Locally-Adaptive Level-of-Detail for Hardware-Accelerated Ray Tracing

JACOB HAYDEL, University of Utah, USA
CEM YUKSEL, University of Utah & Roblox, USA
LARRY SEILER

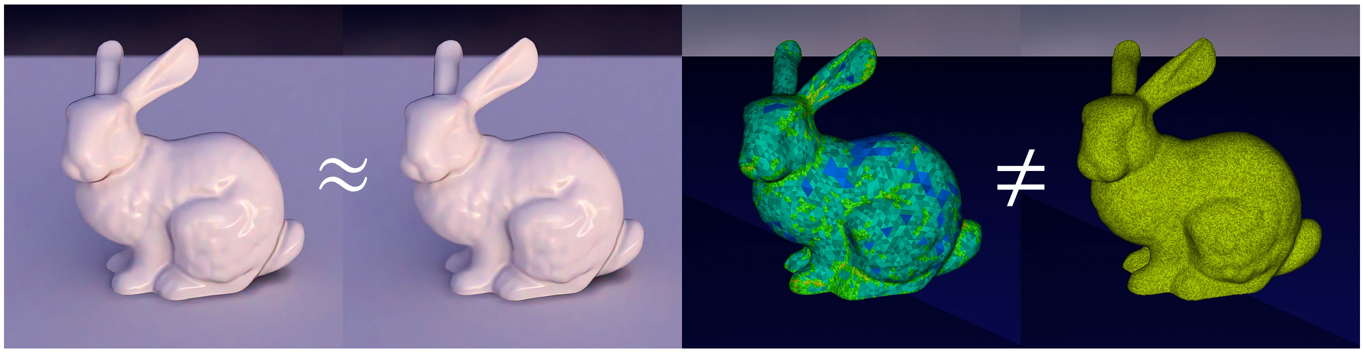


Fig. 1. *Our locally-adaptive level-of-detail for ray tracing renders an image using many fewer triangles than the full-resolution model and produces a visually identical result. The detail level is selected locally for each ray, including both primary and secondary rays, using a screen-space metric. (a) The resulting image is visually indistinguishable from (b) the model rendered at full resolution, even though significantly less data is accessed for rendering a (c) low-resolution mesh, adaptively selected from (d) the full-resolution mesh. Our method provides substantial reductions in data movement that can result in more than an order of magnitude reduction in energy use and render times with sufficient compute resources.*

We introduce an adaptive level-of-detail technique for ray tracing triangle meshes that aims to reduce the memory bandwidth used during ray traversal, which can be the bottleneck for rendering time with large scenes and the primary consumer of energy. We propose a specific data structure for hierarchically representing triangle meshes, allowing localized decisions for the desired mesh resolution per ray. Starting with the lowest-resolution triangle mesh level, higher-resolution levels are generated by tessellating each triangle into four via splitting its edges with arbitrarily-placed vertices. We fit the resulting mesh hierarchy into a specialized acceleration structure to perform on-the-fly tessellation level selection during ray traversal. Our structure reduces both storage cost and data movement during rendering, which are the main consumers of energy. It also allows continuous transitions between detail levels, while locally adjusting the mesh resolution per ray and preserving watertightness. We present how this structure can be used with both primary and secondary rays for reflections and shadows, which can intersect with different tessellation levels, providing consistent results. We also propose specific hardware units to cover the cost of additional compute needed for level-of-detail operations. We evaluate our method using a cycle-accurate simulation of a custom ray tracing hardware architecture. Our results show that, as compared to traditional bounding volume hierarchies, our method can provide more than an order of magnitude reduction in energy use and render time, given sufficient computational resources.

CCS Concepts: • **Computing methodologies** → **Ray tracing**.

Authors' addresses: Jacob Haydel, University of Utah, USA; Cem Yuksel, University of Utah & Roblox, USA; Larry Seiler.

ACM Reference Format:
Jacob Haydel, Cem Yuksel, and Larry Seiler. 2023. Locally-Adaptive Level-of-Detail for Hardware-Accelerated Ray Tracing. *ACM Trans. Graph.* 42, 6, Article 196 (December 2023), 15 pages. https://doi.org/10.1145/3618359

1 INTRODUCTION

Level-of-detail (LOD) is a common technique for reducing render cost by using different resolution versions of an object. Lower-resolution versions are used when the object is further away from the camera and progressively higher-resolutions versions are rendered as the object gets closer. Exactly when and how to switch between LOD levels varies depending on the LOD method.

LOD techniques are commonplace with rasterization, since the cost of geometry processing with rasterization is correlated with the number of triangles rendered. In particular, GPU tessellation shaders and, more recently, mesh shaders offer effective mechanisms for locally determining the mesh resolution on-the-fly at render time, providing substantial savings in computation and data movement costs for rendering.

With ray tracing, acceleration structures provide computation-related benefits of LOD, such that not all triangles are tested for ray intersections when a model is far. Nonetheless, using lower-resolution models is also helpful in ray tracing [Selgrad et al. 2016][Novák and Dachsbacher 2012], particularly for reducing the data movement and the related costs and inefficiencies. Indeed, when ray tracing relatively large scenes with sufficient computational resources, data movement can easily become the bottleneck of performance and it is the main source of energy use on today's computing devices [Vasiou et al. 2018][Ghose et al. 2018].

Unfortunately, there is no effective equivalence of tessellation or mesh shaders with GPU ray-tracing that would reduce data movement. The only LOD options for ray casting are picking an LOD level prior to rendering or stochastically switching between LOD levels per ray [Brandon Lloyd 2020]. The former option results in abrupt changes in the model shape (due to shape differences between LOD levels) without any graceful transition and the latter one leads to noise and increased data movement when multiple levels are sampled during rendering.

In this paper, we propose a locally-adaptive LOD approach that is reminiscent of tessellation shaders. Our goals are the same: reducing the memory bandwidth utilization and achieving locally-adaptive detail levels. Similar to the tessellation pipeline for rasterization, the rendered meshes are defined by a low-resolution mesh (i.e. the coarsest level LOD) and tessellations of its primitives. Unlike the rasterization pipeline, however, we generate the highest-resolution tessellation ahead of rendering, so that we can build a customized acceleration structure, optimized for storage and cache coherency.

Our tessellation structure is designed for reducing data storage and minimizing data movement during rendering. Moreover, it permits continuous transitions between LOD levels and picking the desired LOD level independently for each ray, including secondary rays, without introducing cracks or self-occlusion artifacts. We rely on a pre-defined tessellation structure: each triangle of a lower-resolution level is tessellated into 4 triangles by splitting its edges. The common vertices of consecutive LOD levels maintain their positions for all LOD levels and the other vertices are placed freely. Once built, our tessellation structure can be quickly refit to support animating meshes.

Obviously, handling LOD at render time involves additional computation. In this paper, we target hardware-accelerated ray-tracing with custom hardware logic to perform ray traversal with our LOD operations, which are significantly simpler and more efficient to handle in hardware than a software implementation, allowing us to explore techniques that would not be feasible on existing GPU hardware without our custom units and a fixed traversal pipeline.

We evaluate our method using a cycle-accurate simulation of a custom ray tracing hardware architecture with sufficient compute resources and hardware ray traversal units. Our results show that, depending on the camera distance, our method can achieve more than an order of magnitude reduction in both energy cost and render time, as compared to traditional bounding volume hierarchies. These improvements come at no visible degradation in quality, as shown in Figure 1.

2 RELATED WORK

Currently, tessellation on ray-tracing hardware is handled by pre-tessellating the model and then storing the full-resolution version in memory [Sjoholm 2018]. This model is then rendered at full resolution using a conventional bounding volume hierarchy (BVH) and standard traversal techniques. This has the benefit of fully leveraging the hardware during rendering by determining the desired tessellation level prior to rendering. On the other hand, it does not provide local adaptivity and switching between LOD levels on consecutive frames results in abrupt changes in the model shape. This issue is similar to typical LOD approaches with rasterization without using on-the-fly tessellation.

Stochastic LOD [Brandon Lloyd 2020] works by pre-tessellating the model to form a number of LOD levels. During render time, rays stochastically pick an LOD level to achieve graceful transitions between levels. Unfortunately, such transitions not only introduce noise but also increase the memory bandwidth use, since LOD levels do not share BVH or mesh data and multiple LOD levels are accessed during rendering, instead of just one.

Several other approaches to LOD in ray tracing have been explored by prior works. Lee et al. [2019] proposes an extension to modern ray tracing APIs that would allow for the selection of which acceleration structure and corresponding level of detail should be intersected on the fly at traversal time. The proposed extension would support stochastic LOD. Kulkarni et al. [2019] explores a technique where BVH nodes are shared between multiple LOD levels of a single mesh. This amortizes the cost of switching between levels of detail as some of the nodes will already be cached. Ikeda et al. [2022] proposes a method that uses the bounding boxes of interior BVH nodes as a proxy for the triangle mesh, intersecting them directly, instead of traversing the BVH nodes down to the leaf and intersecting the full-resolution triangles. Similar to our method, this has the effect of reducing memory bandwidth but is only suitable for incoherent rays like those used to approximate Global Illumination, as the BVH nodes do not provide a good approximation of the model when it is directly viewed.

LOD for ray tracing was also explored in the context of software rendering, particularly for high-resolution scenes. Christensen et al. [2003] picks a discrete LOD level for a subpatch to achieve hybridized ray tracing with REYES. Cracks are avoided by moving vertices along subpatch boundaries or inserting gap-filling polygons [Christensen et al. 2006]. Yoon et al. [2006] renders massive models by picking a discrete LOD level per ray, but does not guarantee surface continuity. Hanika et al. [2010] tessellates patches into micropolygons of different LOD levels, relying on conservative bounding boxes and using the boundaries of the nodes to fill cracks on the surface due to varying LOD decisions. The Razor rendering system [Djeu et al. 2011] generates the ray tracing acceleration structure on-the-fly during rendering from a given scene graph. It supports dynamic LOD and geometrically morphing between levels to avoid surface cracks, using a similar approach to ours. However, it is high overhead makes it unsuitable for primary rays and simple secondary rays, such as shadows.

On-the-fly tessellation is an alternative approach for reducing memory use with ray tracing. One option is using displacement maps [Smits et al. 2000]. This avoids the added memory overhead of storing the full-resolution model but requires the computation cost of generating the sub-triangle mesh from a displacement map each time the mesh is rendered. Also, tearing artifacts appear when neighboring pixels disagree on the LOD level due to discontinuities in the triangle edges produced from the displacement map.

Another option for on-the-fly tessellation with ray tracing is using a tessellation cache shared between all threads [Benthin et al. 2015]. This has the effect of reducing the memory storage requirement of a given scene in that the large amount of data required for patches is confined to a certain section of cache. Although patches may be evicted from cache, they can be reconstructed relatively quickly. This means that only a small subset of the full-resolution scene needs to be stored at any given time. This can provide a reduction in render time over full-resolution ray-tracing in software on the CPU. However, this technique requires thread synchronization as well as rebuilding each model's acceleration structure for every frame. Those operations are costly to implement in hardware, so this technique is more suitable for software ray tracing than for selecting detail levels using dedicated ray tracing hardware.

