Reimplementing Masked Software Occlusion Culling

Taylor Folkersen folkerse@ualberta.ca

Program: Master of Science (Crse) in Computing Science

December 7, 2020

[illegible]

Abstract

Modern video games have scenes with increasing size, complexity, and interactivity. Popular games like Grand Theft Auto V, Minecraft, and the genre of battle royale games all have large scenes with lots of objects, many of which are dynamic. If objects are determined to be occluded from view by other objects, they can be skipped in the rendering pipeline, increasing rendering performance if done efficiently. This is called occlusion culling, and is important due to the rising frequency of complex scenes like those mentioned. Some methods of occlusion culling are meant for static scenes, like the idea of "potentially visible sets" -- keeping a list of potentially visible objects for regions of a scene, and only rendering those. Methods for dynamic environments are becoming increasingly important, and more recent methods are better suited for such environments, such as the method discussed in "Masked Software Occlusion Culling", by J. Hasselgren, M. Andersson, and T. Akenine-Möller. My project reimplements this method and has a few scenes meant to illustrate the culling effect, but with the caveat that I do not use AVX or other SIMD instructions, as my system only supports the first version of AVX (which does not support bit shifts) making my solution less performant than the implementation mentioned in their paper. Due to this, and their paper leaving out important implementation details, I am more interested in implementation than just performance.

Project Write Up

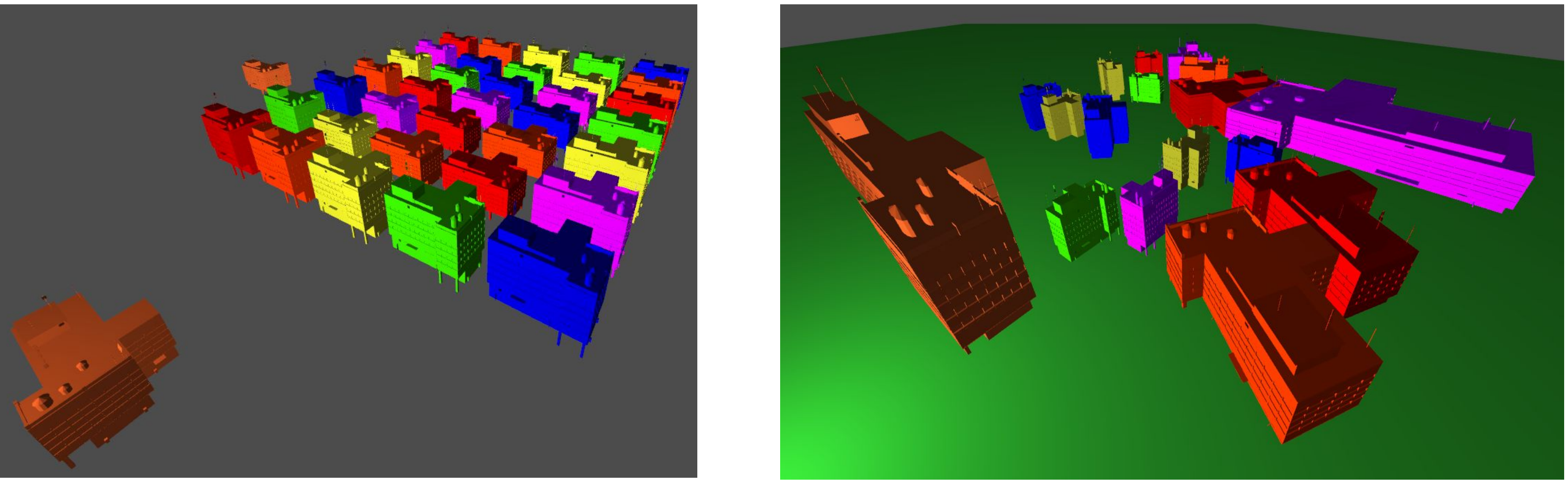


Figure 1: Left: the default scene in my implementation. Right: the alternate scene in my implementation.

I implement an interactive program in OpenGL where the user can fly around in one of two scenes pictured in Figure 1. These scenes have many buildings which can occlude the camera's view of one another depending on where the camera is. Being behind the brown building on the left side of either scene in Figure 1 will occlude vision to all other buildings in the scene, and these occluded buildings need not be rendered.

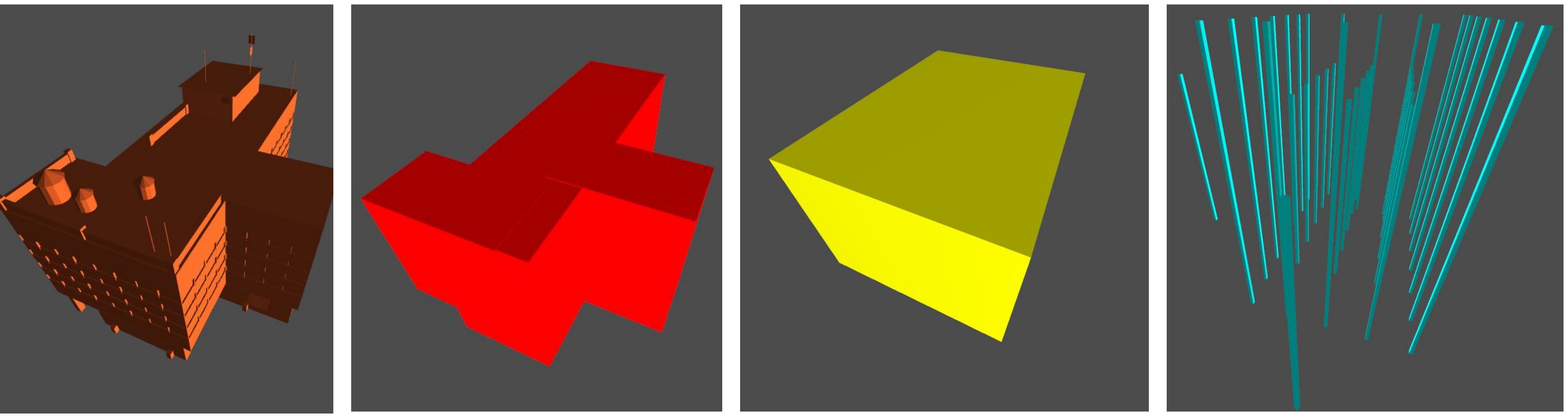


Figure 2: First: the "main mesh" to be rendered in the scene. Second: the object's "occlusion mesh". Third: the object's bounding box. Fourth: view of the default scene in marker mode, showing a "marker" where each object is.

