Continuous remeshing for inverse rendering

Werner Palfinger

Abstract
We present a novel method for joint optimization and remeshing and apply it to inverse rendering. Rapid advances in differentiable rendering during the last years paved the way for fast inverse rendering of complex scenes. But serious problems with gradient-based optimization of triangle meshes remain. Applying gradient steps to the vertices can lead to mesh defects, such as flipped triangles, crumpled regions, and self-intersections. Choosing a good vertex count is crucial for the optimization quality and performance but is usually done by hand. Moreover, meshes with fixed triangulation struggle to adapt to complex geometry. Our novel method tackles all these problems by applying an adaptive remeshing step in each single iteration of the optimization loop. By immediately collapsing suspicious triangles, we avoid and heal mesh defects. We use a closed-loop-controlled location-dependent edge length. We compare our solution to state-of-the-art methods and find that it is faster and more accurate. It produces finer meshes with fewer defects, requires less parameter tuning and can reconstruct more complex objects.

KEYWORDS
differentiable rendering, geometry reconstruction, inverse rendering, mesh optimization, remeshing

1 INTRODUCTION
Inverse rendering is the problem of reconstructing an unknown 3D scene from 2D images. Conventional methods in this field are usually based on finding correspondences between the images. Obviously, the shading of the corresponding surface region needs to be similar for the involved viewing directions. Wherever the surface is not rough enough, or the lighting is not smooth enough, the matching may completely fail.1

In recent years, differentiable rendering came up to solve this problem at its root. Given a powerful enough differentiable renderer and a powerful enough gradient based general purpose optimizer, any inverse rendering problem can, in theory, be solved by minimizing an image loss, using a simple optimization loop. However, solving for a 3D scene, with thousands or millions of strongly coupled parameters, is often a hard and highly non-convex problem.

For differentiable rendering, as with conventional rendering, surfaces are usually represented by triangle meshes. But the simplicity and efficiency of mesh-based surfaces come with a major drawback. Optimizing the vertex positions
Mesh vertices receive two different kinds of gradients from differentiable renderers:

- Shading gradients, coming from deviating surface normals, reflection properties, or lighting, and
- Silhouette gradients, coming from deviating shape

A major source of critical mesh distortions are sparse or inconsistent silhouette gradients. Figure 1 illustrates two important examples. For growing regions (left drawing), silhouette gradients continuously pull the same sparse vertices out of the silhouette, keeping all other vertices unchanged. In contrast, for shrinking regions (right drawing), silhouette gradients push vertices into the silhouette. As a result, when averaged over time, shrinking silhouette gradients are not sparse. But gradients coming from different images may be inconsistent, often leading to strong crumpling in the affected zones.

When using fixed triangulation and standard optimizers, like Adam\(^2\), these problems can only be avoided by using strong regularization terms and small learning rates\(^1\). But these regularizations penalize non-smooth surfaces and, therefore, lead to incorrect results.

Finer meshes require stronger regularization and smaller learning rates. The reason is simply, that with smaller vertex distances, updates can more likely lead to critical distortion. Therefore, starting with a coarse mesh and stepwise refining the mesh during the optimization can greatly speed up the optimization and make it more robust, as seen in previous work.\(^3\)\(^5\) This basically splits the optimization into multiple stages. But this approach requires additional hyperparameters that strongly depend on the complexity of the scene.

We propose a method that generalizes the idea of coarse-to-fine optimization. We keep the triangle size spatially and temporally within a near-optimal range by continuously and adaptively remeshing the surface. Keeping a good triangle size avoids the majority of critical distortions and face flips. Remaining defects are immediately healed by collapsing suspicious edges. Our method is scale-invariant and can handle models of different complexity without parameter-tuning.

Figure 2 shows an intermediate state of our method reconstructing the Lucy model with color-coded triangle sizes for visualization. While red areas are still coarse, blue areas are already close to the target surface and can be refined. The process can be best seen in our supplemental videos.

In summary, our contributions are:

- An optimization rule that continuously estimates the optimal local edge length.
- A GPU-friendly adaptive remeshing algorithm, fast enough to be applied after each optimization step.

We give an overview of related work in Section 2, explain our basic ideas in Section 3, present our method in Section 4 and our implementation in Section 5. In Section 6, we show results and compare our solution to other methods.

\(^1\)We use the common term learning rate here, although we actually mean an optimization step size, with no machine learning involved.
FIGURE 2 Intermediate state of our method reconstructing the Lucy model (see Section 6.2). Colors indicate the adaptive reference edge length $l_{ref}$. Please watch the complete reconstruction in our supplemental videos.

2 | RELATED WORK

2.1 | Differentiable rendering

Great advances have been achieved in the field of differentiable rendering during the last few years. The availability of powerful GPUs and general-purpose libraries opened the door for a brand-new research area. As with conventional rendering, there are two quite different approaches:

- Differentiable rasterizers project all visible primitives (triangles) to the camera image. Occlusions are handled using a depth buffer. Loper et al.\textsuperscript{6} (OpenDR) and Kato et al.\textsuperscript{7} (Neural 3D Mesh Renderer) developed approximate differentiable rasterizers. Liu et al.\textsuperscript{8} proposed Soft Rasterizer, which smoothens the rasterization process to get gradients from discontinuities. PyTorch3D, a very popular pytorch library by Ravi et al.\textsuperscript{9} lowered the barrier for beginners. In 2020 finally, Laine et al.\textsuperscript{10} from NVIDIA published Nvdiffrast, a very fast OpenGL-based differentiable rasterizer.