Recently, Thonat et al. [2021] introduced a method for adding fine-scale displacements on a coarse model. The intersections with the displaced surface are computed using the displacement texture without tessellating the coarse model. This offers a highly effective solution for tiled displacement maps on a larger surface, but it is not suitable for handling arbitrary geometric detail.

Other works have looked into generating meshes with the topology of Loop subdivision [Loop 1987] from arbitrary topology meshes. Lee et al. [2000] uses edge collapses and global optimization to produce a control mesh and then approximate the original surface as a displacement of the limit surface in the normal direction.

Finally, Micro-Meshes [NVIDIA 2023] are similar to our work in that they store up to 5 levels deep of subdivided triangles in a regular data structure that allows selecting different subdivision levels for different parts of the mesh. Unlike our method, micro-meshes do not perform locally adaptive tessellation but instead require an external computation to determine which detail levels to use.

3 LOCALLY-ADAPTIVE LEVEL OF DETAIL

Our locally-adaptive level-of-detail approach is designed to reduce data movement during ray traversal, reminiscent of the GPU tessellation pipeline for rasterization. Unlike rasterization, however, our method requires pre-generating the full-resolution model and building an acceleration structure for it ahead of rendering.

In this section, we describe the details of our method beginning with our multi-resolution model representation (Section 3.1). Then, we present our acceleration structure that combines all mesh resolutions (Section 3.2). An important component of this acceleration structure is the *displacement bound* that represents the geometric differences between our tessellation levels (Section 3.3). We explain how our ray traversal works (Section 3.4) and how we guarantee watertight ray intersections (Section 3.5). Next, we describe the *hit information* we generate when a ray intersection is found with any tessellation level (Section 3.6) and the LOD metric we use in our implementation (Section 3.7). Finally, we present how our method can be extended to secondary rays (Section 3.8).

3.1 Multi-Resolution Mesh Representation

Triangles are the most common primitives with ray tracing (and with rendering in general). Therefore, we build our multi-resolution model representation on triangle meshes.

Our triangle mesh hierarchy contains a number of levels, each with a triangle mesh of a different resolution. To ensure that we can seamlessly transition between these levels, the topologies of these meshes are tightly coupled. Our *tessellation rule* is such that each triangle of a coarser level corresponds to 4 triangles of the next

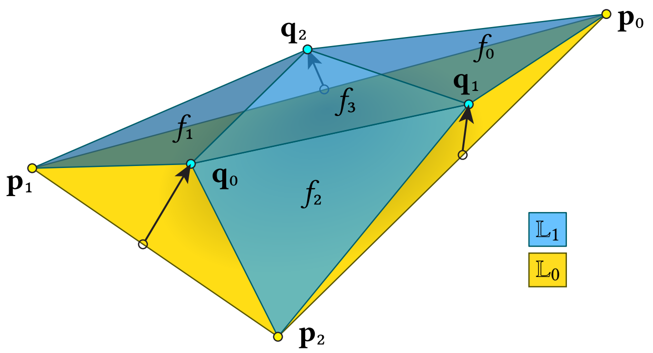


Fig. 2. *Example of the vertex and face ordering of a sub group. The tessellation tree node at $\mathbb{L}_0$ corresponds to the yellow triangle with vertices $\mathbf{p}_0$, $\mathbf{p}_1$, and $\mathbf{p}_2$. Its vertex data stores $\mathbf{q}_0$, $\mathbf{q}_1$, and $\mathbf{q}_2$.*

resolution, formed by splitting the three edges of the triangle, as shown in Figure 2.

Topologically, this matches the tessellation process of Loop subdivision [Loop 1987], but we do not follow the vertex placement rules of it or any other subdivision scheme. The vertices can be positioned arbitrarily.

The three *common vertices* of a triangle and its corresponding 4 triangles at the next level ($\mathbf{p}_0$, $\mathbf{p}_1$, and $\mathbf{p}_2$ in Figure 2) maintain the same positions for both levels, instead of having different positions at each level. This restriction on vertex placement between levels not only provides savings in the overall storage cost, but it is also at the core of our adaptive ray traversal process for deciding when to use higher-resolution levels during rendering.

The coarsest level of detail, $\mathbb{L}_0$, is a triangle mesh with an arbitrary topology. The consecutive levels are formed by tessellating all triangles of the previous level (as shown in Figure 2). Since the common vertices maintain positions, we only need to store the *inserted vertices* of the next level ($\mathbf{q}_0$, $\mathbf{q}_1$, and $\mathbf{q}_2$ in Figure 2).

Using this well-defined ordering, we can easily construct the triangles (i.e. determine each triangle's vertices) of the next level from the previous one. Therefore, we only need to store the triangles of the lowest-resolution mesh and the vertex positions of all resolutions. The triangles of all higher-resolution levels can be constructed during ray traversal without explicitly storing their topology information.

This structure also allows us to continuously transition between levels. Since each inserted vertex corresponds to an edge center of the coarser level, we can easily morph the mesh to take the *exact* shape of the coarser mesh by simply moving each inserted vertex towards the corresponding edge center of the coarser level. This way, we completely avoid abrupt transitions between levels.

3.2 Acceleration Structure

We can use any acceleration structure for $\mathbb{L}_0$. The only change is that the bounding box of each triangle in $\mathbb{L}_0$ must include its tessellated triangles in all subsequent levels.

We use a specialized *tessellation tree* for representing the hierarchy under each triangle in $\mathbb{L}_0$. Thus, we build as many tessellation trees as the number of triangles in $\mathbb{L}_0$.

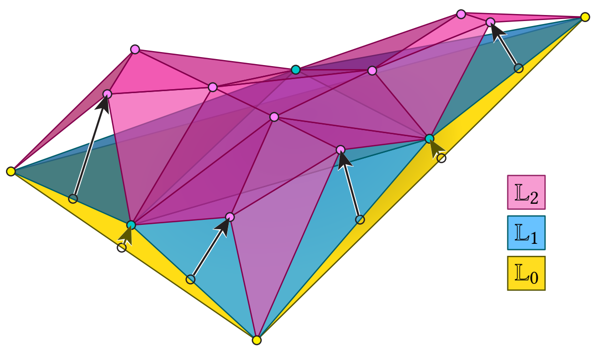


Fig. 3. *An example triangle at $\mathbb{L}_0$ and its tessellated triangles at $\mathbb{L}_1$ and $\mathbb{L}_2$. Notice that the magnitude of the displacement for $\mathbb{L}_2$ is significantly larger than the one for $\mathbb{L}_1$.*

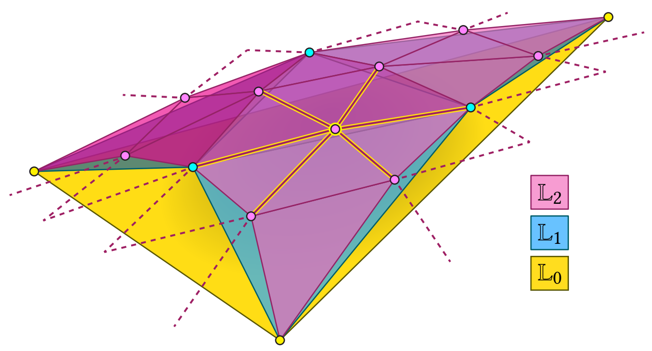


Fig. 4. *A highlighted inserted vertex and its 6 edges. The dashed lines show the edges that belong to neighboring tessellation trees. Notice that inserted vertices along the perimeter of the tessellation tree also have 6 edges, two of which belong to a neighboring tessellation tree and another two are shared by the two tessellation trees.*

Each node of the tessellation tree corresponds to one triangle at a tessellation level. The 4 triangles formed by tessellating this triangle (for the next tessellation level) are associated with its 4 child nodes. However, we do not directly store the triangle information per node. Instead, we store two separate types of information per node:

(1) **Bound data** that contains a bounding box and *displacement bounds* that represent the maximum displacement of the surface near each edge of the node's triangle, as we explain below in Section 3.3. Our LOD decisions during ray traversal rely on this displacement bound. The bounding box contains all triangles in subsequent tessellation levels, which is equivalent to the bounding box of the part of the full-resolution mesh that corresponds to the node.

(2) **Vertex data** that contains the positions of the inserted vertices ($\mathbf{q}_0$, $\mathbf{q}_1$, and $\mathbf{q}_2$ in Figure 2) for the next level.

Note that a tessellation tree node does not store the vertices of its triangle, but the vertices that are needed to form the triangles of its child nodes. Thus, the vertex data is needed only if we decide to traverse deeper into the tessellation tree. Therefore, for minimizing data movement during traversal, we store the bound data and the vertex data separately, so that we can skip reading the vertex data when it is not needed. This is a critical property of our structure that allows minimizing data movement during ray traversal.

The triangle mesh in $\mathbb{L}_0$ can be stored using a typical data structure, where each triangle is represented using 3 vertex indices into a vertex buffer that stores the vertex positions. For the subsequent levels, however, we avoid using an index buffer. Vertex data only contains the positions of the inserted vertices. The mesh connectivity is implicitly defined, so there is no need to store it. We describe the details of how our data structure allows us to quickly find the inserted vertices of a triangle without storing any vertex indices for levels beyond $\mathbb{L}_0$ in Section 4.

We store our tessellation trees as 4-ary BVHs (with each internal node having 4 child nodes) in the form of perfect (i.e. full and complete) trees. This allows a compact representation that does not require storing child node indices, thereby avoiding indirection and reducing data movement.

3.3 Displacement Bound

During ray traversal, the decision of which tessellation level to use for intersecting triangles is made independently for each ray. This is an important property of our approach, allowing localized LOD decisions, which can be different for neighboring rays. Yet, it is crucial that these independent decisions are consistent across all edges for all tessellation levels, since picking different levels on either side of an edge would result in visible cracks on the surface. To ensure consistency, the LOD metric we use is computed per edge.

Our LOD metric (described in Section 3.7) is based on the *maximum displacement* of the surface along and around an edge between all subsequent tessellation levels. We represent it using a single scalar value per edge: the *displacement bound*. Thus, the bound data of our tessellation tree includes 3 scalar displacement bounds per node, one for each edge of the node's triangle.

It is important to note that we cannot simply use the immediate displacement at the current tessellation level (i.e. the distance of an inserted vertex to the corresponding edge center) as the displacement bound. This is because the displacement of an edge at a level (exactly at its center) does not necessarily bound its displacements at the subsequent levels (at different locations along the edge). An example of this is shown in Figure 3, where the displacements of a subsequent level are more significant than its parent level.

