Occluder Generation for Buildings in Digital Games

Kui Wu[iD], Xu He, Zherong Pan[iD], and Xifeng Gao[iD]
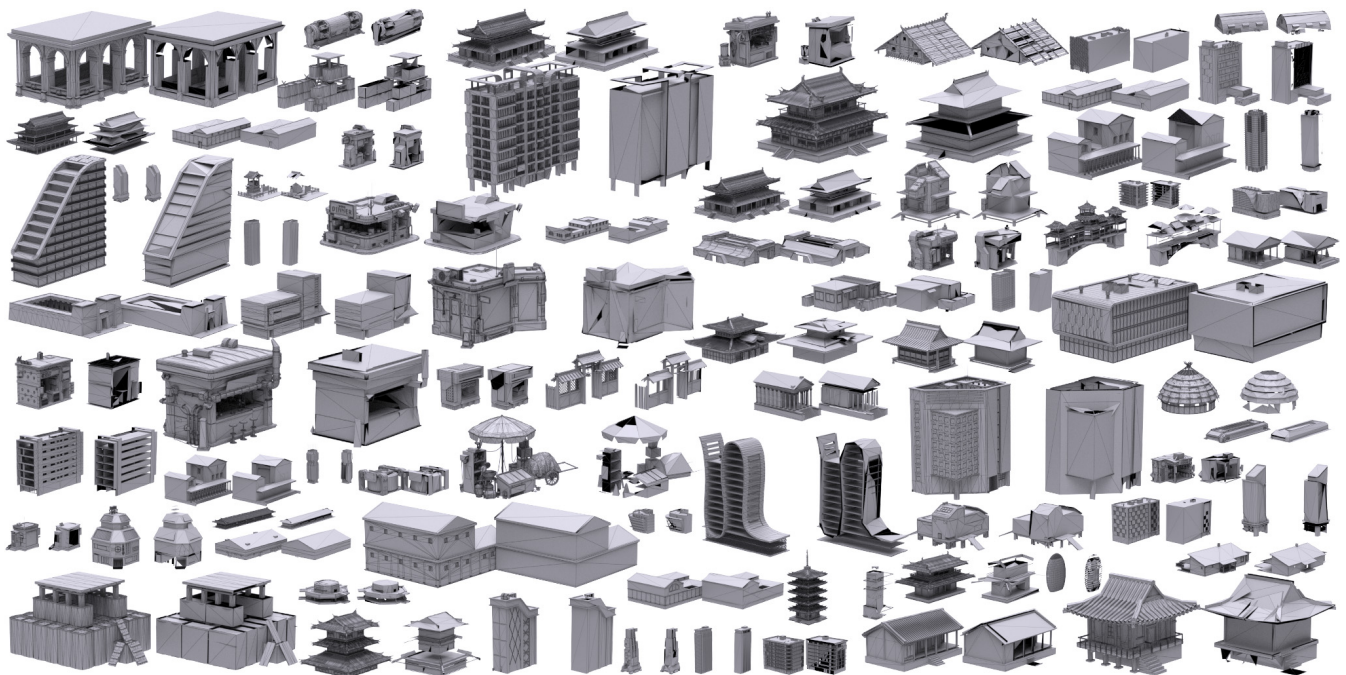
LightSpeed Studios, Tencent

Figure 1: *We show a gallery of high-poly meshes and their corresponding occluders generated using our method, side by side.*

Abstract

Occlusion culling has become a prevalent method in modern game engines. It can significantly reduce the rendering cost by using an approximate coarse mesh (occluder) for culling hidden objects. An ideal occluder should use as few faces as possible to represent the high-resolution input mesh with a high culling accuracy. We address the open problem of automatic occluder generation for 3D building models with complex topology and interior structures. Our method first generates two coarse sets of faces via patch-based and voxel-based mesh simplification techniques. A metric-guided selection algorithm chooses the best subset of faces to form the occluder, achieving a high occlusion rate and accuracy. Over an evaluation of 77 building models, our method compares favorably against state-of-the-arts in terms of occlusion accuracy, occlusion rate, and face number.

CCS Concepts
• Computing methodologies → Mesh geometry models;

1. Introduction

The occlusion culling techniques have been widely used in modern game engines, e.g., Unreal Engine [Gam21], Unity [Tec21], and CryEngine [Cry21]. In particular, raster occlusion culling (ROC) has been widely used in mobile games due to the limited computing power. The key idea is to use an approximate coarse mesh (occluder) rather than the fine rendering mesh (visual mesh) in a lightweight CPU rasterization to cull hidden objects from the GPU
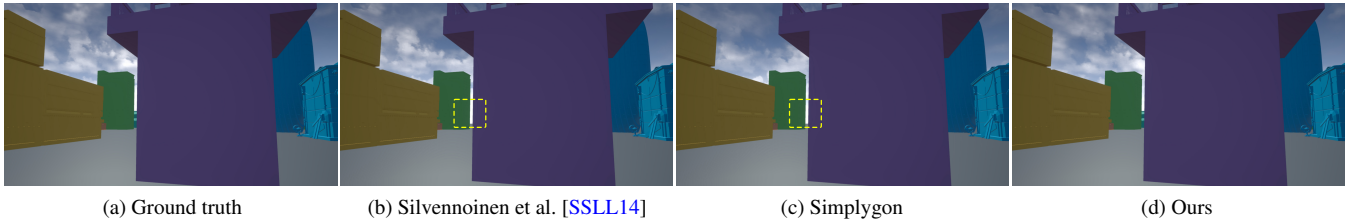
| (a) Ground truth | (b) Silvennoinen et al. [SSLL14] | (c) Simplygon | (d) Ours |

Figure 2: *Exemplary software culling benchmark in Unreal Engine 4. Visible objects are culled by mistake (highlighted with yellow frames) if occluders are non-conservative. (a) is the ground truth and (bcd) are culling results using occluders generated from Silvennoinen et al. [SSLL14], Simplygon [Mic21], and our method, respectively.*
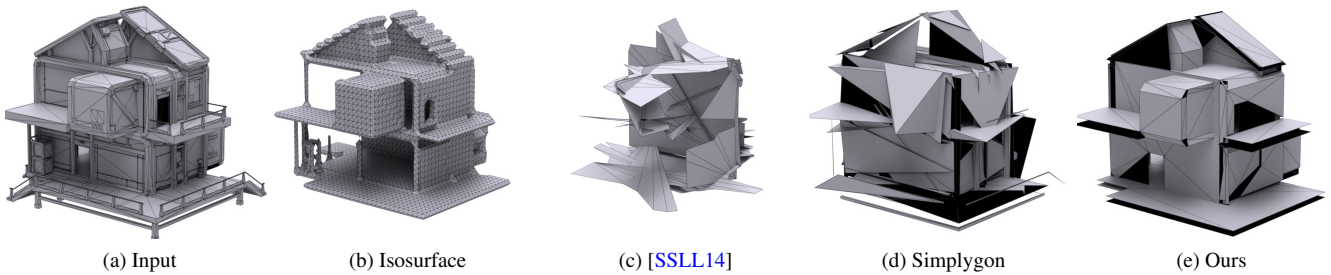

