# Practical Clustered Shading

Emil Persson
Head of Research, Avalanche Studios

AVALANCHE STUDIOS

# Practical Clustered Shading

- History of lighting in the Avalanche Engine
- Why Clustered Shading?
- Adaptations for the Avalanche Engine
- Performance
- Future work

# Lighting in Avalanche Engine

- Just Cause 1
  - Forward rendering
  - 3 global pointlights
- Just Cause 2, Renegade Ops
  - Forward rendering
  - World-space XZ-tiled light-indexing
    - 4 lights per 4m x 4m tile
    - 128x128 RGBA8 light index texture
    - Lights in constant registers (PC/Xenon) or 1D texture (PS3)
  - Per-object lighting
  - Customs solutions

# Lighting in Avalanche Engine

- Post-JC2 unannounced projects
  - Classic deferred rendering
    - 3-4 G-Buffers
    - Flexible lighting setup
      - Point lights
      - Spot lights
        - Optional shadow caster
        - Optional projected texture
      - Area lights
      - Fill lights
  - Transparency a big problem
    - Old forward pass still polluting the code
  - FXAA for anti-aliasing

# Solutions we've been eyeing

- Tiled deferred shading
  - Production proven (Battlefield 3)
  - Faster than classic deferred
  - All cons of classic deferred
    - Transparency, MSAA, memory, custom materials / light models etc.
  - Less modular than classic deferred
- Forward+
  - Production proven (Dirt Showdown)
  - Forces Pre-Z pass
  - MSAA works fine
  - Transparency requires another pass
  - Less modular than classic deferred
- Clustered shading
  - Not production proven (yet)
  - No Pre-Z necessary
  - MSAA works fine
  - Transparency works fine
  - Less modular than classic deferred

# Why Clustered Shading?

- Flexibility
  - Forward rendering compatible
    - Custom materials or light models
    - Transparency
  - Deferred rendering compatible
    - Screen-space decals
    - Performance
- Simplicity
  - Unified lighting solution
  - Actually easier to implement than full blown Tiled Deferred / Forward+
- Performance
  - Typically same or better than Tiled Deferred
  - Better worst-case performance
  - Depth discontinuities? “It just works”

# Depth discontinuities

# Depth discontinuities

# Depth discontinuities

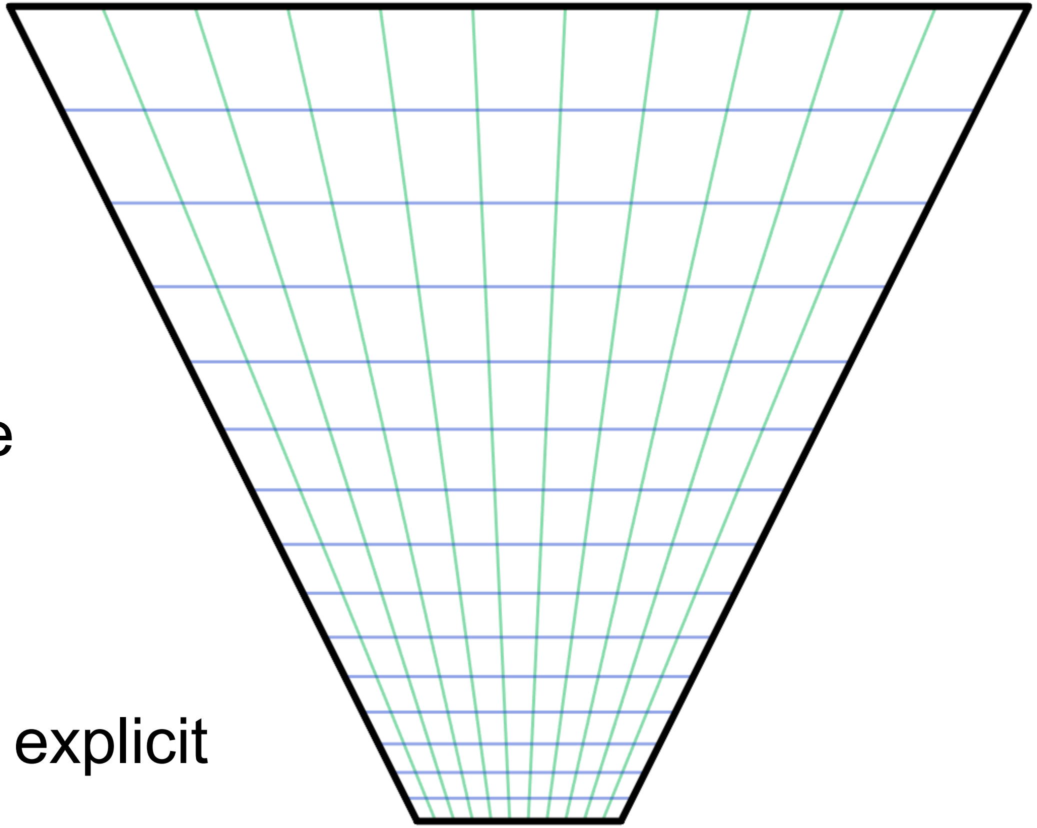
# Depth discontinuities

# Practical Clustered Shading

- What we didn't need
  - Millions of lights
  - Fancy clustering
  - Normal-cone culling
  - Explicit bounds
- What we needed
  - Large outdoor solution
  - No enforced Pre-Z pass
  - Spotlights
  - Shadows
- What we preferred
  - Work with DX10 level HW
  - Tight light culling
  - Scene independence

# The Avalanche solution

- Still a deferred shading engine
  - But unified lighting solution with forward passes
- Only spatial clustering
  - 64x64 pixels, 16 depth slices
- CPU light assignment
  - Works on DX10 HW
  - Allows compacter memory structure
- Implicit cluster bounds only
  - Scene-independent
  - Deferred pass could potentially use explicit



# The Avalanche solution

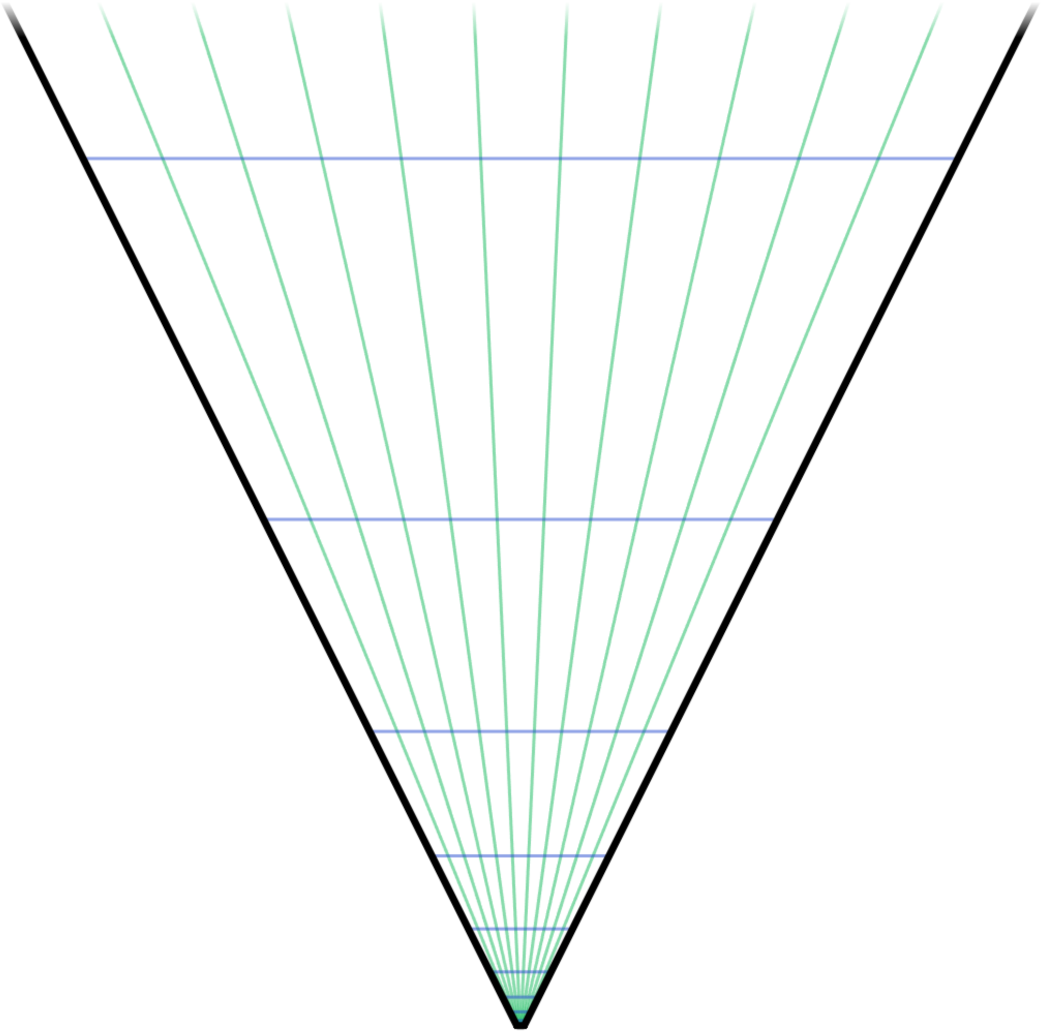
- Exponential depth slicing
  - Huge depth range! [0.1m – 50,000m]
    - Default list
      - [0.1, 0.23, 0.52, 1.17, 2.7, 6.0, 14, 31, 71, 161, 365, 828, 1880, 4270, 9696, 22018, 50000]
      - Poor utilization
    - Limit far to 500
      - We have a “distant lights” systems for light visualization beyond that
      - [0.1, 0.17, 0.29, 0.49, 0.84, 1.43, 2.44, 4.15, 7.07, 12.0, 20.5, 34.9, 59, 101, 172, 293, 500]
    - Special near 0.1 – 5.0 cluster
      - Tweaked visually from player standing on flat ground
      - [0.1, 5.0, 6.8, 9.2, 12.6, 17.1, 23.2, 31.5, 42.9, 58.3, 79.2, 108, 146, 199, 271, 368, 500]