To capture the displacement bound of all subsequent levels, we compute the displacement bound bottom-up, starting with the full-resolution mesh at the last level of our hierarchy $\mathbb{L}_N$ with $N + 1$ levels. The displacement of an edge at its parent level $\mathbb{L}_{N-1}$ can be properly bounded by the distance of the inserted vertex from the edge center. The leaf nodes of our tessellation trees store these values as their displacement bounds. As we go up the hierarchy, each edge at level n with $0 \le n < N$ similarly considers the distance of its center from its inserted vertex at $\mathbb{L}_{n+1}$. In addition, it also considers the displacement bounds of all 6 edges of the inserted vertex at $\mathbb{L}_{n+1}$. The maximum value among them is set as the displacement bound of the edge at $\mathbb{L}_n$.

Note that with our tessellation rule, all inserted vertices are valence 6, i.e. connected to 6 edges (see Figure 4), unless they are on the mesh boundary. For inserted vertices along the tessellation tree boundaries, two of their edges belong to one tessellation tree, another two belong to the neighboring tessellation tree, and the final two are shared between the two tessellation trees. That is why we build all tessellation trees of the mesh hierarchy together.

3.4 Ray Traversal

Our ray traversal begins with the acceleration structure of $\mathbb{L}_0$ down to the bounding boxes of the tessellation trees stored in its leaf nodes. When a ray intersects with the bounding box of a tessellation tree's root node, we begin our LOD process.

For each internal node of a tessellation tree, our LOD metric (described in Section 3.7) determines whether the triangle of the node would be sufficient or if we should traverse into the next level. This decision uses the vertex positions at the current level and the displacement bounds in the bound data. If the current level provides sufficient detail, we perform ray-triangle intersection using the triangle of the node and simply ignore the rest of the tessellation tree below it. Otherwise, we traverse further down into the tessellation tree by intersecting the ray with the bounding boxes of the 4 child nodes at the next level.

Since our LOD decisions are handled separately for each edge, the three edges of a node's triangle may disagree on the tessellation level to be used for the ray. If any one of the three edges requires a higher tessellation level, we must traverse further down into the tessellation tree.

Yet, in the case of any disagreement between the tessellation decisions of the three edges, we cannot simply ignore the edges that did not require further tessellation. As we go down into the next level, those edges that did not require tessellation must maintain their geometries. This is because the neighboring triangles on the other sides of these edges may agree to keep the current tessellation level, so blindly tessellating these edges without preserving their geometries can lead to cracks on the surface. Our solution is to tessellate these edges without applying any displacement. Thus, instead of using the corresponding inserted vertex position stored in the vertex data, we simply use the edge center position for such inserted vertices.

Another possibility is that our LOD metric (described in Section 3.7) for an edge may return an intermediate tessellation value. In that case, we apply a portion of the displacement by placing the inserted vertex somewhere between the inserted vertex position in the vertex data and the corresponding edge center at the current level. When we tessellate an edge without applying any displacement or a portion of the full displacement, we effectively morph the next tessellation level towards the current one. Like geomorphingHoppe [1998] this eliminates temporal artifacts such as popping. After morphing, we continue down the tessellation tree using these inserted vertices with morphed positions.

In addition to morphing the inserted vertex positions, we must also adjust the bounding boxes of the child nodes. This is because these morphed positions may be outside of the original bounding boxes. One might incorrectly assume that modifying these bounding boxes during ray traversal could be avoided by initially building conservative bounding boxes that include the edge centers at the

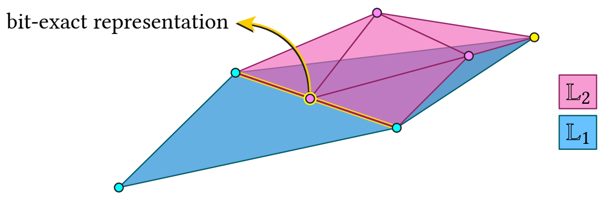


Fig. 5. *Watertight rendering requires bit-exact representation of edge midpoints when one side of the highlighted edge is tessellated and the other side is not.*

previous level. However, this would only account for morphing at one tessellation level. Similar morphing operations may need to be performed at subsequent levels, where the edges might be formed by previously-morphed vertices. Therefore, the alternative of initially building conservative bounding boxes would require considering compound morphing at multiple tessellation levels. Furthermore, such pre-computed conservative bounding boxes would be less tight and, therefore, be less efficient.

As we traverse down the tessellation tree, similar to typical ray tracing, child nodes with bounding boxes that intersect with the ray are placed on a stack. Our stack, however, contains more information, as it also stores the inserted vertex positions (original or morphed). This increase in stack data avoids reading the same vertex position information multiple times, as we pop nodes from the stack. Therefore, it is important for reducing data movement, at the cost of some increase in local storage during ray traversal.

When we reach a leaf node of the tessellation tree, if our LOD metric determines that further detail is needed, we simply read the vertex data for the leaf node, construct all 4 triangles that correspond to the finest tessellation level, and intersect the ray with them. Otherwise, we can skip reading the vertex data and intersect with the one triangle of the leaf node.

3.5 Watertight Tessellation

When applying adaptive tessellation like ours, an immediate challenge is ensuring that the tessellated surface remains watertight. The programmable tessellation shaders of the GPU rasterization pipeline circumvent this challenge by independently specifying the tessellation levels of patch edges and its interior. This can create awkward tessellation patterns, but the resulting surface remains watertight, as long as the shader generates the same edge tessellation level for its two patches and places the tessellated vertices along the edge using the same exact computation.

In our method, however, we cannot completely rely on the same strategy, as our tessellation levels are defined hierarchically and an edge may be tessellated (i.e. the ray may be traversed down to a level with more tessellation) even when the LOD metric (described in Section 3.7) indicates no further tessellation for the edge. This happens when the other edges of a triangle disagree on the tessellation level, as explained above in Section 3.4. As a result, a triangle on one side of an edge may get tessellated while the triangle on the other side of the edge is not, as shown in Figure 5.

Fortunately, our morphing operation geometrically solves this problem by ensuring that, if an edge is tessellated against its decision,

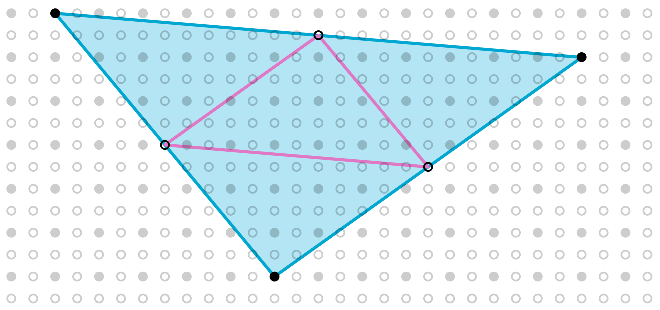


Fig. 6. *Watertight tessellation requires exact representation of edge centers. Dark circles represent possible vertex positions at the lower-resolution level, such that the edge centers are guaranteed to have exact representations on the higher resolution grid.*

the inserted vertex is placed at the edge center. If it were not for the perils of limited numerical precision, this would have been sufficient to ensure watertight tessellation. Unfortunately, using floating-point numbers the computed edge center may not be possible to represent exactly with the available precision, which could result in visible cracks on the rendered surface. Increasing the precision only makes the problem less frequent, but it does not eliminate it.

We solve this problem by ensuring that the edge centers can be represented *exactly* with the numerical precision we use. First, we use a fixed-point representation (i.e. an integer with a global scaling factor and offset per model) for all vertex position values. The scale factor and offset are selected so that the model fits in the range $(-1, +1)$, as described in Section 4.1. To ensure that the midpoint of two vertices can be represented exactly, we just need to make sure that the vertex position values are even integers (with their lowest-precision bits set to zero), since the average of two even numbers is also an integer. This is sufficient for just one tessellation level and the resulting midpoint does not have to be formed by even numbers. To solve this problem for a second level, we need to set the second lowest-precision bit to zero as well.

More generally, for a tessellation tree with $N + 1$ levels, such that $\mathbb{L}_N$ is the full-resolution level, vertex positions at level n are represented with $N - n$ lowest-precision bits set to zero.

It is important to note that the $N - n$ lowest precision bits must also be zero for vertex positions that are computed via morphing. This is because these computed vertex positions are also used when computing the midpoints of edges at the next level.

Guaranteeing that every edge center at intermediate tessellation levels has a bit-exact representation ensures that the edge directions we compute with or without splitting the edge are identical in our fixed-point representation. Figure 6 illustrates the result of splitting edges at the center. Note that all edge centers are on the finer grid used for the next detail level.

In addition, we perform ray-triangle intersections using fixed-point arithmetic. Thus, we can guarantee no truncation (i.e. precision loss) during ray-triangle intersection tests, ensuring that we get consistent results on either side of an edge, regardless of its tessellation on one side. We provide the details of our ray-triangle intersection test and how it guarantees watertightness in Appendix A. The computations we use require only integer arithmetic, using up to 96-bit fixed-point operations, which are much cheaper than 64-bit floating-point addition, since that requires a barrel rotator to align the mantisssas.

3.6 Hit Information

After we find the intersection of a ray with a triangle, we must return the *hit information*, that is, the information needed for shading the intersection point. This is a trivial process when intersecting with the triangles of the finest tessellation level. For triangles of the intermediate LOD levels, however, we must ensure that the hit information we return is consistent with the full-resolution model.

One of the most important components of hit information is the *shading normal*, the surface normal to be used for shading. Often times, shading normal is different from the geometric normal of the intersected triangle, though it is typically precomputed using the geometric normals.

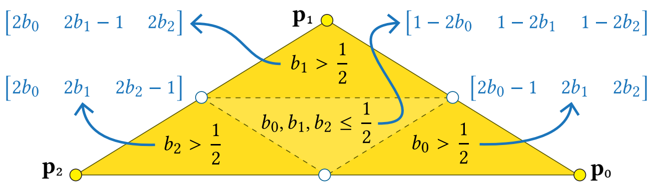
When a ray intersects with a triangle of an intermediate level, we have two options for handling the shading normal: we can return a shading normal at the intersected level or the corresponding shading normal of the full-resolution model.

We favor the latter option, because computing the shading normals of an intermediate level has some important problems. First of all, the shading normal per vertex must be computed and stored independently for each level. Even though we use the same position for a vertex at all tessellation levels, we cannot simply use the surface normal of the vertex computed at the finest level on a lower-resolution tessellation of the model. This is because when these finest surface normals are interpolated on a coarse triangle, the resulting shading normal can be substantially different from the shading normal of the corresponding surface position of the full-resolution model. In fact, even if we compute and store separate shading normals for the intermediate levels, the interpolated shading normal on a coarse triangle can still be substantially different from the full-resolution model.