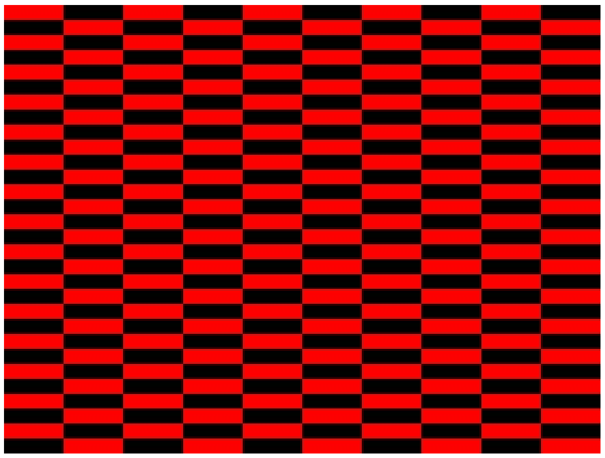
The basic idea of the algorithm proposed by Hasselgren et al. is to maintain a depth buffer on the CPU which is interacted with by using two meshes: an object's bounding box, and its "occlusion mesh", or "occluder", which represents parts of the object that should block vision. The bounding box encloses the main mesh, and the main mesh encloses the occlusion mesh. Before drawing an object, its bounding box is checked against the depth buffer to see if it is visible. If occluded, no more work is done for this object. If this test determines the object to be visible, then the object's occluder is rasterized into the depth buffer, potentially updating depth information in the process, and then the object's main mesh is rendered in OpenGL. Each object in the scene has multiple meshes, shown in Figure 2. The main mesh of the building in my scenes is from the user "lukass12" or "Lukas Carnota" on cgtrader.com. I made the occluder, bounding box, and marker meshes in Blender based on the main mesh.

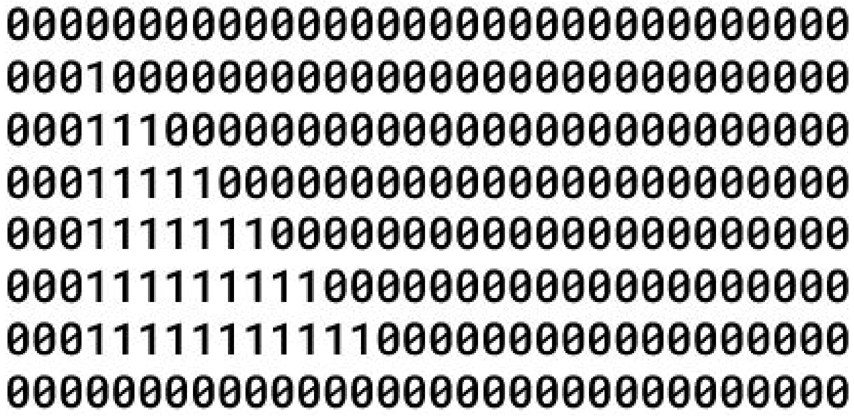
The user can press the F key to cycle rendering modes, changing which mesh is drawn to the screen. The rendering modes do not affect the culling logic. The marker mode makes the culling effect very apparent as the markers can be seen from most orientations in the scene, and when an object is occluded, nothing is rendered for it in any rendering mode, so the user can see these markers disappear when objects are culled.

Algorithm 1 (Rendering objects):

```
On each frame:
Clear depth buffer
Sort scene objects in rough front to back order relative to the camera
For each object o in sorted scene:
    if depthTest(o.boundingBox) then:
        renderToDepthBuffer(o.occluder)
        renderToScreen(o.mainMesh)
```

Algorithm 1 is essentially “traverseSceneTree” from the paper by Hasselgren et al., but without frustum culling. It describes how to render a scene to achieve occlusion culling, and is simple, however, the individual lines of this algorithm are more complicated. I explain my implementation of each component.





```
00000000000000000000000000000000
00010000000000000000000000000000
00011100000000000000000000000000
00011111000000000000000000000000
00011111110000000000000000000000
00011111111100000000000000000000
00011111111111000000000000000000
00000000000000000000000000000000
```

Figure 3: Left: visualization of a tiling of a 320 width by 240 height depth buffer using 32 width by 8 height tiles. Tiles colored with checkerboard pattern for clarity. Right: a triangle rasterized into a tile’s bit mask. In practice it is unlikely that a triangle will fit into one tile unless the triangle is very small.