| (a) Input | (b) Isosurface | (c) [SSLL14] | (d) Simplygon | (e) Ours |

Figure 3: *Example of occluders generated using different methods: (a) the input mesh with 7888 faces, 521 components, 562 boundary loops, and 9034 intersected triangle pairs; (b) the isosurface corresponding to a winding number of 0.5; (c) the result from Silvennoinen et al. [SSLL14] using the isosurface (b), (d) the occluder generated by Simplygon [Mic21], and (e) the occluder generated by our method. Face numbers for (c), (d), and (e) are 150, 231, and 231, respectively.*

rendering pipeline during runtime. As a result, entirely occluded objects can be excluded from the rendering pipeline, significantly reducing GPU bandwidth, draw calls, and extra rendering costs.

The quality of the occluder mesh is crucial to the efficacy and accuracy of culling. On the one hand, the occluder should be a low-poly mesh to reduce the cost of the culling test. On the other, the occluder should be conservative, staying inside the volume of the visual mesh. Indeed, non-conservative occluders can cull visible objects by mistake, causing severe visual artifacts, as shown in Figure 2b-c. We propose two metrics to qualitatively measure the accuracy of an occluder over the 3D domain: *Precision*, which measures the possibility of an object blocked by the occluder being also blocked by the original model, and *Recall*, which computes the possibility of an object blocked by the original model being also blocked by the occluder.

This paper focuses on automatically generating occluders for building models from game assets. Game artists manually craft building models to maximize their visual realism, which typically contains numerous disconnected pieces, large open doors and windows, and interior structures, as demonstrated in Figure 3a. As artists only focus on the buildings' appearance, building models are typically non-manifold, non-watertight, and self-intersecting, rendering conventional meshing processing algorithms inaccessible. Unfortunately, neither voxelization-based method nor progressive face removal can handle cases mentioned above. Nowadays, building occluders are still being handcrafted. An artist typically spends hours to create one occluder for complex buildings via trial and error. Even after careful tuning, handmade occluders can still waste

faces that do not contribute to the recall or violate the conservative constraint.

Instead of relying on one strategy and hoping that it is general enough to handle all building models with different styles, we first generate two coarse meshes from the input mesh using two different methods. We combine the generated coarse meshes to populate the candidate face set as a large solution space. Next, we introduce formulations to evaluate the precision and recall of the occluder with respect to the input model over the 3D evaluation domain. We propose an algorithm based on the metrics to select the best face set from the solution space with a high occlusion rate while preserving the conservativity as much as possible. Furthermore, we introduce several strategies to accelerate metric computation at runtime. We have verified our method on 77 building models with various styles. Overall, our method can generate occluders with a low face count of 260 while achieving an averaged precision of 99.4% and an averaged Recall of 78.0% from all possible viewer positions, including faraway, close-up, and walk-in views. This is 3.7% and 2.9% percent higher than occluders generated by Simplygon [Mic21] in terms of precision and recall, respectively, while using 50 fewer faces on average. The results from another state-of-the-art [SSLL14] only have a recall of 39.7% on average.

2. Related Work

We review related techniques in occlusion culling, mesh simplification, and occluder generation.

Occlusion Culling: For static scenes, one may pre-compute and store a potentially visible set [ARB90] with respect to a single
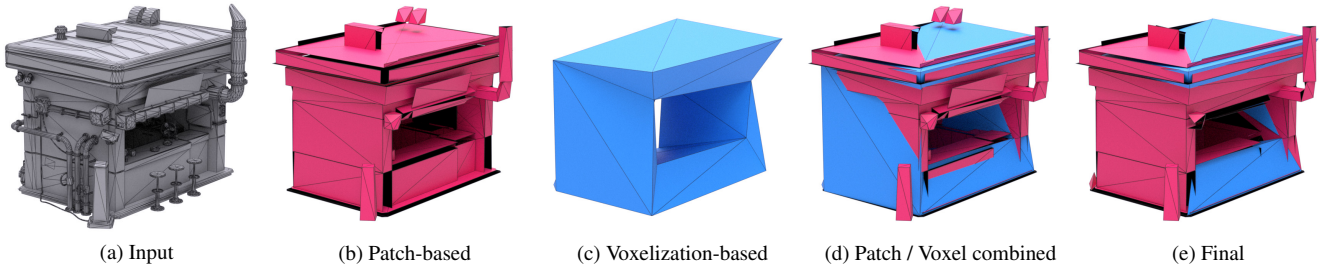
| (a) Input | (b) Patch-based | (c) Voxelization-based | (d) Patch / Voxel combined | (e) Final |

Figure 4: *Our pipeline: (a) input model; (b) the result of patch-based simplification; (c) the result of voxelization-based simplification; (d) the combination of patch-based and voxelization-based results; (e) the final result after the metric-guided mesh simplification, in which faces from patch-based and voxel-based approaches are colored in red and blue, respectively.*

viewpoint [COFHZ98] or a region of viewpoints [BHS98]. Regarding buildings with accessible interiors, cell-and-portal [LG95] decomposes the interior into rooms (cells) connected by doors or windows (portals). However, it is expensive to pre-compute and store the visibility data for complex scenes in large open worlds. To avoid excessive pre-computation and storage, Koltun at el. [KCCO01] maintain a view-dependent subset of the input mesh as *virtual occluders* at runtime. We refer to the surveys [COCSD03, PT02] for more details on various occlusion culling techniques. It is important to note that, while the proposed method focuses on the occlusion performance of single occluder, several authors have proposed hierarchical occlusion culling solutions for complex models [MBW08, MBJ*15, LJSL21].

Mesh Simplification: It is prevalent to use software rasterization for rendering the coarse mesh (occluder) into a depth buffer, which is then used to cull hidden objects in a very early stage [Val11]. The problem of generating an approximate coarse mesh from a fine-detailed one has been investigated for several decades. One classical way is to keep collapsing edges satisfying certain conditions or minimizing certain metrics, e.g., the Quadric Error Metrics (QEM) [GH97]. Various metrics have since been added to the collapsing conditions for specific applications, e.g., Zhang and Turk [ZT02] defined a surface visibility metric. Other works propose to satisfy hard constraints during remeshing. For example, progressive hulls are introduced to guarantee all vertices are outside the input mesh [PT03, SGG*00]. Sacht et al. [SVJ15] generated coarse meshes while maintaining strict nesting. Unfortunately, none of these methods can work in our topologically inconsistent cases. Recently, Gao et al. [GWP22] introduced a visual-driven method to robustly generate low-poly meshes for building models, however, their output cannot be used as occluder since it could violate conservative constraints.