# The Avalanche solution

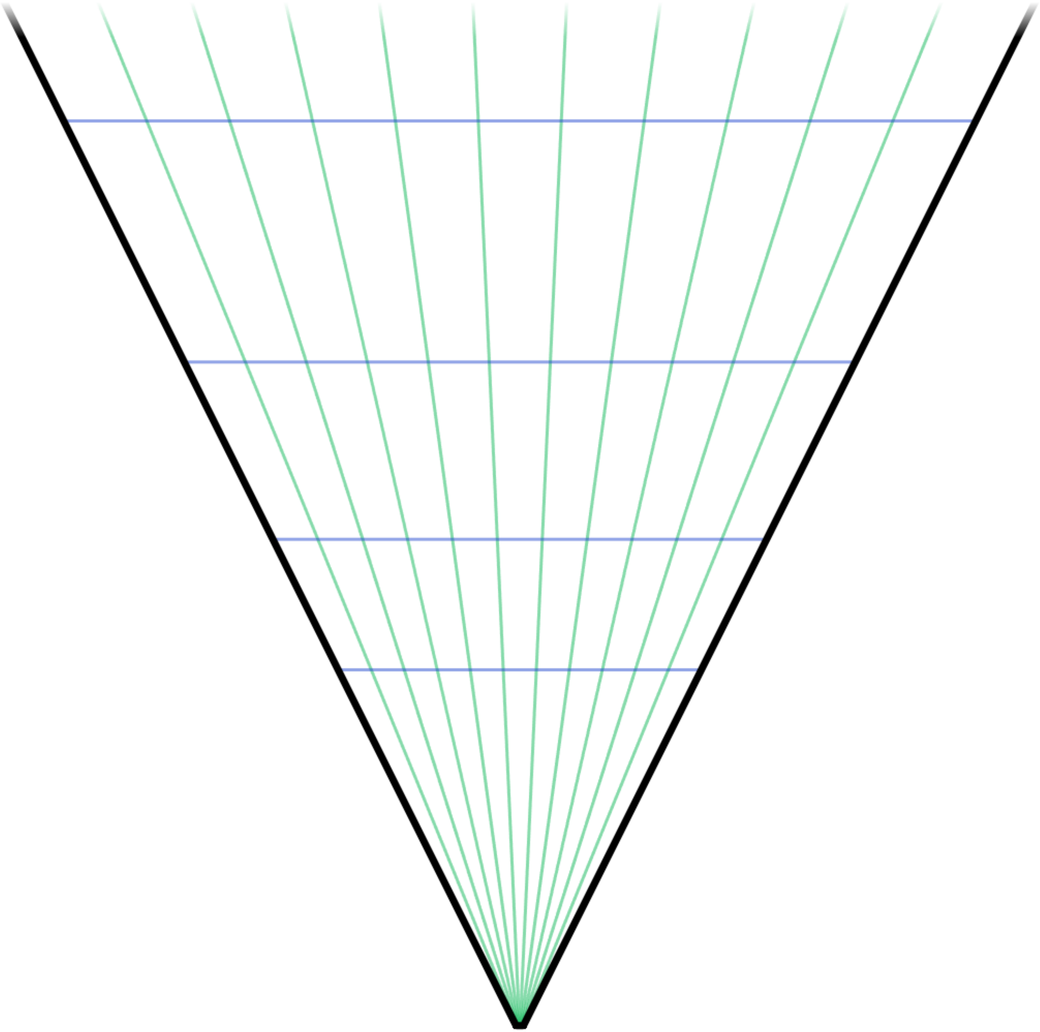
- Separate distant lights system

# The Avalanche solution

## Default exponential spacing

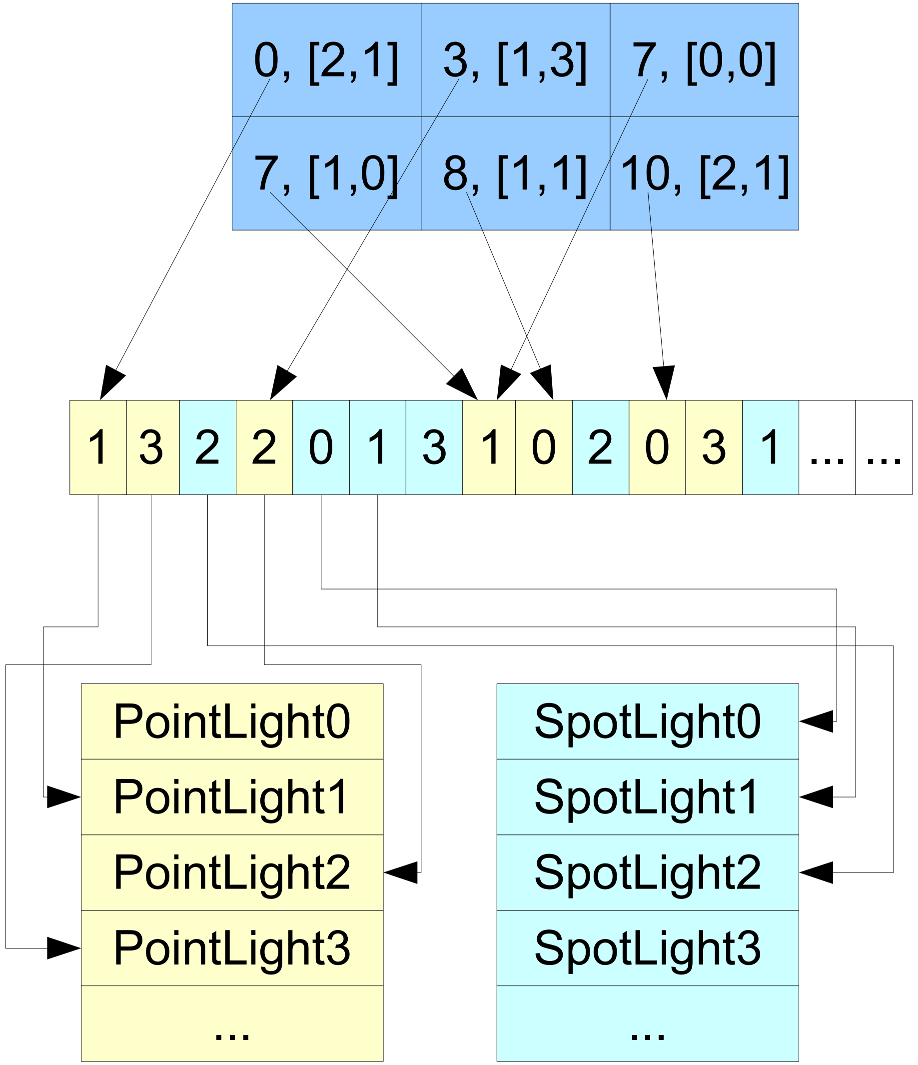


## Special near cluster



# Data structure

- Cluster “pointers” in 3D texture
  - R32G32_UINT
    - R=Offset
    - G=[PointLightCount, SpotLightCount]
- Light index list in texture buffer
  - R16_UINT
  - Tightly packed
- Light data in constant buffer
  - PointLight = 2 x float4
  - SpotLight = 3 x float4



# Shader

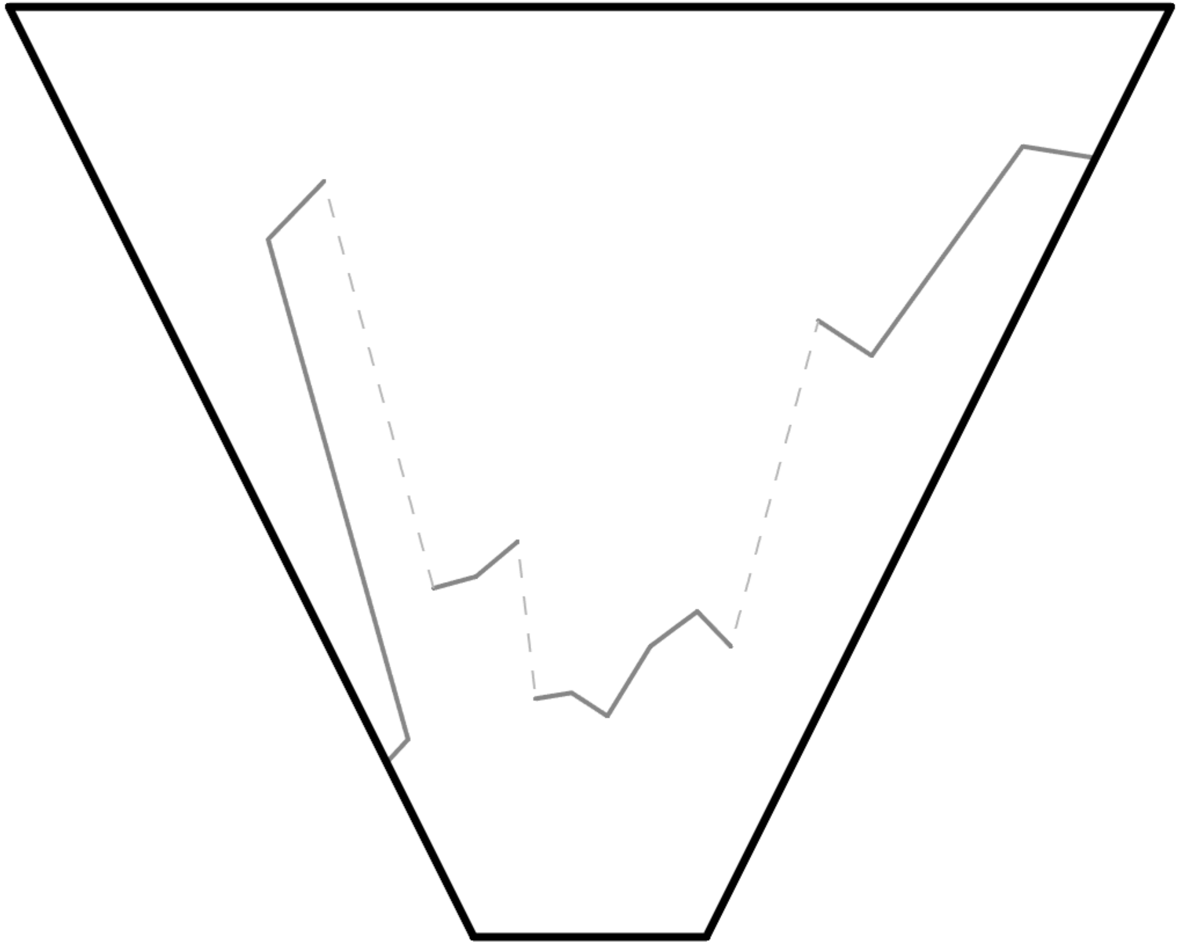
```
int3 tex_coord = int3(In.Position.xy, 0);                  // Screen-space position ...
float depth = Depth.Load(tex_coord);                       // ... and depth

int slice = int(max(log2(depth * ZParam.x + ZParam.y) * scale + bias, 0)); // Look up cluster
int4 cluster_coord = int4(tex_coord >> 6, slice, 0);       // TILE_SIZE = 64

uint2 light_data = LightLookup.Load(cluster_coord);        // Fetch light list
uint light_index = light_data.x;                           // Extract parameters
const uint point_light_count = light_data.y & 0xFFFF;
const uint spot_light_count  = light_data.y >> 16;

for (uint pl = 0; pl < point_light_count; pl++) {          // Point lights
    uint index = LightIndices[light_index++].x;

    float3 LightPos = PointLights[index].xyz;
    float3 Color    = PointLights[index + 1].rgb;
    // Compute pointlight here ...
}

for (uint sl = 0; sl < spot_light_count; sl++) {           // Spot lights
    uint index = LightIndices[light_index++].x;

    float3 LightPos = SpotLights[index].xyz;
    float3 Color    = SpotLights[index + 1].rgb;
    // Compute spotlight here ...
}
```

