Low-poly Mesh Generation for Building Models

Xifeng Gao
Lightspeed & Quantum Studios, Tencent America
Seattle, WA, USA
xifgao@tencent.com

Kui Wu
Lightspeed & Quantum Studios, Tencent America
Los Angeles, CA, USA
kwwu@tencent.com

Zherong Pan
Lightspeed & Quantum Studios, Tencent America
Seattle, WA, USA
zrpan@tencent.com

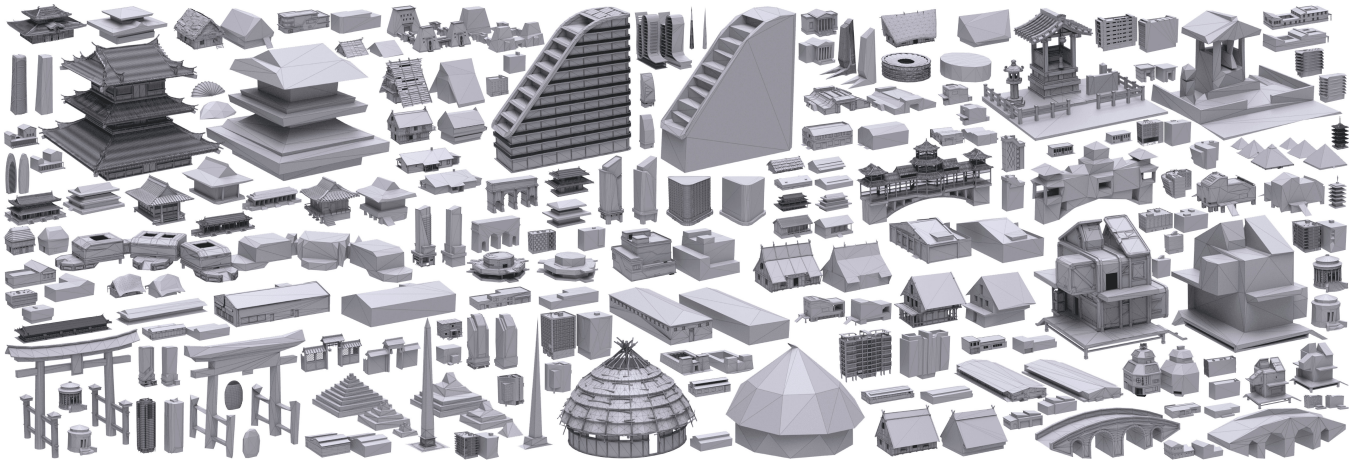

Figure 1: A gallery of high-poly meshes and their corresponding low-poly counterparts generated using our method.

ABSTRACT

As a common practice, game modelers manually craft low-poly meshes for given 3D building models in order to achieve the ideal balance between the small element count and the visual similarity. This can take hours and involve tedious trial and error. We propose a novel and simple algorithm to automate this process by converting high-poly 3D building models into both simple and visually preserving low-poly meshes. Our algorithm has three stages: First, a watertight, self-collision-free visual hull is generated via Boolean intersecting 3D extrusions of input's silhouettes; We then carve out notable but redundant structures from the visual hull via Boolean subtracting 3D primitives derived from parts of the input; Finally, we generate a progressively simplified low-poly mesh sequence from the carved mesh and extract the Pareto front for users to select the desired output. Each stage of our approach is guided by visual metrics, aiming to preserve the visual similarity to the input. We have tested our method on a dataset containing 100 building models with different styles, most of which are used in popular digital games. We highlight the superior robustness and quality by comparing with state-of-the-art competing techniques. Executable program for this paper is at lowpoly-modeling.github.io.

SIGGRAPH '22 Conference Proceedings, August 7–11, 2022, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9337-9/22/08…$15.00
https://doi.org/10.1145/3528233.3530716

CCS CONCEPTS

• **Computing methodologies** → **Mesh geometry models**.

KEYWORDS

mesh simplification, low-poly mesh, building model

ACM Reference Format:
Xifeng Gao, Kui Wu, and Zherong Pan. 2022. Low-poly Mesh Generation for Building Models. In *Special Interest Group on Computer Graphics and Interactive Techniques Conference Proceedings (SIGGRAPH '22 Conference Proceedings), August 7–11, 2022, Vancouver, BC, Canada.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3528233.3530716

1 INTRODUCTION

Building models are essential assets that make up the metaverse for the virtual world. Unfortunately, building modelers only focus on polishing visual appearances without maintaining a clean mesh representation. Typically, a highly detailed building model can have complicated topology and geometry properties, i.e., disconnected components, open boundaries, non-manifold edges, and self-intersections (see Fig. 1 and Table 1). On the other hand, it can be expensive to render detailed building models for real-time applications, and the level-of-details (LOD) technique has been widely used to maximize the run-time performance. Instead of sticking to a highly detailed (high-poly) 3D model, the LOD renderer uses a low-element-count (low-poly) mesh at the distant view. As a result, the low-poly mesh must have a reasonably small element count while preserving the appearance of the high-poly model as much as possible.

Many prior works are proposed to create low-poly meshes from input high-poly models, such as mesh simplification [Khan et al.

2020], simple mesh reconstruction from point clouds [Nan and Wonka 2017], high-poly mesh simplification to remove small-scale details [Calderon and Boubekeur 2017; Mehra et al. 2009], and mesh optimization and re-meshing techniques by commercial software [AB 2021; Nerurkar 2021; Sweeney 2021]. However, these methods are less suitable for the rising applications at the low-end platforms, e.g., mobile devices. Indeed, even the finest building mesh on mobile platforms has only thousands of triangles, which is already considered low-poly by traditional LOD generation approaches. In terms of the coarsest level, these models typically have no more than a few hundred elements. To generate such "extremely" low-poly meshes, none of the above mentioned methods perform well for our tested building dataset (Fig. 1). Sadly, the standard process for creating this type of low-poly meshes is still labor-intensive and involves tedious trial and error in the current game industry. Thus, we conclude that generating extremely low-poly meshes for building models used by mobile applications is a challenging problem, for which effective and robust solutions are still elusive.

In this work, we propose a robust method for effectively generating extremely low-poly meshes that can be used as the coarsest level. We first define a *visual metric* to quantitatively measure the visual difference between the low- and the high-poly meshes. We then design our method, consisting of three stages: First, we construct a coarse visual hull by intersecting a small set of 3D primitives selected greedily to minimize our visual metric. These primitives are generated by computing and analyzing silhouettes of the input from a number of view directions. The result of the first stage is denoted as *visual hull*, which captures the input's silhouette but can miss important concave features. Our second stage generates a *carved mesh* from the visual hull by subtracting redundant volumes to recover concave features. We again deploy a greedy strategy to select the craving primitives by minimizing the visual metrics between the carved mesh and the input. Since all the 3D shapes involved in the Boolean operations are watertight and we employ exact arithmetics for computation, the generated carved mesh is guaranteed to be watertight and self-intersection-free. Our final stage derives the *low-poly mesh* by progressively applying edge-collapse and edge-flip to the carved mesh.

We observe that high-quality, low-poly meshes occur indefinitely among the mesh sequence. Therefore, we keep a history of the simplified meshes and order them into a Pareto set [Kung et al. 1975] with two objectives: the number of triangles and the visual closeness to the input. A game modeler can then explore and pick an ideal mesh as the final result. Our method is empirically compared with state-of-the-art rival techniques in terms of both the effectiveness and efficacy, where we use a dataset containing 100 building models with varying styles that are manually crafted by artists and used by real-world games. Our method exhibits a significant improvement in terms of achieving a low element count and a high visual similarity at the same time. Visual results of all the models can be found in the attached video.