In addition, besides the storage cost of these additional surface normals for the intermediate levels, they also introduce computational complexity at render time: since we morph between tessellation levels, we would also need to morph the shading normals. Note that morphing the normals is more complicated than morphing the vertex positions, because they involve combining different LOD metrics computed separately for all three vertices of the intersected triangle. Morphing the surface normal can also be problematic with secondary reflection rays, the direction of which can change significantly as the shading normal changes.

Because of these problems, regardless of the tessellation level we use for the final ray-triangle intersection, in our hit information we always return the corresponding shading normal of the full-resolution mesh. This requires quickly finding the corresponding triangle of the full-resolution model without having to traverse down to the leaf nodes of our tessellation trees.

Fortunately, this is a simple process with our tessellation rule. Using the barycentric coordinates of the hit point at the current level, we can quickly determine which one of the four triangles at the next level correspond this hit point and calculate the barycentric



$\mathbf{p}_1$
$\mathbf{p}_2$
$\mathbf{p}_0$
$\begin{bmatrix}2b_0 & 2b_1-1 & 2b_2\end{bmatrix}$
$\begin{bmatrix}1-2b_0 & 1-2b_1 & 1-2b_2\end{bmatrix}$
$\begin{bmatrix}2b_0 & 2b_1 & 2b_2-1\end{bmatrix}$
$\begin{bmatrix}2b_0-1 & 2b_1 & 2b_2\end{bmatrix}$
$b_1 > \frac{1}{2}$
$b_0, b_1, b_2 \leq \frac{1}{2}$
$b_2 > \frac{1}{2}$
$b_0 > \frac{1}{2}$

Fig. 7. *Computation of the barycentric coordinates at the next level for each of the 4 sub-triangles, given a hit point* $\mathbf{x}$ *with barycentric coordinates* $\begin{bmatrix}b_0 & b_1 & b_2\end{bmatrix}$*, such that* $\mathbf{x} = b_0\mathbf{p}_0 + b_1\mathbf{p}_1 + b_2\mathbf{p}_2$*. In corner triangles, inserted vertices replace two triangle vertices. In the central triangle, each inserted vertex replaces the opposite triangle vertex.*

coordinates for that triangle, as shown in Figure 7. We repeat this process until we reach the full-resolution level. Since we are not computing new hit points for the preceding levels, we do not need to read vertex positions of those levels or any other data. The corresponding triangle of the full-resolution mesh and its barycentric coordinates can be computed directly using these simple operations.

An additional benefit of finding the triangle of the corresponding full-resolution model and its barycentric coordinates for the hit point is that we can gather any other shading-related information we need directly from the full-resolution mesh. For example, texture mapping seams do not need to respect the edges of the coarsest level $\mathbb{L}_0$ and such seams can safely pass through the triangles of $\mathbb{L}_0$ without introducing any problems during rendering. Even shading operations that rely on the triangle index of the full-resolution model are compatible with our solution.

In short, the hit information we return corresponds to a position on the full-resolution model, even if the exact hit position might be (and often is) different and the ray may not intersect with the corresponding hit position on the full-resolution model. Nonetheless, we achieve consistent rendering/shading across all pixels of an image and between multiple frames of an animation with varying LOD decisions.

3.7 Level of Detail Metric

Our tessellation approach can be used with any LOD metric, as long as it can be computed based on the maximum displacement we store per edge. Here, we present the LOD metric we use in our tests.

For our LOD decision, we use a screen-space metric based on the displacement bound stored for each edge. We use ray cones to convert world-space distances at this reference plane to screen-space quantities.

To evaluate a displacement bound we first compute the vectors between the ray origin $\mathbf{x}$ and the two vertices $\mathbf{p}_0$ and $\mathbf{p}_1$ that form the corresponding edge. We then project these vectors onto the ray direction $\mathbf{d}$ and select the minimum distance ℓ

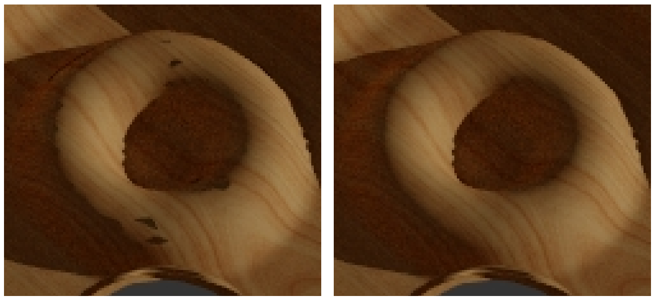
$$\ell = \min\big((\mathbf{p}_0 - \mathbf{x}) \cdot \mathbf{d}\, , (\mathbf{p}_1 - \mathbf{x}) \cdot \mathbf{d}\big) \ . \tag{1}$$

This distance ℓ along the ray is used to conservatively approximate the ray cone radius r near the edge, such that $r = \ell \tan\theta$, where θ is the ray-cone half angle. It is important that this is conservative, as we cannot terminate traversal unless we are sure that nothing in the sub-tree needs tessellation. The screen-space displacement h is then approximated using the corresponding displacement bound $h_{\max}$, such that $h = h_{\max}/2r$. Finally, we compute an *edge state* s with a user-defined parameter δ that controls the LOD quality, using

$$s = \text{clamp}\,(h\,\delta - 1, 0, 1)\ . \tag{2}$$

We use $\delta = 1$ in all examples in this paper, which corresponds to an offset of 1 pixel size in screen space.

When $s = 0$ for all three edges of a tessellation tree node, we can use the coarser level. For $s > 0$, we traverse into the next level by moving the vertex positions $\mathbf{q}_i$ at the next level toward the edge center by a factor of $1 - s$.



(a) *Independent secondary LOD* **(b)** *Dependent secondary LOD*

Fig. 8. *Computing shadows with secondary rays: (a) when secondary rays independently compute their LOD levels, incorrect self-occlusion artifacts appear. (b) Using the LOD decisions of the primary within its influence region fixes such artifacts.*

3.8 Secondary Rays

The LOD metric we presented above works for secondary rays as well. However, when a secondary ray originates on a surface with our adaptive tessellation, we cannot purely rely on the ray cone of the secondary ray. This is because the LOD metric for the secondary ray can easily be different than the one for the primary ray. Therefore, purely relying on the secondary ray for the tessellation decisions may result in inconsistent LOD decisions with the primary ray, causing the secondary ray to originate at a point that may not be on the surface (for the primary ray), intersecting with a different surface and producing incorrect self-intersections. Figure 8 demonstrates an example of this problem.

We solve this by including the primary ray in the LOD decisions for the secondary ray. More specifically, during traversal of the secondary ray, we check if we are traversing the same tessellation tree as the primary ray or a *neighboring tessellation tree*, i.e. one that shares an edge with it at $\mathbb{L}_0$. If it is the same tessellation tree, we use the primary ray's LOD decisions. If it is a neighboring tessellation tree, we use the primary ray's LOD decisions only for the shared edge. This way, we avoid intersecting with an inconsistent surface, the primary and secondary rays agree on the point where the secondary ray originates, and we transition from the primary ray's LOD decisions to the secondary ray's LOD decisions without introducing cracks on the surface. The primary ray's *influence region* only covers the tessellation tree that contains the primary hit and the edges of the neighboring three tessellation trees.

This solution can be applied to the next ray further along the camera path as well. However, simply considering the immediately preceding ray for the LOD decisions is not sufficient to guarantee consistent LOD decisions. This is because the LOD decisions when traversing the immediately preceding ray may have been impacted by a previous ray. Therefore, we must consider the prior rays that contributed to the LOD decisions made for finding the last hit point, where this final ray originates.

Within the same tessellation tree of the last hit point, we use the same set of prior rays for the LOD decisions as the immediately previous ray, overriding the current ray's LOD decisions. When we traverse into a neighboring tessellation tree, we can begin using the current ray's LOD decisions, except for the shared edge between the tessellation trees of the previous hit point and the current neighboring tree. For this, we only need to consider the one prior ray that determined the LOD decisions for the shared edge between the tessellation trees. Thus, the LOD decisions of a ray can be overridden by up to 2 prior rays: the one that determined the LOD level for the tessellation tree of the previous hit and the one that determined the LOD level of the edge (if any) between the currently traversed tessellation tree and the tessellation tree of the previous hit. Note that these two ray may or may not be the same ray, but they may not be the immediately 2 previous rays.

4 IMPLEMENTATION DETAILS

The LOD method we describe above can be implemented in various ways. An efficient implementation, however, should not only consider the computation and storage cost, but also minimize data movement. In this section, we present the details of our implementation and suggestions for further improvements.

4.1 Number Format

We use a 32-bit signed fixed-point representation in the processor registers and temporary local storage in the traversal stack. All operations with vector components are performed in fixed point, using additional bits internally to fully preserve precision (up to 96 bits for ray-edge tests) and returning truncated results in 32 bits.

For data storage in our tessellation trees, however, using a 28-bit fixed-point format provides sufficient precision to represent mesh vertices. This is because, within the range $(-1, 1)$ 28-bit fixed-point provides even better precision than the standard 32-bit floating-point format (IEEE 754), which only has 24 bits of fractional precision (23 mantissa bits plus the hidden bit) within $\pm[1/2, 1)$. Though the floating-point format offers better precision for values closer to zero, e.g. 3 extra bits of precision within $\pm[1/16, 1/8)$, there is little value in getting more precision for a fixed, small part of the model space. The values stored as 28-bit fixed-point are converted to 32-bit fixed-point by simply padding zeros for the additional 4 bits.

The advantage of the 28-bit fixed-point representation is that it allows packing 9 values into a single typical cache line size of 32 bytes (with 4 extra bits to spare). This allows fitting both bound data and vertex data of our tessellation trees in single cache lines, as we explain below. This is used for minimizing data movement, since data is moved (from DRAM and between caches) in chunks of cache line size. [1].

[1]Even with today's CPUs and GPUs that use larger cache lines (64B or 128B) it is common to move data in chunks of 32B, using partially-filled cache lines.

Note that, using N levels, our watertight tessellation requires setting up to N lower-precision bits as zeros (see Section 3.5). 4 of these bits are automatically set to zero due to the 28-bit representation. For $N \leq 8$, which corresponds to each triangle in $\mathbb{L}_0$ to tessellate up to $4^8 = 65,536$ triangles at $\mathbb{L}_N$, we suffer no effective precision loss as compared to 32-bit floating-point representation with 24 bits of precision within $\pm[1/2, 1)$. With deeper tessellation trees, only the vertices at the lower-resolution levels have any precision loss. All our experiments (Section 6) use $N \leq 8$.

Ray-triangle intersections with fixed-point numbers involve representing the ray origin and direction in fixed-point as well. This is handled by performing the computations in model space. Rays are generated using floating-point numbers and converted to fixed-point numbers for computing ray-triangle intersections. This involves moving the ray origin to the first intersection point of the ray with the model's bounding box, since the fixed-point representation only spans the bounding box of the model. Because the intersections are computed in model space, we can easily handle instanced models with different transformations by simply transforming the rays differently for each instance. Note that the conversion to fixed point after transformation is exact (i.e. introduces no truncation error), except for the range close to zero, $(-1/256, 1/256)$ for 32-bit fixed point. In the range $(-1/256, 1/256)$ the model space error introduced by fixed point conversion is bounded by $\pm 2^{-32}$, i.e. a single bit in fixed point.