Occluder Generation: Bergen [vdB21] recently used conservative mesh simplification to generate occluders for terrain patches in games, assuming clean topologies. Indeed, existing mesh simplification methods can be used to generate occluders if topologies can be made consistent. Only a few prior works have attempted automatic occluder generation for buildings, which can be classified into two strategies. A prominent strategy first voxelizes the input mesh, extract the isosurface, and then simplify the output isosurface. Next, either axis-aligned boxes [Dar11] or cutting planes [SSLL14] are inserted to form the occluder. How-

ever, some essential features, e.g., thin walls, cannot be captured at an affordable resolution, as shown in Figure 3b. Moreover, nested and open structures in building models can cause ill-defined orientations, for which the isosurface cannot be extracted accurately and correctly. An alternative strategy progressively removes faces from the input mesh through error-guided element-removal operations [vdB21, Mic21]. However, those operations, e.g., edge-collapse, often generate results with large gaps and parts outside the visual mesh, which violates conservativity, as illustrated in Figure 3d. Last but not least, all these methods assume buildings are viewed faraway, and none of them respect the concave and interior building structures. As a result, when game characters enter those areas, these occluders would fail to provide accurate occlusion predictions. The work most close to ours is [SL17]. They greedily find a set of planes inside the voxelized input model to form an occluder with bounded occlusion error. However, their work relies on the voxelized mesh as the input, which can introduce a large occlusion error during voxelization. Also, buildings with nested structures can be non-orientable and cannot be voxelized, while thin walls cannot be captured at an affordable resolution. In the game industry, to avoid creating buildings occluders by hand, Valient [Val11] uses the collision mesh as starting mesh for simplification. Still, collision meshes are typically larger than the input mesh, significantly violating conservativity.

3. Method

An input building model is represented as triangle/polygon soups, including hundreds of (possibly self-intersecting) disconnected components, nested structures, and thin features. Due to the topological complexity, neither voxelization nor conventional mesh simplification can work well by themselves. Instead, we observe that some disconnected components contain a number of large patches that are useful candidates to form the final occluder mesh, while other large volumetric features can be well captured by voxelization. Therefore, we propose to use a hybrid approach, as demonstrated in Figure 4, combining the outputs from the two mesh simplification strategies, patch-based simplification (Section 3.1.1) and voxelization-based simplification (Section 3.1.2), to form a large candidate set of faces, as the initial occluder M_{occluder}. Next, we formulate the two evaluation metrics: precision $\mathcal{P}$ and recall $\mathcal{R}$ for M_{occluder} (Section 3.2). Finally, we introduce a metric-guided mesh simplification of M_{occluder} to extract high-quality final oc-

cluder from the initial occluder and terminates when user-specified quality bounds are reached (Section 3.3).

3.1. Initial Occluder Generation

In this section, we propose a two-way hybrid method to form the initial face candidate set.

3.1.1. Patch-based Mesh Simplification

Our first approach performs each of the following steps to generate a coarse mesh:

1. *planar patches grouping*: group faces into planar patches by thresholding the dihedral angle of each interior edge (we choose a small threshold of 1×10^{-3});
2. *curved patches grouping*: group planar patches generated from the last step into curved patches if the dihedral angle of one shared edge is less than the user-specified threshold ε_a. Note that using planar patches leads to an occluder with a smaller number of triangles, while using curved patches captures more details. Combining two patch sets leads to a large candidate set, from which our final occluder is selected. One example is shown in Figure 5.
3. *simplification*: simplify each curved patch with QEM-guided mesh simplification [GH97]; for the remaining planar patches, simplify the boundaries of each 2D-projected patch using the Ramer-Douglas-Peucker algorithm [Ram72] and re-triangulate the boundary into a triangle mesh using constrained Delaunay triangulation;
4. *hole filling*: fill holes that do not originally exist in M_{Input};
5. *reduction*: sort all planar and curved patches by their areas and keep adding patches to the final mesh until the face count of which is larger than a user-specified number N_P.

Hole-filling: Hole-filling is a standard typical mesh-repairing practice. For building models, some building decorators will lead a hole after removing small patch, such as the example shown in Figure 6, while windows and doors must be left open for the conservativity of the occluder. Therefore, we propose a verification strategy to check whether a hole corresponds to a decorator or not. As shown in Figure 6, we first triangulate the hole into several faces and then
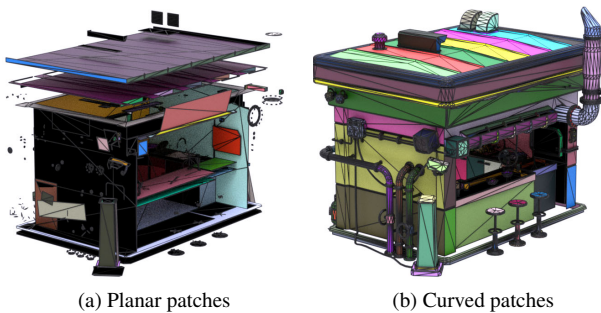
place a set of testing line segments with length l_s uniformly sampled inside each face along its normal direction. As long as there is one segment that does not intersect with M_{input}, we decide that there is a hollow structure in M_{input} and keep the hole open. If all segments hit the input mesh, then no openings exist in M_{input} and the hole can be filled safely. After hole-filling, we mesh-simplify the patches to further reduce the face count.
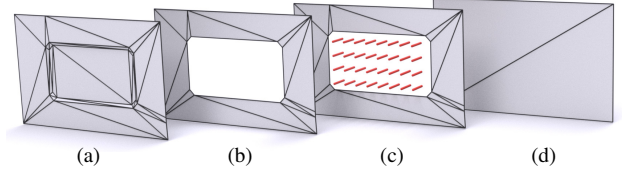

Figure 6: *Example of hole filling: (a) the input mesh with a concave "window"; (b) the planar patch with a hole extracted from the input mesh; (c) testing line segments placed within the hole (red); (d) the output mesh after hole filling and remeshing.*

3.1.2. Voxel-based Mesh Simplification

Our second approach performs each of the following steps sequentially to generate a coarse mesh:

1. *voxelization*: voxelize M_{input} with voxel size l/N_v, where N_v is an user-defined volelization resolution parameter and l is the diagonal length of M_{input}'s bounding box, compute the 3D winding number for each voxel [JKSH13], and extract the 0.5-isosurface using the marching cube algorithm (Figure 7b);
2. *remeshing*: simplify the isosurface into a coarse mesh based on QEM [GH97] (Figure 7c);
3. *conservative enforcement*: project the simplified mesh back onto the 0.5-isosurface to enforce the conservative.
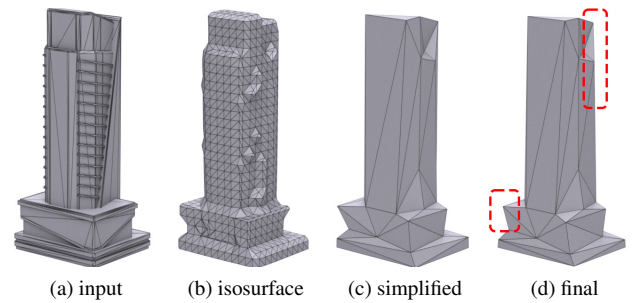