2 RELATED WORK

We first review prominent methods related to low-poly mesh generation. We then briefly survey representative works on visual hull, on which our method is built.

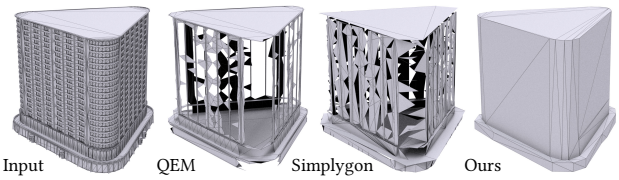

Figure 2: For a manifold model with 39620 triangles and 1894 disconnected components, the overly aggressive remeshing operators by QEM [Garland and Heckbert 1997] or Simplygon [AB 2021] lead to salient artifacts with 1000 triangles, while our method faithfully represents the overall structure with 128 triangles.

Low-poly Meshing: The first and foremost large group of methods directly re-mesh the raw inputs through progressively error-guided element-removal operations, such as edge collapse [Garland and Heckbert 1997; Lescoat et al. 2020; Salinas et al. 2015; Sander et al. 2000], or mesh segmentation into patches that can later be re-triangulated into simplified ones [Cohen-Steiner et al. 2004; Li and Nan 2021]. Khan et al. [2020] provides an inclusive survey on this type of methods. These re-meshing approaches perform well when handling medium- to high-poly meshes with benign topology and geometry. However, they are not suitable for generating extremely coarse low-poly meshes and overly aggressive direct re-meshing can lead to salient, detrimental visual artifacts. Fig. 2 illustrates meshes with these artifacts generated using edge-collapse guided by the QEM [Cignoni et al. 2008; Garland and Heckbert 1997] and the mesh reduction module in the commercial software - Simplygon [AB 2021].

Another type of approaches [Calderon and Boubekeur 2017; Mehra et al. 2009] voxelizes the raw inputs and then applies feature-guided re-triangulation to generate low-poly outputs by assuming the input meshes are watertight, i.e., come with a unique definition of inside/outside. However, building models in most real-time applications are often not watertight due to holes and self-intersecting elements. By incorporating various rules and assumptions, Verdie et al. [2015] propose to generate LODs for multi-view stereo reconstructed urban scenes, while our goal is to create extremely low-poly meshes for buildings designed by artists.

Following the idea of fitting simple primitives to point clouds and obtaining polyhedral meshes through combinatorial selection of elements intersected from the primitives, a series of works on 3D reconstruction have been proposed [Bauchet and Lafarge 2020; Chauve et al. 2010; Fang and Lafarge 2020; Fang et al. 2018; Kelly et al. 2017; Nan and Wonka 2017]. Among them, [Bauchet and Lafarge 2020; Nan and Wonka 2017] stand out as ways to generate low-poly meshes. However, these methods require properly detected plane primitives to form enough polygon candidates for the final output, which is a challenging task by itself for complex models. The learning method described in [Chen et al. 2020] demonstrates low-poly results with promising quality reconstructed from images. However, to adapt it for building models, it requires not only a large building dataset for the network training, but voxelized meshes with well-defined in/out segmentation, while the data we aim to handle often have large holes and nested structures that do not permit a clear in/out definition. A similar idea is adopted for shape abstraction [Huang et al. 2014; Mehra et al. 2009; Yang and

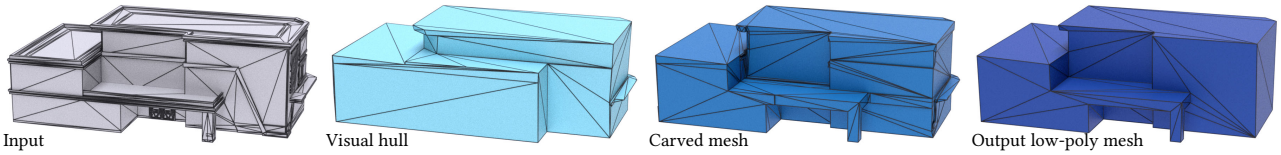

Figure 3: We illustrate the three major steps of our method. Given a topologically and geometrically dirty input mesh, we first construct a visual hull using a small set of greedily selected primitives. With such a small set, some concave features are erroneously flattened. We then correct them by carving concave features using another set of primitives, deriving the carved mesh. We finally simplify the carved mesh to a set of final output low-poly mesh candidates.

Chen 2021], re-interpreting shapes using primitives such as boxes, spheres, curves, etc. But these methods are not directly targeted at low-poly mesh generation.

We are also aware of a recent trend in differentiable rendering [Laine et al. 2020] leading to appearance-driven 3D reconstruction, shape deformation, and re-meshing methods [Hasselgren et al. 2021; Luan et al. 2021; Nicolet et al. 2021]. These methods can nicely approximate the original shape implied in the given images. However, due to the poor initial guess and the discontinuity of the inverse-rendering-based optimization when mesh connectivity changes, they cannot be directly employed for generating low-poly models.

Visual Hull: An object has a unique and well-defined visual hull, which is the maximal shape with the same silhouette as the object from any view directions [Laurentini 1994; Szeliski 1993]. Visual hulls were originally used for 3D reconstruction from images or videos [Matusik et al. 2001, 2000; Mikhnevich and Hebert 2011; Svetlana Lazebnik 2007] by considering the image boundaries as silhouettes. As the main advantage, visual hulls are inherently watertight, topologically simple meshes from arbitrarily dirty inputs. For faithful reconstructions, users typically prefer more views to capture as much detail as possible. Theoretically, an infinite number of views are needed for obtaining the exact visual hull from a general 3D model. For example, in order to construct the exact visual hull using viewpoints inside the convex hull, Laurentini [1997] constructed an algorithm using $O(n^5)$ silhouettes. Visual hull also has its own disadvantage, e.g., concave features cannot be represented well. To tackle this issue, Rivers et al. [2010] proposed to construct the visual hull part-by-part via Boolean operations. Our method borrows all these ideas but adapt them to the problem of mesh simplification by carefully selecting and limiting the number of views.

3 METHOD

In this section, we provide our problem statement, define the visual metric to measure the quality of our low-poly meshes, give an pipeline overview, and explain each step in detail.

Problem Statement: Given a non-orientable, non-watertight, high-poly 3D building model M_i, our goal is to generate a low-poly mesh M_o that satisfies three qualitative requirements. To be used as the coarsest mesh in a LOD hierarchy, the *visual appearance* of M_o should resemble that of M_i from any faraway view points. To maximize the rendering efficacy, the number of *geometric elements* (e.g., faces) should be as few as possible. We further require this number of elements to be user-controllable. Finally, M_o needs to be *watertight* to enable the automatic downstream mesh editing operations.

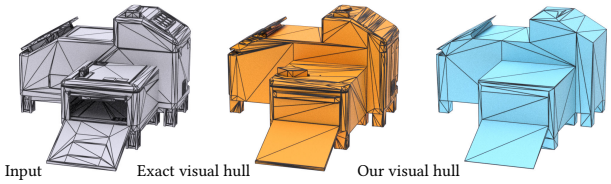

Figure 4: Given an input mesh, the exact visual hull is approximated by intersecting silhouettes from 13 view directions, leading to 277k faces. In comparison, our visual hull only has 368 faces, obtained by 3 primitives.