- Differentiable ray tracers, on the other hand, cast rays through each pixel of the camera image. While that enables handling all kinds of physical interactions of the light, rendering is usually much slower than with rasterizers. Li et al.\textsuperscript{11} handle discontinuities by sampling additional rays on the object edges. In 2019, Nimier-David et al.\textsuperscript{12} released Mitsuba 2, an extremely versatile physically correct renderer. Since then, new methods were developed to improve and greatly speed up differentiable ray tracing.\textsuperscript{13-15}
2.2 | Mesh optimization for inverse rendering

Various methods exist to avoid distortions, face flips, and self-intersections:

- **Mesh regularization** adds different kinds of penalty terms to the rendering loss to smooth the mesh.\(^9,16\) Laplacian and bi-Laplacian are the most popular terms.
- **Diffeomorphic mesh flows** utilize ordinary differential equations to force smooth surfaces.\(^4,17\) While that solves the problem of face-flipping and self-intersection, the method typically requires large neural networks and is therefore comparably slow.
- Nicolet et al.\(^3\) optimize differential coordinates instead of raw vertices to smooth out sparse gradients. This can also be seen as a Laplacian preconditioning. The method is fast and robust for smooth surfaces, but the reconstruction of higher frequencies is typically slow.
- For all methods above, **coarse-to-fine optimization** is in general beneficial to robustness and performance as reported by Nicolet et al.,\(^3\) Cheng et al.,\(^4\) and Luan et al.\(^5\) The optimization loop is split into about 2–10 stages. After each stage, the mesh is refined using standard remeshing methods.

2.3 | Remeshing

Back in 1993, Hoppe et al.\(^18\) first proposed an explicit remeshing procedure, using edge collapse, edge split and edge flip. Botsch et al.\(^19\) further investigated the method. Surazhsky et al. propose an area-based smoothing technique.\(^20\) Jakob et al.\(^21\) optimize both the edge orientations and vertex positions.

2.4 | Adam optimizer

The well-known Adam\(^2\) optimization rule is in short:

\[
\begin{align*}
g & \leftarrow \frac{\partial \Phi}{\partial x}, \\
m_1 & \leftarrow \beta_1 m_1 + (1 - \beta_1)g, \\
m_2 & \leftarrow \beta_2 m_2 + (1 - \beta_2)g^2, \\
\hat{m}_1 & \leftarrow m_1 / (1 - \beta_1^t), \\
\hat{m}_2 & \leftarrow m_2 / (1 - \beta_2^t), \\
x & \leftarrow x - \alpha \frac{\hat{m}_1}{\sqrt{\hat{m}_2} + \epsilon},
\end{align*}
\]

where \(\Phi\) is the objective function (loss), \(x\) is a parameter to optimize, \(g\) is the parameter gradient, \(m_1\) is the first moment (momentum), \(m_2\) is the second moment (uncentered variance), and \(\hat{m}_i\) are the bias-corrected moments.

3 | IDEA

3.1 | IsotropicAdam

When applied to triangle meshes, \(x\) in Equations (1) and (6) usually holds the vertex positions. With \(V\) being the number of vertices, there are \(3V\) degrees of freedom. Optimizing these parameters independently breaks spatial symmetry in Equation (6). So it is useful to see \(x\) as a vector-valued parameter array (denoted in bold), in other words, \(x, g, m_1\) become matrices of size \(V \times 3\). Now it is easy to fix the symmetry break in \(m_2\) by using the spatial L2-norm of \(g\). We will call this...
slightly modified update rule ISOTROPICADAM:

\[ g \leftarrow \frac{\partial \Phi}{\partial x}, \quad (7) \]

\[ m_1 \leftarrow \beta_1 m_1 + (1 - \beta_1) g, \quad (8) \]

\[ m_2 \leftarrow \beta_2 m_2 + (1 - \beta_2) ||g||^2, \quad (9) \]

\[ x \leftarrow x - \frac{\alpha}{\sqrt{m_2}} m_1. \quad (10) \]

\( || \cdot || \) in Equation (9) denotes the L2-norm over the three spatial coordinates. Note that \( m_2 \) is now scalar-valued again, so ISOTROPICADAM requires substantially less calculations than ADAM.