# Data structure

- Memory optimization
  - Naive approach: Allocate theoretical max
    - All clusters address all lights
      - Not likely
    - Might be several megabytes
    - Most never used
  - Semi-Conservative approach
    - Construct massive worst-case scenario
      - Multiply by 2, or what makes you comfortable
      - Still likely only a small fraction of theoretical max
    - Assert at runtime that you never go over allocation
      - Warn if you ever get close

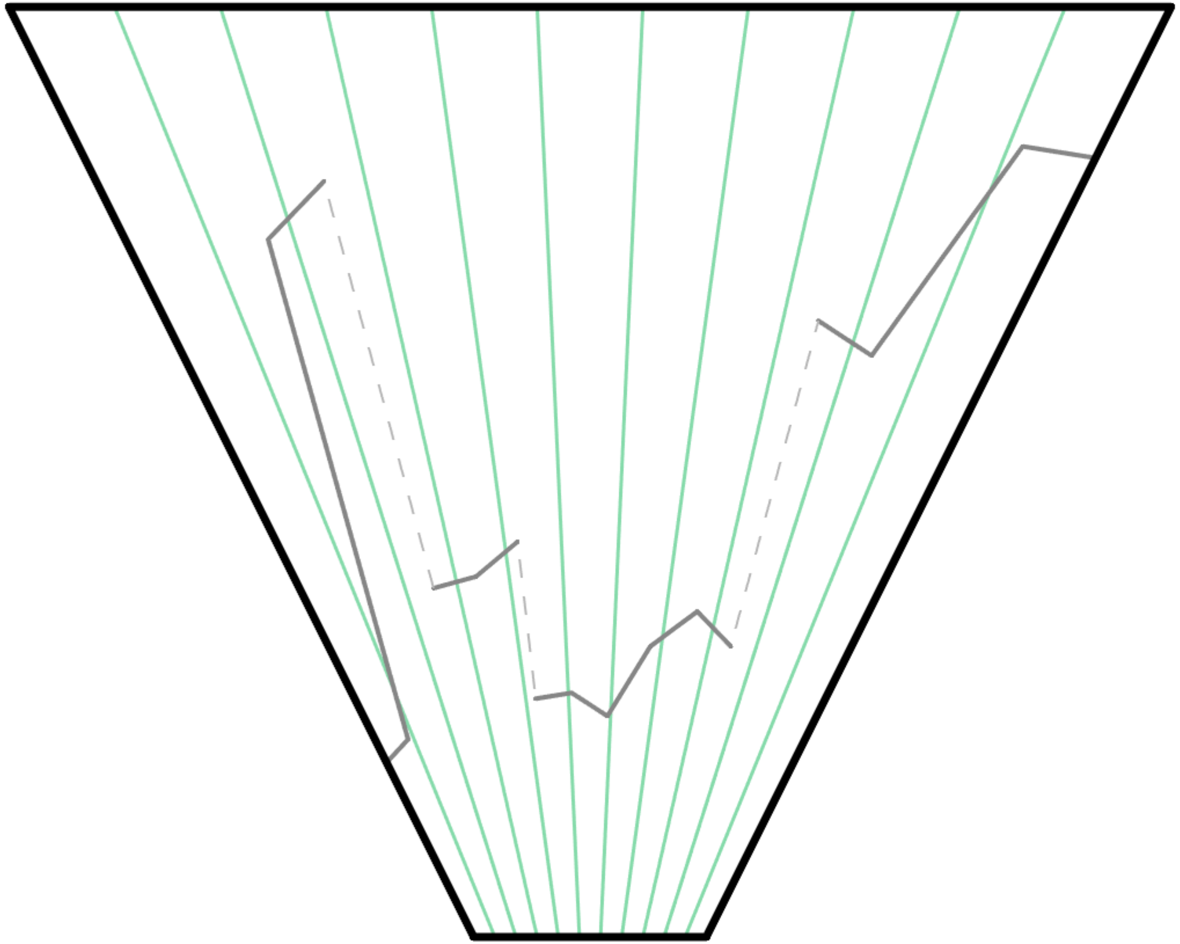
# Clustering and depth

- Sample frustum with depths



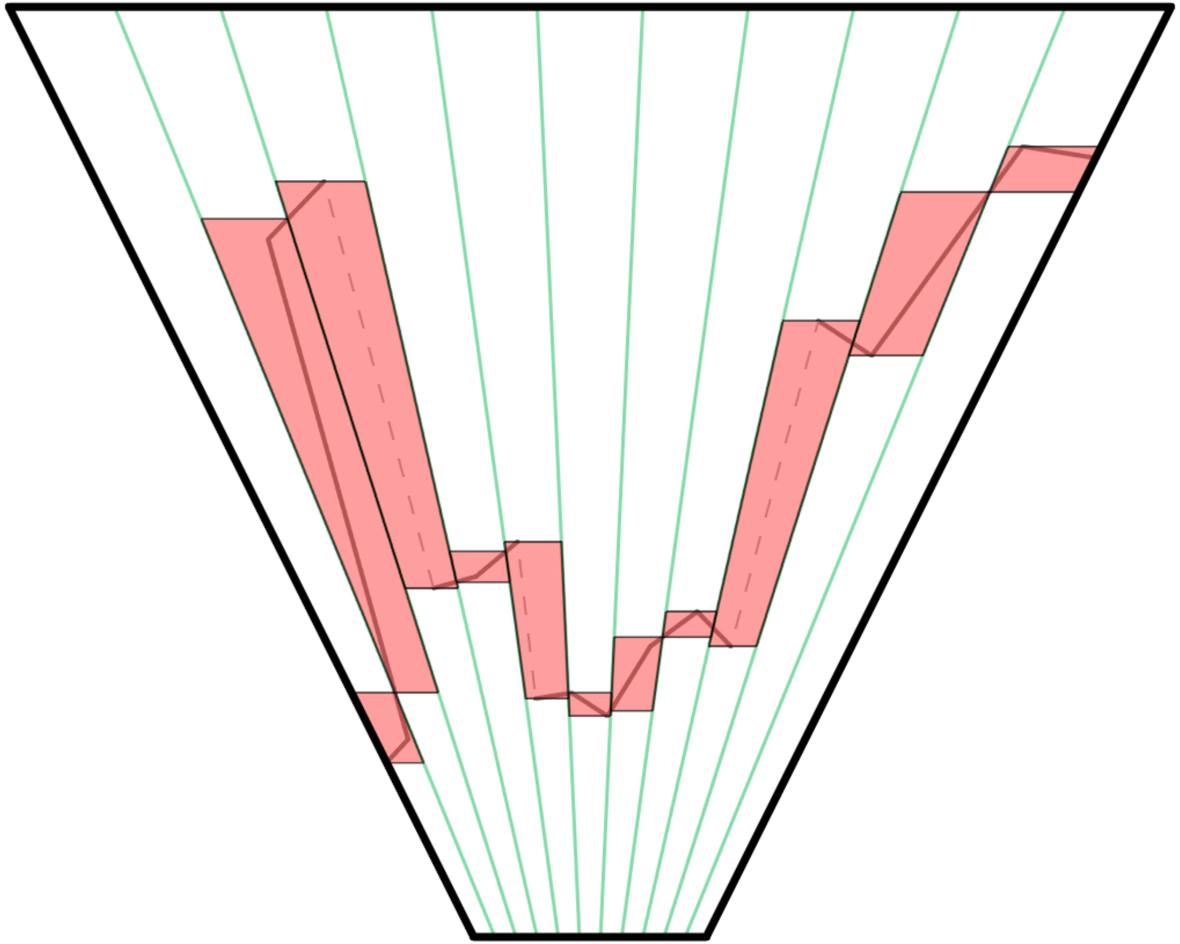
# Clustering and depth

- Tiled frustum



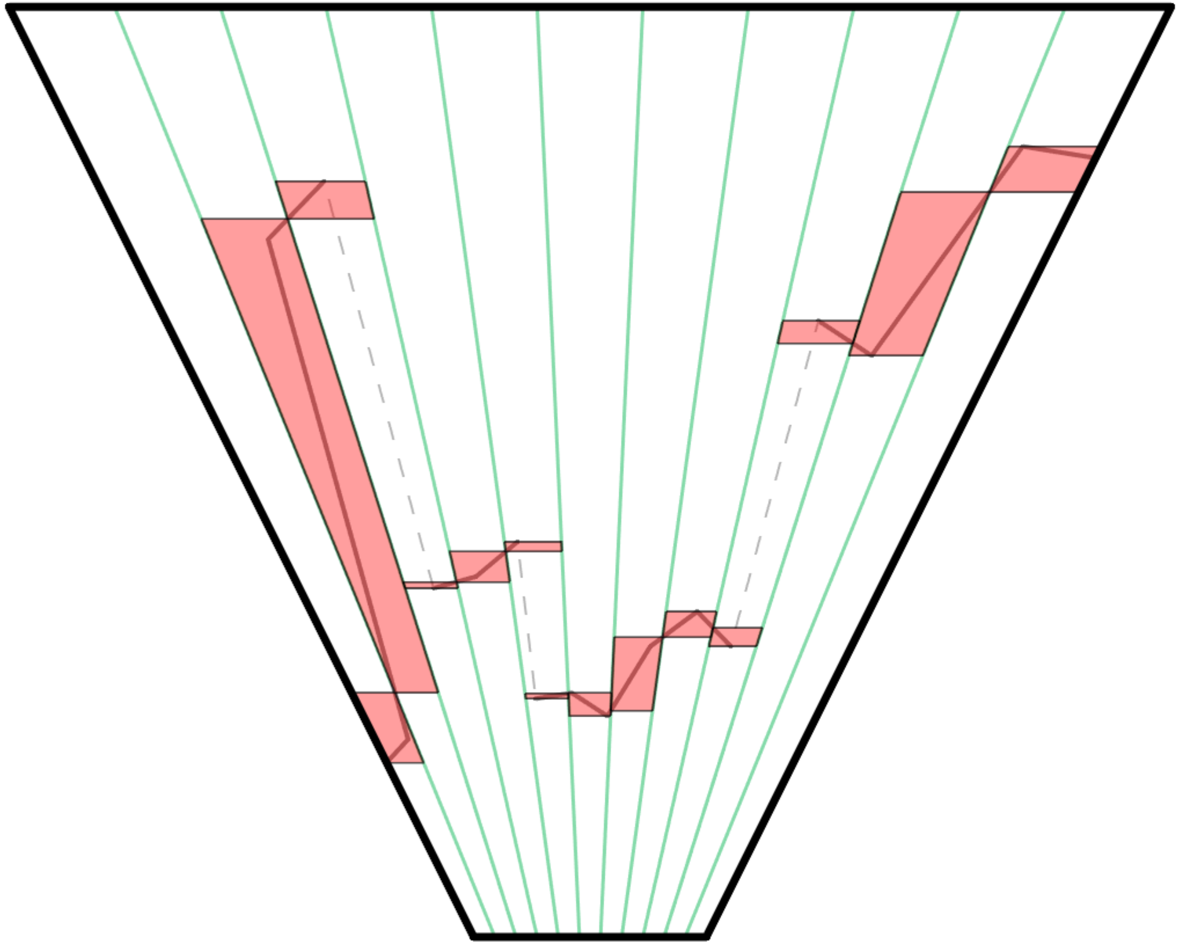
# Clustering and depth

- Depth ranges for Tiled Deferred / Forward+



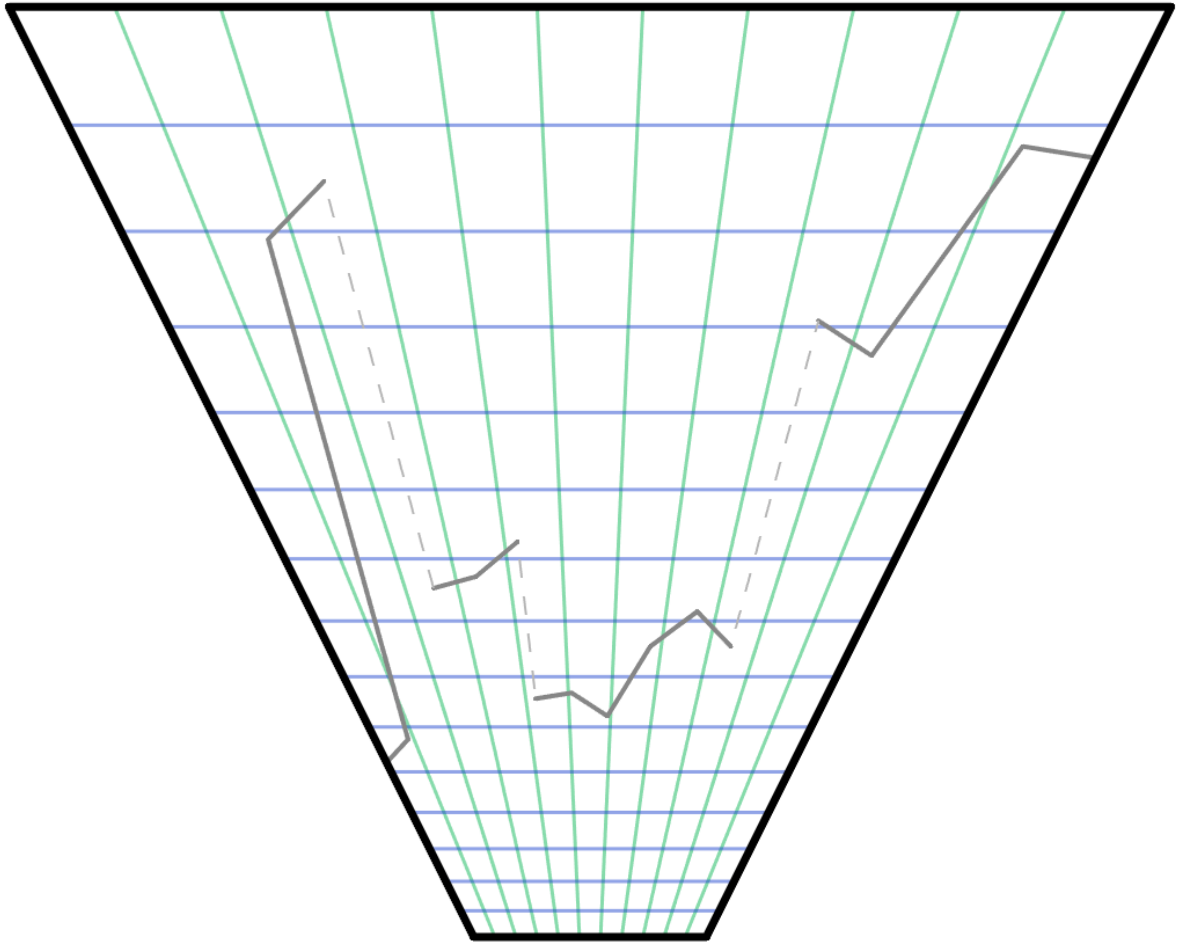
# Clustering and depth

- Depth ranges for Tiled Deferred / Forward+ with 2.5D culling