Figure 7: *Example of voxel-based mesh simplification. Note that we get the final mesh (d) by pushing some faces (highlighted with red frames) from the simplified mesh (c) in the final mesh to enforce the conservative.*

Conservative Enforcement (CE): QEM-based simplification is not conservative, i.e., some resulting vertices can be outside the input mesh, causing false negatives during occlusion culling. To alleviate this issue, we formulate an optimization problem to push the obtrusive parts back inside. Specifically, denoting the simplified mesh as M_{coarse}, we first compute a signed distance field ϕ from the

(a) Planar patches   (b) Curved patches

Figure 5: *Example of patch grouping: Given the input mesh shown in Figure 4, all faces can be grouped into two sets, (a) planar patches and (b) curved patches. Note the faces belonging to the same patch are rendered with the same color.*

isosurface mesh (Figure 7b) and then solve the following optimization problem:

$$\min_{\mathbf{x} \in \mathbb{R}^{3n}} \sum_i E_i(\mathbf{x}) \quad \text{s.t.} \quad \phi(\mathbf{p}) \leq 0, \quad \forall \mathbf{p} \in M_{\text{coarse}}, \quad (1)$$

where $\mathbf{x}$ is a vector corresponding to the vertex positions of M_{coarse} and $\mathbf{p}$ can be any point on M_{coarse}. Each edge in M_{coarse} is defined as a spring energy $E_i = \frac{1}{2}(\|\mathbf{p_0} - \mathbf{p_1}\| - r)^2$, where $\mathbf{p_0}$ and $\mathbf{p_1}$ are edge ending points and r is the edge length before performing conservative enforcement.

To formulate the unilateral constraints in the above optimization problem, we detect continuous contacts between the triangle mesh M_{coarse} and the signed distance field ϕ using [MEM*20]. We keep detecting continuous collisions during the optimization. When a collision happens, the following soft penalty energy based on signed distance field (SDF) is added to the objective function:

$$E_{\text{SDF}}(\mathbf{p}) = \begin{cases} \frac{1}{2}\phi(\mathbf{p})^2 & \phi(\mathbf{p}) > 0 \\ 0 & \phi(\mathbf{p}) \leq 0 \end{cases}, \quad (2)$$

to replace the hard constraints in Equation 1. Putting things together, we reformulate the optimization into the following unconstrained form:

$$\text{argmin}_{\mathbf{x} \in \mathbb{R}^{3n}} \sum_i E_i(\mathbf{x}) + \sum_i E_{\text{SDF}}(\mathbf{p_i}). \quad (3)$$

Note this method is essentially a penalty method for handling hard constraints with automatic parameter tuning. Although we do not introduce a weight for E_{SDF}, if a same continuous collision happens repeatedly, more E_{SDF} terms will be added, essentially increasing its weight. Since M_{coarse} typically has less than 100 vertices, we use Quasi-Newton method [LBK17] to solve the optimization. As in the sharp corners highlighted in Figure 7d, our method pushes the coarse mesh into the building to improve precision.

3.2. Occlusion Evaluation

We formulate two metrics, precision $\mathcal{P}$ and recall $\mathcal{R}$ of an occluder M_{occluder} with respect to the input model M_{input}, to evaluate its quality and guide the follow-up mesh simplification procedure.

Originally proposed by Silvennoinen et al. [SSLL14], the notions of precision and recall are not new. However, their original definition uses a rather restrictive setting, putting both the occluder and the building in origin and the viewer outside the building facing the origin. As demonstrated in Figure 8, we argue that measuring occlusion from the outside only is insufficient for many models in video games, such as caves, tunnels, and buildings, where players are allowed to walk through. In these latter cases, camera views from interiors and concave areas are essential.

Following the above reasoning, we present a new way to compute precision $\mathcal{P}$ and recall $\mathcal{R}$ over the 3D evaluation space. Theoretically, a player can omnidirectionally look into the ambient space at any accessible location over the 3D space. To compute $\mathcal{P}$ and $\mathcal{R}$, we first define the evaluation domain as $B' = (1 + \varepsilon_{\text{padding}})B - M_{\text{input}}$, i.e., B' is the enlarged volume of M_{input}'s bounding box by a factor of $(1 + \varepsilon_{\text{padding}})$ minus the volume of M_{input}. The overall precision $\mathcal{P}$ and recall $\mathcal{R}$ are given as:

$$\mathcal{P} = \frac{1}{|\mathcal{B}'|} \int_{\mathcal{B}'} \mathcal{P}_\mathbf{x} d\mathbf{x}, \quad \text{and} \quad \mathcal{R} = \frac{1}{|\mathcal{B}'|} \int_{\mathcal{B}'} \mathcal{R}_\mathbf{x} d\mathbf{x}. \quad (4)$$

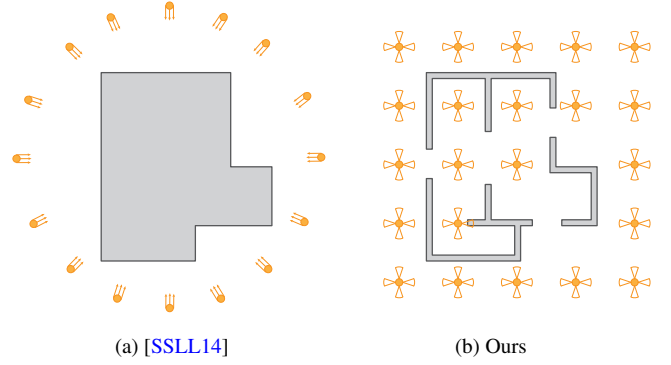

(a) [SSLL14]   (b) Ours

Figure 8: *Comparison between [SSLL14]'s and ours view sampling methods: (a) Silvennoinen et al. [SSLL14] sample views around the building directed towards the origin; (b) our method samples over the 3D space and evaluates all cameras views, including interiors and concave areas.*

To numerically approximate the intractable integrals above, we uniformly divide B' into volume blocks with spacing Δx and mark all volume blocks outside M_{input} as valid. Therefore, the overall precision and recall can be computed as:

$$\mathcal{P} = \frac{1}{\sum \Delta V} \sum^N \mathcal{P}_\mathbf{x} \Delta V = \frac{1}{N} \sum^N \mathcal{P}_\mathbf{x},$$
$$\mathcal{R} = \frac{1}{\sum \Delta V} \sum^N \mathcal{R}_\mathbf{x} \Delta V = \frac{1}{N} \sum^N \mathcal{R}_\mathbf{x}, \quad (5)$$

where N is the number of valid blocks and $\Delta V = \Delta x^3$ is block volume. $\mathcal{P}_\mathbf{x}$ and $\mathcal{R}_\mathbf{x}$ denote the precision and recall at block center $\mathbf{x}$.