3.2 Relative velocity

The change of \( x \) in Equations (6), (10) is the discrete vertex velocity. It is the product of the learning rate \( \alpha \) and a term that the authors of ADAM call signal noise ratio,\(^2\) which is quite misleading. In fact, the term is related to signal and noise, but it is not their ratio. As we are dealing with vertex velocity, we prefer the name relative velocity \( \nu \). We omit \( \epsilon \) from here for better reading.

\[ \nu = -\frac{m_1}{\sqrt{m_2}}. \quad (11) \]

It is easier to understand the meaning of \( \nu \), if we investigate the one-dimensional case, neglect the different values for \( \beta_1 \) and \( \beta_2 \) and introduce a real signal noise ratio \( SNR = m_1/\sigma \), using the standard deviation \( \sigma \) and its well-known property \( \sigma^2 = m_2 - m_1^2 \):

\[ -\frac{m_1}{\sqrt{m_2}} = -\frac{m_1}{\sqrt{\sigma^2 + m_1^2}} = -\frac{SNR}{\sqrt{1 + SNR^2}}. \quad (12) \]

The last expression in Equation (12) is a sigmoid function of \( SNR \), shown in Figure 3. We can see from this simplified version of \( \nu \), that \( || \nu || \) is close to one for large \( SNR \) (uniform gradients far away from the target surface) and goes to zero for small values of \( SNR \) (varying gradients close to the target surface).

\(^2\)To avoid confusion, we do not use the symbol \( v \), which often denotes \( m_2 \).
The learning rate $\alpha$ defines a trust region with radius $r = \alpha \cdot l$ around the vertices.

We optimize a 1D mesh (black line) to match a target shape (gray dashed line). Please also watch our supplemental video.

The same holds in the three-dimensional case. ISOTROPICADAM updates vertices with uniform gradients at a maximum speed close to the learning rate $\alpha$. When the vertices get closer to the target surface, gradients typically become more and more noisy and ISOTROPICADAM reduces the update speed accordingly.

### 3.3 Learning rate and edge length

As explained above, the learning rate $\alpha$ can be seen as a maximum vertex speed. Note, however, that because of the different values for $\beta_1$ and $\beta_2$, $||v||$ can actually be larger than one in some cases. The authors of ADAM interpret $\alpha$ as a trust region around the current parameter value. In the context of triangle meshes, a single iteration should not flip triangles, so the trust radius is directly related to the edge length, for example $l/2$, see Figure 4. Knowing that $\alpha$ in (10) should be proportional to $l$, we modify the update rule and redefine $\alpha$ to get a scale-invariant learning rate:

$$x \leftarrow x - \alpha \cdot l \cdot v.$$  \hfill (13)

### 3.4 Optimal edge length

We have seen in Sections 1 and 3.3, that choosing a good triangle size (edge length) is crucial for performance and robustness of the optimization. Let's look at a very simple 1D example to find out how to choose the optimal edge length. We optimize a chain (black line in Figure 5) to match a target curve (gray/yellow). According to the shading gradients in the 3D case, we minimize the mean squared error of the chain normals, using ADAM with a learning rate proportional to the edge length, see Equation (13). The gray lines in Figure 7 show the results for different fixed vertex counts. As expected, the coarse chains converge faster, but once they reach the target surface, they get stuck with no further improvement. Ideally, we would adapt the edge length to the distance to the surface. But the surface is of course unknown, and so we need to estimate the distance somehow.

Fortunately, the relative speed $|v|$ is closely related to the distance to the target surface, as discussed in Section 3.2. $v$ depends on the actual triangulation, so it cannot serve as an absolute distance estimation, but at least it indicates whether the current edge length is too short, good or too long. Knowing this, we can build a linear closed-loop controller for estimating the optimal edge length $l_{ref}$. We initialize $l_{ref}$ with the initial edge length and update it in each iteration using

$$l_{ref} \leftarrow l_{ref} + G \cdot (|v| - v_{ref}).$$  \hfill (14)
We control the reference edge length (green dashed line) using the deviation of the relative velocity $\nu$ (red line) from a reference value $\nu_{\text{ref}}$ (dashed red line).

Continuously reducing the edge length would introduce unnecessary interpolation errors. So, we use a relative edge length tolerance of $\text{tol} = 0.6$ and remesh once the deviation $|l - l_{\text{ref}}| > l_{\text{ref}} \cdot \text{tol}$ (green area in Figure 6). The green line in Figure 7 shows the results of the adaptive remeshing. As intended, the result is at any point during optimization better than the respective best result with fixed vertex count.

where $G$ is a gain constant, $|\nu|$ is the mean relative speed of all vertices and $\nu_{\text{ref}} \in [0, 1]$ is a fixed reference value, indicated in Figure 3 with blue lines. We choose $\nu_{\text{ref}} = 0.3$ and $G = 0.5$ for the 1D experiment. (It turns out that the same numbers also work great for 3D meshes.) Figure 6 illustrates the edge length controller.

Continuous reducing the edge length would introduce unnecessary interpolation errors. So, we use a relative edge length tolerance of $\text{tol} = 0.6$ and remesh once the deviation $|l - l_{\text{ref}}| > l_{\text{ref}} \cdot \text{tol}$ (green area in Figure 6). The green line in Figure 7 shows the results of the adaptive remeshing. As intended, the result is at any point during optimization better than the respective best result with fixed vertex count.