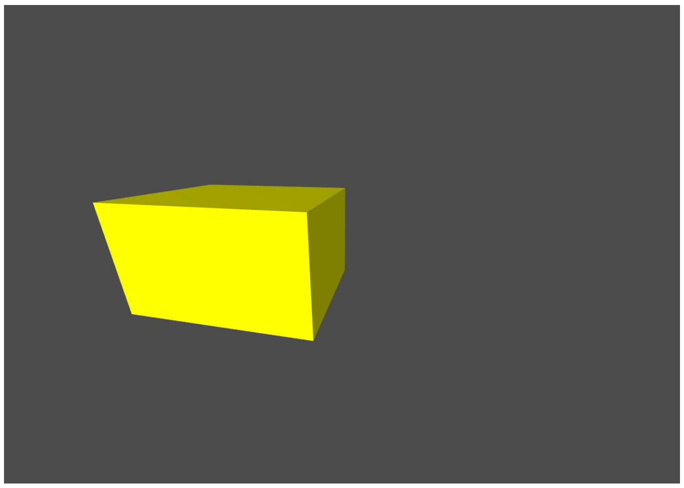
I use the depth buffer and its operations as proposed by Hasselgren et al. in their paper. The depth buffer spans the entire window and normally has the same dimensions as the window, meaning it has some data for every pixel of the window. The buffer is partitioned into tiles, with each tile representing a rectangular region of pixels, as shown in the left of Figure 3. A tile in the buffer stores three values: zReference, zWorking, and a 32 width by 8 height bit mask, shown in the right of Figure 3. A tile is 32 by 8 because in Hasselgren et al.’s implementation, AVX instructions can operate simultaneously on 8 unsigned 32 bit integers, handling operations for a tile with only a few instructions. zReference is the “reference depth”, representing the depth of pixels in the background of the tile. zWorking is the “working depth”, representing the maximum depth of pixels in the foreground of the tile. The mask has one bit for each pixel of the tile, with a 0 meaning the pixel belongs to the reference layer, and a 1 meaning the pixel belongs to the working layer. In the right of Figure 3, the rasterized triangle belongs to the working layer of the tile. Because zReference represents the depth of the background of the tile, every pixel in the tile is at most zReference depth away from the camera, so this value is used in the depth test with the bounding box. When a triangle of an occlusion mesh is rasterized into the buffer, mask bits may be set and the value of zWorking may be moved deeper, often for multiple tiles. zWorking is not used for depth testing -- it is there so that when the mask of a tile fills up, zReference takes on the value of zWorking, then zWorking is reset to a depth of 0 (minimum depth) and the mask is zeroed. The minimum depth value is 0.0 and the maximum depth value is 1.0. To clear the depth buffer at the start of a frame, each tile's mask is zeroed, zReference set to 1.0, and zWorking set to 0.0. Hasselgren et al. actually use AVX shuffle operations to implement tile sizes of 8 by 4, but I use 32 by 8, which may slightly negatively affect culling of small or distant objects, but is simpler and likely makes culling logic faster in the absence of AVX instructions. I also assume the screen width is a multiple of 32, and the screen height is a multiple of 8.

To sort scene objects, I store each object's center point with the object's data, and each frame apply the model and view transformations to convert this point into camera space, then I use the squared distance of the resulting coordinates as the object's distance. Objects are rendered in rough front to back order so that significant occluders hopefully get rendered first

$$p = view * model * objectCenterPoint$$
$$p' = p / p.w$$
$$dist^2 = p'.x^2 + p'.y^2 + p'.z^2$$

Distance computation used in sorting scene objects

A slightly more efficient solution would compute the distance using the world coordinates of the object and camera, using fewer, or no matrix multiplications, but I use this method for simplicity.



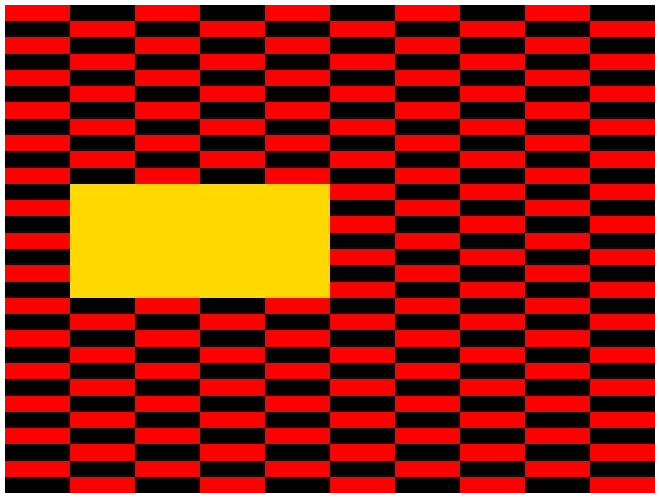


Figure 4: Left: bounding box of an object. Right: depth buffer tiles overlapped by corresponding bounding rectangle shown in yellow.

To do the depth test, Hasselgren et al. transform an object's bounding box into a bounding rectangle in screen space, and then its minimum depth, zMin, is compared to the reference depths of all overlapped tiles in the depth buffer. An object is occluded if the zMin of its bounding box is greater than the reference depths of all overlapped tiles of the depth buffer -- this means that the bounding box is behind the backgrounds of all overlapped tiles. First I apply the model and view transformations to each triangle of the bounding box, then clip resulting triangles against the near plane. Each clipping of a triangle may output 0, 1, or 2 triangles. After clipping, the perspective transformation is applied. After perspective transformation, the min and max of the X and Y coordinates over all triangles specifies the bounding rectangle, and the min of the Z coordinates is the depth to use in comparison with each overlapping tile's zReference.

Assuming pixel space has (0, 0) at the top left corner, with right and down being positive directions:

Algorithm 2 (My implementation of the depth test from the paper by Hasselgren et al.)

```
//returns true if bounding box determined to be visible
depthTest(boundingBox): //assume boundingBox is passed by value and not by reference
//these are OpenGL coordinates
minX = minY = minZ = ∞
```

```
maxX = maxY = -∞
for each Triangle t in boundingBox:
    for each Vertex v in t:
        v = view * model * v
        v = v / v.w
    clippedTriangles = clipToNearPlane(t) //might make 0, 1, or 2 triangles
    for each Triangle t2 in clippedTriangles:
        for each Vertex v in t2:
            v = perspective * v
            v = v / v.w
            minX = min(minX, v.x)
            minY = min(minY, v.y)
            minZ = min(minZ, v.z)
            maxX = max(maxX, v.x)
            maxY = max(maxY, v.y)
convert minX, minY, minZ, maxX, maxY from GL coordinates to pixel coordinates
//following are depth buffer coordinate ranges of overlapped tiles, where (0, 0) is the top left
//tile; right and down are positive
left = max(floor(minX / 32), 0)
right = min(ceiling(maxX / 32), screenWidth / 32 - 1)
top = max(floor(minY / 8), 0)
bottom = min(ceiling(maxY / 8), screenHeight / 8 - 1)
for i from left to right:
    for j from top to bottom:
        if minZ < tiles[i, j].zReference:
            return true //in front of tile, so visible
return false
```

Although I do not frustum cull the bounding box like Hasselgren et al. do, a side effect of my implementation of this depth test is that off-screen objects are usually culled because their bounding rectangles are non-existent, either because they are behind the near plane and all their triangles are deleted by clipping, or the left/right or top/bottom ranges are invalid. This depth test could be somewhat expensive as even though the bounding box is just a box, a few matrix multiplications are done for each of its triangles, as well as clipping each triangle, which could create a few more triangles, and result in a few more matrix multiplications. I tried this same test without clipping (i.e. by just applying model, view, and perspective transformations to each triangle of the bounding box and taking the min/max of those coordinates), and while this would slightly improve performance of the depth test, it results in erroneous culling, as shown in Figure 5, in places where the unclipped test culls more objects than the clipped one.