Visual Metric: We use an image-space metric to measure the visual difference between M_i and M_o. Given a view direction d, we can render a mesh into the image space via perspective projection, which is denoted as the operator R_n. $R_n(M, d)$ renders the three Cartesian components of the surface normal vector into a framebuffer, and we define the visual difference as the averaged pixel-wise distance:

$$d_n(M_\mathrm{i}, M_\mathrm{o}, d) = \|R_n(M_\mathrm{i}, d) - R_n(M_\mathrm{o}, d)\|/N, \tag{1}$$

where N is the number of pixels. We further define our visual appearance metric as the marginalized visual difference over all directions:

$$\tau_n(M_\mathrm{i}, M_\mathrm{o}) \triangleq \int_{S^2} d_n(M_\mathrm{i}, M_\mathrm{o}, d(s))ds,$$

where S is a sphere enclosing M_i and M_o. τ_n can be approximated via Monte-Carlo sampling in practice.

Method Overview: As illustrated in Fig. 3, we tackle our problem by first computing a visual hull M_v from M_i, which is both geometrically enclosing M_i and silhouette preserving. Instead of directly mesh-simplifying M_i itself, M_v is benign to existing remeshing operators, geometrically a tighter bound of M_i than its convex hull, and preserves M_i's silhouette which is important for visual appearance (Section 3.1). Then, we subtract redundant blocks from M_v to obtain the carved mesh M_c, enriching its visual appearance with notable concave features (Section 3.2). Finally, we mesh-simplify M_c to generate the low-poly output M_o (Section 3.3).

3.1 Visual Hull Generation

This stage aims to generate a topologically simple and geometrically clean visual hull while capturing the visual appearance of salient structures. However, as shown in Fig. 4, generating an exact visual hull would lead to too many small features and details. Instead, we propose a novel procedure to generate a simplified visual hull as described in Algorithm 1. Given M_i, we first generate one silhouette

for each of the top k view directions. As our first point of departure from the standard visual hull construction method (e.g., [Matusik et al. 2001]), we then perform a self-intersection-free simplification in the 2D space for each silhouette. At this point, a standard visual hull can be constructed through the intersection of extruded silhouettes. Our second point of departure lies in a further decomposition of silhouette into connected 2D loops, and we denote an extruded loop as a *primitive*. Instead of considering a set of extruded silhouettes, a larger set of primitives allows finer complexity control of the visual hull. Specifically, we initialize M_v as the bounding box of M_i and use a greedy algorithm to iteratively intersect M_v with the next-best primitive P through Boolean operations. Details of each step are described as follows.

Algorithm 1 Visual Hull Construction

Input: $M_\text{i}, N, \epsilon_\tau$
Output: M_v
1: $\mathcal{P} \leftarrow \emptyset$
2: Extracting a set of view directions and pick top k as $\mathcal{D}$
3: **for each** $d \in \mathcal{D}$ **do**
4: Generate silhouette S along d
5: Simplify silhouette S
6: **for each** connected loop $L \in S$ **do**
7: $\mathcal{P} \leftarrow \mathcal{P} \cup \{\text{Extrude}(L)\}$
8: $n \leftarrow 0, M_\text{v} \leftarrow \text{BBox}(M_\text{i}), \tau \leftarrow \tau_s(M_\text{v}, M_\text{i})$
9: **while** $n < N$ **do** ▷ Maximal primitive count
10: $\Delta\tau_\text{best} \leftarrow 0, P_\text{best} \leftarrow \emptyset$
11: **for each** $P \in \mathcal{P}$ **do**
12: $\tau_P \leftarrow \tau_s(\text{Intersect}(M_\text{v}, P), M_\text{i})$
13: $\Delta\tau_P \leftarrow \tau - \tau_P$
14: **if** $\Delta\tau_P > \Delta\tau_\text{best}$ **then**
15: $\Delta\tau_\text{best} \leftarrow \Delta\tau_P, p_\text{best} \leftarrow P$
16: **if** $\Delta\tau_\text{best} \geq \epsilon_\tau$ **then** ▷ Minimal primitive improvement
17: $M_\text{v} \leftarrow \text{Intersect}(M_\text{v}, P_\text{best}), \mathcal{P} \leftarrow \mathcal{P}/\{P_\text{best}\}$
18: $n \leftarrow n+1, \tau \leftarrow \tau - \Delta\tau$
19: **else** Break

Generating View Directions (Line 2): For mesh simplicity, a limited number of view directions can be used to generate silhouettes and corresponding primitives, so the quality of the view directions would significantly impact the quality of M_v. For instance, the ideal view directions for a cube should be parallel to the cube faces, along which a Boolean intersection with two extruded silhouettes would carve out the exact cube. Based on this observation, we introduce the following four-stage strategy to extract the potential view directions given M_i. First, we group triangles from each connected component into regions, where two triangles are merged if their dihedral angle is close to π up to a threshold α. Next, we fit a plane for each region using the $\mathcal{L}_2$ metric [Cohen-Steiner et al. 2004], resulting in a set of fitting planes $\mathcal{K}$. Third, for each pair of planes in $\mathcal{K}$, the cross product of their normal directions would result a direction parallel to both planes, resulting in the view direction set $\mathcal{D}$. We consider two directions d_i and d_j duplicate if $|\cos(d_i, d_j)| \geq \cos\beta$, where β is a hyper-parameter set to a small value. Finally, we associate a weight with each view direction in $\mathcal{D}$, which equals to the sum of areas of the two planar regions. Empirically, we find that a higher weight indicates more surface regions can be captured by the silhouette. Therefore, we sort the view directions by their weights and pick the top k directions as the final direction set.

Computing Silhouettes (Line 4): To compute a silhouette from a view direction d, we project all M_i's faces onto the plane perpendicular to d. 2D Boolean union is then used to obtain the corresponding silhouette shape, where the vertex coordinates are represented by integers for fast and robust computation [Vatti 1992]. The generated silhouette shape is guaranteed to have no self-intersections.

Simplifying Silhouettes (Line 5): Even with a small number of view directions, their generated visual hull can still contain too many small-scale details due to their complex silhouettes. To tackle this problem, we bring each silhouette through a 2D simplification and a shape-size filtering process for further complexity reduction. Our 2D simplification uses a triangulation simplification approach [Dyken et al. 2009] implemented under rational number arithmetics, which first generates a triangulation conforming to the set of silhouette loops, and then simultaneously simplifies the loops through triangle mesh simplification in which topology consistency, flip-free, and self-intersection free can be guaranteed. The simplification stops on a squared distance criterion ϵ_d^2. After simplification, we compute the area of each loop of the silhouette and directly discard it from the silhouette if its area is smaller than ϵ_a.

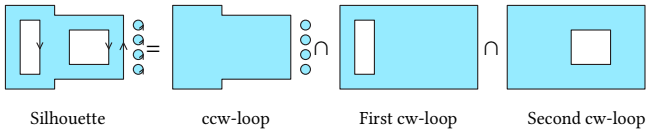

Figure 5: Example of silhouette decomposition: Given a silhouette (leftmost), which contains five ccw loops and two cw loops, we merge all the ccw loops into one outer loop and convert each cw loop into a separate loop by subtracting it from the 2D bounding box. Note this conversion is lossless and order-independent, aka., the input silhouette can be recovered by Boolean intersecting all the loops in any order.

Generating Primitives (Line 6): To derive the primitive set $\mathcal{P}$ from the silhouette, we extract all boundary loops from the silhouette, in which the counterclockwise (ccw) loops are marked as solid, and clockwise (cw) loops are marked as hollow. As shown in Fig. 5, all solid loops are merged into one ccw-loop, while each hollow loop L is extracted as a standalone cw-loop by computing Subtract(BBox, L), where BBox is the 2D bounding box of the silhouette and Subtract$(\bullet, \bullet)$ is the Boolean subtraction operator. Finally, each loop is extruded along the view direction to derive a set of 3D primitives. Fig. 6 shows an example of primitives generated from the input mesh.