The hit position of a ray on a triangle may not be exactly represented in either fixed point or floating point. This leads to the well-known problem of self-occlusion with secondary rays that originate at the previous hit position. In fixed point, the error is bounded by a single bit. This corresponds to an error of $\pm 2^{-32}$ in model space. However, the conversion of the hit point to world space in floating point (a typical process for a ray tracing API) can incur a much larger error, thereby eliminating the precision advantage of the fixed-point representation.

4.2 Vertex Data Storage

The vertex data includes 3 inserted vertex positions (i.e. 9 values) per node. Therefore, using the 28-bit fixed-point format, we can easily store the vertex data for each node into a single 32-byte cache line. This provides the optimal solution for minimizing data movement.

However, this results in duplicated storage of inserted vertex positions. Since each inserted vertex along an edge is shared by two triangles on either side of the edge, we end up storing each inserted vertex twice, once for each of the two nodes that use it.

This duplication cost could be (mostly) eliminated with a more elaborate storage scheme, which reduces the storage cost, but increases data movement instead. This is because without duplicate storage of inserted vertices we cannot fit the vertex data of *all* nodes into single cache lines. Thus, the solutions we tested for avoiding this duplicated storage ended up significantly increasing the data movement. Since data movement is much more important than storage cost and our representation already significantly reduces the storage cost even with this duplication (as compared to a typical storage of the full-resolution mesh alone using vertex indices), we recommend duplicated storage of vertex data.

4.3 Bound Data Storage

The bound data for each tessellation tree node includes the bounding box (6 values) and the displacement bounds of the three edges of the node's triangle. Thus, using our 28-bit fixed-point representation, we can pack these 9 values into a single 32-byte cache line.

It is important to note that during traversal the node data for all 4 child nodes of a node are accessed simultaneously. Thus, packing the bound data for 4 sibling nodes into less than 4 (i.e. 2 or 3) cache lines would certainly reduce the data movement. Furthermore, the bound data for the 4 sibling nodes contain shared information: the displacements bounds of the 3 internal edges (i.e. edges $\overline{\mathbf{q}_0\mathbf{q}_1}$, $\overline{\mathbf{q}_1\mathbf{q}_2}$, and $\overline{\mathbf{q}_2\mathbf{q}_0}$ in Figure 2). Therefore, packing the bound data of 4 sibling nodes into a single data structure can save space by preventing duplicate storage of these displacement bounds.

It is actually possible to achieve further reduction in storage and data movement by using lower-precision representations for the bound data. This does not lead to any rendering errors, since lower-precision bound data can simply be conservative and less tight.

For packing the 4 sibling nodes into 3 cache lines, it is sufficient to reduce the precision of the 9 displacement bounds we must store down to 10 bits. Since displacement bounds are relatively small values, we find the largest displacement bound and use it as a scaling factor of all displacement bounds. We also take the square root of the displacement before quantization to provide more resolution for finer details. As a result 10 bits provide sufficient precision for representing the displacement bounds.

We can pack the 4 sibling nodes into 2 cache lines by using a 17-bit fixed-point format for the bounding boxes. Alternatively, we can use 9-bit fixed-point for displacement bounds and 18-bit fixed-point for bounding boxes, which would require one extra bit. Instead of storing this extra bit elsewhere, we can simply reduce the precision of one of the bounding box numbers.

In our tests (Section 6), we use a less optimized packing of bound data, using 16-bit values for bounding boxes and 10-bit values for displacement bounds. Thus, the bound data for each node can be independently packed into half a cache line (16 bytes) with 2 bits to spare. Thus, the node data for the 4 sibling nodes can be packed into 2 cache lines. This option significantly simplifies the implementation and only had 1 or 2 fewer bits of precision for bounding boxes as compared to the more optimized alternatives described above.

4.4 Packetized Ray Casting

As mentioned in Section 3.4, our traversal stack contains additional information, storing the inserted vertex positions. This overhead can be reduced using ray packets by tracing groups of rays together. Ray packets, particularly for ray casting (i.e. primary rays), can amortize the cost of extra compute and local data by combining the LOD-related computation and using a shared traversal stack.

To get the most out of ray packets, we use a shared edge state computation that forms a common LOD state for all rays in a packet. The computed intermediate inserted vertex positions can be safely shared by all rays in the packet.

Note that primary rays in a packet diverge from each other as they move away from the camera. In screen-space, however, they maintain their relative positions. That is why our LOD approach is a good fit for packatized ray casting. Even though it may not perfectly match the LOD decisions computed independently for each ray in the packet, the differences are typically minor and can be safely approximated using ray packets.

5 HARDWARE IMPLEMENTATION

It is important to consider a hardware implementation to handle the additional compute of our LOD operations. In this section we describe how our technique could be implemented in hardware. The traversal loop of our technique should use special purpose hardware for some operations and ideally the entire loop.

Figure 9 shows a top level block diagram for how our technique processes a single ray against a triangle. Some of the steps are well understood. In particular, ray-triangle and box intersection are the same as in modern ray tracing hardware. Early termination is also the same, except for passing on displacement bounds for rays that do not terminate. The Sort Child Nodes block updates the Traversal Stack by placing intersected child nodes in the order of intersection.

The primary difference from existing hardware ray tracers is in the process to decide whether to intersect the ray with the current triangle or with its tessellated triangles. The LOD Test block inputs the displacement bounds and checks to see whether any of the edges require the triangle to be tessellated. If not, the triangle passes immediately to the Triangle Intersect block. Otherwise, the edge displacements are sent to the Tessellation block, which uses the three inserted vertices stored in the node's vertex data to compute four sub-triangles. Then, the vertex positions are morphed between the specified locations and the edge midpoints, if necessary. If the current node is leaf, all four are sent to the Triangle Intersect block. Otherwise, the traversal continues with testing the bounding boxes of the four child nodes. The intersecting child nodes are sorted and stored in the traversal stack, along with the vertex positions of their triangles.

The added compute required by the new stages is moderate. E.g. checking the edge quality requires 45 operations and morphing the inserted vertices requires 9 operations. Our technique also requires computing sub-triangle indices and barycentric coordinates within the sub-triangle when a triangle is tessellated. Figure 10 illustrates the simple logic needed to compute sub-triangle barycentric coordinates, which replaces a dozen instructions if implemented in software. Computing sub-triangle indices is similarly complex in software and even simpler in custom hardware.

6 EVALUATION AND RESULTS

We provide an evaluation of our method using a cycle-accurate simulation of a highly-parallel ray tracing hardware architecture (Section 6.1). We include another evaluation that includes packetized ray traversal for ray casting (Section 6.2). Finally, we present our results with secondary rays (Section 6.3).

6.1 Ray Casting Simulation

The LOD technique we describe in this paper is designed to reduce data movement when rendering high-resolution models from a distance, such that they correspond to a relatively low-resolution part of the rendered image. This is the key for efficiently rendering

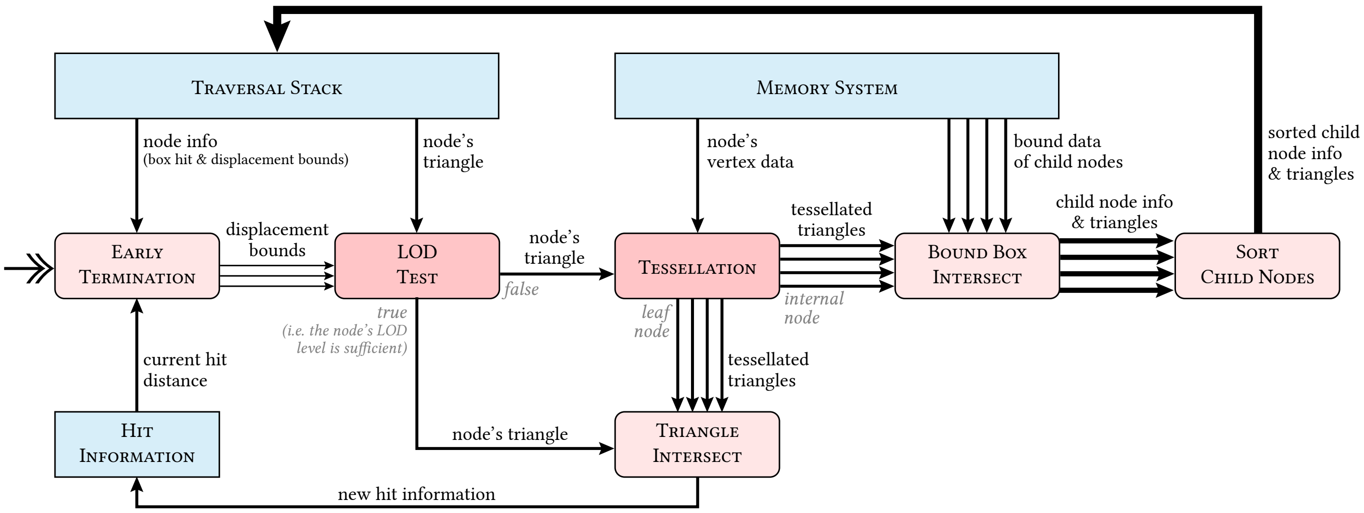


Fig. 9. *Block diagram of our ray traversal. Our addition is to perform the LOD Test and apply Tessellation if it fails (i.e. the current node's level is insufficient). The rest of the blocks are typical for ray traversal without LOD.*

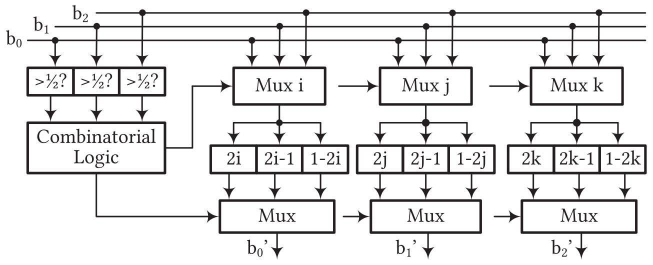


Fig. 10. *Logic to calculate the barycentric coordinate in a child triangle from the barycentric coordinate in the parent triangle (see Figure 7). Either one or none of the parent barycentrics is greater than one half, which selects how to compute the child barycentrics.*

high-resolution scenes with a highly-parallel ray tracing hardware architecture that contains sufficient computation power, augmented with special-purpose ray traversal and intersection units.

Therefore, our performance evaluation concentrates on data movement within the entire memory hierarchy. This includes data movement between the processor cores and their L1 caches, between multiple L1 caches and a chip-wide L2 cache, and between the L2 cache and an off-chip DRAM. We measure the detailed data movement using a cycle-accurate simulation of a generic ray tracing hardware architecture and assume that there are no delays due to computation.