To compute $\mathcal{P}_\mathbf{x}$ and $\mathcal{R}_\mathbf{x}$ at a fixed camera position $\mathbf{x}$, we discretize the full view direction space into 6 view frustums along the $\pm X, Y, Z$ axes, each having a $90°$ view angle. We then reduce the occlusion computation in 3D to 2D screen space by comparing the 2D areas occluded by M_{occluder} and M_{input}. For each view frustum, we approximate $\mathcal{P}_\mathbf{x}$ and $\mathcal{R}_\mathbf{x}$ using Monte-Carlo sampling. In particular, N_{quad} randomly sampled, axis-aligned quads are rasterized as occludees over the screen space. Accordingly, there are three cases:

- *true positive:* a quad is covered by both M_{occluder} and M_{input}. The number of pixels in this quad is denoted as N^t;
- *false positive:* a quad is entirely covered by M_{input} but not M_{occluder}. The number of pixels in this quad is denoted as P^f;
- *false negative:* a quad is completely covered by M_{occluder} but some pixels are not covered by M_{input}. The number of pixels uncovered by M_{input} in this quad is denoted as N^f.

Like [SSLL14], our discretized $\mathcal{P}_\mathbf{x}$ and $\mathcal{R}_\mathbf{x}$ can be computed as follows:

$$\mathcal{P}_\mathbf{x} = \frac{\sum N_i^t}{\sum N_i^t + \sum N_i^f}, \quad \text{and} \quad \mathcal{R}_\mathbf{x} = \frac{\sum N_i^t}{\sum N_i^t + \sum P_i^f}, \quad (6)$$

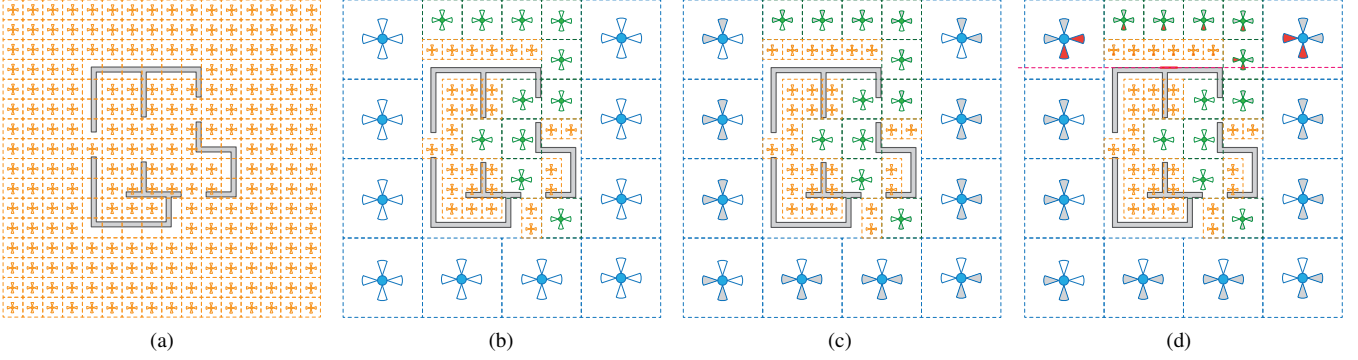where the summation is over all six directions $\pm X, Y, Z$.

Figure 9: *Occlusion measure acceleration:* *(a) We first discretize the empty space into a number of uni-size blocks; (b) We merge samples in a similar way as an octree; (c) We mark the view directions that cannot see the building in gray; (d) If the face in red is removed, only views in red on the positive side of the face normal need to be updated.*

3.3. Metric-Guided Occluder Simplification (MGOS)

Combining the results from patch-based (Section 3.1.1) and voxel-based (Section 3.1.2) simplification methods, a high-quality face candidate set is generated. As our last step, we use a metric-guided face reduction algorithm to select a face subset and form our final occluder. A naive approach to this end is to greedily remove faces that produce the smallest recall reduction $\Delta\mathcal{R}$. However, this is too computationally expensive. Assuming that the combined mesh M_{combined} has m faces and we use n view position samples to compute the recall, then we have to do $6n\prod_{i=m}^{i=m-k} i$ visual evaluation in order to remove k faces. Instead, our simplification algorithm performs the following steps: We first check all faces, if removing face f_i would lead to a precision change $\mathcal{P}(M) - \mathcal{P}(M - f_i) > \varepsilon_P$, we remove f_i from M_{combined}. Note we don't update $\mathcal{P}$ after discarding f_i, since removing one face from the occluder will not increase $\mathcal{P}(M - f_i)$. Our second step is to recheck all the remaining faces. If removing f_i would lead to $\Delta\mathcal{R} < \varepsilon_R$, we remove f_i and update $\mathcal{R}$. Here we need to update $\mathcal{R}$ during each iteration, since removing one face may increase other faces' contributions to the recall.

To further accelerate the metric-guided occluder simplification, we deploy two strategies: reducing adjacent views and skipping unnecessary evaluations.

Sampled View Reduction: If two views are close to each other, the difference between their occlusion results will be small. Based on this observation, we reduce the number of view samples by merging neighboring ones. In particular, we first uniformly divide the domain $\mathcal{B}$ into N blocks as shown in Figure 9a. Next, we merge $2 \times 2 \times 2$ adjacent blocks, whose centers do not collide with the building until no merges can be performed, as shown in Figure 9b. Accordingly, we can reformulate the Equation 5 as:

$$\mathcal{R} = \frac{1}{\sum \Delta V_i} \sum^N \mathcal{R}_\Omega(\mathbf{x_i})\Delta V_i, \qquad (7)$$

where ΔV_i and $\mathbf{x_i}$ are the ith sample's block volume and center location, respectively.

Sample Skipping: The building might be outside the view frustum of many view locations, in which case N^t, P^f, and N^f will always be zero. Therefore, before the simplification, we test and

mark all views and skip those that cannot see the building at all. Figure 9c shows these marked views in grey. Further, when a face is removed, it does not change the metric values measured from views that cannot see the face. Hence, we can separate the evaluation domain into two parts based on the orientation of the face. We only update metric values for those view locations lying on the positive side of the face, as demonstrated in Figure 9d.

4. Results

We implement our method in C++ with CGAL [FT15] and libigl [JP*18]. To evaluate occluders and compute our metrics, we implement a fast parallel CPU software rasterization in Unreal Engine 4 [Gam21]. Specifically, we rasterize the input model and occluder into the depth buffer. Then, we randomly generate a number of quads on the depth buffer and calculate the precision and recall rates using Equation 6. To avoid too many empty pixels in the depth buffer, we use the minimum length of the input model's bounding box as the $\varepsilon_{\text{padding}}$. We did all experiments on a computer with an AMD Ryzen Threadripper 3970X 32-Core Processor @3.69 GHz and 256 GB RAM.