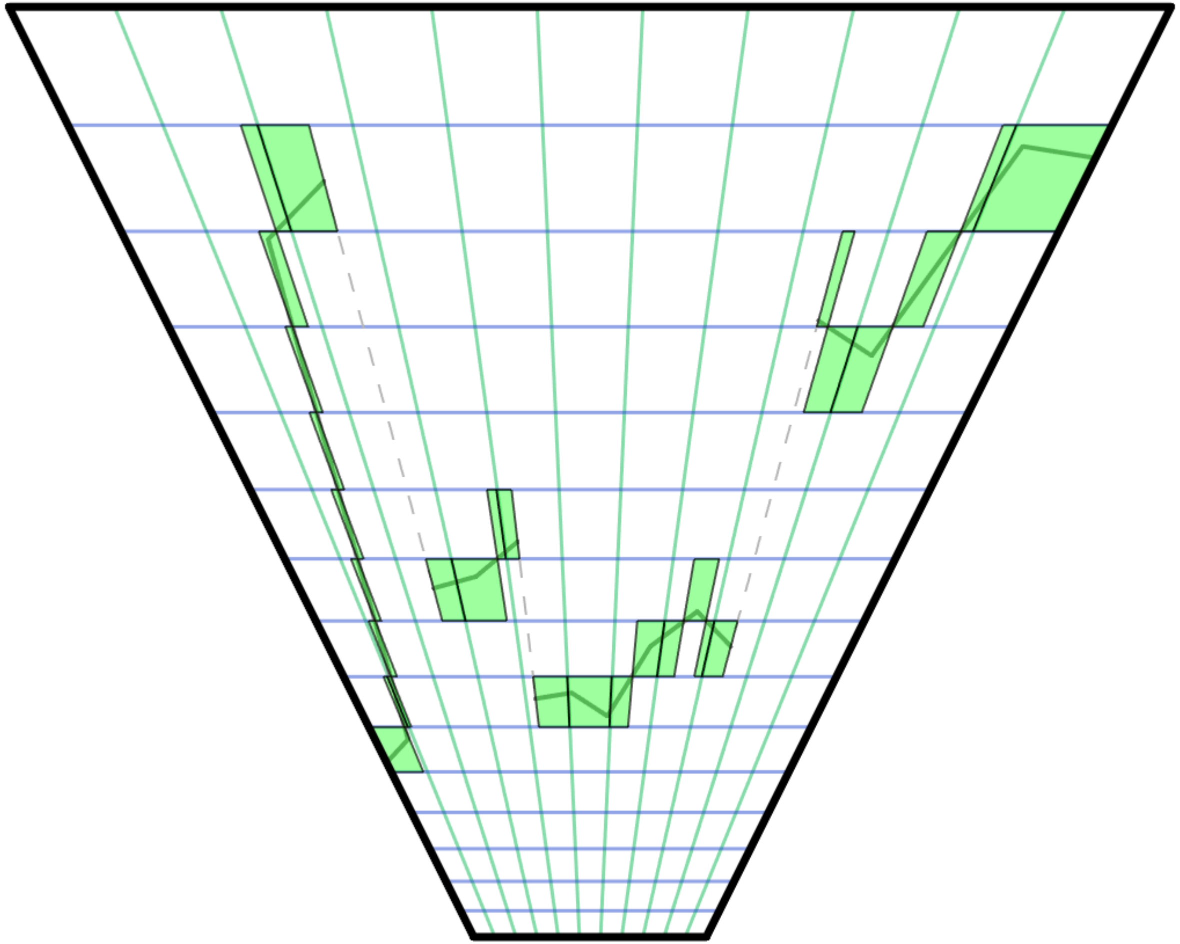
# Clustering and depth

- Clustered frustum



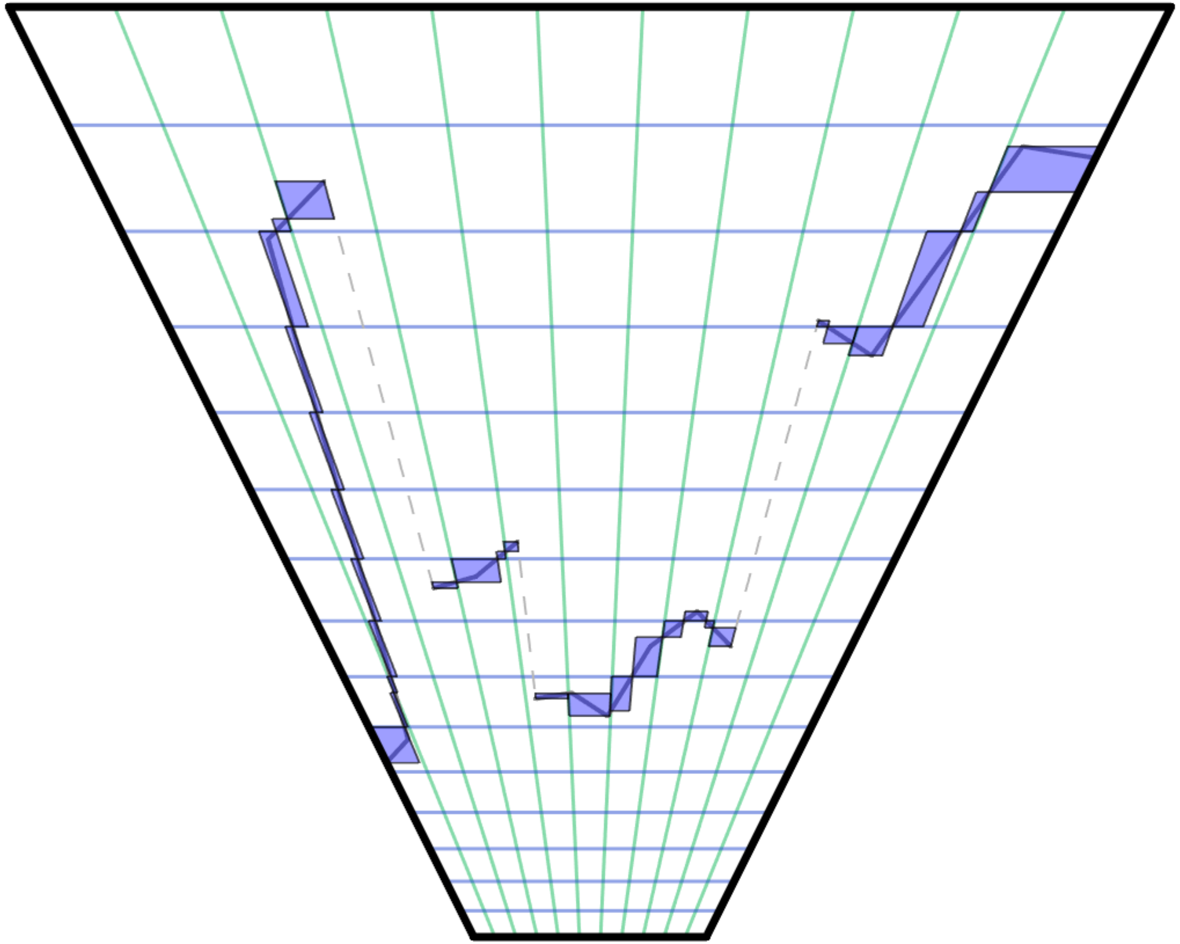
# Clustering and depth

- Implicit depth ranges for clustered shading



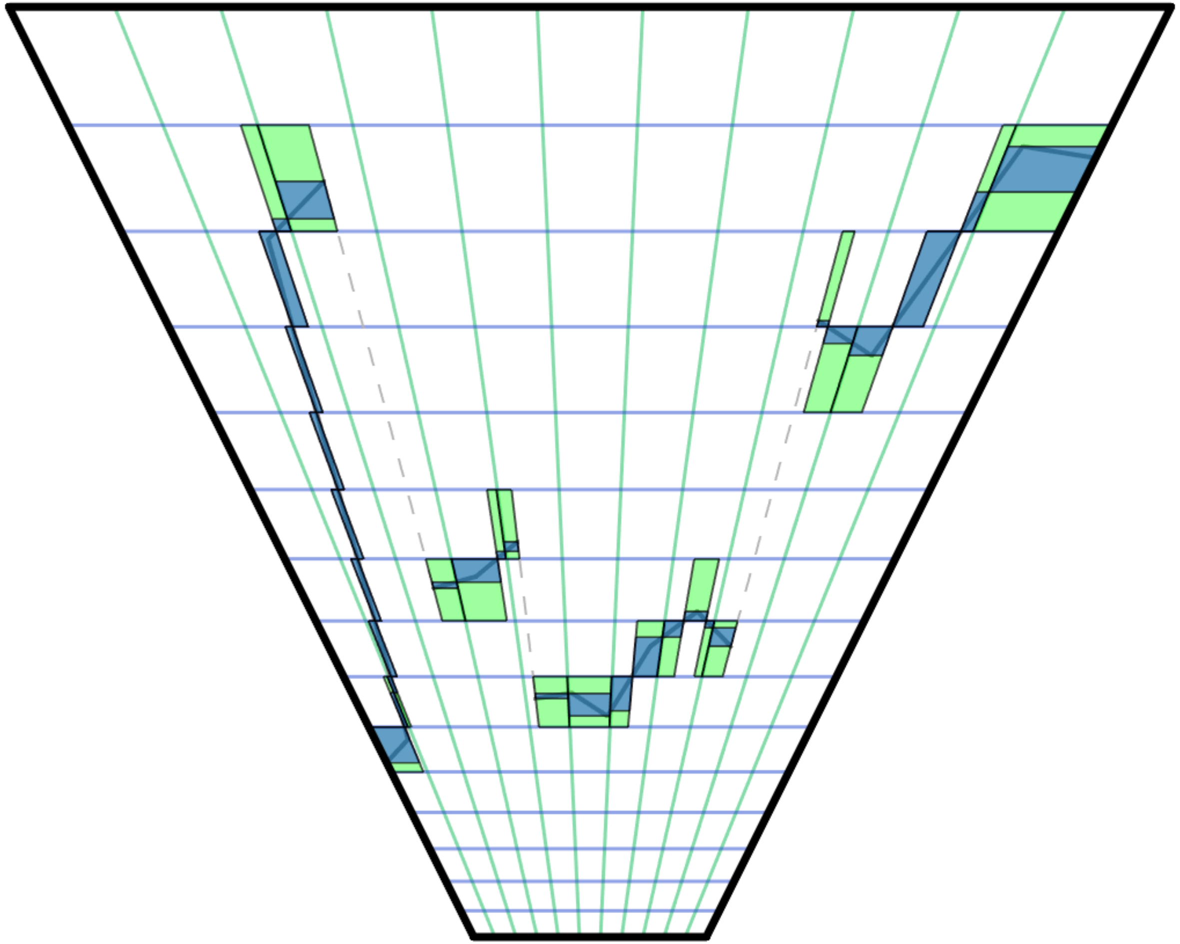
# Clustering and depth

- Explicit depth ranges for clustered shading



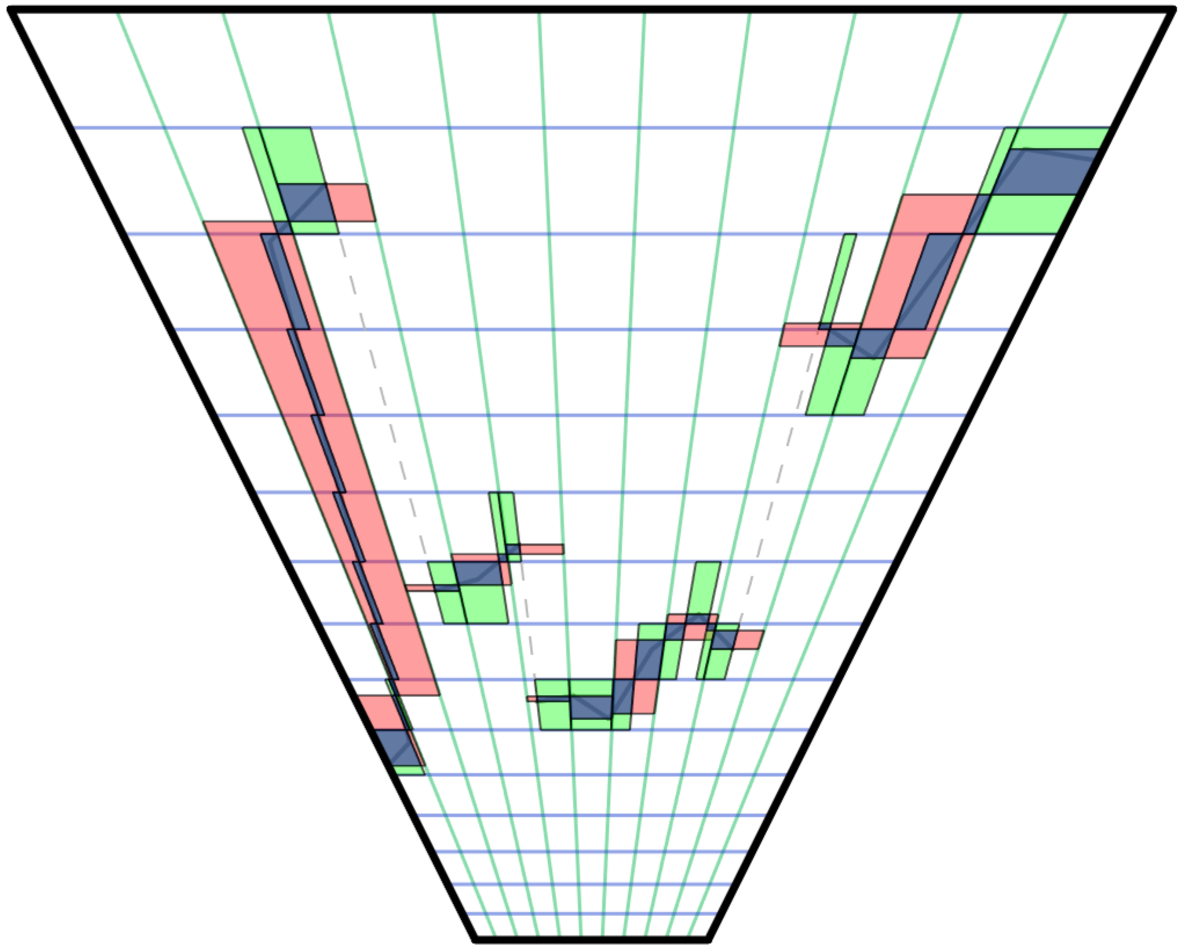
# Clustering and depth

- Explicit versus implicit depth ranges



