DOI: 10.1111/cgf.14401
Pacific Graphics 2021
E. Eisemann, K. Singh, and F.-L Zhang
(Guest Editors)

Volume 40 (2021), Number 7

Conservative Meshlet Bounds for Robust Culling of Skinned Meshes

J. Unterguggenberger, B. Kerbl, J. Pernsteiner and M. Wimmer

TU Wien, Institute of Visual Computing & Human-Centered Technology

Abstract

Following recent advances in GPU hardware development and newly introduced rendering pipeline extensions, the segmentation of input geometry into small geometry clusters—so-called meshlets—has emerged as an important practice for efficient rendering of complex 3D models. Meshlets can be processed efficiently using mesh shaders on modern graphics processing units, in order to achieve streamlined geometry processing in just two tightly coupled shader stages that allow for dynamic workload manipulation in-between. The additional granularity layer between entire models and individual triangles enables new opportunities for fine-grained visibility culling methods. However, in contrast to static models, view frustum and backface culling on a per-meshlet basis for skinned, animated models are difficult to achieve while respecting the conservative spatio-temporal bounds that are required for robust rendering results. In this paper, we describe a solution for computing and exploiting relevant conservative bounds for culling meshlets of models that are animated using linear blend skinning. By enabling visibility culling for animated meshlets, our approach can help to improve rendering performance and alleviate bottlenecks in the notoriously performance- and memory-intensive skeletal animation pipelines of modern real-time graphics applications.

1. Introduction

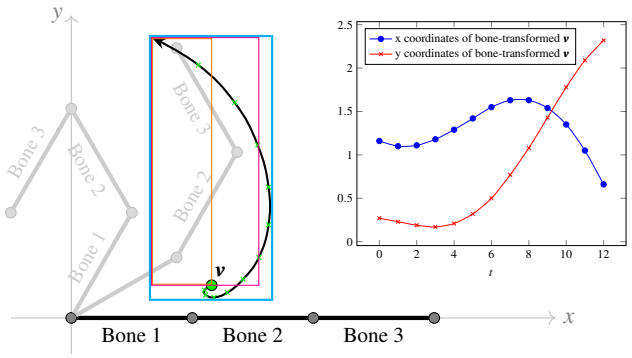
Recently, a strong trend towards extremely high geometric detail and dividing meshes into primitive clusters has emerged in the field of real-time rendering. Technologies like Epic Games' Nanite [KSW21] enable extremely high geometric detail for static meshes, using cluster culling to minimize frame times. The same level of detail is bound to become relevant for *animated* models in the near future to match the static environment. To this end, we propose a novel approach to help realize this goal: we describe an algorithm for calculating conservative spatio-temporal bounds for positions and normal orientations of primitive clusters. The so precomputed information can be used for efficient view frustum culling (VFC), backface culling (BFC), and rendering in compute-based pipelines, but also applies to hardware rasterization on modern graphics processing units (GPUs) with support for task and mesh shaders.

Task and *mesh* shaders (named *amplification* and *mesh* shaders in DirectX [Mic21], respectively) have been introduced as new shader stages with Nvidia's Turing microarchitecture to replace the multi-layered geometry processing of a conventional graphics pipeline with a more streamlined alternative [NVI18]. The key point of task and mesh shaders is to allow more fine-grained control over primitive processing and dynamic workload distribution via two tightly coupled compute shader-style geometry processing stages within rasterization-based graphics pipelines, while still utilizing their later hardware-accelerated fixed-function stages. The new setup encourages the division of geometry workload into smaller packages that can be efficiently processed by the GPU. For Nvidida's Turing microarchitecture, optimal efficiency can be achieved by dividing