The specific hardware system we simulate is a version of the TRaX architecture [Spjut et al. 2008, 2009] packed with specialized hardware units for handling all ray traversal and intersection logic that can process the data as fast as the memory hierarchy can deliver. This ensures that there is no computation related latency that could be easily resolved by increasing the computational resources and that our results are not narrowly limited to one particular hardware configuration, but can be interpreted somewhat more broadly.

Nonetheless, cycle-accurate simulation requires a detailed specification of a part of the hardware configuration. In our tests, we use a configuration that is in line with today's GPUs. More specifically, the processor we simulate contains 1024 independent ray tracing cores issuing memory requests at a 2 GHz clock rate. These cores are grouped into 64 thread multi-processors (TM), each containing a 16 KB 2-way set associative L1 cache with 4 banks and 1 cycle latency. Each L1 cache services 16 ray tracing cores within a TM and is connected to a 2 MB 4-way set associative chip-wide L2 cache with 32 banks and a latency of 4 cycles. This L2 cache is connected to GDDR5 DRAM with 8 channels running at 5 GHz.

Figure 11 shows the results of ray casting in 3 different test scenes with different camera distances, comparing our method to a standard quad-BVH (each node with 4 child nodes) of the highest-resolution model, running on the same hardware. These scenes are represented with full geometry without any instancing optimization. In these tests we use 16-bit fixed-point bounding boxes for both BVH and our method. This way, we can ensure that the 4 child node bounding boxes of a node perfectly fit in 2 cache lines, similarly optimizing the data movement for both BVH and our method.

As can be seen in the graphs in Figure 11, our method uses significantly less energy, up to 20× less in these tests. The main reduction is in DRAM energy, as our method has 5× to 30× fewer DRAM accesses, which dominate the total energy cost. Notice that with BVHs the energy and render time costs peak at the distance when the whole model is visible on the image. With our method, however, the peak can be earlier, when more rays traverse deeper into our tessellation trees.

Not surprisingly, when there is sufficient computation power, improvements in energy cost and render time are correlated, as can be seen in Figure 11. This is because DRAM is not only the main energy consumer, but also the reason for most data-dependent stalls. The improvements of our DRAM accesses result in up to 16× faster render times in some of these experiments. After all, regardless of

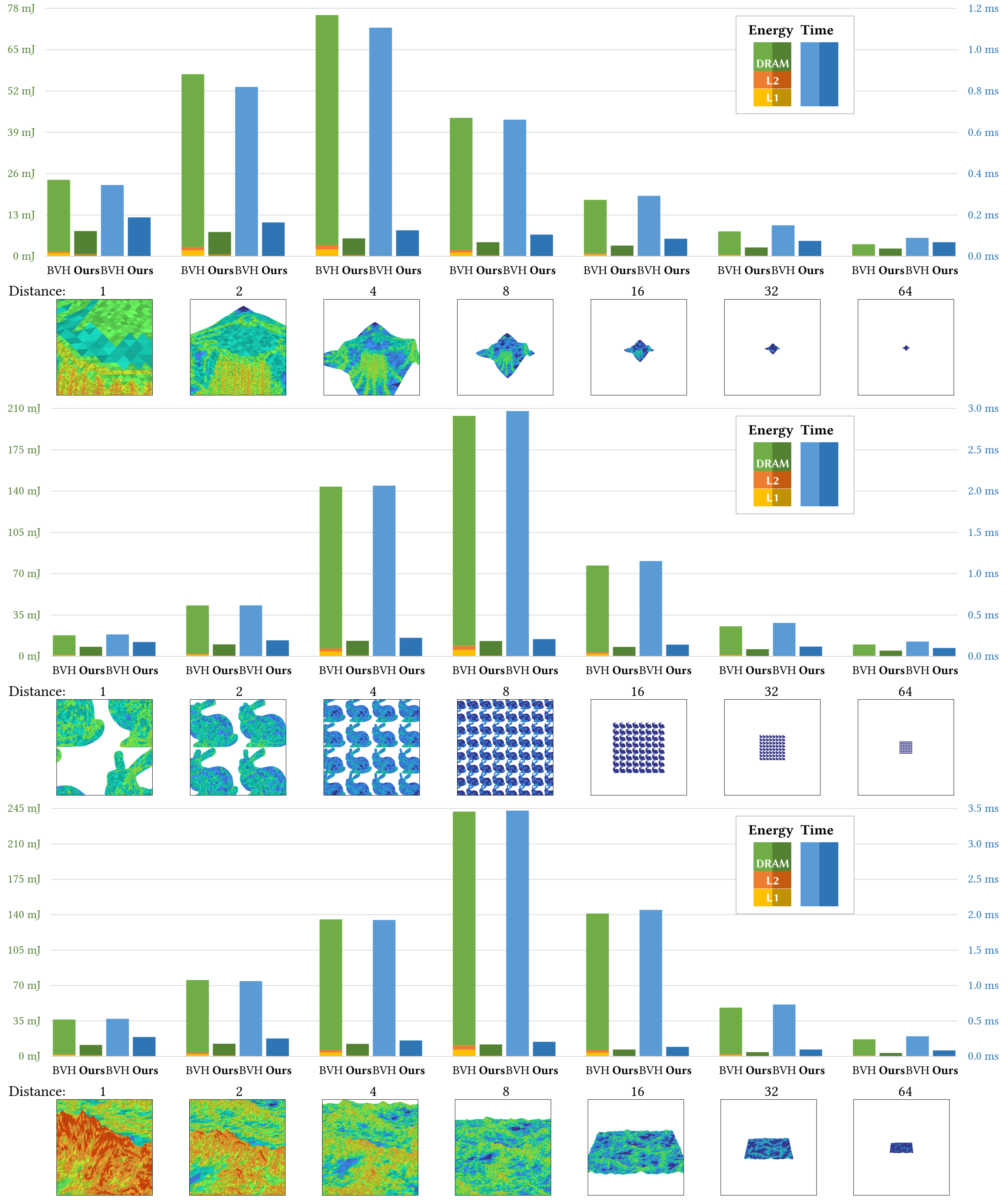


Fig. 11. *Breakdown of total energy and time per frame for different distances comparing our method to regular BVHs in three test scenes.*

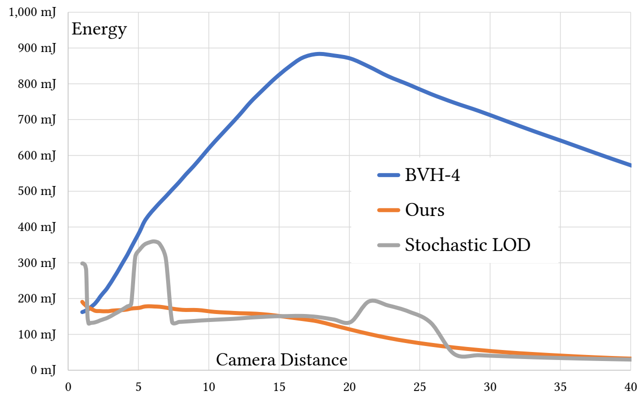
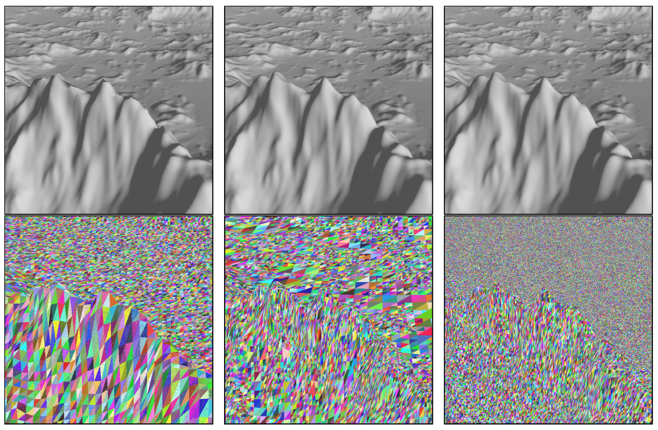


Fig. 12. *The estimated data movement energy cost during rendering for different camera distances with the terrain scene. The LOD level for stochastic LOD is selected based on the average LOD level that our method uses; therefore, it results in a similar energy profile, except for the peaks when it transitions between levels. Standard quad-BVH consumes signifincantly more energy for most distances.*



(a) *Stochastic LOD* **(b)** ***Ours*** **(c)** *Reference*

Fig. 13. *An example terrain scene (camera distance=1.5), showing the difference in the intersected triangles. Stochastic LOD uses too low a resolution near the camera and too high a resolution farther from the camera, since its LOD decision is not per ray, like ours.*

the compute capabilities of the processor, a scene cannot be rendered faster the time it takes for the DRAM to deliver the necessary data.

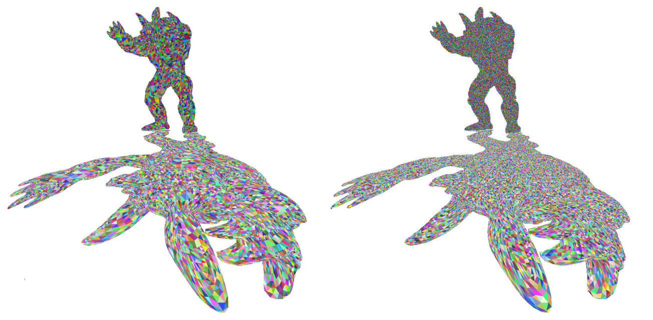
Our experiments with full-precision BVHs showed that they consistently performed slower and consumed more energy than the reduced-precision BVHs in Figure 11 by about 10% to 40%.

6.2 Comparisons with Packetized Ray Casting

We also provide a simpler evaluation of data movement energy for packetized ray casting, using a processor with a single thread, connected to an L1 cache of 32 KB. Between L1 and DRAM, we place an L2 cache of 256 KB. In these tests, we assume a relative energy cost of 1× for L1, 1.5× for L2, and 40× for DRAM.

In Figure 12 we present test results with packetized ray-casting using 32 rays per packet. We compare our method both to the reference full-resolution mesh and a mesh using discrete level-of-detail with stochastic transitions, i.e. stochastic LOD [Brandon Lloyd 2020]. The stochastic LOD uses the same mesh resolution levels as our method, but selects two levels prior to rendering and stochastically alternates between them per ray during LOD transition regions. To provide a direct comparison and make sure that this choice of the LOD level would not impact our comparisons, we pick the LOD level for stochastic LOD that matches the average LOD level used by all rays with our method, after rendering the scene with our method first. This circumvents one important difficulty with stochastic LOD: deciding which level to use prior to rendering. On the other hand, it limits the quality with stochastic LOD, as it cannot pick a higher-resolution level where it is needed. Nonetheless, it allows us to achieve a similar geometric complexity for both methods.