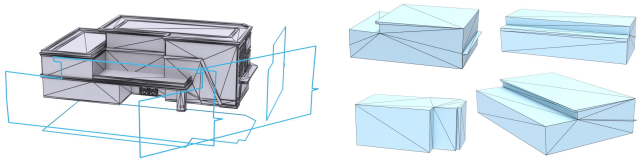

Figure 6: Example of primitives (right) generated from the input mesh and their loops (left).

Selecting the Next-Best Primitive (Line 9): Our visual hull is constructed by Boolean intersecting greedily selected $P_{\text{best}} \in \mathcal{P}$. To select the next P_{best}, we traverse all the primitives $P \in \mathcal{P}$ and create a tentative Intersect(M_{v}, P), where Intersect$(\bullet, \bullet)$ is the Boolean intersection operator. By measuring the visual difference between the tentative mesh and M_{i}, we pick P_{best} as the primitive leading to a most decreased the visual difference. Note that we use a slightly different version of visual difference from d_n for selecting primitives. This is because Algorithm 1 is focused on generating similar silhouettes, and we do not care about the geometry of the interior. Indeed, the mesh is projected onto the 2D plane, removing all the geometric features inside the silhouette. Therefore, we propose another operator $R_s(M, d)$ which renders the mesh M into a stencil buffer, binary masking the occluded pixels and discarding the normal information. Correspondingly, we define:

$$d_s(M_{\text{i}}, M_{\text{o}}, d) \triangleq \|R_s(M_{\text{i}}, d) - R_s(M_{\text{o}}, d)\|/N, \tag{2}$$

$$\tau_s(M_{\text{i}}, M_{\text{o}}) \triangleq \int_{S^2} d_s(M_{\text{i}}, M_{\text{o}}, d(s))ds, \tag{3}$$

as the visual silhouette difference and visual silhouette metric, respectively, which are used in Algorithm 1. Note that the output of $R_s(M, d)$ is binary, so computing d_s amounts to performing a pixel-wise XOR operator.

Stopping Criteria: There are two stopping criteria for the visual hull construction. First, if the improvement of the visual difference $\Delta\tau$ is smaller than the user-specified threshold ϵ_τ, meaning there is barely any room for further improvement, the construction stops. Second, we terminate the construction when the number of selected primitives, n, reaches the user-specified upper limit N. Note that there is the parameter k determining the number of view directions in $\mathcal{D}$, which further controls the total number of primitives in $\mathcal{P}$. After large-scale experiments, however, we find the best strategy is to use a sufficiently large k, leaving a sufficiently large search space for the greedy primitive selection algorithm to optimize M_{v}.

Algorithm 2 Carved Mesh Generation

Input: $M_{\text{i}}, M_{\text{v}}, N, \epsilon_\tau$
Output: M_{c}
1: $\mathcal{P} \leftarrow \emptyset$
2: Pick top k plane from $\mathcal{K}$
3: **for each** $K \in \mathcal{K}$ **do**
4: Cut M_{i} and keep the positive part $M_{\text{i}}^{K^+}$
5: Generate the silhouette of $M_{\text{i}}^{K^+}$ on K, denoted as S
6: Simplify silhouette S
7: Compute M_{i}'s extended 2D bounding square B on K
8: $\mathcal{P} \leftarrow \mathcal{P} \cup \{\text{Extrude}_K^+(\text{Subtract}(B, S))\}$
9: $n \leftarrow 0, M_{\text{c}} \leftarrow M_{\text{v}}, \tau \leftarrow \tau_n(M_{\text{c}}, M_{\text{i}})$
10: **while** $n < N$ **do** ▷ Maximal primitive count
11: $\Delta\tau_{\text{best}} \leftarrow 0, P_{\text{best}} \leftarrow \emptyset$
12: **for each** $P \in \mathcal{P}$ **do**
13: $\tau_P \leftarrow \tau_n(\text{Subtract}(M_{\text{c}}, P), M_{\text{i}})$
14: $\Delta\tau_P \leftarrow \tau - \tau_P$
15: **if** $\Delta\tau_P > \Delta\tau_{\text{best}}$ **then**
16: $\Delta\tau_{\text{best}} \leftarrow \Delta\tau_P, P_{\text{best}} \leftarrow P$
17: **if** $\Delta\tau_{\text{best}} \geq \epsilon_\tau$ **then** ▷ Minimal primitive improvement
18: $M_{\text{c}} \leftarrow \text{Subtract}(M_{\text{c}}, P_{\text{best}}), \mathcal{P} \leftarrow \mathcal{P}/\{P_{\text{best}}\}$
19: $n \leftarrow n + 1, \tau \leftarrow \tau - \Delta\tau$
20: **else** Break

3.2 Carved Mesh Generation

Our visual hull (Fig. 7e) inherits the limitation of only capturing the silhouette while ignoring other features. One such salient feature is the concave mesh parts as illustrated in (Fig. 7a). To overcome this limitation, we propose to refine M_{v} into M_{c} by carving out redundant volume blocks and further reducing the visual difference between M_{c} and M_{i}. As described in Algorithm 2, we take the following steps. First, we consider the top-k planes in the plane set $\mathcal{K}$ generated in Section 3.1 that are sorted by their corresponding region areas. Each plane $K \in \mathcal{K}$ is used to slice M_{i} into two parts, and we only keep the one on the positive side, denoted as $M_{\text{i}}^{K^+}$ (Fig. 7b). Note that, the normal and location of a plane is predetermined during the plane fitting, and the positive part is chosen to avoid unnecessary carving for the bottom of the buildings. Next, we project $M_{\text{i}}^{K^+}$ onto K to obtain its silhouette S. We also compute the enlarged bounding square B of S on K (Fig. 7c). Finally, we derive the carving primitive P as follows:

$$P \triangleq \text{Extrude}_K^+(\text{Subtract}(B, S)),$$

where $\text{Extrude}_K^+(\bullet)$ is the extrusion operator along the positive side of the plane K (see Fig. 7b-d for an illustration). Note that we set B large enough to enclose the entire M_{v}'s silhouette on K so that the extruded primitive P is guaranteed to include the entire volume of M_{v}, which is outside M_{i}. Similar to the visual hull generation, we further simplify and regularize S through 2D simplification and shape size filtering. Fig. 8 shows an example of carving the visual hull to enrich the concave details.

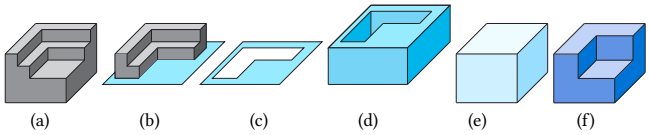

Figure 7: Example of mesh carving: (a) the input mesh M_{i}; (b) the plane K (in cyan) extracted from M_{i} and cut the submesh $M_{\text{i}}^{K^+}$ from the positive side of K; (c) the enlarged 2D bounding square on the plane K minus $M_{\text{i}}^{K^+}$'s silhouette S on K; (d) the primitive P, which is the extrusion of (c); (e) the visual hull M_{v} that does not have the concave part of $M_{\text{i}}^{K^+}$; (f) the carved mesh M_{c} after subtracting P from M_{v}.

Selecting the Next-Best Carving Primitive (Line 10): The overall structure of Algorithm 2 is very similar to that of Algorithm 1. Our carved mesh M_{c} is constructed by greedily selecting $P_{\text{best}} \in \mathcal{P}$ as the candidate carving primitive. To identify P_{best}, we traverse all the primitives $P \in \mathcal{P}$ and create a tentative Subtract(M_{c}, P). By measuring the visual difference between the tentative mesh and M_{i}, we pick P_{best} as the primitive that can best decrease the visual difference. The same stopping criteria to the visual hull construction are used to terminate the carving process. It is worth noting that, since all the 3D primitives for constructing the visual hull and the redundant volumes are watertight, the generated carved mesh is guaranteed to be watertight and self-intersection-free.