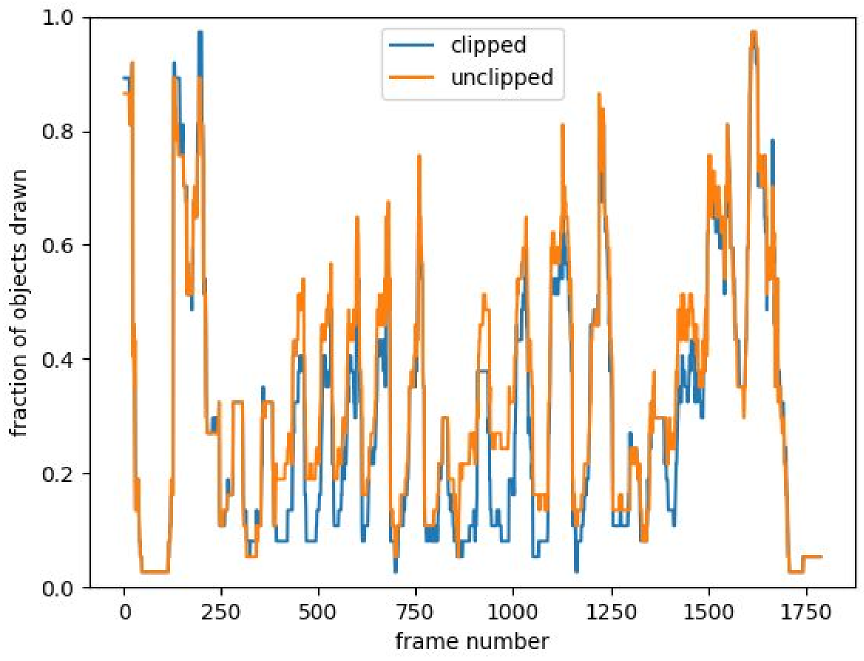
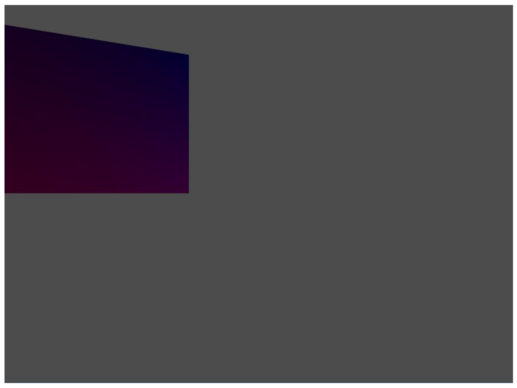
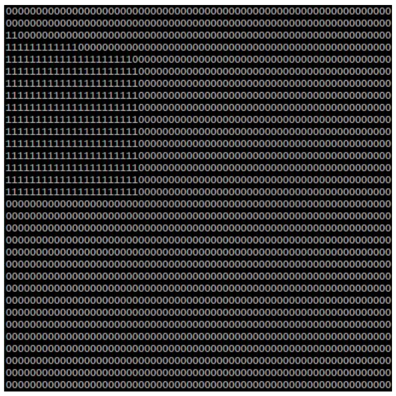
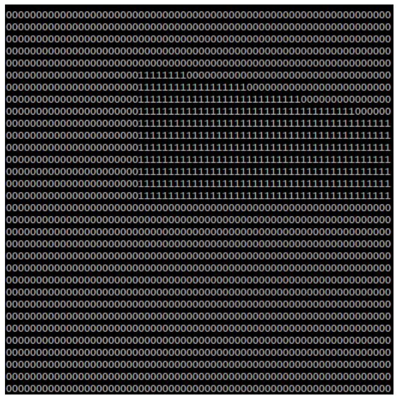


Figure 5: Culling for the same walk around the default scene using clipped and unclipped depth tests.





```
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
1100000000000000000000000000000000000000000000000000000000000000
1111111111110000000000000000000000000000000000000000000000000000
1111111111111111111110000000000000000000000000000000000000000000
1111111111111111111111000000000000000000000000000000000000000000
1111111111111111111111000000000000000000000000000000000000000000
1111111111111111111111000000000000000000000000000000000000000000
1111111111111111111111000000000000000000000000000000000000000000
1111111111111111111111000000000000000000000000000000000000000000
1111111111111111111111000000000000000000000000000000000000000000
1111111111111111111111000000000000000000000000000000000000000000
1111111111111111111111000000000000000000000000000000000000000000
1111111111111111111111000000000000000000000000000000000000000000
1111111111111111111111000000000000000000000000000000000000000000
1111111111111111111111000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
```



```
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000111111110000000000000000000000000000000000
0000000000000000000000111111111111111111000000000000000000000000
0000000000000000000000111111111111111111111111111000000000000000
0000000000000000000000111111111111111111111111111111111111000000
0000000000000000000000111111111111111111111111111111111111111111
0000000000000000000000111111111111111111111111111111111111111111
0000000000000000000000111111111111111111111111111111111111111111
0000000000000000000000111111111111111111111111111111111111111111
0000000000000000000000111111111111111111111111111111111111111111
0000000000000000000000111111111111111111111111111111111111111111
0000000000000000000000111111111111111111111111111111111111111111
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
```

Figure 6: Left: scene view of triangle with a vertex behind the near plane. Middle: correct rasterization into bit masks of said triangle. Right: incorrect rasterization of said triangle due to lack of clipping.