**METHOD**

To recap the 1D example in Section 3.4, we used the mean relative speed $|\nu|$ of all vertices as a feedback value to control the reference edge length $l_{\text{ref}}$ in Equation (14). We then applied a global remeshing every time the reference edge length deviated too much from the actual edge length. We could do the same global remeshing for triangle meshes. But for
non-trivial surfaces, the relative speed (the distance to the target) is always locally varying, so we use a per-vertex reference edge length with continuous adaptive remeshing. Putting everything together, our final update rule is:

\[ g \leftarrow \frac{\partial \Phi}{\partial x}, \]  
\[ m_1 \leftarrow \beta_1 \cdot m_1 + (1 - \beta_1) \cdot g, \]  
\[ m_2 \leftarrow \beta_2 \cdot m_2 + (1 - \beta_2) \cdot \|g\|^2, \]  
\[ \hat{m}_1 \leftarrow \frac{m_1}{(1 - \beta_1^t)}, \]  
\[ \hat{m}_2 \leftarrow \frac{m_2}{(1 - \beta_2^t)}, \]  
\[ \nu \leftarrow -\frac{\hat{m}_1}{\sqrt{\hat{m}_2}}, \]  
\[ x \leftarrow x + \alpha \cdot \text{ref} \cdot \nu, \]  
\[ l_{\text{ref}} \leftarrow l_{\text{ref}} + G(\|\nu\| - \nu_{\text{ref}}). \]

In the last Equation (23) we apply a remeshing step using the vertices \( x \), faces \( f \) and reference edge length \( l_{\text{ref}} \) as explained in Section 4.1.

### 4.1 Remeshing

We remesh after each vertex update to immediately heal any defects. A full remeshing at each iteration would be too expensive, but we can apply a single step of an iterative remeshing algorithm, embedding the remeshing loop continuously into the optimization loop.

Our remeshing method is based on the popular explicit surface remeshing\(^{18-20}\) which iteratively applies edge-split, edge-collapse, edge-flip, smoothing, and relocating, where, in our case, relocating is already covered by the gradient-based updates. These steps normally iterate sequentially over the edges. We adapt the algorithm to process the vertices in parallel to be able to run the complete remeshing on a GPU. This is not trivial, as interference of parallel operations must be prevented. Moreover, we extend edge-collapse to better heal defects. Our remeshing algorithm is quite complex and includes many heuristics, but it is based on proven standard methods and so it works out of the box without any parameter tuning, using only the mesh and target edge length as inputs.

Remeshing during the optimization covers three main objectives:

- Keep the (local) edge length close to the optimum.
- Heal defects coming from sparse or inconsistent vertex gradients.
- Keep the mesh uniform and well triangulated.

The first objective is covered by edge-split and edge-collapse. Splitting and collapsing seriously impacts the triangulation quality, so it is crucial to avoid unnecessary splits and collapses. Therefore, we use an edge length tolerance of typical \( tol = 0.5 \).

\[ l_{\text{min}} = l_{\text{ref}} \cdot (1 - tol), \]  
\[ l_{\text{max}} = l_{\text{ref}} \cdot (1 + tol). \]
FIGURE 8  *Left:* Selected edges of an existing mesh (black/gray) are split. We use an equal-length example for better illustration. New vertices are inserted at the midpoints (red).  *Right:* The corners of affected triangles are moved counter-clockwise to the new vertices (c₂ and c₃ in this case), shrinking the existing triangle (gray). At each of these corners, a new triangle is inserted (blue, green).

FIGURE 9  We detect face flips (folds) by comparing each face normal with the mean normal of its corner vertices.

The second objective is covered by *edge-collapse* and the third by *edge-flip* together with Laplacian regularization. We now describe all remeshing steps in detail:

### 4.2 Edge split

We split all edges that are longer than \( l_{\text{max}} \) in parallel. Figure 8 illustrates the algorithm.

First, all existing triangles (gray) are processed: If an edge of a triangle is to be split, the corner to the left of the edge is moved to the midpoint of the edge. That shrinks affected triangles in-place. Then, the resulting holes are filled with new triangles (colored).

### 4.3 Edge collapse

This is the most challenging part of our method. *Edge-collapse* is responsible for avoiding and healing defects. On the other hand, edges to be collapsed must be chosen carefully to prevent interference with other collapses running in parallel. Moreover, surface topology must be preserved.

We check all faces for face flips in each iteration. Face flips are actually small folds as illustrated in Figure 9, and there is no obvious dividing line between real folds in the target surface and optimization defects. We use the normalized area-weighted sum of face normals \( n_f \) as vertex normals. For each triangle, we calculate the mean of its corner vertex normals and use that as a reference face normal \( n_N \). A face is considered flipped if the dot product \( \langle n_f, n_N \rangle < 0 \) (see Figure 9). That works best, if none of the adjacent triangles is flipped. So, again, it is important to apply the remeshing in each iteration of the optimization, before more triangles are flipped.

Due to the topology preserving rule (see below), narrow triangles may permanently resist collapsing, although they have edges smaller than \( l_{\text{min}} \). That leads to narrow tube artefacts in the results. Therefore, we also try to collapse all faces with an area \( A < l_{\text{min}}^2/2 \).

For each face candidate (flipped or small), we choose an edge to be collapsed. Normally the shortest edge of the triangle is the best choice, but again, the topology preserving rule may prevent an edge collapse, so we randomly choose one of the two other edges with a 10% chance respectively.
We rank the candidate edges using a collapse priority:

\[
p = \max \left( 1 - \frac{l}{l_{\text{min}}}, 0 \right) + \begin{cases} 
1 & \text{if face collapse} \\
0 & \text{otherwise.}
\end{cases}
\]  

Then we remove all candidates that are closer than three edges to a candidate with higher priority rank to prevent interference between the collapses. For each remaining candidate, we count the number of edge-pairs, that connect the start and end vertex of the candidate. If it is larger than two, a collapse would cause a topology change, so the candidate is removed.