3.3 Low-Poly Mesh Generation

After the first two steps, the carved mesh M_{c} can largely preserve the visual appearance of the input but may have more triangles

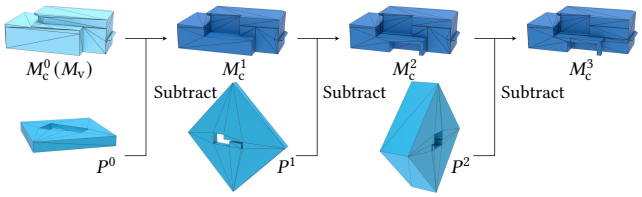

Figure 8: Example of carving the mesh to enrich the details.

than users desire. Our last step is to simplify M_c to obtain a low-poly mesh while maintaining a low visual difference from the input. Specifically, we first progressively re-mesh M_c through edge-collapse and edge-flip operators (see details in Section 4) until no more triangles can be removed, and store only the meshes with their numbers of triangles less than T, where T is the largest element count users can tolerate. We then employ the Pareto efficiency concept to rank these meshes with two metrics: the number of faces and visual differences τ_n, and keep those that are in the Pareto set. Since picking M_o from this mesh set often involves subjective factors in practice, as illustrated in Fig. 9, we finally visualize the meshes in the Pareto set and have a game modeler manually pick M_o as the output. If it is desired, we can also automatically export a mesh from the Pareto set right before an obvious increase of τ_n.

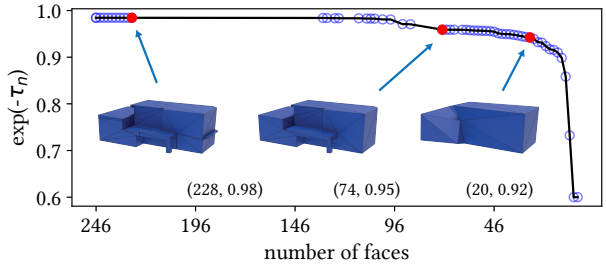

Figure 9: By measuring the number of faces and $e^{-\tau_n}$, the generated Pareto set contains 55 meshes out of 123 in the original sequence.

4 IMPLEMENTATION DETAILS

We implemented our algorithm in C++, using GLSL for metric computation, Eigen for linear algebra routines, CGAL [Brönnimann et al. 2017] for rational number computations, the Clipper library [Vatti 1992] for silhouette computations, mesh arrangement [Zhou et al. 2016] encapsulated in libigl [Jacobson et al. 2018] for 3D exact Boolean operations.

Computing Visual Metrics: We compute the visual difference between any two meshes using GLSL. Given a view direction, we set the camera to be $3l$-away from the meshes, where l is the diagonal length of BBox($M_i \bigcup M_o$). We render each mesh to a 128×128 framebuffer and compute d_s and d_n using Equation 2 and Equation 1, respectively. $\tau_{n,s}$ are evaluated by repeating the process for C uniformly sampled view directions on S^2. Since we target at building models that are assumed to be oriented upright, only the upper half of S^2 is used for sampling. We have experimented with the number of view directions and its influence on corresponding visual metrics. The metric values converge when $C \geq 250$. To be conservative, we set $C = 10^3$ for all experiments below.

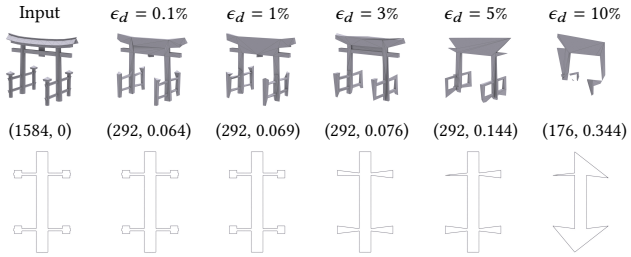

Figure 10: A larger ϵ_d leads to a more simplified silhouette (second row) and therefore a final mesh (first row) with less details preserved. (•, •) denotes (face number,τ_n).

Re-meshing Operators: In our third step, we execute an edge-collapse and an edge-flip iteratively while maintaining M_c's topology. For edge-collapse, we employ QEM [Garland and Heckbert 1997] to rank all the edges and add a virtual perpendicular plane with a small weight for each edge to fight against coplanar degeneracy, which is a common numerical instability in QEM. We perform an edge-flip if any pair of adjacent triangles has an obtuse dihedral angle larger than θ_π or if the exterior dihedral angle is smaller than $\theta_{2\pi}$, where θ_π (resp. $\theta_{2\pi}$) is a threshold close to π (resp. 2π).

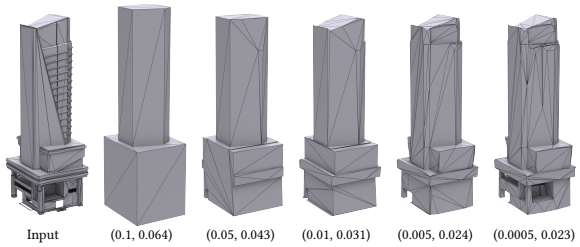

Figure 11: A smaller stopping criteria of ϵ_τ leads to M_c with richer details. (•, •) denotes (ϵ_τ,τ_n).

Hyper Parameters: We use the following default parameter settings: $\alpha = 175°$, which controls the number of regions generated from M_i, $\beta = 1°$, which is the threshold for two directions to be considered duplicate, $k = 50$ for choosing the view direction set (Section 3.1) and the slicing plane set (Section 3.2), $\theta_\pi = 175°$ and $\theta_{2\pi} = 5°$ for edge-flip (Section 3.3). ϵ_d specifies the maximum distance for simplifying of silhouettes. It plays an important role for generating the final mesh. Through extensive experiments, we found that $\epsilon_d = 1\%$ of the largest diagonal length of all silhouettes' bounding boxes achieves a good balance in terms of simplicity and visual appearance preservation. Fig. 10 shows an example of the different final meshes with one simplified silhouette corresponding to varying ϵ_d. The filtering process would discard all the loops with the area size less than 1% of the maximal surface area of the bounding squares of all the silhouettes. We choose $N = 20$ to allow a sufficiently large number of primitives. While ϵ_τ can greatly affect the complexity of the resulting meshes (as illustrated in Fig. 11), it is not intuitive for end-users to fine-tune and we empirically set $\epsilon_\tau = 1 \times 10^{-3}$ by default. In practice, we expose only a single parameter to users, the maximal number of triangles T of the output low-poly mesh. For all of our experiments, we set $T = 600$.

5 EXPERIMENT

We run all our experiments on a workstation with a 32-cores Intel processor clocked at 3.5 Ghz and 64 Gb of memory. We use OpenMP to parallelize the silhouette related computations, the candidate primitive selection in Section 3.1, as well as 3.2. We employ τ_n and light field distance (LFD, [Chen et al. 2003]) to measure the visual preservation of the low-poly models. Note that, conventionally used metrics, e.g. Hausdorff distance, may not fit well for the rendering applications, since they are designed to measure the geometry preservation, instead of visual preservation, of re-meshing operators. However, our input building models often involve geometries inside the building that contribute little to the visual perception.

Table 1: Statistics for input models shown in Fig. 2, 3, 6, 12, and 13, including number of vertices N_V, faces N_F, genus N_G, intersected face pairs N_S, components N_C, and holes N_H and manifoldness M_{an}. Note that genus and number of holes are not well-defined for non-manifold meshes.