The depth buffer update also needs this same clipping. Triangles going behind the near plane result in incorrect rasterization as shown in the right of Figure 6. A less accurate, but slightly faster solution, could just discard triangles having a vertex behind the near plane, and I initially did this when I could not determine the cause of my incorrect rasterization, though this results in large occluders often not affecting the depth buffer, so I do not consider this further after solving the issue. I do clipping to the near plane with the Sutherland-Hodgman algorithm (Sutherland et al. 1974), taking the clip polygon as the deeper side of space split by the near plane. My implementation of this clipping algorithm works as so: to clip a triangle, an empty list of output vertices is made, then the triangle vertices are arbitrarily labelled as t_0, t_1, and t_2, then pairs (t_0, t_1), (t_1, t_2) and (t_2, t_0) are considered in this order. Each case is handled as shown in Figure 7. The output vertices are then turned into triangles, resulting in 0, 1, or 2 triangles.

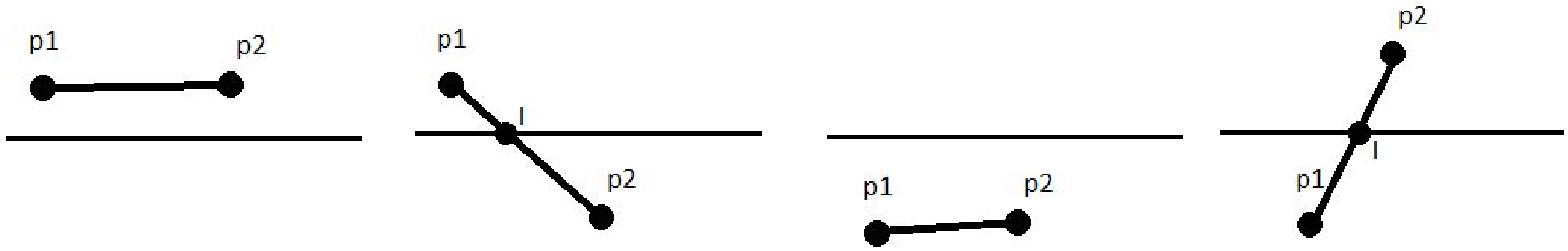
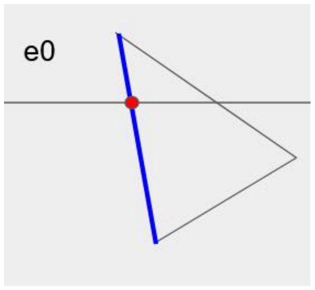
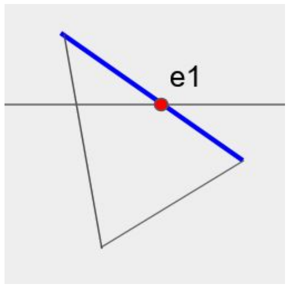
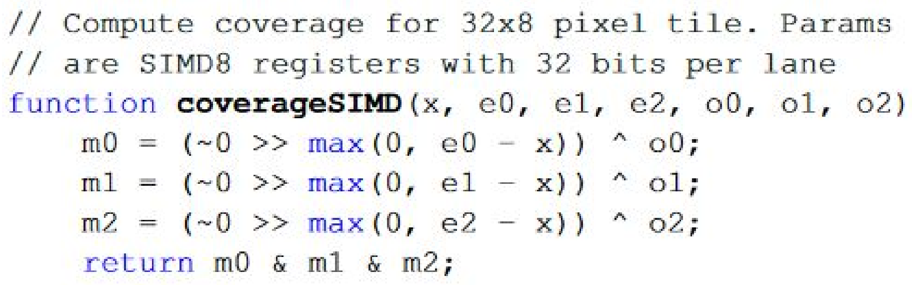


Figure 7: Horizontal line is the near plane. Lower part of image is behind the plane (behind viewer). In each case, p1 is the first vertex in the pair and p2 is the second vertex in the pair. Case 1: both points are inside the near plane, output p2. Case 2: going from inside to outside the near plane, compute I and output it. Case 3: p1 and p2 are both outside the near plane, output nothing. Case 4: going from outside to inside the near plane: compute I, output I and then p2.

I update the depth buffer following the method of Hasselgren et al., by using their rasterization method and depth update, though with minor differences. First I explain my implementation of the rasterization, and then the depth update. Right of Figure 8 shows the coverage function by Hasselgren et al., and is a screenshot from their paper. The parameter x is the X coordinate of the left edge of the buffer tile being updated. Parameters e0, e1, and e2 are the "events", which are X coordinates of intersections between horizontal scanlines and each triangle edge (each e_i is for a different edge). Each SIMD lane is one scanline, so each e_i represents the events for 8 scanlines and one triangle edge. On each scanline, for each edge of the triangle, a separate 32 bit register of all 1 bits is right shifted to the edge's event, e_i, then each of these results is XORed with their corresponding o_i, which is a mask of all 0s if the edge is left-facing, or all 1s if the edge is right-facing (this flips the result for right-facing edges). The bitwise AND of these results produces 32 bit registers whose 1s correspond to pixels within the triangle, and whose 0s correspond to pixels outside the triangle. This 32 width by 8 height mask is then bitwise ORed into the tile.







```
// Compute coverage for 32x8 pixel tile. Params
// are SIMD8 registers with 32 bits per lane
function coverageSIMD(x, e0, e1, e2, o0, o1, o2)
    m0 = (~0 >> max(0, e0 - x)) ^ o0;
    m1 = (~0 >> max(0, e1 - x)) ^ o1;
    m2 = (~0 >> max(0, e2 - x)) ^ o2;
    return m0 & m1 & m2;
```

Figure 8: Left: scanline intersecting left-facing triangle edge. Center: scanline intersecting right-facing triangle edge. Right: coverage function from Hasselgren et al. paper (this is a screenshot from their paper).

Due to not using AVX instructions, my version of the coverage function only computes coverage for one scanline at a time, as my parameters are 32 bit integers instead of vectors of 8 32 bit integers, though my version of this function is the same otherwise. Figure 9 visualizes regions of 1 bits for each edge according to the coverage function, assuming the pictured space is one tile. This describes the rasterization given the events.