For all remaining edge candidates, the start and end vertices are merged into the edge midpoints. We also merge \( m_1, m_2, \nu, \) and \( l_{\text{ref}} \) jointly with the vertex positions, so the remeshing smoothly integrates with the vertex update.

### 4.4 Edge flip

The purpose of edge flip is to keep the degree (valence) of the vertices close to their optimal value. The degree \( d_i \) of a vertex \( i \) is the number of associated edges, its optimal value is 6 (or 4 for boundary edges).\(^9\) We minimize the sum of squared degree errors \( \hat{d}_i \):

\[
\Phi_d = \sum_{i=1}^{V} \hat{d}_i^2 ,
\]

\[
\hat{d}_i = \begin{cases} 
d_i - 4 & \text{if vertex } i \text{ is on boundary} \\
d_i - 6 & \text{otherwise.}
\end{cases}
\]

Figure 10 illustrates a case where a single edge flip reduces the total \( \Phi_d \). We renumber the involved vertices with \( i = \{ 1, ..., 4 \} \). For a single edge flip to reduce \( \Phi_d \), with the degree changes \( \Delta \hat{d}_i \in (-1, 1) \), we get the simple inequation

\[
\Delta \Phi_d = \sum_{i=1}^{4} \left( \left( \hat{d}_i + \Delta \hat{d}_i \right)^2 - \hat{d}_i^2 \right) < 0,
\]

\[
\hat{d}_1 + \hat{d}_2 - \hat{d}_3 - \hat{d}_4 > 2.
\]
4.5 | Smoothing and regularization

Explicit remeshing always includes a smoothing step, usually Laplacian smoothing. This smoothing is essential to achieve a good triangulation while splitting, collapsing and flipping edges. Some methods apply a tangential smoothing to avoid surface shrinking, but that requires reliable normal vectors, and so it did not work out in our experiments.

Note that, as we embed the iterative remeshing into the optimization loop, this smoothing step is equivalent to a Laplacian regularization term in the optimization loss.

As already discussed, Laplacian regularization is required not only as part of the remeshing, but also to smooth out sparse or inconsistent gradients. This is particularly important when the vertex updates are large with respect to the edge length, in other words, when the relative speed $v = \|\mathbf{v}\|$ is large. On the other hand, when the mesh is close to the final result and the relative speed is low, smoothing is undesirable, as it impacts the result. So, we use a speed-weighted smoothing

$$
\mathbf{x} \leftarrow \lambda v L \mathbf{x},
$$

(31)

where $\lambda$ is a smoothing hyperparameter and $L$ is the combinatorial Laplacian matrix.

5 | IMPLEMENTATION

We implemented our method in pure python, using pytorch for remeshing and optimization. Source code and data are available at https://github.com/Profactor/continuous-remeshing.

Pytorch is designed to efficiently work on fixed-sized tensors, so we first experimented with remeshing operations on static tensor shapes. It turned out, that this is only faster for small meshes ($V \leq 10^3\ldots10^4$). For large meshes, due to the vast amount of useless operations, the calculation overhead outweighs the saved resizing overhead. So, we use dynamic shapes, but avoid resizing where possible.

Pytorch’s scatter operations are heavily used for smoothing and remeshing operations. In order to avoid resizing, we use the value zero to mask unused elements in referencing tensors (e.g., faces, edges) and prefix the referenced tensor (e.g., vertices) with a single dummy element. Using this trick, we can perform fast “masked” scattering and other operations on fixed-sized tensors.

Using pytorch’s just-in-time compiler (TorchScript) can accelerate remeshing, but the saving is usually less than 10%. We believe that there is great potential for optimization using for example custom CUDA kernels.

Where not otherwise stated, the experiments in this article were made with 16 images per reconstruction and an image size of 512 × 512 pixels. We chose the fast rasterizer Nvdiffrast for differentiable rendering. As we focus on remeshing and optimization, the simplest possible shading was used, a linear mapping of Cartesian normal vector components to RGB components. This is equivalent to a first order spherical harmonics approximation of a diffuse reflection.

One of the major strengths of our solution is its strict scale-invariance. All operations are taking the edge length or reference edge length into account, for example in the update rule (21). This enables our method to optimize meshes from coarse to very fine using the same hyperparameters. In order to achieve scale invariance for the whole optimization, also the gradients coming from the differentiable renderer must be linear with respect to scaling. It is not at all obvious that the rendering loss provides such a linearity and that the renderer gives correct linear gradients.

We tested the scale-linearity of Nvdiffrast and observed very good results for shading gradients but significant limitations with silhouette gradients at small triangle sizes. This is a known limitation of Nvdiffrast and is related to its silhouette search that relies on the silhouette triangles being visible in the image. We found from our experiments, that severe mesh defects arise if we reduce the edge length below the size of one pixel (0.005 for our setup).