Models	N_V	N_F	M_{an}	N_G	N_S	N_C	N_H
32 (Fig. 2)	27k	40k	Yes	1	42k	1894	2632
16 (Fig. 3)	4k	6k	No	-	5k	151	-
20 (Fig. 6)	4k	6k	Yes	4	6k	234	285
50 (Fig. 10)	1k	3k	Yes	0	2k	20	0
24 (Fig. 11)	2k	3k	Yes	1	4k	112	147
17 (Fig. 12 top)	3k	4k	No	-	5k	150	-
95 (Fig. 12 bottom)	13k	30k	Yes	891	5k	34	0
55 (Fig. 13top)	6k	10k	Yes	1	18k	280	114
63 (Fig. 13 bottom)	6k	10k	Yes	0	6k	99	135
43 (Fig. 14)	5k	10k	Yes	3	14k	151	34

Dataset: We collect 100 building models with various styles commonly appear in today's digital games (Fig. 1). The models in our dataset have complex geometry and topology, where 39% (resp. 88%) are non-manifold (resp. non-watertight). Of these models, the average number of triangles, intersecting face pairs, and disconnected components are 20k, 35.6k, and 685, respectively. For models that are manifold, the average number of genus and holes are 136 and 640, respectively. Table 1 summarizes the statistics of the input models that appear in Fig. 2, 3, 6, 12, and 13. The full statistics of the dataset can be found in our supplementary file. Fig. 1 demonstrates our results for the entire dataset. A short clip for each model can be found in the the attached video, comparing our method with state-of-the-art competing techniques.

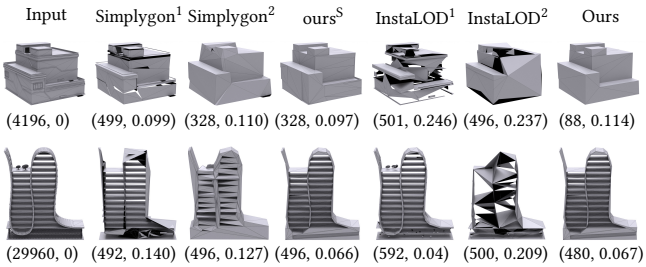

Figure 12: Comparison with commercial software. $(\bullet, \bullet)$ denotes (N_F, τ_n).

Comparison with Commercial Software: InstaLOD [Nerurkar 2021] and Simplygon [AB 2021] are state-of-the-art commercial solutions that can automatically generate simplified meshes and are popularly used by game studios. We compare with results generated by the various modules from InstaLOD and Simplygon, including the InstaLOD optimization (InstaLOD[1]), the InstaLOD re-meshing (InstaLOD[2]), the Simplygon reduction (Simplygon[1]), and the Simplygon re-meshing (Simplygon[2]). Table 2 summarizes the statistics in terms of the average and standard deviation of faces number, τ_s, τ_n, and the simplification rate. By default, we manually pick a mesh from the Pareto front as our final result. For fairness, we further extract from the Pareto front a mesh with the same number of faces as one generated by Simplygon[2] (Ours[S]). We hightlight our method against Simplygon[2], because Simplygon[2] performs the best among the four modules provided by both InstaLOD and Simplygon. Our approach generates results having smaller element numbers and better visual appearance preservation. We highlight this improvement in Fig. 12 for two sample models.

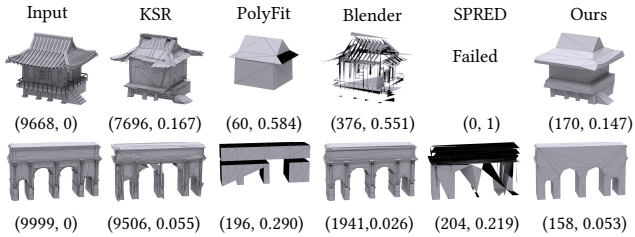

Figure 13: Comparison with academia and open-source solutions, where $(\bullet, \bullet)$ denotes (face number,τ_n). Note that, inverted faces appears in the results of PolyFit even after reorientation use MeshLab [Cignoni et al. 2008].

Comparison with Academia Solutions: KSR [Bauchet and Lafarge 2020], PolyFit [Nan and Wonka 2017], SPRED [Salinas et al. 2015], and the decimate modifier in Blender [Community 2022] are employed for comparison. To employ KSR and PolyFit for low-poly meshing, we uniformly sample each 3D model into a point cloud using the point cloud library (PCL, [Rusu and Cousins 2011]) by setting 1M point samples as the upper bound and 1×10^{-4} of the diagonal length of its bounding box as the sampling interval. We batch process the dataset by KSR's public binary program using parameters $n_{min} = 50$, $K = 2$, and $\lambda = 0.3$, PolyFit's CGAL implementation with the default parameter settings, SPRED's public program with steps: load mesh→detect primitives→regularize primitive→filter small proxies→recompute graph→edge collapse with the target $N_V = 300$, and decimate modifier's collapse option in Blender with the target simplification ratio for each model corresponding to $N_F = 300$. Because not enough primitives are found, KSR and PolyFit fail to produce any results for 40 and 9 out of 100 input models, respectively. SPRED crashes for 49 models. Blender processes all inputs but cannot simplify the mesh further to achieve the target N_F for many models. As shown in Table 2, our method produce results having smaller $N_F^{Avg.}$ but higher visual preservation than those from KSR, SPRED, and Blender. Since PolyFit generates fewer number of triangles on average, we summarize the results that are successfully handled by PolyFit and extract our results (Ours[P]) by matching their face numbers. As illustrated in Table 2, with the same number of faces, Ours[P] ($\tau_n^{\text{Avg.}} = 0.0946$) has a much better visual appearance preservation than PolyFit ($\tau_n^{\text{Avg.}} = 0.3576$). Fig. 13 shows two sample models for visual comparison of these methods.

(10000, 0) (260934, 0.008) (380, 0.545) (2200, 0.118) #V = 313000 (366158, 0.223) (1289, 0.302) (199, 0.296) (96, 0.117)

Figure 14: Comparison with alternative pipelines (from left to right): the input, surface mesh repaired by [Diazzi and Attene 2021], mesh simplification [Cignoni et al. 2008] with and without topology preservation, point cloud by virtual-scanning the input, mesh by poisson reconstruction, mesh simplification with and without topology preservation, and the result of our method. $(\bullet, \bullet)$ denotes (face number, τ_n).

Comparison with Alternative Pipelines: While not being publicly documented, two alternative pipelines could be employed for generating the desired low-poly meshes: 1) applying mesh repairing first, e.g. [Diazzi and Attene 2021; Huang et al. 2018, 2020], and then conducting re-meshing algorithms, e.g. QEM-guided mesh simplification using meshlab [Cignoni et al. 2008], and 2) virtual scanning the input using [Gschwandtner et al. 2011] to obtain a point cloud, Poisson reconstruction [Kazhdan and Hoppe 2013] of the point cloud, and then run QEM re-meshing. We compare our method with these two pipelines. As demonstrated in Fig. 14, each pipeline has its own issues. For the repair-and-QEM pipeline, although mesh-repairing approaches can fix the mesh to some extent, local re-meshing operators cannot generate satisfactory results when the desired element count is extremely small. For the scan-reconstruct-QEM pipeline, it often reconstructs surfaces with the following issues: 1) very high genus; 2) redundant surfaces near open regions of the input; and 3) smoothed out sharp features of the buildings.