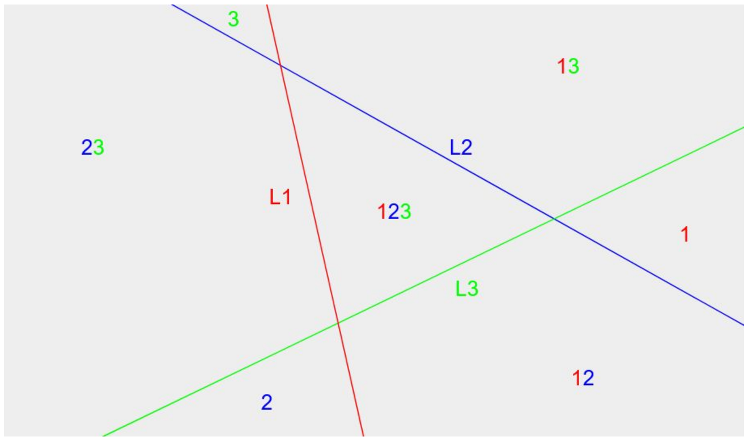


Figure 9: triangle edges are labelled L1, L2, L3, and labelled regions of 1 bits corresponding to each edge, as computed by the coverage function, are shown. L1 is left-facing so its 1 bits appear on its right side, while L2 and L3 are right-facing, so their 1 bits appear to their left sides. The bitwise AND on the last line of the coverage function corresponds to the intersection of these regions of 1s, which is the rasterized triangle itself, as shown in this figure.

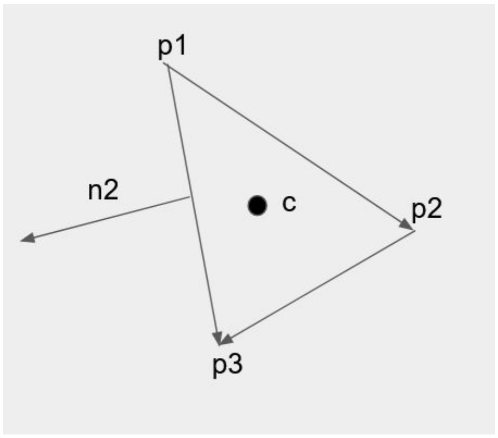


Figure 10: sorted/labelled triangle vertices, downward-pointing vectors for edges, triangle centroid, and left-pointing normal vector

Given a triangle in GL coordinates, I have devised a way of computing edge facing directions and tracking events. First, the triangle's vertices are sorted and then labelled in order of decreasing height, so that as in Figure 10, p1 has the greatest height and p3 has the least height. The edges (p2 - p1), (p3 - p1), and (p3 - p2) are then downward-pointing vectors. If two vertices have the same height, I add distinct, small epsilons to them until no vertices have the same height. For each triangle edge, E_i, a left-pointing normal vector, n_i, is derived as:

$$n_i = (E_i.y, \ -E_i.x)$$

Because E_i is downward-pointing, its Y coordinate is negative, so the X component of n_i is negative, meaning n_i points left. The centroid, C, of the triangle is computed as:

$$C = (p1 + p2 + p3) / 3$$

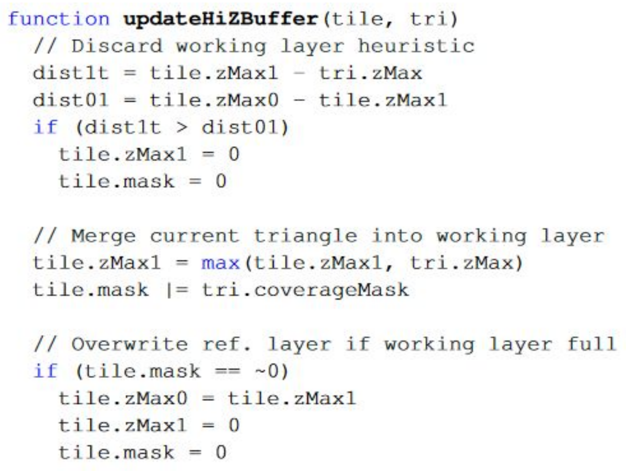
Then the sign of the expression:

$$n_i \cdot (C - p_i)$$

Where p_i is either vertex on E_i, gives the edge orientation. If negative, E_i is left-facing, and if positive, E_i is right-facing. To track events, I extrapolate each edge vector E_i back to the top of the screen to get a point F_i, and then compute S_i to be the Δx/Δy slope of E_i. Then the event at each scanline is computed as

$$e_i = (F_i - S_i * scanLineNumber).x$$

The subtraction term is because scanLineNumber increases as scanlines move toward the bottom of the screen (Y is decreasing). This e_i is then converted into pixel space and used by the coverage function to do rasterization.



```
function updateHiZBuffer(tile, tri)
  // Discard working layer heuristic
  dist1t = tile.zMax1 - tri.zMax
  dist01 = tile.zMax0 - tile.zMax1
  if (dist1t > dist01)
    tile.zMax1 = 0
    tile.mask = 0

  // Merge current triangle into working layer
  tile.zMax1 = max(tile.zMax1, tri.zMax)
  tile.mask |= tri.coverageMask

  // Overwrite ref. layer if working layer full
  if (tile.mask == ~0)
    tile.zMax0 = tile.zMax1
    tile.zMax1 = 0
    tile.mask = 0
```

Figure 11: Depth buffer update from Hasselgren et al. paper (screenshot from their paper). Updates a depth buffer tile given a triangle after rasterization. tile.zMax1 is the working layer depth, and tile.zMax0 is the reference layer depth.

This concludes the clipping and rasterization used in renderToDepthBuffer(). I use the depth buffer update from the Hasselgren et al. paper shown in Figure 11. First the working layer is discarded if the distance between the triangle and working layer is greater than the distance between the two layers. The paper does this to prevent objects in the background from leaking into the foreground (Hasselgren et al. 2016). However, on the 3rd line from the bottom I instead use tile.zMax0 = min(tile.zMax0, tile.zMax1). This prevents the reference layer of a tile from moving deeper, which unnecessarily reduces the culling effect. I get better experimental results with this change, and it has a negligible performance impact. The function renderToDepthBuffer() applies model, and view transformations to each triangle of the occluder, clips these triangles as discussed previously, applies the perspective transformation, then rasterizes them using the coverage function for all overlapping tiles, and then uses the updateHiZBuffer function proposed by Hasselgren et al..