For unknown reasons, we sometimes observe huge gradients on single pixels coming from Nvdiffrast. This happens only once in hundreds of images but can lead to severe artefacts. As a workaround, we limit the gradients to $|g| < 10 \cdot |m_1|$ which has no impact on regular gradients.

6 | RESULTS

All experiments in this article were performed on an Intel i7-11700K CPU and an NVIDIA RTX 3090 GPU.
FIGURE 11  We benchmark our remeshing implementation on meshes of different size. The durations of the three remeshing steps are stacked, so the top line shows the total duration.

6.1  |  Benchmarks

Figure 11 shows results from our remeshing benchmarks. We first measure edge-split alone, then edge-collapse alone and finally all three steps together. In total, one remeshing iteration takes about five milliseconds for small meshes and computing time grows linearly with respect to vertex count for large meshes.

While the remeshing duration depends only on the vertex count, rendering heavily depends on image size and count as well as shading. With our very simplistic shading, an iteration without remeshing takes about 10 ms. Physically based shading typically takes much longer, which makes the remeshing costs often negligible.

6.2  |  Comparison

We compare our solution to two state-of-the-art methods:

- ADAM with Laplacian regularization.
- Adam-Remesh: Adam optimization with regularization and remeshing at fixed intervals using the method of Botsch et al.\textsuperscript{19} and implemented in C++ by Sellán.\textsuperscript{24}

We selected seven test models. The two baseline methods struggle to reconstruct the two challenging models Deer and Smilodon, so we grouped the models by complexity:

- Basic models: Nefertiti, Bunny, Armadillo, Lucy, Horse.\textsuperscript{25}
- Complex models: Deer,\textsuperscript{26} Smilodon.\textsuperscript{27}

All models were scaled and shifted to fit into a unit sphere (radius $r = 1$). A sphere with $r = 0.5$ was used as initial mesh, using 10,242 vertices for ADAM and 162 vertices for ADAM-REMESH and OURS. Images were rendered as explained in Section 5. The mean absolute pixel difference (RGBA) was used as objective function.

All methods use a linear warm-up phase (learning ramp) at the beginning of the optimization (and after each remeshing for ADAM-REMESH). The actual learning rate is calculated as

$$\tilde{\alpha} = \alpha \cdot \min \left( 1, \frac{n}{\text{ramp}/(1 - \beta_1)} \right).$$

(32)

The hyperparameters of all three methods were carefully tuned to minimize the overall mean squared surface distance of the five Basic Models after three seconds of optimization. (We calculate distances both from the reconstructed mesh to the target mesh and vice versa.) In addition, ADAM-REMESH was tuned again, using only the two Complex Models

\textsuperscript{3}We also tried the Laplacian preconditioned optimization proposed by Nicolet al.,\textsuperscript{1} but with relatively poor results.
### Table 1: Tuned hyperparameters for the compared methods

| Parameter                  | Ours | Adam  | Adam-Remesh | Adam-Remesh Complex |
|----------------------------|------|-------|-------------|--------------------|
| Learning rate \( \alpha \) | 0.37 | 0.33  | 0.21        | 0.41               |
| Learning ramp              | 1.5  | 2.5   | 10          | 5                  |
| \( \beta_1 \)              | 0.8  | 0.81  | 0.81        | 0.87               |
| \( \beta_2 \)              | 0.8  | 0.81  | 0.99        | 0.96               |
| \( \lambda \)              | 0.013| 8.4   | 0.86        | 0.3                |
| Gain \( G \)               |      |       |             | 0.3                |
| \( I_{\text{min.}} \)      | 0.01 | 0.005 | 0.005       |                   |
| \( I_{\text{max}} \)       | 0.15 |       |             |                    |
| Remesh interval            |      | 64    | 100         |                   |
| Remesh ratio               |      | 0.41  | 0.45        |                   |

*Note: Ours, Adam, and Adam-Remesh were tuned using the five Basic Meshes. Adam-Remesh Complex was tuned using the two Complex Meshes Deer and Smilodon, leading mainly to a much larger remesh interval.*

---

**Figure 12** Overview of the results of all methods after three seconds of optimization. Lower values are better. Models are sorted by complexity from left to right.

(Results shown as Adam-Remesh Complex), leading to a larger remesh interval and much coarser meshes. All tuned parameters are listed in Table 1.

Figure 12 gives an overview of the results after three seconds of optimization. Our method produces the best results for all models. While the differences for the simple first three models are small, our results are substantially better for Lucy and Horse. Without being tuned for the two complex models Deer and Smilodon, our method is still able to reconstruct them well. As expected, Adam-Remesh Complex gives better results for the Complex Models than Adam-Remesh but worse results for the Simple Models.

While the RMS distance gives a first idea about the reconstruction quality, it is important to also investigate the mesh quality. Figure 13 compares the RMS distance, average edge length and face flip ratio of all models and methods. The face flip ratio was calculated by comparing all face normals with the normal of the closest point on the target mesh. Then the area of all flipped triangles was divided by the total mesh area. It is clear to see, that our method converges faster and produces much finer meshes with fewer defects for all models. Without being tuned for Deer and Smilodon, it clearly outperforms the tuned Adam-Remesh Complex for these two models.