# Clustering and depth

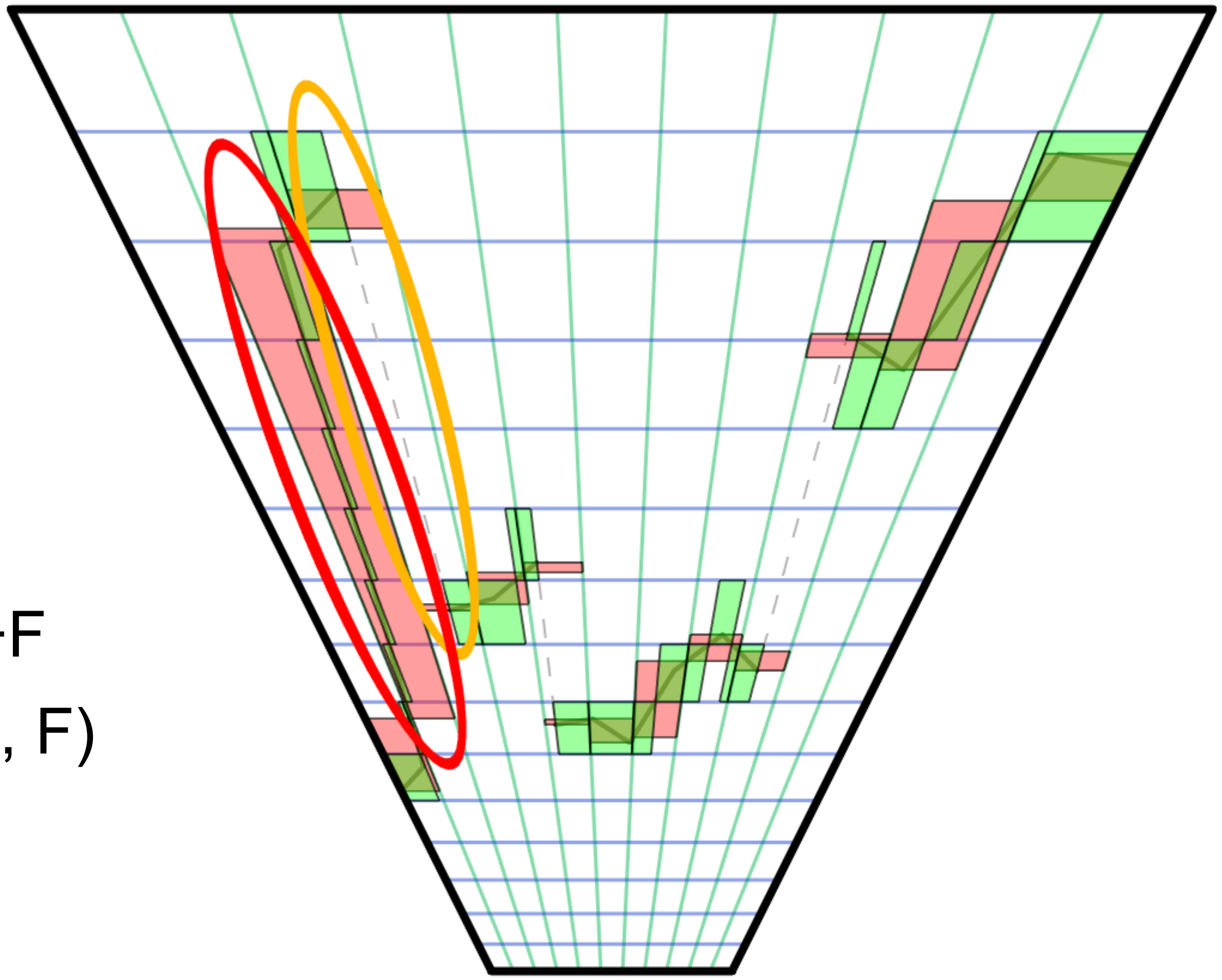
- Tiled vs. implicit vs. explicit depth ranges



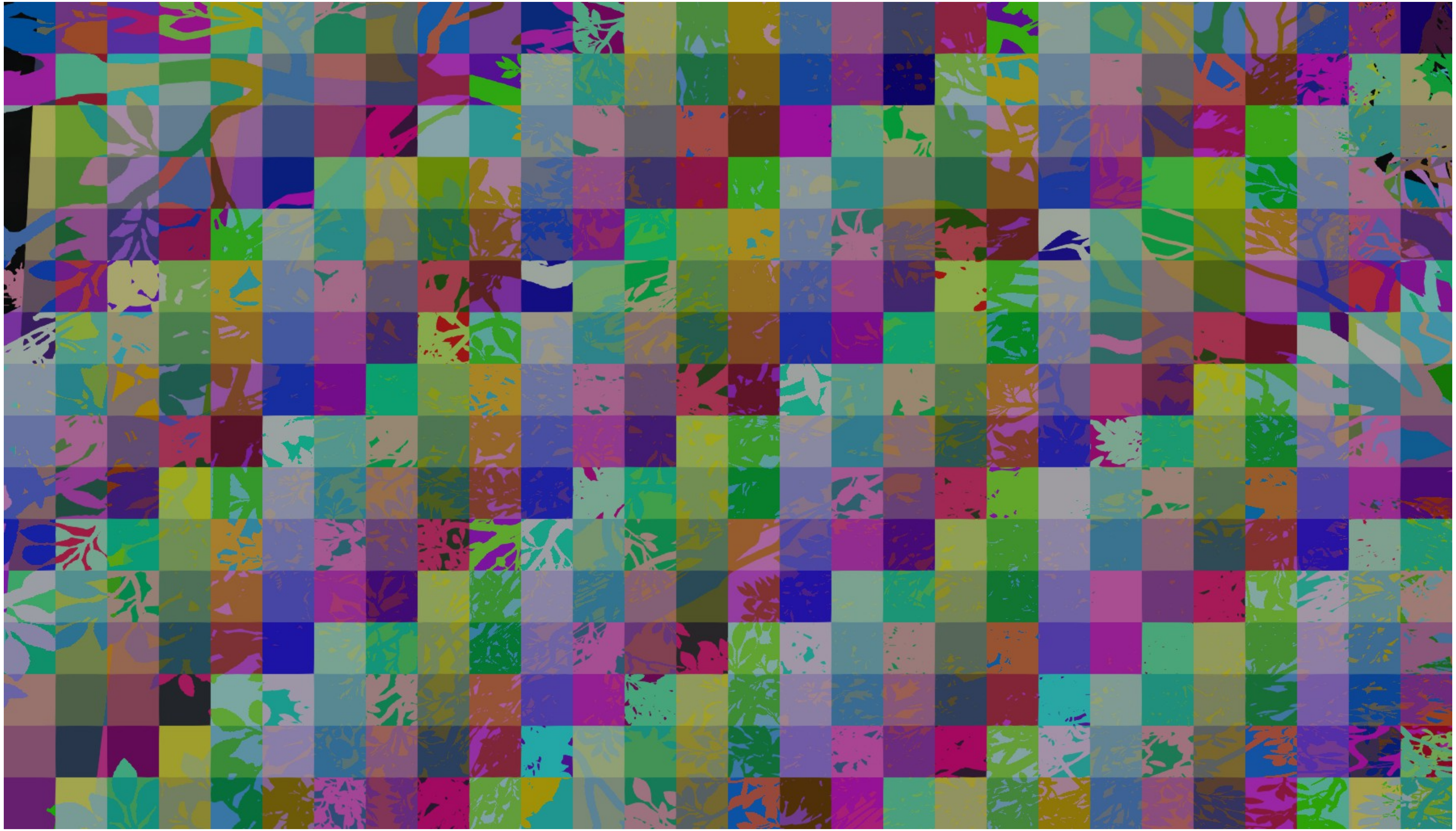
# Wide depths

- Depth discontinuity range A to F
  - Default Tiled: A+B+C+D+E+F
  - Tiled with 2.5D: A + F
  - Clustered: ~max(A, F)

- Depth slope range A to F
  - Default Tiled: A+B+C+D+E+F
  - Tiled with 2.5D: A+B+C+D+E+F
  - Clustered: ~max(A, B, C, D, E, F)