Figure 1: *Skinned models* GAWAIN *(198k vertices, 261k triangles, 136 bones, 9044 meshlets),* GIANT WORM *(169k vertices, 329k triangles, 80 bones, 10382 meshlets),* BUTCHER *(287k vertices, 477k triangles, 224 bones, 14915 meshlets), and* WYVERN *(267k vertices, 512k triangles, 88 bones, 16015 meshlets), divided into color-coded meshlets, fit for rendering with task and mesh shaders.*

indexed triangle meshes into parts that consist of no more than 64 vertices and 126 triangles [Kub18a]. Each one of these small geometry packets is referred to as a *meshlet*. Since the new shader stages and meshlet-based rendering see wide-spread support in recently released GPUs—including the GPUs utilized in the consoles of the latest generation: PlayStation 5, and Xbox Series S and X, which are based on AMD's RDNA 2 microarchitecture [Bla20]—we expect developer adoption and research interest in task and mesh shader-based solutions to rise considerably. The task shader runs early in the graphics pipeline and can serve as a work scheduler for the mesh

(a) *On the borders of the screen, the effect of premature frustum culling can be observed during animations of* GIANT WORM *and* BUTCHER *(cyan rectangles). Inside the* GIANT WORM*'s mouth, some meshlets are prematurely backface-culled as a result of disregarding how the normals of triangles assigned to meshlets change through animation (marked in magenta).*



(b) *The path of a single vertex* $\boldsymbol{v}$*, animated via LBS from one keyframe to the next. The vertex is strongly weighted toward Bone 3. Sampling intermediate positions underestimates the spatial bounds (orange box: start and end, magenta box: two intermediate samples). As multiple rotations are chained, the path of* $\boldsymbol{v}$ *can become arbitrary and coordinate extrema difficult to predict.*

Figure 2: *Premature culling can result in very noticeable visual artifacts, as shown in Figure 2a. Through animation, the shape of a meshlet can change significantly. Underestimating the vertex bounds can lead to premature view frustum culling, while not taking into account the possible changes in surface normals of a meshlet's triangles can lead to erroneous backface culling. For both, vertex positions and surface normals, merely sampling along an animation interval can be insufficient for calculating conservative bounds, as illustrated in Figure 2b.*

shader, similar to the dynamic work generation performed by tessellation control shaders in a conventional pipeline. Only meshlets that are not culled by the task shader are processed in the later shader stages. Precomputation of positional and normal bounds for static models can be achieved easily [Wih16]. However, generation and usage of such information are significantly more difficult to achieve for animated models, since a meshlet's shape can change under animation. Dividing modern skinned 3D models into meshlets can yield results like those shown in Figure 1.

Vertex skinning is one of the most popular and widely used animation methods for 3D models. With vertex skinning, vertices are assigned to one or multiple bones of a skeleton in a weighted manner. When the skeleton's bones move into different positions over the course of an animation, assigned vertices move according to their weighting. With respect to meshlets—each of which represents a part of the skin—this means that geometry primitives associated with a meshlet can change their shape since the referenced vertices will have different bone assignments and weights in general. For example, a meshlet could be stretched in one or multiple directions, thus increasing its bounds w.r.t. a resting or configuration pose. Also, since bone assignment and weighting can differ between the vertices of a triangle, animation can produce face normal directions that were not present in the initial pose. Figure 2 visualizes artifacts of VFC and BFC that can occur if these bounds are underestimated.

In this paper, we describe and evaluate a solution for computing conservative spatio-temporal vertex bounds under animation for arbitrary animation clips. We show how conservative bounds for a meshlet's extents and also for its face normals distribution can be computed from the vertex bounds of its associated vertices in a CPU-based precomputation step. Evaluating the precomputed data in task shaders allows for robust VFC and BFC on a per-meshlet basis, extending the advantages of the additional visibility culling granularity of task shaders in rasterization-based graphics pipelines from static meshes to models with skeletal animation. Exploiting the computed bounds to perform robust culling can accelerate rendering and reduce bottlenecks in skeletal animation pipelines on modern GPU architectures while preserving the fidelity of the rendered scenes. Hence, this paper describes the following contributions:

- We derive an adaptive procedure to compute spatio-temporal axis-aligned bounding boxes (AABBs) on a per-meshlet basis for a given interval of an animation clip that are suitable for linear blend skinning (LBS). Our approach yields conservative bounds over all continuous joint orientations within a given animation and can be parameterized to produce arbitrarily tight bounds.
- We describe a method for efficiently computing the maximum extents described by a given rotation quaternion, based on a derivative of Rodrigues' rotation formula.
- Given spatial bounds for individual mesh vertices, we show that we can obtain a conservative estimate on the maximum deviation of a triangle's surface normal from an initial state during a given animation interval. We apply this concept to entire meshlets to obtain normal cones for robust backface culling with LBS.
- We evaluate our spatio-temporal meshlet bounds for animated models in task and mesh shader-based rasterization, enabling VFC and BFC on a per-meshlet basis. The assessed methods incur negligible memory overhead and improve run-time performance.