This concludes the culling logic. Now I show experimental results. All results are generated using a replay system that does the same actions when run. The replay used is the one included with my implementation. It walks around the perimeter of the default scene, looking towards the center. The scene has 37 buildings. A screen and depth buffer resolution of 1440 by 1024 was used. Windows 10 and an i7-3930k was used. For each data collection run, I run it twice to allow caches to warm up.

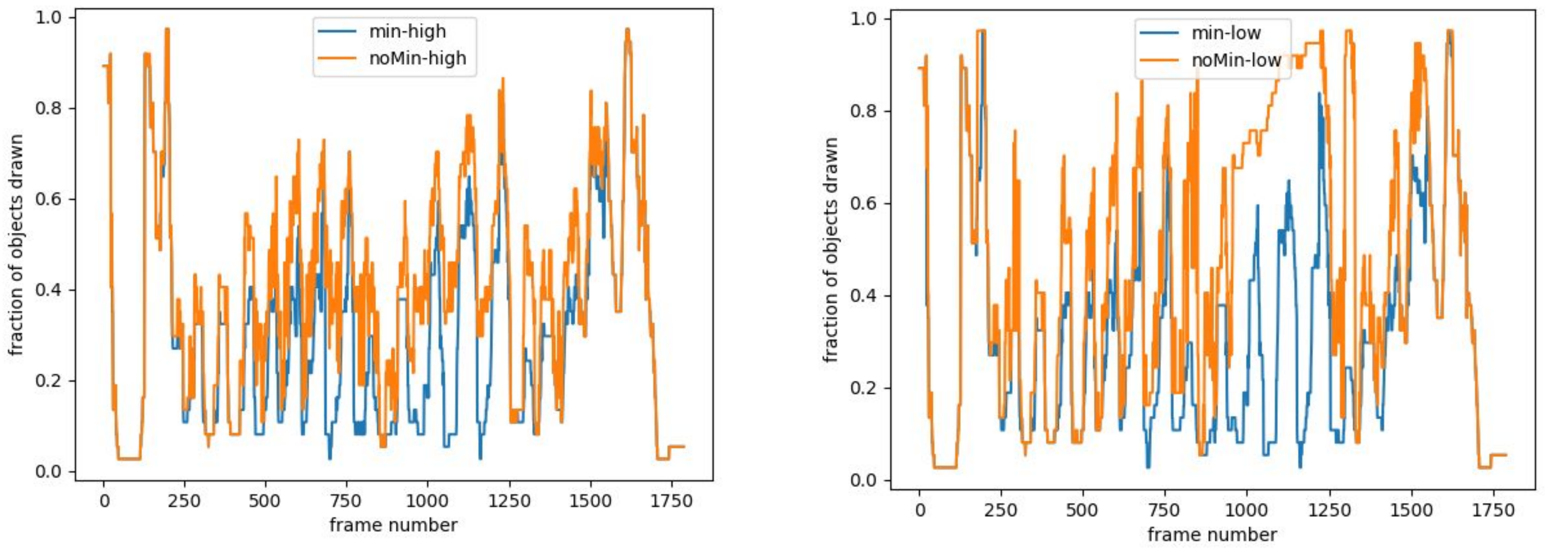


Figure 12: comparing fractions of scene objects drawn between the original depth update (noMin) and modified depth update (min). Lower is better. Left: high polygon count occluders. Right: low polygon count occluders.

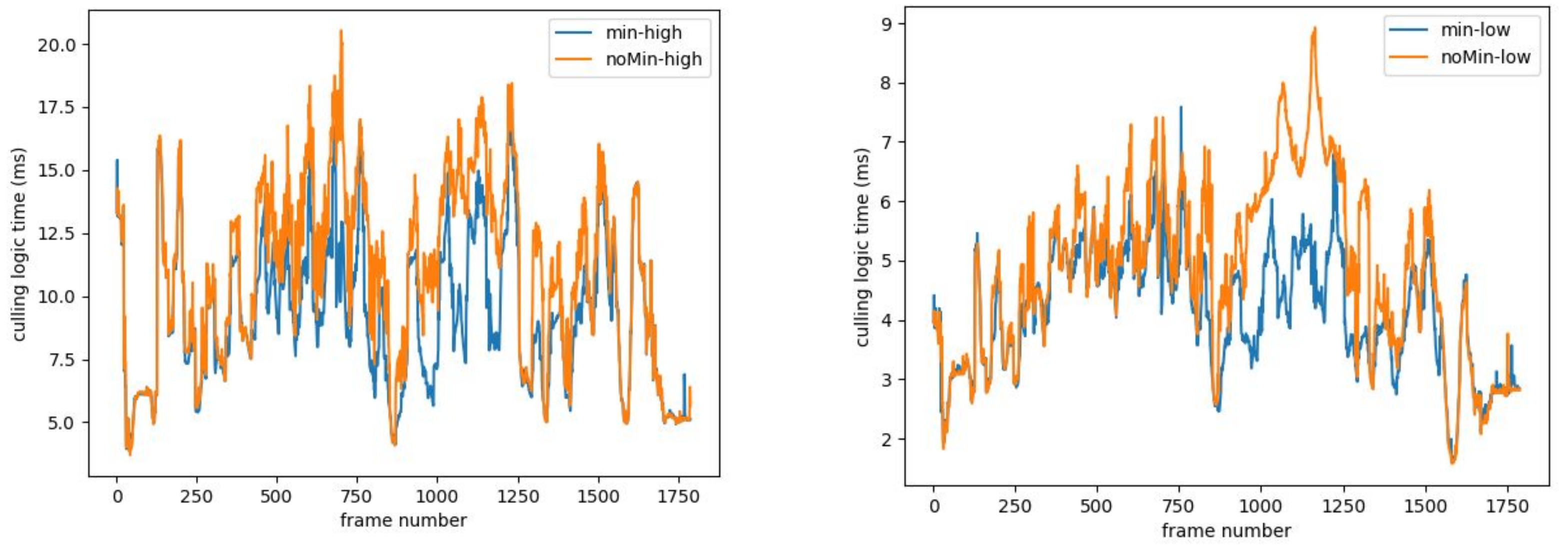
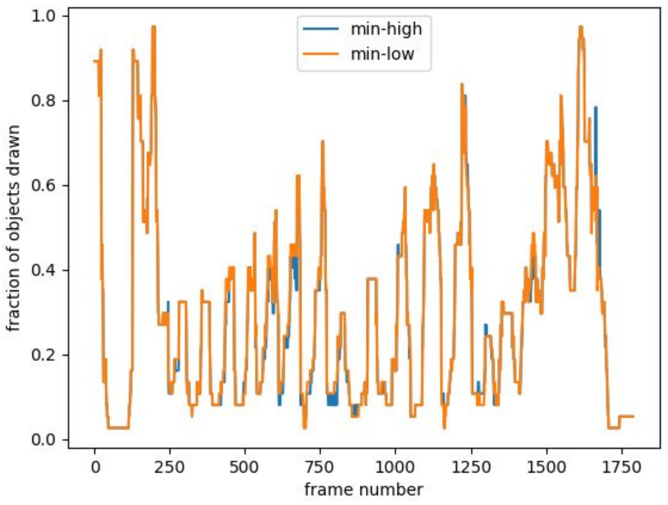