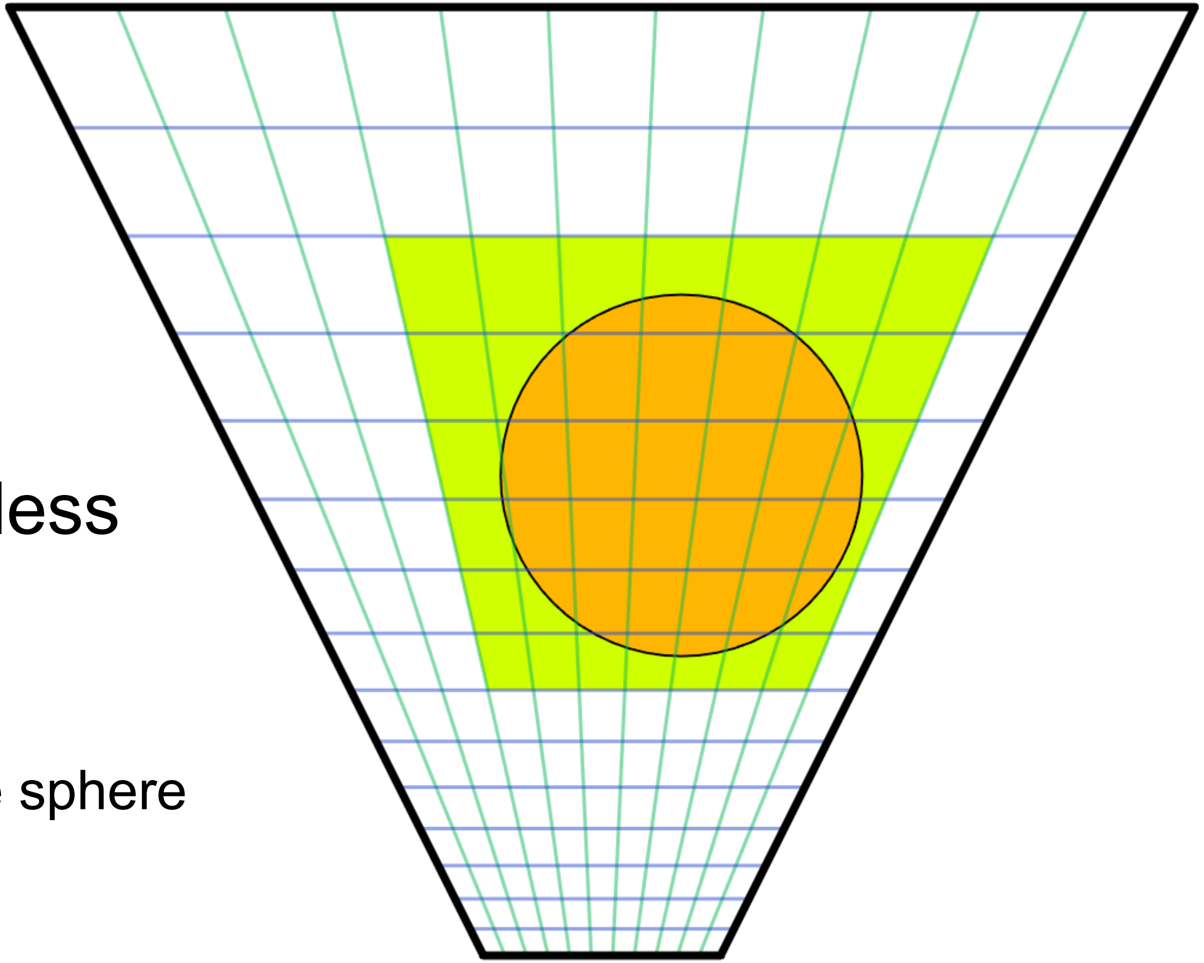
# Data coherency



# Branch coherency

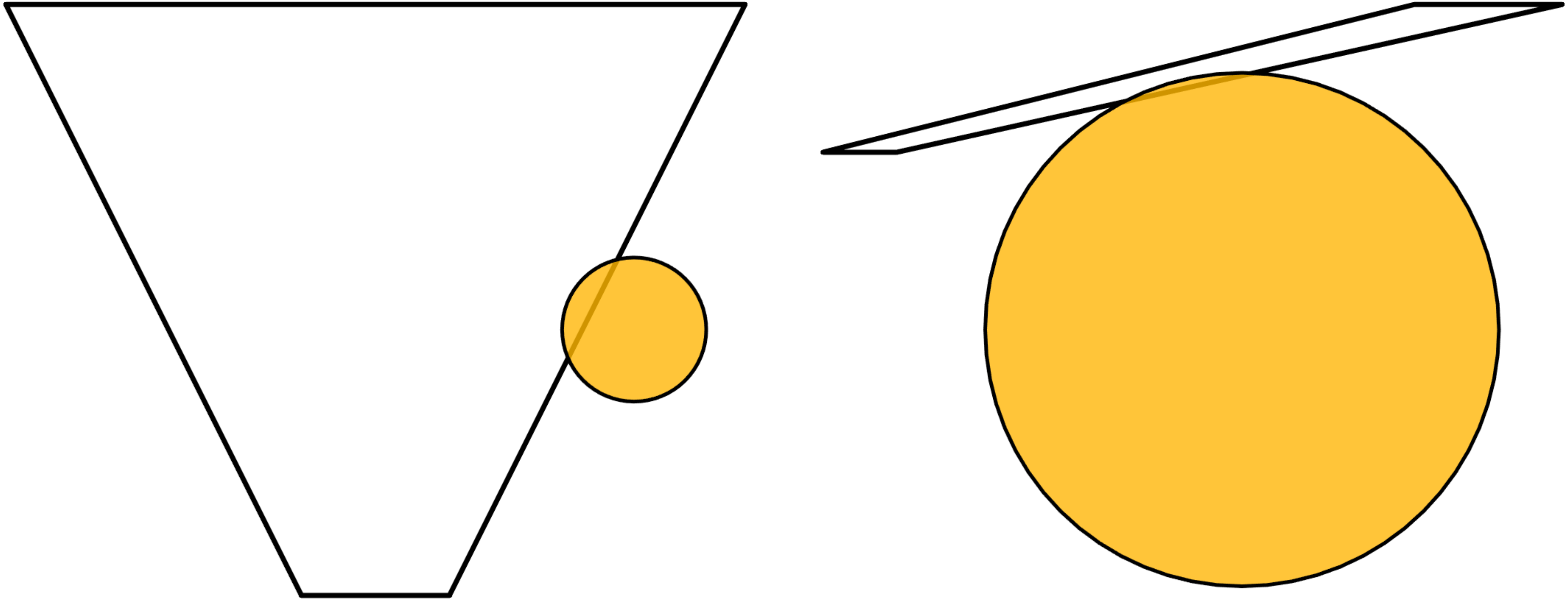
# Culling

- Want to minimize false positives
- Must be conservative
  - But still tight
  - Preferably exact
    - But not too expensive
    - Surprisingly hard!
- 99% frustum culling code useless
  - Made for view-frustum culling
    - Large frustum vs. small sphere
    - We need small frustum vs. large sphere
  - Sphere vs. six planes won't do



# Culling

- Your mental picture of a frustum is wrong!

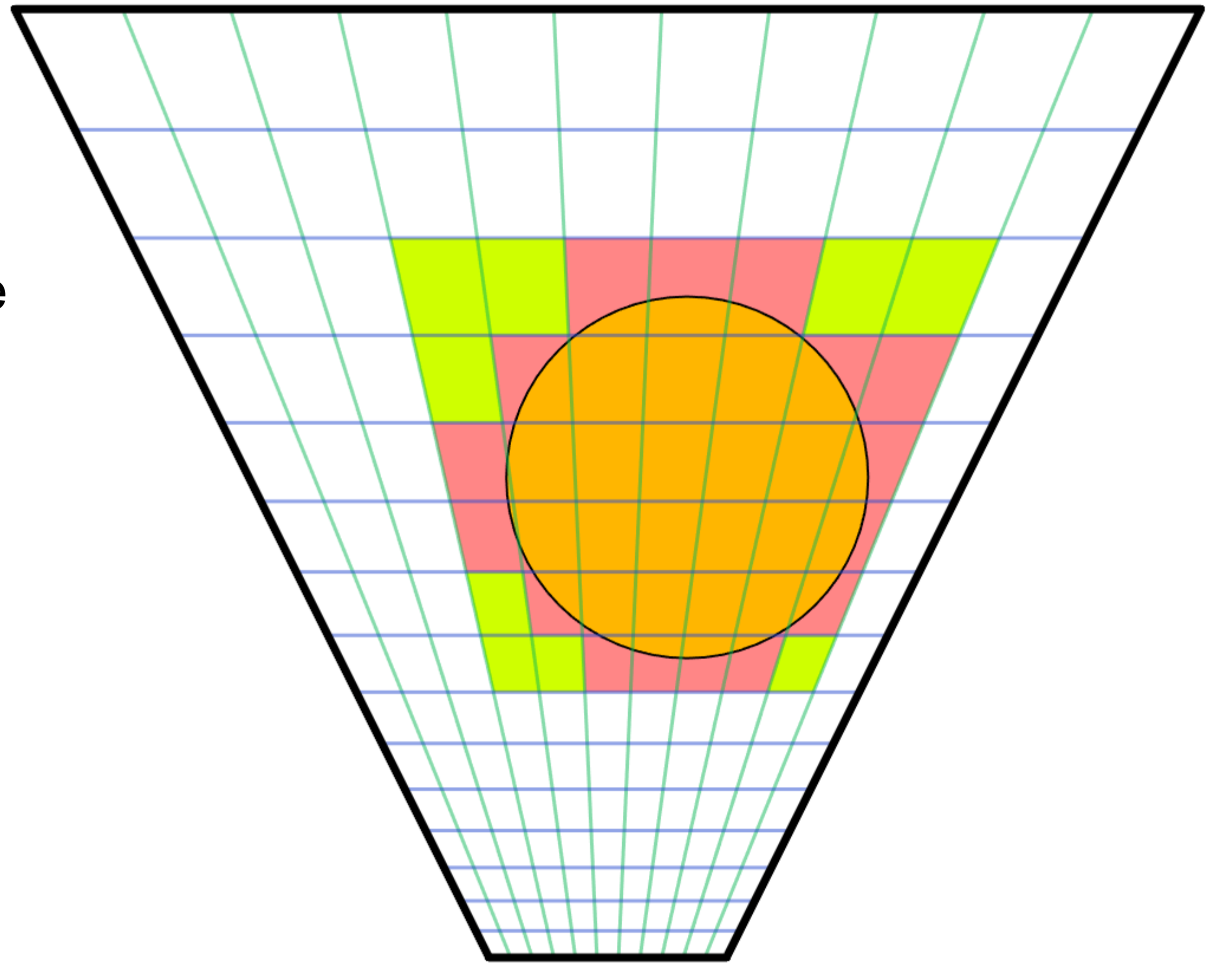


# Culling

- “Fun” facts:
  - A sphere projected to screen is not a circle
  - A sphere under projection is not a sphere
  - The widest part of a sphere on screen is not aligned with its center
  - Cones (spotlights) are even harder
- Frustums are frustrating (pun intended)
- Workable solution:
  - Cull against each cluster's AABB

# Pointlight Culling

- Our approach
  - Iterative sphere refinement
    - Loop over z, reduce sphere
    - Loop over y, reduce sphere
    - Loop over x, test against sphere
  - Culls better than AABB
    - Similar cost
    - Typically culling 20-30%