As can be seen in Figure 12, stochastic LOD results in a similar data movement, as expected (due to matching our average LOD level), except when stochastic LOD is transitioning between two levels (by stochastically picking one of the two consecutive levels per ray), which results in significantly increased data movement.



(a) ***Ours*** **(b)** *Reference*

Fig. 14. *Secondary rays of our method can properly pick the LOD level that would maintain the quality of the secondary effects, such as shadows. This visualization shows the final triangles intersected. With our method, part of the shadow that is close to the camera uses higher resolution, while the rest of the shadow and the model use lower-resolution tessellations.*

This can be clearly seen as spikes of energy use for stochastic LOD in Figure 12.

Nonetheless, both LOD methods substantially outperform the reference using quad-BVH of the full-resolution model.

It is important to note that, because the geometric complexity of stochastic LOD is homogeneous (on average), it generally results in lower quality tessellation than our method. Figure 13 shows an example sample from the same tests. As can be seen in this image, stochastic LOD results in much lower-resolution triangulation for parts of the model that are close to the camera, while using too fine triangulation for further away parts. Obviously, this is an expected behavior with uniform LOD level selection prior to rendering.

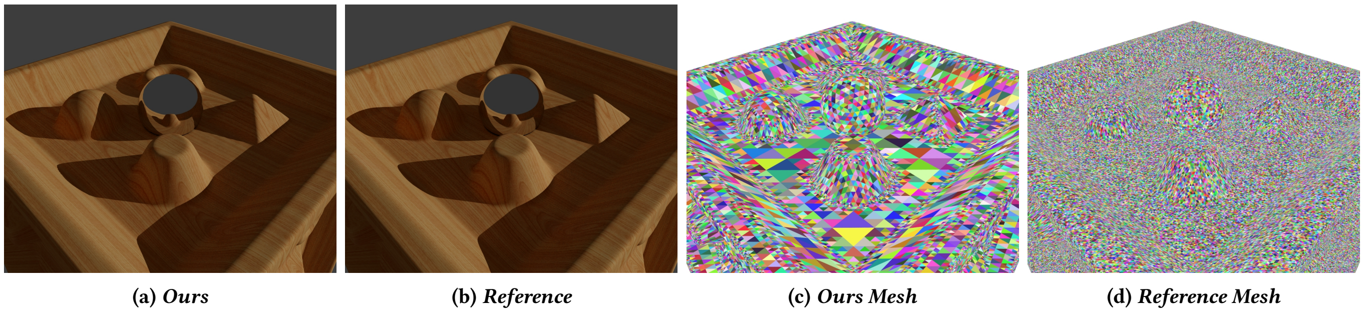


Fig. 15. *Example of using secondary rays for shadows, reflections, and diffuse global illumination. Shadows and lighting are rendered with no visible artifacts, despite using large triangles for parts of the model with flat shape and low geometric detail.*

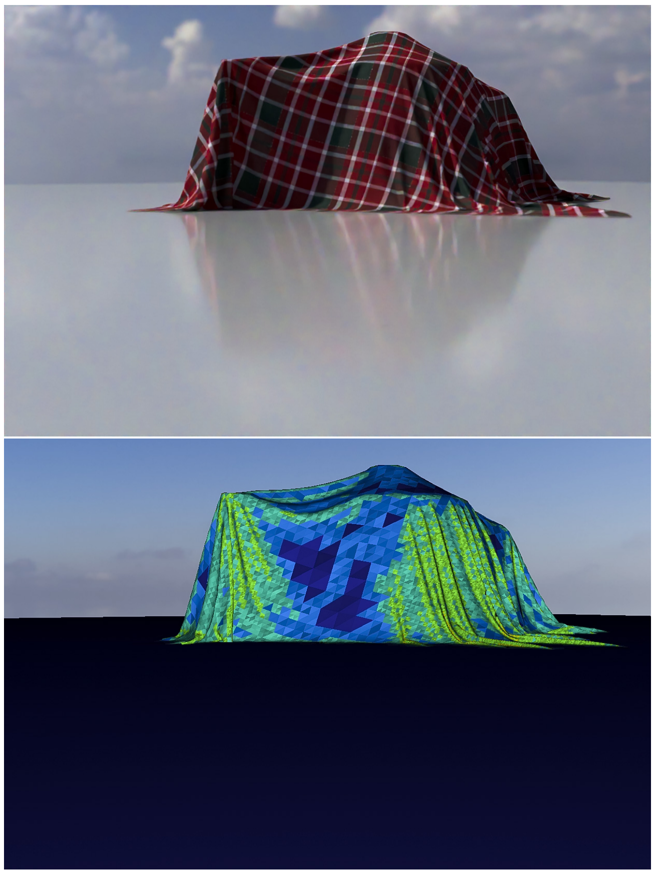


Fig. 16. *A cloth model on a teapot, rendered using our method with glossy reflections, shadows, and diffuse global illumination. Notice that our method automatically uses large screen space triangles for relatively flat parts of the model.*

6.3 Secondary Rays

Our method can also properly adjust the LOD level for secondary rays. This is demonstrated in the shadow example in Figure 14, visualising the triangles used for the final intersection tests. Notice that the mesh resolution on the parts of the shadow that is closer to the camera is similar to the reference, but the model itself, which is further away from the camera, is rendered using a lower-resolution triangulation. Also, notice that our method results in a similar triangle size for the entire image in this example.

When rendering models with more flat/smooth features and less details, our adaptive LOD method can automatically use lower-resolution triangles, as apparent in Figure 15. In this example, screen-space geometric error is kept below the size of a pixel, though our method picks triangles that are much larger than a pixel. Notice that reflections and shadows are properly computed without visible visual artifacts, while rendering significantly fewer triangles.

Figure 16 shows a cloth model on a teapot, rendered using our method. Notice that our method automatically uses a low-resolution tessellation for the flat parts of the model, even when it leads to relatively-large triangles in screen space.

Figure 1 shows a comparison of our method to the full-resolution model, including secondary effects like shadows, reflections, and diffuse global illumination. Our LOD metric ensures that the final image of our method is indistinguishable from the reference, despite accessing significantly fewer triangles during rendering.

To provide a quantitative comparison, we placed the same cloth model in a box with an open side and rendered with path tracing, using a Lambertian material. The results with our cycle-accurate simulation using the same hardware setup as the tests in Section 6.1 are shown in Figure 17 in comparison to a quad-BVH with reduced-precision bounds. In this test, our method consumes about 8× less energy and renders about 7.5× faster using primary rays only. The overhead of secondary rays with a single bounce is significant, though BVH has more than 13.5× the render time overhead as our method. Using additional bounces in this scene leads to a smaller render time overhead, as each additional bounce has progressively fewer rays in this scene.

7 DISCUSSION

While our method provides substantial savings in data movement, this reduction is not free. It comes at an additional computation complexity. Therefore, it is not suitable for software implementations running on hardware that lacks sufficient computation power to absorb the additional compute load. On the other hand, considering hardware implementation, the additional logic required for our ray traversal is relatively straightforward, as explained in Section 5.

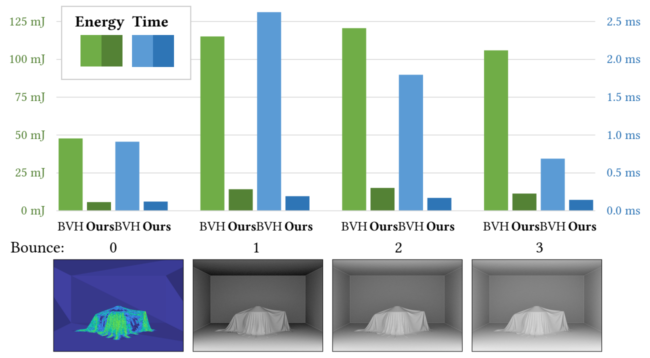


Fig. 17. *Energy and frame time overhead of rays for each bounce in a path tracer, comparing a standard quad-BVH with reduced-precision bounds to ours. The overhead values in the graph are calculated as the difference between the total energy and render times for simulations with different bounce counts. For bounce 0 the graph shows the total energy and render times. Bounce 1 in the graph shows the difference of total values between bounce 1 and 0. Bounce 2 and 3 overheads are calculated similarly, as differences from bounce 1 and 2, respectively.*

Integrating our technique into a modern API primarily requires a new mesh type for locally adaptive tessellation meshes and a tuneable parameter to allow the user to specify the amount of acceptable error. The ray type would also have to include the ray cones used to compute perceived error. This is similar to the kind of API changes required to support micro-meshes [NVIDIA 2023].

Other than that, the meshes can be treated as normal meshes. The hit record can be populated by computing the barycentric coordinates and triangle ID of the highest-resolution version of the mesh that can be trivially mapped to the coarser triangle using an equal division the coarser polygons. This means that shading code can just treat the mesh like the highest-resolution mesh.

Our method can also be used with animating models. After the tessellation trees are built, they can easily be refit to a deformed model. Since the structure of the tessellation trees are pre-defined, there is no need to rebuild the acceleration structure beyond a simple refit, except for the lowest-resolution level, which uses a standard BVH. Therefore, our data structure can be quickly updated for animating models.

An important limitation of our approach is that it does not support arbitrary mesh topologies. The lowest-resolution mesh can have arbitrary topology, but the topology of all subsequent levels must be defined by the tessellation rule we use. Models that do not support this topological structure must be remeshed before they can be used with our method.

8 CONCLUSION

We have presented a locally-adaptive level-of-detail method for ray tracing, aimed at reducing the energy cost due to data movement via adaptive tessellation. By building a specialized data structure alongside our BVH, we are able to support adaptive tessellation without storing multiple versions of the model or BVH. Through the use of screen-space metrics, we are able to use this data structure to dynamically reduce the number of triangles accessed at render time. Our results show that this technique is capable of reducing the energy cost of rendering a mesh, while only marginally degrading quality, the amount of which is controlled by a user-defined parameter. Additionally, we show that this technique is capable of handling secondary rays without introducing self-intersection artifacts.

Since the primary function of this method is to reduce memory bandwidth use at the expense of compute, our approach would not very helpful for software implementations of ray-tracing that are compute bound. However, in memory bound contexts, such as hardware-accelerated ray tracing, it would follow that our method would lead to performance improvements. It also is highly beneficial in any system where performance is limited by power use.

ACKNOWLEDGMENTS

This project was supported in part by a grant from Meta Reality Labs and a gift from AMD.