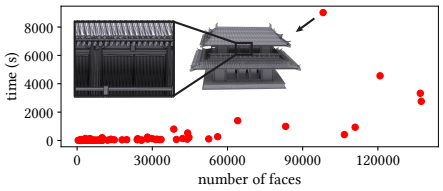

Figure 15: Timing plots of our approach on the dataset.

Table 2: Statistics of the results generated for the entire dataset, including percentage of results that are watertight W, average number of faces $N_F^{\text{Avg.}}$, average and standard deviation of LFD, silhouette difference, normal difference, and simplification ratio, respectively ($L^{\text{Avg.}}$, L^{SD}, $\tau_s^{\text{Avg.}}$, τ_s^{SD}, $\tau_n^{\text{Avg.}}$, τ_n^{SD}, $R^{\text{Avg.}}$, R^{SD}). For all listed numerical metrics, smaller values are more desired.

	W	$N_F^{\text{Avg.}}$	$L^{\text{Avg.}}$	L^{SD}	$\tau_s^{\text{Avg.}}$	τ_s^{SD}	$\tau_n^{\text{Avg.}}$	τ_n^{SD}	$R^{\text{Avg.}}$	R^{SD}
InstaLOD[1]	17%	527	1415	1690	0.035	0.045	0.099	0.093	0.103	0.121
InstaLOD[2]	97%	499	1309	1007	0.0411	0.049	0.135	0.101	0.109	0.148
Simplygon[1]	16%	499	1222	1378	0.0368	0.057	0.102	0.107	0.109	0.148
Simplygon[2]	100%	206	949	835	0.0235	0.009	0.084	0.043	0.030	0.033
Ours[S]	100%	196	907	902	0.015	0.011	0.066	0.042	0.029	0.032
KSR	61%	3121	8015	3040	0.167	0.215	0.200	0.205	0.80	1.04
SPRED	0%	962	2921	2899	0.11	0.14	0.235	0.144	0.10	0.12
Blender	21%	2848	4347	6268	0.105	0.10	0.186	0.139	0.135	0.119
PolyFit	100%	103	3700	2568	0.138	0.109	0.358	0.173	0.019	0.030
Ours[P]	100%	103	1285	1196	0.033	0.044	0.095	0.074	0.020	0.031
Ours	100%	152	817	651	0.016	0.012	0.068	0.047	0.026	0.044

Discussion and Limitations: In the above comparisons, while other approaches, e.g. InstaLOD[1], Simplygon[1], Simplygon[2], Blender, take only 1-2 seconds on average to process each model in our dataset, InstaLOD[2] needs 1 minute, and ours requires 6 minutes. Over the 100 models, it takes our method from a second to tens of minutes to process a model. Our computational bottleneck lies in the 2D Boolean operations during the silhouette computation for building models with excessive details, such as the one illustrated in Fig. 15. This step accounts 96.8% of the total computing time. For the dataset, our algorithm terminates within 3 mins for 85% of the models and the average computational cost is 45 secs. As a future work, we plan to address this bottleneck by combining rasterization-based silhoutte-extraction approaches.

Our method is specially designed to optimize the visual quality preservation of the to-be-generated low-poly models, which is different from most of the comparing approaches that designed for optimizing shape differences that could be employed for general purposes. Therefore, when artists choose the testing dataset, more weights are put on diversifying the building styles instead of geometry or topology complexities. However, for inputs with relatively simple topology and benign geometry properties or when the desired element count of the output is not extremely low, our approach might not be the best choice. Even if the input has a simple topology, the first two stages (e.g. Section 3.1 and Section 3.2) of our method cannot ensure that the produced mesh has a consistent topology, which may be a limitation for certain applications.

6 CONCLUSION

We propose a new and effective approach to generate a low-poly representation for commonly used 3D building models in digital games. Our key idea is to rely on the visual hull to generate topologically simple proxy meshes, and we design novel algorithms to construct and carve visual hulls using selected primitives for structural simplicity. We emphasize that, when the desired number of triangles of the low-poly mesh is larger than $1k$, traditional re-meshing methods and the commercial solutions are often good enough. Our method is designed exclusively for generating the coarsest mesh in the LOD hierarchy for building assets used by mobile applications.

ACKNOWLEDGMENTS

We would like to thank our colleagues from Lightspeed & Quantum Game Studios at Tencent — Guanli Hou, Guangyi Cai, Tianyu Gao, and Tiexiong Chen for running comparisons, preparing the executable program and the dataset. We also thank our colleagues — Ka Chen, Fengquan Wang, and Dong Li for their project support.

REFERENCES

Donya Labs AB. 2021. Simplygon 9. https://www.simplygon.com/Home/Index#section-solutions

Jean-Philippe Bauchet and Florent Lafarge. 2020. Kinetic Shape Reconstruction. *ACM Trans. Graph.* 39, 5, Article 156 (jun 2020), 14 pages.

Hervé Brönnimann, Andreas Fabri, Geert-Jan Giezeman, Susan Hert, Michael Hoffmann, Lutz Kettner, Sylvain Pion, and Stefan Schirra. 2017. 2D and 3D Linear Geometry Kernel.

Stéphane Calderon and Tamy Boubekeur. 2017. Bounding Proxies for Shape Approximation. *ACM Trans. Graph.* 36, 4, Article 57 (jul 2017), 13 pages.

Anne-Laure Chauve, Patrick Labatut, and Jean-Philippe Pons. 2010. Robust piecewise-planar 3D reconstruction and completion from large-scale unstructured point data. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. IEEE Computer Society, USA, 1261–1268.

Ding-Yun Chen, Xiao-Pei Tian, Yu-Te Shen, and Ming Ouhyoung. 2003. On Visual Similarity Based 3D Model Retrieval. *Computer Graphics Forum* 22, 3 (2003), 223–232.

Zhiqin Chen, Andrea Tagliasacchi, and Hao Zhang. 2020. BSP-Net: Generating Compact Meshes via Binary Space Partitioning. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, Seattle, WA, United States.

Paolo Cignoni, Marco Callieri, Massimiliano Corsini, Matteo Dellepiane, Fabio Ganovelli, and Guido Ranzuglia. 2008. MeshLab: an Open-Source Mesh Processing Tool. In *Eurographics Italian Chapter Conference*, Vittorio Scarano, Rosario De Chiara, and Ugo Erra (Eds.). The Eurographics Association, Salerno, Italy, 1–1.

David Cohen-Steiner, Pierre Alliez, and Mathieu Desbrun. 2004. Variational Shape Approximation. *ACM Trans. Graph.* 23, 3 (aug 2004), 905–914.

Blender Online Community. 2022. *Blender - a 3D modelling and rendering package*. Blender Foundation, Stichting Blender Foundation, Amsterdam. http://www.blender.org

Lorenzo Diazzi and Marco Attene. 2021. Convex Polyhedral Meshing for Robust Solid Modeling. *ACM Trans. Graph.* 40, 6, Article 259 (dec 2021), 16 pages.

Christopher Dyken, Morten Dæhlen, and Thomas Sevaldrud. 2009. Simultaneous curve simplification. *Journal of Geographical Systems* 11, 3 (sep 2009), 273–289.

Hao Fang and Florent Lafarge. 2020. Connect-and-Slice: An Hybrid Approach for Reconstructing 3D Objects. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, USA, 13487–13495.

Hao Fang, Florent Lafarge, and Mathieu Desbrun. 2018. Planar Shape Detection at Structural Scales. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. IEEE Computer Society, USA, 2965–2973.

Michael Garland and Paul S. Heckbert. 1997. Surface Simplification Using Quadric Error Metrics. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '97)*. ACM Press/Addison-Wesley Publishing Co., USA, 209–216.