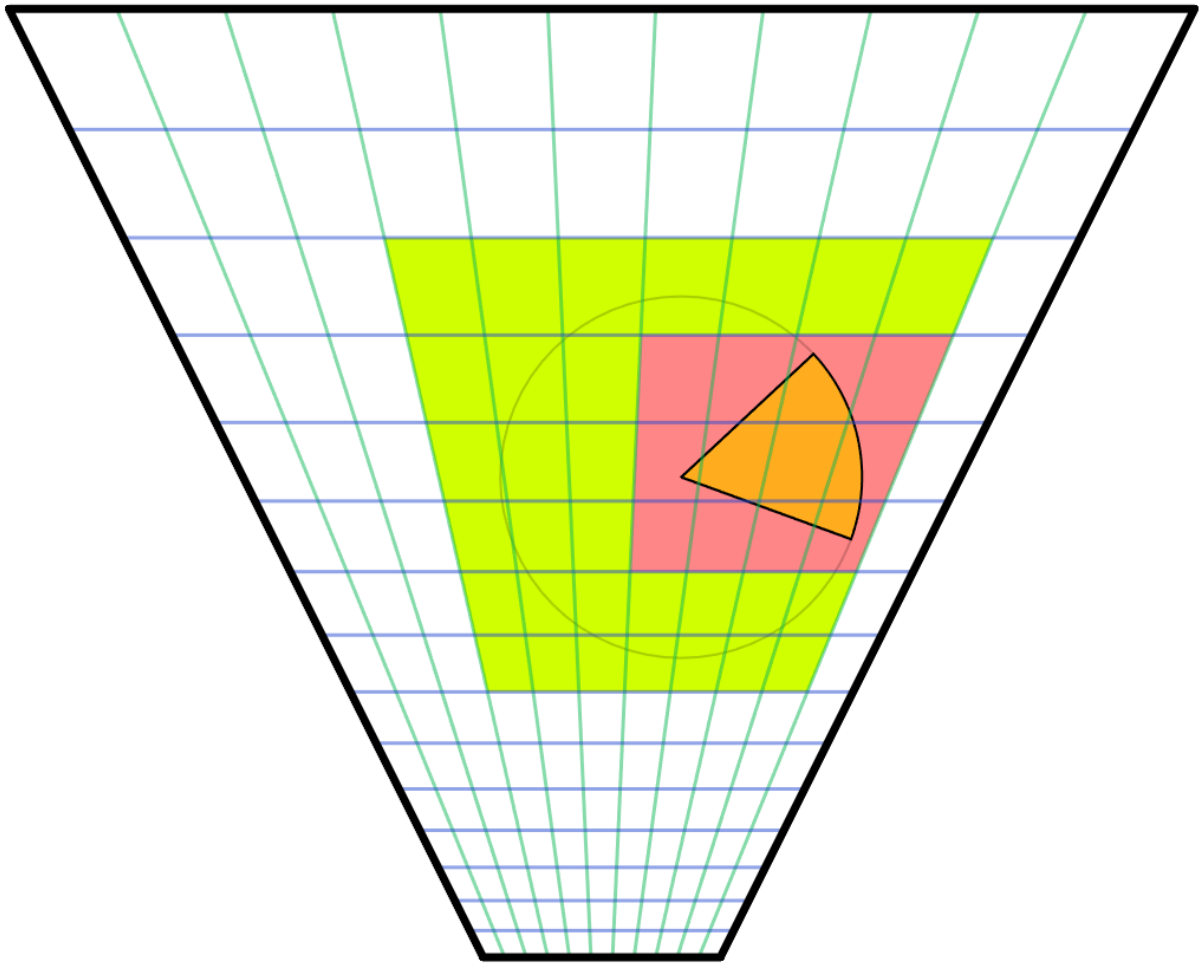


# Culling pseudo-code

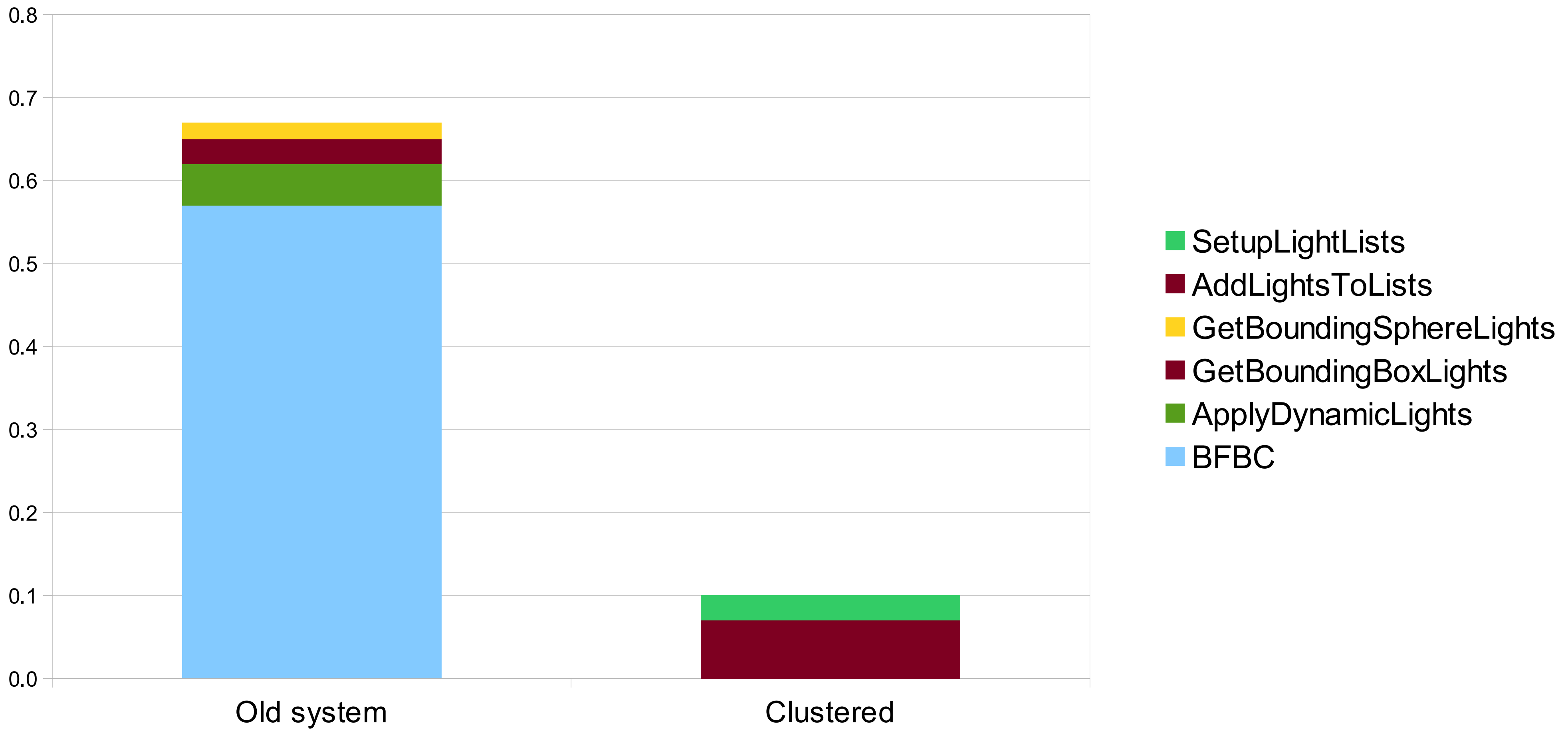
```
for (int z = z0; z <= z1; z++) {
     float4 z_light = light;
     if (z != center_z) { // Use original in the middle, shrunken sphere otherwise
          const ZPlane &plane = (z < center_z)? z_planes[z + 1] : -z_planes[z];
          z_light = project_to_plane(z_light, plane);
     }
     for (int y = y0; y < y1; y++) {
          float3 y_light = z_light;
          if (y != center_y) { // Use original in the middle, shrunken sphere otherwise
               const YPlane &plane = (y < center_y)? y_planes[y + 1] : -y_planes[y];
               y_light = project_to_plane(y_light, plane);
          }
          int x = x0;
          do {           // Scan from left until with hit the sphere
               ++x;
          } while (x < x1 && GetDistance(x_planes[x], y_light_pos) >= y_light_radius);

          int xs = x1;
          do {           // Scan from right until with hit the sphere
               --xs;
          } while (xs >= x && -GetDistance(x_planes[xs], y_light_pos) >= y_light_radius);

          for (--x; x <= xs; x++)      // Fill in the clusters in the range
               light_lists.AddPointLight(base_cluster + x, light_index);
     }
}
```

# Spotlight Culling

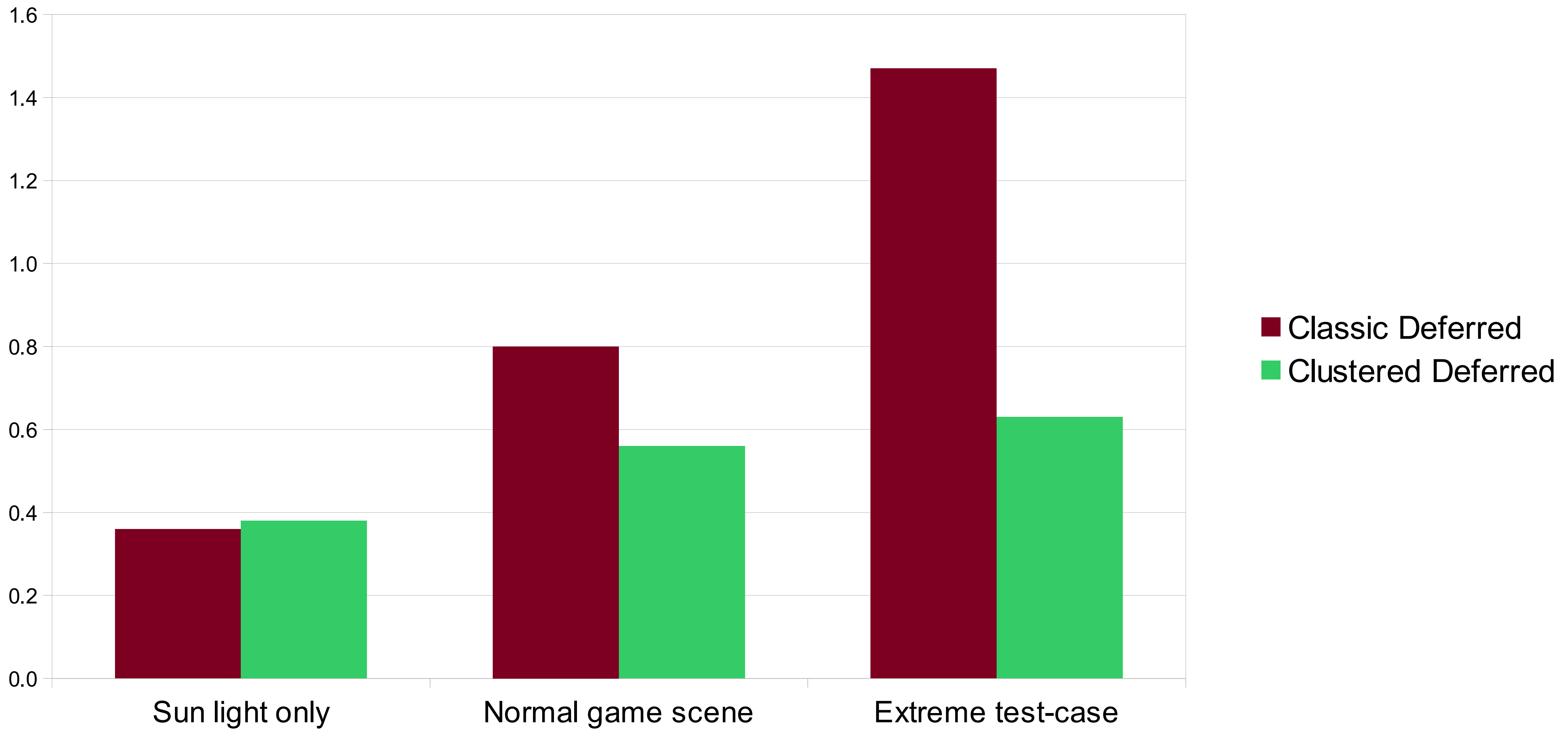
- Our (not so optimal) approach
  - Iterative plane narrowing
    - Find sphere cluster bounds
    - In each six directions
      - Do plane-cone test and shrink
    - Fill remaining “cube”



# CPU Performance



# GPU Performance



# Future work

- Clustering strategies
  - Screen-space tiles + depth
  - Screen-space tiles + distance
  - View-space cascades
  - World space
    - Allows light evaluation outside of view-frustum (reflections etc.)
  - Dynamic adjustments?
- Shadows
  - Need all shadow buffers up-front
  - May need more data per light

# Conclusions

- Clustered shading is practical for games
  - It's fast
  - It's flexible
  - It's simple
  - It opens up new opportunities
    - Evaluate light anywhere
    - Ray-trace your volumetric fog

# Questions?

@_Humus_
emil.persson@avalanchestudios.se

AVALANCHE STUDIOS