4.1. Ablation study

Metric Discretization Precision: We first evaluate the impact of the number of screen quads N_{quad} on the precision of approximate metric computation on one building model (shown in Figure 3). In particular, we compare metrics computed using 100, 1000, 5000, 10000, 20000, 30000, and 100000 quads per view, with a resolution of 256×256, a common size used in mobile games. For each quad number, we compute our metrics ten times with different sets of randomly picked quads. Figure 10a shows the standard deviation of the ten metrics. We use 5000 quads per view throughout the rest of the experiments to balance computational cost and accuracy.

Sampling Distance: We then test how the view sample spacing Δx impacts the metric approximation accuracy on all 77 models. We experiment with 8%, 4%, 2%, 1%, and 0.5% of the largest diagonal length of M_{input}' bounding boxes. Halving the sampling distance would increase the computational cost by a factor of 8.
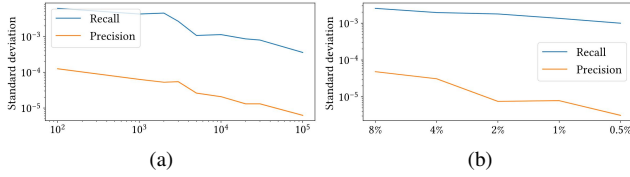
Figure 10: *Standard deviation for the recall and precision of quad numbers per view (a) and sampling distances (b).*
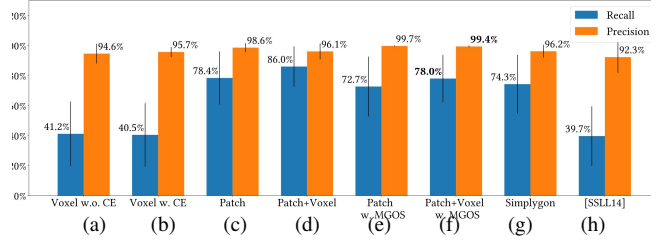

Figure 11: *Average recall and precision of the dataset: (a) voxel-based result without conservative enforcement (CE); (b) voxel-based result with conservative enforcement; (c) patch-based result; (d) combined face candidate set from voxel-based and patch-based results; (e) patch-based result with metric-guided occluder simplification (MGOS); (f) combined voxel-based and patch-based results with MGOS as our final result; (g) occluder generated by Simplygon; and (h) planar occluder [SSLL14].*

Similarly, we compute metrics ten times under each sampling distance. Figure 10b shows the standard deviations for both recall and precision. Since the standard deviation of recall at 4% and 2% are very close, 1.9×10^{-3} and 1.8×10^{-3} respectively, we use 4% as the sampling distance throughout the rest of the experiments.

Samples Reduction: We use sample reduction and sample skipping to accelerate the metric computation, which in turn speeds up metric-guided mesh simplification. After testing all 77 models, we find that the total number of evaluation tests is reduced to 16.3% of the number of tests without using any evaluation acceleration techniques. The ratio is further reduced to 13.1%, then 5.9% after skipping occluded views and removed faces, respectively. Overall, sample reduction achieves a speed up of $5.56\times$ with no impact to the evaluation accuracy. Note that sample skipping does not impact the accuracy of the evaluation results.

4.2. Experiments

We evaluate our method using a dataset of 77 building models used for games, as shown in Figure 1. In practice, we expect the number of faces to be as small as possible. It is worth noting that each building has hundreds of components and thousands of intersecting triangle pairs, which make it impossible to be processed with conventional mesh simplification methods. To evaluate our generated occluder, we rasterize 5000 randomly generated quads on the depth buffer to collect information for computing precision and recall. We set $l_s = 1\%$ of the largest diagonal length of M_{input} bounding boxes. All parameters are listed in Table 1. On average, occluders generated by our method have 260 faces with a recall of 78.0% and a precision of 99.4%, as shown in Figure 11 and Table 2. Figure 12 also shows 10 example buildings and their corresponding occluders generated by Simplygon [SSLL14] and our method. Visual results of all the models can be found in the attached video.

Table 1: *Parameters*

Parameters	Value
ε_a	$\pi / 3$
N_v	64
N_P	600
$\varepsilon_{\mathcal{P}}$	1×10^{-3}
$\varepsilon_{\mathcal{R}}$	1×10^{-3}

	Voxel-based	Patch-based	Patch/Voxel Combined	Metric-Guided Simplification	Ours Final	Simplygon	[SSLL14]
Face #	53	500	553	-	260	300	127*
Time	5 s	24 s	-	135 s	154 s	3 s	267 s

*For single-sided rendering, the number needs to be doubled.

Table 2: *Average face number and computation time on the dataset using our method, Simplygon, and planar occluder [SSLL14].*

Conservative Enforcement: Figure 11 shows that our conservative enforcement can improve the precision from 94.6% (Figure 11a) to 95.7% (Figure 11b). With the improved precision, however, comes a reduction in the recall, which is understandable. When making a given coarse mesh strictly within the input mesh, there must be some areas that the coarse mesh cannot fill. Regardless, in our application, we prefer higher precision even with a loss of recall since false culling leads to severe visual artifacts. Note that the precision of voxel-based results is lower than that of patch-based results since our isosurface generation is dependent on correct winding numbers.

Combining Patch/Voxel-based results: In Figure 11, the occluder has a recall of 40.5% and 78.4% and a face number of 500 and 53, from the patch-based (Figure 11b) and voxel-based (Figure 11c) method, respectively. The combined candidate set (Figure 11d) leads to a significantly higher recall of 86.0% with 7.6% improvement over patch-only results (Figure 11b).

Results after metric-guided simplification: As mentioned, a low recall is preferred over a low precision in practice, since a low precision will lead to serious visual artifacts. As shown in Figure 11f, our metric-guided occluder simplification can improve combined voxel- and patch-based method to a 99.4% in precision at the cost of 8.0% decrease in recall, as compared with the results without MGOS (Figure 11d). Moreover, the combined solution (Figure 11f) achieves a 5.3% higher recall than patch-only results (Figure 11e). This implies the contribution of the voxel-based solution is significant. One example of combining patch-based and voxel-based results can be found in Figure 4.

Timing: Throughout the 77 testing building model, our method takes 155 seconds on average (Table 2), of which patch-based mesh simplification and voxel-based mesh simplification take 5 seconds and 24 seconds, respectively. It takes 126 seconds to further reduce the face number from 553 to 260 by metric-guided simplification. In contrast with hours of manual tuning by artists, our method is automatic and orders of magnitude faster.