Michael Gschwandtner, Roland Kwitt, Andreas Uhl, and Wolfgang Pree. 2011. BlenSor: Blender sensor simulation toolbox. In *International Symposium on Visual Computing*. Springer, Berlin, Germany, 199–208.

Jon Hasselgren, Jacob Munkberg, Jaakko Lehtinen, Miika Aittala, and Samuli Laine. 2021. Appearance-Driven Automatic 3D Model Simplification. In *Eurographics Symposium on Rendering - DL-only Track*, Adrien Bousseau and Morgan McGuire (Eds.). The Eurographics Association, Prague, Czech Republic, 19 pages. https://doi.org/10.2312/sr.20211293

Jin Huang, Tengfei Jiang, Zeyun Shi, Yiying Tong, Hujun Bao, and Mathieu Desbrun. 2014. ℓ1-Based Construction of Polycube Maps from Complex Shapes. *ACM Trans. Graph.* 33, 3, Article 25 (jun 2014), 11 pages. https://doi.org/10.1145/2602141

Jingwei Huang, Hao Su, and Leonidas J. Guibas. 2018. Robust Watertight Manifold Surface Generation Method for ShapeNet Models. *CoRR* abs/1802.01698 (2018). arXiv:1802.01698

Jingwei Huang, Yichao Zhou, and Leonidas J. Guibas. 2020. ManifoldPlus: A Robust and Scalable Watertight Manifold Surface Generation Method for Triangle Soups. *CoRR* abs/2005.11621 (2020). arXiv:2005.11621

Alec Jacobson, Daniele Panozzo, et al. 2018. libigl: A simple C++ geometry processing library. https://libigl.github.io/.

Michael Kazhdan and Hugues Hoppe. 2013. Screened Poisson Surface Reconstruction. *ACM Trans. Graph.* 32, 3, Article 29 (jul 2013), 13 pages.

Tom Kelly, John Femiani, Peter Wonka, and Niloy J. Mitra. 2017. BigSUR: Large-Scale Structured Urban Reconstruction. *ACM Trans. Graph.* 36, 6, Article 204 (nov 2017), 16 pages. https://doi.org/10.1145/3130800.3130823

Dawar Khan, Alexander Plopski, Yuichiro Fujimoto, Masayuki Kanbara, Gul Jabeen, Yongjie Zhang, Xiaopeng Zhang, and Hirokazu Kato. 2020. Surface Remeshing: A Systematic Literature Review of Methods and Research Directions. *IEEE Transactions on Visualization and Computer Graphics* 01, 01 (2020), 1–1.

H. T. Kung, F. Luccio, and F. P. Preparata. 1975. On Finding the Maxima of a Set of Vectors. *J. ACM* 22, 4 (oct 1975), 469–476.

Samuli Laine, Janne Hellsten, Tero Karras, Yeongho Seol, Jaakko Lehtinen, and Timo Aila. 2020. Modular Primitives for High-Performance Differentiable Rendering. *ACM Transactions on Graphics* 39, 6 (2020), 1–14.

A. Laurentini. 1994. The Visual Hull Concept for Silhouette-Based Image Understanding. *IEEE Trans. Pattern Anal. Mach. Intell.* 16, 2 (feb 1994), 150–162.

Aldo Laurentini. 1997. How Many 2D Silhouettes Does It Take to Reconstruct a 3D Object? *Comput. Vis. Image Underst.* 67 (1997), 81–87.

Thibault Lescoat, Hsueh-Ti Derek Liu, Jean-Marc Thiery, Alec Jacobson, Tamy Boubekeur, and Maks Ovsjanikov. 2020. Spectral Mesh Simplification. *Computer Graphics Forum* 39, 2 (2020), 315–324.

Minglei Li and Liangliang Nan. 2021. Feature-preserving 3D mesh simplification for urban buildings. *ISPRS Journal of Photogrammetry and Remote Sensing* 173 (March 2021), 135–150.

Fujun Luan, Shuang Zhao, Kavita Bala, and Zhao Dong. 2021. Unified Shape and SVBRDF Recovery using Differentiable Monte Carlo Rendering.

Wojciech Matusik, Chris Buehler, and Leonard McMillan. 2001. Polyhedral Visual Hulls for Real-Time Rendering. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*. Springer-Verlag, Berlin, Heidelberg, 115–126.

Wojciech Matusik, Chris Buehler, Ramesh Raskar, Steven J. Gortler, and Leonard McMillan. 2000. Image-Based Visual Hulls. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '00)*. ACM Press/Addison-Wesley Publishing Co., USA, 369–374.

Ravish Mehra, Qingnan Zhou, Jeremy Long, Alla Sheffer, Amy Gooch, and Niloy J. Mitra. 2009. Abstraction of Man-Made Shapes. *ACM Trans. Graph.* 28, 5 (dec 2009), 1–10.

Maxim Mikhnevich and Patrick Hebert. 2011. Shape from Silhouette Under Varying Lighting and Multi-viewpoints. In *2011 Canadian Conference on Computer and Robot Vision*. IEEE, St. Johns, Newfoundland Canada, 285–292.

Liangliang Nan and Peter Wonka. 2017. PolyFit: Polygonal Surface Reconstruction from Point Clouds. In *2017 IEEE International Conference on Computer Vision (ICCV)*. IEEE, Venice, Italy, 2372–2380.

Manfred M. Nerurkar. 2021. InstaLOD. https://instalod.com

Baptiste Nicolet, Alec Jacobson, and Wenzel Jakob. 2021. Large Steps in Inverse Rendering of Geometry. *ACM Trans. Graph.* 40, 6, Article 248 (dec 2021), 13 pages.

Alec Rivers, Frédo Durand, and Takeo Igarashi. 2010. 3D Modeling with Silhouettes. *ACM Trans. Graph.* 29, 4, Article 109 (jul 2010), 8 pages.

Radu Bogdan Rusu and Steve Cousins. 2011. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, Shanghai, China, 1–4.

D. Salinas, F. Lafarge, and P. Alliez. 2015. Structure-Aware Mesh Decimation. *Computer Graphics Forum* 34, 6 (2015), 211–227.

Pedro V. Sander, Xianfeng Gu, Steven J. Gortler, Hugues Hoppe, and John Snyder. 2000. Silhouette Clipping. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '00)*. ACM Press/Addison-Wesley Publishing Co., USA, 327–334.

Jean Ponce Svetlana Lazebnik, Yasutaka Furukawa. 2007. Projective Visual Hulls. *Int J Comput Vision* 74 (dec 2007), 137–165.

Tim Sweeney. 2021. Unreal Engine 5. https://www.unrealengine.com/en-US/unreal-engine-5

Richard Szeliski. 1993. Rapid Octree Construction from Image Sequences. *CVGIP: Image Underst.* 58, 1 (jul 1993), 23–32.

Bala R. Vatti. 1992. A Generic Solution to Polygon Clipping. *Commun. ACM* 35, 7 (jul 1992), 56–63.

Yannick Verdie, Florent Lafarge, and Pierre Alliez. 2015. LOD Generation for Urban Scenes. *ACM Trans. Graph.* 34, 3, Article 30 (may 2015), 14 pages.

Kaizhi Yang and Xuejin Chen. 2021. Unsupervised Learning for Cuboid Shape Abstraction via Joint Segmentation from Point Clouds. *ACM Trans. Graph.* 40, 4, Article 152 (jul 2021), 11 pages.

Qingnan Zhou, Eitan Grinspun, Denis Zorin, and Alec Jacobson. 2016. Mesh Arrangements for Solid Geometry. *ACM Trans. Graph.* 35, 4, Article 39 (jul 2016), 15 pages.