Figure 14 shows some intermediate and final meshes for four of the models, colored by per-vertex distance to the target mesh. Again, it is easy to see, that our method has major advantages, mainly for the more complex models. Note that while Adam-Remesh Complex and Ours give similar RMS distance for Smilodon, the reconstruction using Ours has
much finer details, see also the edge length plot in Figure 13. Please also watch our supplemental videos of some complete reconstructions.

### 6.3 Fine to coarse

Another advantage of our method is the ability to take a step back in the coarse-to-fine optimization. Normally the reference edge length $l_{\text{ref}}$ is steadily decreased in Equation (22), but in some cases it may increase. Figure 15 shows an example where a part of the mesh is only connected by a thin wire that requires very small triangles to be reconstructed. After that, increasing the edge length for the missing part greatly speeds up the reconstruction.

### 6.4 Parameter robustness

We analyze the dependency of the overall RMS distance on our parameters $v_{\text{ref}}$ and $G$ in Figure 16. The Basic Models and corresponding parameters were used for evaluation. The gain $G$ is a quite uncritical parameter, bad values typically just
FIGURE 14 Intermediate \((t=1\, s, 2\, s)\) and final \((t=3\, s)\) meshes from the comparison in Section 6.2. Colors indicate per-vertex distance to the target mesh. Please zoom in to see the triangulation. Our method is faster and produces much finer and more accurate meshes with fewer defects for all models. All baseline methods fail to completely reconstruct the antlers of the Deer. Adam-Remesh Complex is able to reconstruct Smilodon's hind legs, but produces very coarse meshes. Please also watch our supplemental videos of complete reconstructions.

slow down the optimization. The reference velocity \(v_{ref}\) is much more critical, but a value of \(v_{ref} = 0.3\) gave good results in all our experiments.

7 CONCLUSION AND OUTLOOK

We presented a novel method that tackles fundamental problems of mesh optimization. We demonstrated that our method is faster for simple models and able to reconstruct complex models where baseline methods completely fail.

The results presented in Section 6 give just a very first idea of what might be possible with our method. The ability to freely optimize the triangulation locally without touching other parts of the mesh is the prerequisite for reconstructing much larger and more complex meshes than shown in this article. A lot of additional research is needed to evaluate how good our method performs when it is applied to real-world problems with physically based shading. Reconstructing
Some ribs of the *Smilodon* are only connected by thin wires. After reducing the edge length and reconstructing the wires, the edge length locally increases to quickly reconstruct the rib. Colors indicate the reference edge length $l_{\text{ref}}$. Please watch our supplemental video. (Optimized with $l_{\text{min}} = 0.005$ and 25 images.)

We calculate the overall RMS distance depending on our hyperparameters $v_{\text{ref}}$ and $G$, while all other parameters are fixed to their optimal value.

Textures and reflection properties can greatly benefit from our cautious vertex updates. On the other hand, it is not clear how to extend our solution to continuously optimized UV mapping.

Applying a remeshing step per iteration is only possible due to the simplicity of our remeshing method that only takes the direct neighborhood of each vertex into account. That also means, that global self-intersections cannot be detected. Furthermore, topology changes would be crucial to be able to reconstruct more complex objects. Both problems require additional global operations that could be combined with our method.

**FUNDING INFORMATION**

Upper Austria, European Regional Development Fund, Investment for Growth and Jobs (IGI/ERDF), Priority 7, REACT-EU, ZDM_OmniScan, Upper Austria, ZeroP, Zero Defect Manufacturing für die nachhaltige Produktion

**ORCID**

Werner Palfinger https://orcid.org/0000-0002-7991-6139

**REFERENCES**

1. O'Riordan A, Newe T, Dooly G, Toal D. Stereo vision sensing: review of existing systems. Proceedings of the 2018 12th International Conference on Sensing Technology (ICST); 2018. p. 178-84. https://doi.org/10.1109/ICSensT.2018.8603605
2. Kingma DP, Ba J. Adam: a method for stochastic optimization. In: Bengio Y, LeCun Y, editors. Proceedings of the 3rd International Conference on Learning Representations, ICLR 2015. San Diego, CA: arXiv; 2015, Conference Track Proceedings (2015). http://arxiv.org/abs/1412.6980

3. Nicolet B, Jacobson A, Jakob W. Large steps in inverse rendering of geometry. ACM Trans Graph (Proc SIGGRAPH Asia). 2021;40(6):1-13. https://doi.org/10.1145/3478513.3480501

4. Cheng Z, Li H, Hartley RI, Zheng Y, Sato I. One ring to rule them all: a simple solution to multi-view 3D-reconstruction of shapes with unknown BRDF via a small recurrent resnet. arxiv abs/2104.05014; 2021

5. Luan F, Zhao S, Bala K, Dong Z. Unified shape and SVBRDF recovery using differentiable Monte Carlo rendering. Comput Graph Forum. 2021;40:1-13.