Comparison with Simplygon [Mic21]: Simplygon is the state-of-the-art close-sourced meshing processing tool commonly used

Input	[SSLL14]	Simplygon	**Ours**	Input	[SSLL14]	Simplygon	**Ours**

(4733, -, -)	(75*, 0.613, 0.601)	(300, 0.669, 0.903)	**(118, 0.805, 0.983)**	(30156, -, -)	(150*, 0.314, 0.994)	(300, 0.520, 0.986)	**(301, 0.888, 0.997)**
(40002, -, -)	(75*, 0.355, 0.971)	(300, 0.785, 0.994)	**(233, 0.805, 0.999)**	(39999, -, -)	(150*, 0.139, 0.933)	(300, 0.728, 0.878)	**(271, 0.708, 0.996)**
(4999, -, -)	(149*, 0.386, 1.000)	(300, 0.569, 0.969)	**(187, 0.746, 0.995)**	(9999, -, -)	(150*, 0.081, 0.912)	(300, 0.656, 0.967)	**(274, 0.910, 0.999)**
(29960, -, -)	(75*, 0.053, 0.888)	(300, 0.886, 0.851)	**(279, 0.912, 0.989)**	(33498, -, -)	(150*, 0.521, 0.994)	(300, 0.631, 0.946)	**(145, 0.798, 0.994)**
(3548, -, -)	(78*, 0.682, 0.972)	(300, 0.819, 0.960)	**(216, 0.837, 0.993)**	(6066, -, -)	(150*, 0.558, 0.978)	(300, 0.844, 0.963)	**(266, 0.890, 0.996)**

Figure 12: *Our method compared with Simplygon and [SSLL14]. (●,●,●) denotes (face number, recall, and precision).*

in the game industry. We use the visibility-driven mesh simplification pipeline in Simplygon with the target triangle count set as 300, which is the occluder face count we commonly use in mobile games. The method works well in terms of preserving the silhouette with a reasonably fast processing speed (3 seconds per model). Simplygon achieves a recall of 74.3%, which is 3.7% lower than ours. However, it fails to maintain conservativity (with a precision of 96.2% versus our 99.4%). More importantly, the standard deviation of our precision is only 0.6% while that of Simplygon is 4.3%, showing that our method is more stable across inputs. As mentioned before, a lower precision is more detrimental than a lower recall. With a low recall, the system has to render more hidden objects, which will hinder the rendering efficacy. However, a low precision is fatal, since an object can be wrongly culled even when it is visible to the player, as shown in Figure 2. It is worth noting that, for the first model in Figure 12, even if the occluder generated by Simplygon can capture the overall input shape, its recall is only 66.9% due to the small cracks in between the disconnected walls, which significantly weaken its occlusion power.

Comparison with Planar Occluder [SSLL14]: [SSLL14] first voxelizes the input mesh and generates an isosurface. Since their method assumes the building model is viewed from far away, it only generates one plane for each view direction. After choosing a set of planes, [SSLL14]'s method greedily removes the triangle with the minimal area rather than minimizing the loss of recall. Thus, we slightly improve their method by evaluating the occlusion whenever a face is discarded for better accuracy. Even with such an optimization, the output planar occluders can only achieve an averaged recall of 39.7% and a precision of 92.3%. One reason is that our testing models contain thin walls and nested structures that cannot be voxelized correctly. Thus, the output isosurface can only capture a small portion of the input mesh with a large precision error, which can also be observed in our voxel-based results (only a recall of 41.2%). In terms of the computation time, in our implementation, [SSLL14]'s method takes 267 seconds per model, which is slower than ours, while their occlusion metric is only half that of ours. It is also important to note that the method in [SSLL14] assumes double-sided rendering. For the culling meth-

ods using single-sided rendering, the face number of occluders has to be doubled. We set their target output face count as 150 and double the face count at runtime, since our game engine (based on Unreal Engine 4) uses single-sided rendering.

5. Conclusion

We propose an occluder generation method that combines patch-based and voxel-based face generation techniques. We then select the best face subset to form an occluder based on novel evaluation metrics. We further introduce two evaluation metrics over the 3D domain to measure the quality of occluders and several strategies to accelerate the procedure of evaluation. By testing our method on 77 buildings in Unreal Engine 4, we highlight that our method generates occluders with a higher precision and recall than several prior works [SSLL14, Mic21].

Limitations & Future Works: Due to the reliance of state-of-the-art mesh processing techniques on inputs being manifold and watertight, we turn to heuristics and practical techniques which require parameters. Furthermore, since our approach utilizes only patch-based and voxel-based mesh simplification tools, our method could fail in two typical cases. First, if a mesh has complex nested structures, its orientation is ambiguous and the voxelization result (Figure 13b) might be incorrect. Second, if a mesh has many small, disconnected patches, our grouping algorithm might not be able to merge them, which lead a hollow structures with a low recall, such as Figure 13c. We speculate that fusing results from multiple different mesh simplification techniques can improve the quality of face candidate sets. Finally, our metric-guided occluder simplification has only a single operation, aka. face reduction.
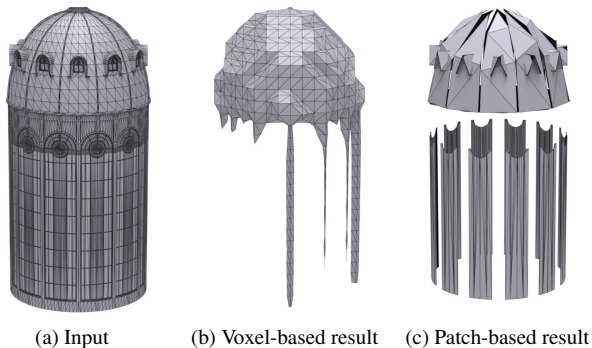

(a) Input   (b) Voxel-based result   (c) Patch-based result

Figure 13: *A failure case of our method.*

Acknowledgements

References

[ARB90] AIREY J. M., ROHLF J. H., BROOKS F. P.: Towards image realism with interactive update rates in complex virtual building environments. In *Proceedings of the 1990 Symposium on Interactive 3D Graphics* (New York, NY, USA, 1990), I3D '90, Association for Computing Machinery, p. 41–50. 2

[BHS98] BITTNER J., HAVRAN V., SLAVIK P.: Hierarchical visibility culling with occlusion trees. In *Proceedings. Computer Graphics International (Cat. No.98EX149)* (1998), pp. 207–219. 3

[COCSD03] COHEN-OR D., CHRYSANTHOU Y., SILVA C., DURAND F.: A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics 9*, 3 (2003), 412–431. 3

[COFHZ98] COHEN-OR D., FIBICH G., HALPERIN D., ZADICARIO E.: Conservative visibility and strong occlusion for viewspace partitioning of densely occluded scenes. *Computer Graphics Forum 17*, 3 (1998), 243–253. 3