Figure 13: comparing culling logic time between the original depth update (noMin) and modified depth update (min). Lower is better. Left: high polygon count occluders. Right: low polygon count occluders.



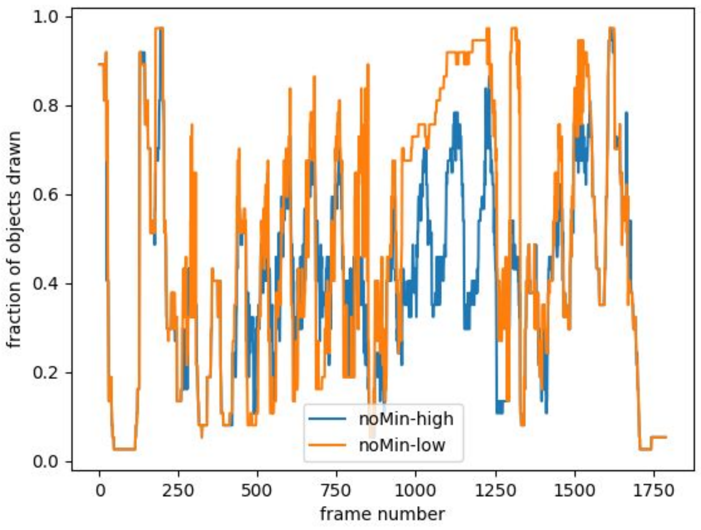


Figure 14: fraction of objects rendered for each depth update variant, using the different occluders. Lower is better.

In these plots, culling logic time is the time to do all occlusion culling computations, separated from OpenGL logic. First the occlusion culling is done, and then objects are rendered after. Fraction of objects drawn is the fraction of all scene objects not culled. The high poly and low poly occluders used to generate results look the same from the outside, but the low poly one is made of 3 overlapping boxes, while the high poly one is made of 16 overlapping boxes. The modified depth update using the min function generally produces better culling as shown in Figure 12, and even runs faster as shown in Figure 13. Figure 14 shows that the original depth update is more sensitive to the poly count of the occluder used. I recommend the modified update using the min function, and low poly count occluders based on these results, as both of these generally give the best running time and culling. I did not encounter false positives (culling objects that should be visible) in any of my time working on this. In the default scene with 37 buildings, the modified update with low poly occluders took on average of 4.12 milliseconds per frame, with a maximum of 7.59 milliseconds. 60 frames per second is about 16.6 milliseconds per frame, so my implementation uses about 25% to 50% of a frame's time in this replay, which is too slow for practical applications, though AVX instructions should speed up this implementation by a factor of up to 8.

References

Hasselgren, Jon, Magnus Andersson, and Tomas Akenine-Möller. "Masked software occlusion culling." Proceedings of High Performance Graphics. 2016. 23-31.

Sutherland, Ivan E., and Gary W. Hodgman. "Reentrant polygon clipping." Communications of the ACM 17.1 (1974): 32-42.

Time Spent

Reading papers: 6 hours

Initial prototyping in JavaScript: 12 hours

Implementation: about 60 hours based on github commits. Underestimate because of work not committed (prototyping, some debugging efforts, etc.) though this is hard to estimate because some of this work did get committed. A lot of trial and error was done as the original paper leaves a few details out, like how to track scanline events or compute edge orientations. I did not use the source code of the authors' implementation as I could not build it, and even if I could, I would not be able to run it, and it was a bit difficult to read.

Report: 24 hours

Code Used

I used the tutorial code from this website http://www.opengl-tutorial.org/ in particular, this repository https://github.com/opengl-tutorials/ogl as a starting point (to have all the necessary libraries and build configuration). It comes with a few libraries (AntTweakBar, Assimp, Bullet, GLEW, GLFW, GLM, and “rpavlik-cmake-modules”), and some other miscellaneous code from the tutorials. From this I use the build configuration, GLEW, GLFW, GLM, and the tutorial code to load and compile shaders for OpenGL. I also use their instructions for building.

The office building model is by the user "lukass12" or "Lukas Carnota" on cgtrader.com, and can be found here:
https://www.cgtrader.com/free-3d-models/exterior/office/free-office-building

The vertex and fragment shaders I use are adapted from assignment 4 of CMPUT 511.

From the Hasselgren et al. paper I use logic similar to “traverseSceneTree()”, I use the “coverageSIMD()” and “updateHiZBuffer()” functions, and their definition of the depth buffer, and the idea of the depth test. This is mentioned in comments in my source code, where I use these things.

I use a variant of the Sutherland-Hodgman algorithm for clipping to the near plane. This is mentioned in a comment in my source code where I use it.

All code I made is in the directory: Culling/render. Stuff here is mine except for v.glsl, f.glsl (shaders mentioned earlier), and the models and code fragments mentioned above (and mentioned in comments throughout my code).

I use Matplotlib to generate plots.

[illegible]