6. Loper MM, Black MJ. OpenDR: an approximate differentiable renderer. Proceedings of the Computer Vision – ECCV 2014, Lecture Notes in Computer Science. Berlin: Springer International Publishing; Vol. 8695, 2014, p. 154–169. DOI https://doi.org/10.1007/978-3-319-10584-0_11

7. Kato H, Ushiku Y, Harada T. Neural 3D mesh renderer. Proceedings of the 2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT; June 18-22, 2018, p. 3907–3916. Computer Vision Foundation / IEEE Computer Society (2018). https://doi.org/10.1109/CVPR.2018.00411

8. Li S, Chen W, Li T, Li H. Soft rasterizer: differentiable rendering for unsupervised single-view mesh reconstruction. arXiv preprint arXiv:1901.05567, 2019.

9. Ravi N, Reizenstein J, Novotný D, Gordon T, Lo W, Johnson J, et al. Accelerating 3D deep learning with pytorch3d. CoRR abs/2007.08501; 2020. https://arxiv.org/abs/2007.08501

10. Laine S, Hellsten J, Karras T, Seo Y, Lehtinen J, Aila T. Modular primitives for high-performance differentiable rendering. ACM Trans Graph. 2020;39(6):1-14. https://doi.org/10.1145/3414685.3417861

11. Li TM, Aittala M, Durand F, Lehtinen J. Differentiable Monte Carlo ray tracing through edge sampling. ACM Trans Graph. 2018;37(6):1-11. https://doi.org/10.1145/3272127.3275109

12. Nimier-David M, Vicini D, Zeltner T, Jakob W. Mitsuba 2: a retargetable forward and inverse renderer. ACM Trans Graph. 2019;38(6):1-17. https://doi.org/10.1145/3355089.3356498

13. Zhang C, Wu L, Zheng C, Gkioulekas I, Ramamoorthi R, Zhao S. A differential theory of radiative transfer. ACM Trans. Graph. 2019;38(6):1-16. https://doi.org/10.1145/3355089.3356522

14. Nimier-David M, Speierer S, Ruiz B, Jakob W. Radiative backpropagation: an adjoint method for lightning-fast differentiable rendering. Trans Graph (Proc SIGGRAPH). 2020;39(4):146:1–146:15. https://doi.org/10.1145/3386569.3392406

15. Vicini D, Speierer S, Jakob W. Path replay backpropagation: differentiating light paths using constant memory and linear time. Trans Graph (Proc SIGGRAPH). 2021;40(4):108:1–108:14. https://doi.org/10.1145/3450626.3459804

16. Gkioxari G. Mesh r-cnn. ICCV 2019. 9784-9794.

17. Gupta K, Chandraker M. Neural mesh flow: 3D manifold mesh generation via diffeomorphic flows. ArXiv abs/2007.10973, 2020.

18. Hoppe H, DeRose T, Dachau T, McDonald J, Stuetzle W. Mesh optimization. Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH ’93. New York, NY: ACM; 1993. p. 19–26. DOI https://doi.org/10.1145/166117.166119

19. Botsh M, Kobbelt L. A remeshing approach to multiresolution modeling. Proceedings of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing, SGP ’04. Association for Computing Machinery, New York, NY, 2004. p. 185–92. doi https://doi.org/10.1145/1057432.1057457

20. Surazhsky V, Gotsman C. Explicit surface remeshing. In: Kobbelt L, Schroeder P, Hoppe H, editors. Eurographics Symposium on Geometry Processing. Geneva, Switzerland: The Eurographics Association; 2003. https://doi.org/10.2312/SGP/SGP03/020-030

21. Jakob W, Tarini M, Panozzo D, Sorkine-Hornung O. Instant field-aligned meshes. ACM Trans Graph (Proc SIGGRAPH Asia). 2015;34(6):189:1–189:15. https://doi.org/10.1145/2816795.2818078

22. Paszke A., Gross S., Massa F., Lerer A., Bradbury J., Chanan G., et al. Pytorch: an imperative style, high-performance deep learning library Cite arxiv:1912.01703 NeurIPS, 2019. http://arxiv.org/abs/1912.01703

23. Ramamoorthi R, Hanrahan P. An efficient representation for irradiance environment maps. Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH ’01. New York, NY: ACM; 2001. p. 497–500. https://doi.org/10.1145/383259.383317

24. Sellán S. Libigl botsch-kobbelt local remesher; 2022. https://github.com/sgsellan/botsch-kobbelt-remesher-libigl

25. Jacobson A. Common 3D models. Accessed May 06, 2022. https://github.com/alecjacobson/common-3d-test-models

26. Zhou Q, Jacobson A. Thingi10k: a dataset of 10,000 3d-printing models. arXiv preprint arXiv:1605.04797, 2016.

27. Vienna NHM. 3D scan of saber-toothed cat. Accessed May 06, 2022. https://sketchfab.com/NHMWien
AUTHOR BIOGRAPHY

Werner Palfinger is a Senior Researcher in the Machine Vision Department of Profactor GmbH in Austria. He received a Masters's degree in Technical Physics from the Technical University of Vienna (2004). His current research interests are physically-based rendering, 3D-reconstruction and surface inspection.

SUPPORTING INFORMATION

Additional supporting information can be found online in the Supporting Information section at the end of this article.

How to cite this article: Palfinger W. Continuous remeshing for inverse rendering. Comput Anim Virtual Worlds. 2022;33(5):e2101. https://doi.org/10.1002/cav.2101