[Cry21] CRYTEK: cryengine, 2021. URL: https://www.cryengine.com/. 1

[Dar11] DARNELL N.: Automated occluders for gpu culling. *Game Developer Magazine (Sep. 2011)* (2011). 3

[FT15] FOGEL E., TEILLAUD M.: The computational geometry algorithms library cgal. *ACM Commun. Comput. Algebra 49*, 1 (jun 2015), 10–12. 6

[Gam21] GAMES E.: Unreal engine, 2021. URL: https://www.unrealengine.com. 1, 6

[GH97] GARLAND M., HECKBERT P. S.: Surface simplification using quadric error metrics. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques* (USA, 1997), SIGGRAPH '97, ACM Press/Addison-Wesley Publishing Co., p. 209–216. 3, 4, 10

[GWP22] GAO X., WU K., PAN Z.: Low-poly mesh generation for building models. In *ACM SIGGRAPH 2022 Conference Proceedings* (New York, NY, USA, 2022), SIGGRAPH '22, Association for Computing Machinery. 3

[JKSH13] JACOBSON A., KAVAN L., SORKINE-HORNUNG O.: Robust inside-outside segmentation using generalized winding numbers. *ACM Trans. Graph. 32*, 4 (jul 2013). 4

[JP*18] JACOBSON A., PANOZZO D., ET AL.: libigl: A simple C++ geometry processing library, 2018. https://libigl.github.io/. 6

[KCCO01] KOLTUN V., CHRYSANTHOU Y., COHEN-OR D.: Hardware-accelerated from-region visibility using a dual ray space. In *Rendering Techniques 2001* (Vienna, 2001), Gortler S. J., Myszkowski K., (Eds.), Springer Vienna, pp. 205–215. 3

[LBK17] LIU T., BOUAZIZ S., KAVAN L.: Quasi-newton methods for real-time simulation of hyperelastic materials. *ACM Trans. Graph. 36*, 4 (may 2017). 5

[LG95] LUEBKE D., GEORGES C.: Portals and mirrors: Simple, fast evaluation of potentially visible sets. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics* (New York, NY, USA, 1995), I3D '95, Association for Computing Machinery, p. 105–ff. 3

[LJSL21] LEE G. B., JEONG M., SEOK Y., LEE S.: Hierarchical raster occlusion culling. *Computer Graphics Forum 40*, 2 (2021), 489–495. 3

[MBJ*15] MATTAUSCH O., BITTNER J., JASPE A., GOBBETTI E., WIMMER M., PAJAROLA R.: Chc+rt: Coherent hierarchical culling for ray tracing. *Computer Graphics Forum 34*, 2 (2015), 537–548. 3

[MBW08] MATTAUSCH O., BITTNER J., WIMMER M.: Chc++: Coherent hierarchical culling revisited. *Computer Graphics Forum 27*, 2 (2008), 221–230. 3

[MEM*20] MACKLIN M., ERLEBEN K., MÜLLER M., CHENTANEZ N., JESCHKE S., CORSE Z.: Local optimization for robust signed distance field collision. *Proc. ACM Comput. Graph. Interact. Tech. 3*, 1 (apr 2020). 5

[Mic21] MICROSOFT: Simplygon, 2021. URL: https://www.simplygon.com/. 2, 3, 7, 9

[PT02] PANTAZOPOULOS I., TZAFESTAS S.: Occlusion culling algorithms: A comprehensive survey. *Journal of Intelligent and Robotic Systems 35*, 2 (2002), 123–156. 3

[PT03] PLATIS N., THEOHARIS T.: Progressive hulls for intersection applications. *Computer Graphics Forum 22*, 2 (2003), 107–116. 3

[Ram72] RAMER U.: An iterative procedure for the polygonal approximation of plane curves. *Computer Graphics and Image Processing 1*, 3 (1972), 244–256. 4

[SGG*00] SANDER P. V., GU X., GORTLER S. J., HOPPE H., SNYDER J.: Silhouette clipping. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques* (USA, 2000), SIGGRAPH '00, ACM Press/Addison-Wesley Publishing Co., p. 327–334. 3

[SL17] SILVENNOINEN A., LEHTINEN J.: Real-time global illumination by precomputed local reconstruction from sparse radiance probes. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia) 36*, 6 (Nov. 2017), 230:1–230:13. 3

[SSLL14] SILVENNOINEN A., SARANSAARI H., LAINE S., LEHTINEN J.: Occluder simplification using planar sections. *Computer Graphics Forum 33*, 1 (2014), 235–245. 2, 3, 5, 7, 8, 9

[SVJ15] SACHT L., VOUGA E., JACOBSON A.: Nested cages. *ACM Trans. Graph. 34*, 6 (oct 2015). 3

[Tec21] TECHNOLOGIES U.: Unity, 2021. URL: https://unity.com/. 1

[Val11] VALIENT M.: Practical occlusion culling in killzone 3. In *ACM SIGGRAPH 2011 Talks* (New York, NY, USA, 2011), SIGGRAPH '11, Association for Computing Machinery. 3

[vdB21] VAN DEN BERGEN G.: Conservative mesh decimation for collision detection and occlusion culling. Game Developers Conference (GDC), 2021. 3

[ZT02] ZHANG E., TURK G.: Visibility-guided simplification. In *IEEE Visualization, 2002. VIS 2002.* (2002), pp. 267–274. 3

Appendix: Patch-Based Simplification

We provide more details of patch-based mesh simplification.

Planar patches grouping: We employ a patch-growing algorithm to group faces into planar patches. The algorithm begins forming a patch with an arbitrary seed face and grows it, one face at a time. During this process, we keep track of the boundary B of the patch and the set of all faces belonging to the patch. Specifically, each growing step consists of two steps, edge selection and expansion. Edge selection randomly picks an edge e in B that neighbors a face f not belonging to the patch. Then, if the dihedral angle between two faces neighboring e is smaller than 1×10^{-3}, f is included in the patch and the data structures (boundary B and the set of all faces) are updated.

Curved patches grouping: We employ a patch-growing algorithm to group curved patches similar to that of planar patches. The algorithm takes all planar patches generated from the previous step as the input. Then, it begins forming a curved patch with an arbitrary seed planar patch and grows it, one planar patch at a time. We also keep track of the boundary B of the curved patch. If the dihedral angle of any e in B is smaller than ε_a, the expansion attaches the patch containing e's face into the curved patch and add all non-shared edges to B. Note that only the planar patch that is expanded during this process will be labeled as the curved patch.

QEM-based simplification: In addition to the conventional QEM guided edge collapse [GH97], we add a small weight, i.e. 10^{-3}, for each edge to fight against the singular degenerate issue when solving for the metric on a coplanar neighborhood of an edge. For boundary preservation, we employ a large weight, i.e. 10^3, so that those edges on the boundary are collapsed later. We also avoid edge collapse operations, if they introduce topological changes of the mesh, e.g. non-manifoldness and genus changes.