REFERENCES

Carsten Benthin, Sven Woop, Matthias Nießner, Kai Selgrad, and Ingo Wald. 2015. Efficient Ray Tracing of Subdivision Surfaces Using Tessellation Caching. In *Proceedings of the 7th Conference on High-Performance Graphics* (Los Angeles, California) *(HPG '15)*. Association for Computing Machinery, New York, NY, USA, 5–12. https://doi.org/10.1145/2790060.2790061

Martin Stich Brandon Lloyd, Oliver Klehm. 2020. Implementing Stochastic Levels of Detail with Microsoft DirectX Raytracing. Thing's Credible!, blog. https://developer.nvidia.com/blog/implementing-stochastic-lod-with-microsoft-dxr/

Per H. Christensen, Julian Fong, David M. Laur, and Dana Batali. 2006. Ray Tracing for the Movie 'Cars'. In *2006 IEEE Symposium on Interactive Ray Tracing*. 1–6. https://doi.org/10.1109/RT.2006.280208

Per H. Christensen, David M. Laur, Julia Fong, Wayne L. Wooten, and Dana Batali. 2003. Ray Differentials and Multiresolution Geometry Caching for Distribution Ray Tracing in Complex Scenes. *Computer Graphics Forum* 22, 3 (2003), 543–552. https://doi.org/10.1111/1467-8659.t01-1-00702

Peter Djeu, Warren Hunt, Rui Wang, Ikrima Elhassan, Gordon Stoll, and William R. Mark. 2011. Razor: An Architecture for Dynamic Multiresolution Ray Tracing. *ACM Trans. Graph.* 30, 5, Article 115 (oct 2011), 26 pages. https://doi.org/10.1145/2019627.2019634

Saugata Ghose, Abdullah Giray Yaglikçi, Raghav Gupta, Donghyuk Lee, Kais Kudrolli, William X. Liu, Hasan Hassan, Kevin K. Chang, Niladrish Chatterjee, Aditya Agrawal, Mike O'Connor, and Onur Mutlu. 2018. What Your DRAM Power Models Are Not Telling You: Lessons from a Detailed Experimental Study. *Proc. ACM Meas. Anal. Comput. Syst.* 2, 3, Article 38 (dec 2018), 41 pages. https://doi.org/10.1145/3224419

Johannes Hanika, Alexander Keller, and Hendrik P. A. Lensch. 2010. Two-Level Ray Tracing with Reordering for Highly Complex Scenes. In *Proceedings of Graphics Interface 2010* (Ottawa, Ontario, Canada) *(GI '10)*. Canadian Information Processing Society, CAN, 145–152.

Hugues Hoppe. 1998. Smooth View-Dependent Level-of-Detail Control and Its Application to Terrain Rendering. In *Proceedings of the Conference on Visualization '98* (Research Triangle Park, North Carolina, USA) *(VIS '98)*. IEEE Computer Society Press, Washington, DC, USA, 35–42.

Sho Ikeda, Paritosh Kulkarni, and Takahiro Harada. 2022.. Multi-Resolution Geometric Representation using Bounding Volume Hierarchy for Ray Tracing. AMD GPUOpen. Vol. 32. (2022.). https://gpuopen.com/download/publications/GPUOpen2022_FusedLOD.pdf

Paritosh Kulkarni, Sho Ikeda, and Takahiro Harada. 2019.. Fused BVH to Ray Trace Level of Detail Meshes. AMD GPUOpen. Vol. 32. (2019.). https://gpuopen.com/download/publications/GPUOpen2022_FusedLOD.pdf

Aaron Lee, Henry Moreton, and Hugues Hoppe. 2000. Displaced Subdivision Surfaces. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '00)*. ACM Press/Addison-Wesley Publishing Co., USA, 85–94. https://doi.org/10.1145/344779.344829

Won-Jong Lee, Gabor Liktor, and Karthik Vaidyanathan. 2019. Flexible Ray Traversal with an Extended Programming Model. In *SIGGRAPH Asia 2019 Technical Briefs* (Brisbane, QLD, Australia) *(SA '19)*. Association for Computing Machinery, New York, NY, USA, 17–20. https://doi.org/10.1145/3355088.3365149

Charles Loop. 1987. *Smooth Subdivision Surfaces Based on Triangles.* Master's thesis. University of Utah.

Jan Novák and Carsten Dachsbacher. 2012. Rasterized Bounding Volume Hierarchies. *Computer Graphics Forum (Proc. of Eurographics)* 31, 2 (2012), 403–412.

NVIDIA. 2023. Micro-Mesh. https://developer.nvidia.com/rtx/ray-tracing/micro-mesh

Kai Selgrad, Alexander Lier, Magdalena Martinek, Christoph Buchenau, Michael Guthe, Franziska Kranz, Henry Schäfer, and Marc Stamminger. 2016. A Compressed Representation for Ray Tracing Parametric Surfaces. *ACM Trans. Graph.* 36, 1, Article 5 (nov 2016), 13 pages. https://doi.org/10.1145/2953877

Juha Sjoholm. 2018. Effectively Integrating RTX Ray Tracing into a Real-Time Rendering Engine. Thing's Credible!, blog. https://developer.nvidia.com/blog/effectively-integrating-rtx-ray-tracing-real-time-rendering-engine/

Brian Smits, Peter Shirley, and Michael Stark. 2000. Direct Ray Tracing of Displacement Mapped Triangles. *Rendering Techniques 2000 (Proc. 11th Eurographics Workshop on Rendering)*, 307–318. https://doi.org/10.1007/978-3-7091-6303-0_28

Josef Spjut, Solomon Boulos, Daniel Kopta, Erik Brunvand, and Spencer Kellis. 2008. TRaX: A Multi-Threaded Architecture for Real-Time Ray Tracing. In *Proceedings of the 2008 Symposium on Application Specific Processors (SASP '08)*. IEEE Computer Society, Washington, DC, USA, 108–114. https://doi.org/10.1109/SASP.2008.4570794

Josef Spjut, Andrew Kensler, Daniel Kopta, and Erik Brunvand. 2009. TRaX: A Multicore Hardware Architecture for Real-time Ray Tracing. *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 28, 12 (Dec. 2009), 1802–1815. https://doi.org/10.1109/TCAD.2009.2028981

Theo Thonat, Francois Beaune, Xin Sun, Nathan Carr, and Tamy Boubekeur. 2021. Tessellation-Free Displacement Mapping for Ray Tracing. *ACM Trans. Graph.* 40, 6, Article 282 (dec 2021), 16 pages. https://doi.org/10.1145/3478513.3480535

Elena Vasiou, Konstantin Shkurko, Ian Mallett, Erik Brunvand, and Cem Yuksel. 2018. A Detailed Study of Ray Tracing Performance: Render Time and Energy Cost. *The Visual Computer (Proceedings of CGI 2018)* (April 2018). https://doi.org/10.1007/s00371-018-1532-8

Sung-Eui Yoon, Christian Lauterbach, and Dinesh Manocha. 2006. R-LODs: fast LOD-based ray tracing of massive models. *The Visual Computer* 22, 9 (01 Sep 2006), 772–784. https://doi.org/10.1007/s00371-006-0062-y

A FIXED-POINT RAY-TRIANGLE INTERSECTION

Our ray-triangle intersection uses Plucker coordinates with bit-perfect precision in fixed-point. Let $\mathbf{p}_m$ represent the midpoint of an edge that connects vertices $\mathbf{p}_a$ and $\mathbf{p}_b$. Note that we guarantee that $\mathbf{p}_m$ can be exactly represented in fixed-point without any truncation. To guarantee watertightness using Plucker coordinates, we simply need to show that rays are guaranteed to provide exactly the same intersection test results for the full edge $\overline{\mathbf{p}_a\mathbf{p}_b}$ and the half edges $\overline{\mathbf{p}_a\mathbf{p}_m}$ and $\overline{\mathbf{p}_m\mathbf{p}_b}$. Further tessellations of the edge follow the same proof via recursion.

Given a ray with (fixed-point) origin $\mathbf{x}$, Plucker coordinates are computed using the cross product of each edge and the vector from $\mathbf{x}$ to the first vertex of the edge, forming an edge normal direction. With our three edges ($\overline{\mathbf{p}_a\mathbf{p}_b}$, $\overline{\mathbf{p}_a\mathbf{p}_m}$, and $\overline{\mathbf{p}_m\mathbf{p}_b}$), we get the following normals:

$$\begin{aligned}
\mathbf{n}_{ab} &= (\mathbf{p}_b - \mathbf{p}_a) \times (\mathbf{p}_a - \mathbf{x}) \\
\mathbf{n}_{am} &= (\mathbf{p}_m - \mathbf{p}_a) \times (\mathbf{p}_a - \mathbf{x}) \\
\mathbf{n}_{mb} &= (\mathbf{p}_b - \mathbf{p}_m) \times (\mathbf{p}_m - \mathbf{x})
\end{aligned}$$

For the intersection tests to be consistent, these normal vectors must have the exact same direction, but their magnitudes can be different. Using 32-bit fixed-point values (for $\mathbf{p}_a$, $\mathbf{p}_b$, $\mathbf{p}_m$, and $\mathbf{x}$), the resulting normals require 64-bit fixed-point to exactly represent without truncation.

Since our representation guarantees $(\mathbf{p}_b - \mathbf{p}_a) = 2(\mathbf{p}_m - \mathbf{p}_a)$ is exactly true by ensuring that both $\mathbf{p}_a$ and $\mathbf{p}_b$ are even numbers, it is easy to see that $\mathbf{n}_{ab} = 2\mathbf{n}_{am}$ is also guaranteed. We can also show that $\mathbf{n}_{mb} = \mathbf{n}_{am}$ with the following steps:

$$\begin{aligned}
\mathbf{n}_{mb} &= (\mathbf{p}_b - \mathbf{p}_m) \times (\mathbf{p}_m - \mathbf{x}) \\
&= (\mathbf{p}_m - \mathbf{p}_a) \times (\mathbf{p}_m - \mathbf{x}) \\
&= (\mathbf{p}_m - \mathbf{p}_a) \times \big((\mathbf{p}_a - \mathbf{x}) + (\mathbf{p}_m - \mathbf{p}_a)\big) \\
&= (\mathbf{p}_m - \mathbf{p}_a) \times (\mathbf{p}_a - \mathbf{x}) + (\mathbf{p}_m - \mathbf{p}_a) \times (\mathbf{p}_m - \mathbf{p}_a) \\
&= (\mathbf{p}_m - \mathbf{p}_a) \times (\mathbf{p}_a - \mathbf{x}) \\
&= \mathbf{n}_{am}
\end{aligned}$$

The final step of the intersection tests for the edge is to compute the dot product of the edge normal and the fixed-point ray direction $\mathbf{d}$. Using 32-bit fixed-point $\mathbf{d}$, computing the exact dot product involves 96-bit adders, though we only need the sign of this dot product in the intersection test. Since $\mathbf{n}_{ab} = 2\mathbf{n}_{am} = 2\mathbf{n}_{mb}$, we can guarantee the same sign for all three edges.

Note that this requires using the same fixed-point representation for the ray origin $\mathbf{x}$ as the mesh vertices. However, our fixed-point representation is only valid within the bounding box of the mesh and the ray origin may be outside of this bounding box. In that case, this can be easily remedied by taking $\mathbf{x}$ as the point that the ray enters the bounding box of the mesh for performing the ray-triangle intersections.