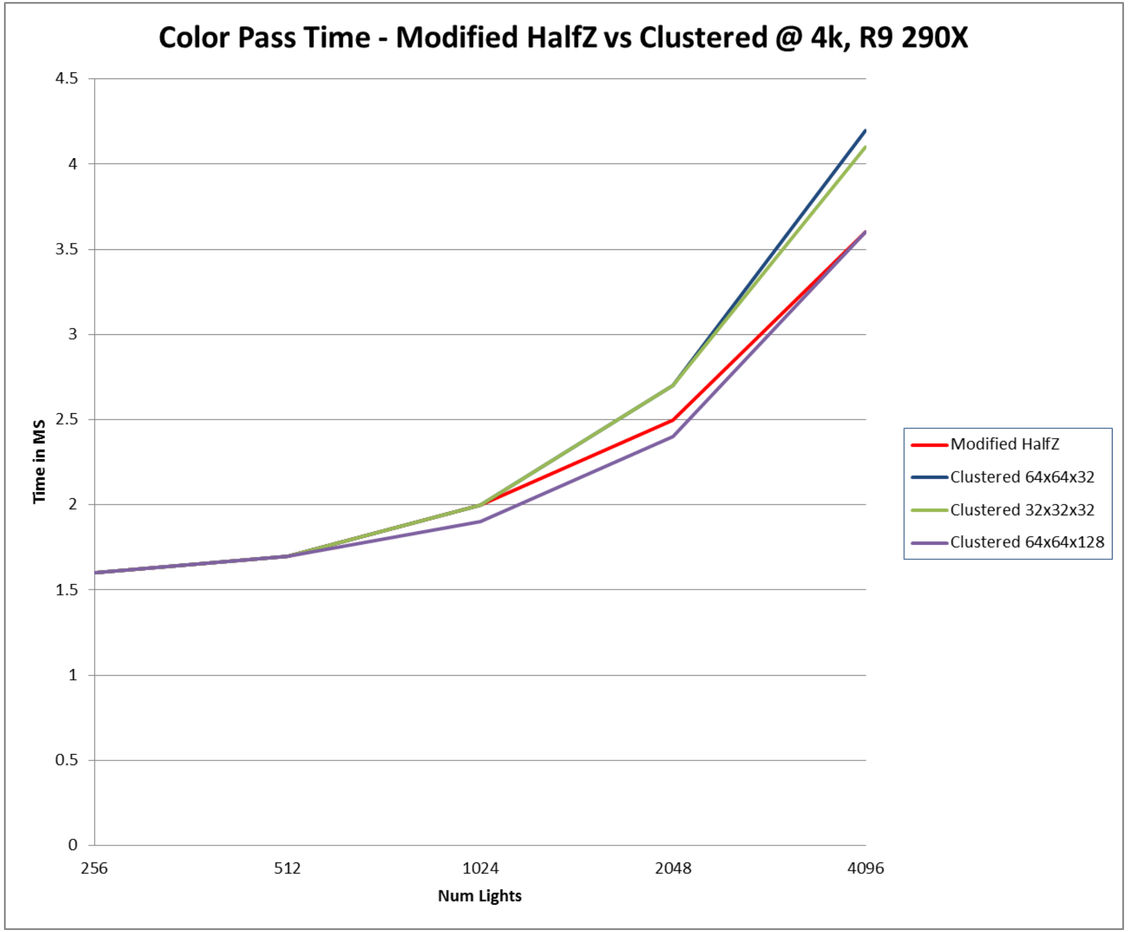
- Option 1:
  - Do all culling on CPU [Olsson et al12] [Persson13][Dufresne14]
    - But some of the lights may be spawned by the GPU
    - My CPU is a precious resource!
- Option 2:
  - Cull on GPU
  - Keep track of how many lights per slice in TGSM
  - Write table of offsets in light list header
  - Only need maxLights x “safety factor” per tile

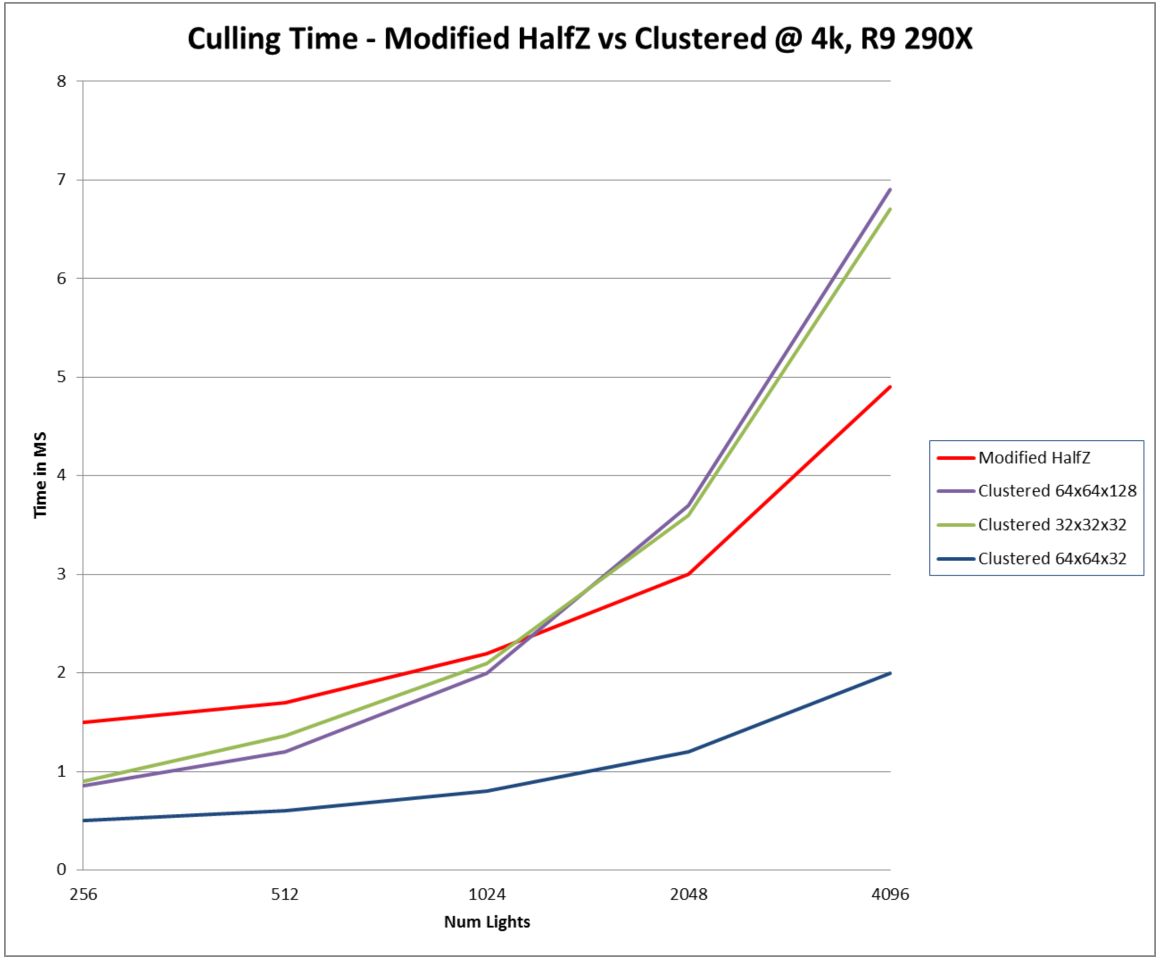
# Coarse Grid

- Example:
  - 4k resolution
  - 64x64 pixel tiles with 64 slices
  - maxLights = 512
  - 60 x 34 tiles x 64 slices x 512 x uint16 = 128MB




Total Time - Modified HalfZ vs Clustered @ 4k, R9 290X
12
10
8
6
4
2
0
Time in MS
256
512
1024
2048
4096
Num Lights
Modified HalfZ
Clustered 32x32x32
Clustered 64x64x128
Clustered 64x64x32


Color Pass Time - Modified HalfZ vs Clustered @ 4k, R9 290X
Time in MS
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
256
512
1024
2048
4096
Num Lights
Modified HalfZ
Clustered 64x64x32
Clustered 32x32x32
Clustered 64x64x128


Culling Time - Modified HalfZ vs Clustered @ 4k, R9 290X
Time in MS
0
1
2
3
4
5
6
7
8
256
512
1024
2048
4096
Num Lights
Modified HalfZ
Clustered 64x64x128
Clustered 32x32x32
Clustered 64x64x32

# Z Prepass

- Very scene dependant
- Often considered too expensive
- DirectX12 can help draw submission cost
- *Should* already have a super optimized depth only path for shadows!
  - Position only streams
  - Index buffer to batch materials together
- A partial prepass can really help lighten the geometry load

# Conclusions

- Parallel Reduction - faster than atomic min/max
- AABB-Sphere test in conjunction with Modified HalfZ is a good choice
- Clustered shading
  - Potentially a big saving on the tile culling
  - Less overhead for low light numbers
  - Offers other benefits over 2D tiling
- Aggressive culling is very worthwhile
  - The best optimisation for your expensive color scene

# References


- [Andersson09] Johan Andersson, “Parallel Graphics in Frostbite – Current & Future”, Beyond Programmable Shading, SIGGRAPH 2009
- [Harada et al12] Takahiro Harada, Jay McKee, Jason C Yang, “Forward+: Bringing Deferred Lighting to the Next Level”, Eurographics 2012
- [Harris07] Mark Harris, “Optimizing Parallel Reduction in CUDA”, NVIDIA 2007
- [Engel14] Wolfgang Engel, “Compute Shader Optimizations for AMD GPUs: Parallel Reduction”, Confetti 2014
- [Harada12] Takahiro Harada, “A 2.5D Culling for Forward+”, Technical Briefs, SIGGRAPH Asia 2012
- [Arvo90] Jim Arvo, “A simple method for box-sphere intersection testing”, Graphics Gems 1990
- [Dufresne14] Marc Fauconneau Dufresne, “Forward Clustered Shading”, Intel 2014
- [Persson13] Emil Persson, “Practical Clustered Shading”, Avalanche 2013
- [Olsson et al12] Ola Olsson, Markus Billeter, Ulf Assarsson, “Clustered Deferred and Forward Shading”, HPG 2012
- [Schulz14] Nicolas Schulz, “Moving to the Next Generation – The Rendering Technology of Ryse”, GDC 2014

# Thanks

- Jason Stewart, AMD
- Epic Rendering Team
- Emil Persson, Avalanche Studios

# Questions?

gareth.thomas@amd.com