The remainder of this paper is structured as follows: in Section 2, we discuss relevant background and related work our approach builds on. In Section 3, we derive our solution for the adaptive computation of arbitrarily tight vertex bounds. In Section 4, we show how this information can be exploited to compute conservative surface normals distributions of animation intervals. Implementation details are described in Section 6. Sections 5 and 7 assess the usage of the computed bounds for visibility culling and resulting performance impact. Finally, we discuss possible extensions of the presented approach as well as open problems in Section 8.

2. Related Work

Vertex skinning plays an integral role in the animation of characters in visual applications. The fundamental concept of connecting vertices with manually or automatically defined weights to an underlying skeleton can be implemented in a variety of ways [MLT89; BP07]. Arguably, one of the most widespread methods to this day is LBS, in which the final animated position of each vertex is the result of a linear combination of independently computed results. Obvious shortcomings and frequent artifacts (e.g., the "candy-wrapper" effect) gave rise to alternative approaches, such as spherical blend skinning [KŽ05b] and log-matrix skinning [CM04], which manage to eliminate some of these artifacts, but introduce others and exhibit a higher performance penalty. In contrast, dual-quaternion skinning is comparable to LBS in terms of performance, while resolving most of its issues [KCŽO07]. Instead of opting for mathematically involved solutions at runtime, the selection of optimized centers of rotation as an isolated preparatory step for animation has been recently suggested [LH16]. While pursuing either of these methods would be worthwhile, we will only consider the fastest of these methods, LBS, for the derivation of our conservative bounds.

The task of computing conservative spatio-temporal bounds for meshes—or more generally the vertices and faces of its meshlets—undergoing skeletal animation is more challenging than it may seem at first glance. A key requirement for conservative bounds which are suitable for visibility culling is that they encompass all possible positions that vertices can occupy, as well as all possible face normal orientations that can emerge during an animation clip or subintervals thereof. Clearly, these challenges are related to the field of collision detection. A wide range of efficient solutions exists for this topic, which requires computing positional bounds for particular instants or ranges. *Reduced deformation* solutions effectively decouple bound computations from the geometry and perform them on influencing factors only (e.g., bones) before applying them to entire clusters of primitives. Several such approaches present solutions that target individual animation frames [KŽ05a; SBT06; SOG08], but not the interval in-between. Furthermore, their application usually entails a non-negligible run-time cost for computing and updating bound information, which impedes complex animated scenes with many differently animated (e.g., temporally offset) models. The same is true for established reduced deformation approaches for bounding normals, such as normal trees [SGO09]. Temporally continuous collision detection (e.g., swept volume approaches) have been applied to rigid objects [AA00; KVLM03; RLM04], but remain a challenging problem for deformable meshes, which we target in our work. Specialized methods for reduced deformable models, such as BD-Trees [JP04] provide excellent opportunities to accelerate bound queries for entire primitive groups. However, they require precomputed displacement fields and imply the creation and maintenance of hierarchical data structures, which we strive to avoid in order to minimize run-time overhead in complex animated scenes. Although spatial bounds may be approximated from analysis of the model data, available proprietary solutions make no claim about computed bounds being conservative [Uni21], while sampling-based methods can easily miss extreme positions and orientations and provide no guarantee for robustness [GFSS06], so that they can still lead to undesirable artifacts such as those described in Figure 2a.

Task and mesh shaders were first introduced with the Nvidia Turing architecture [NVI18]. The new shader stages can be used as alternative geometry processing stages within rasterization-based graphics pipelines. Consequently, the usage of task and mesh shaders, and the usage of classical geometry processing shaders (vertex, tessellation, and geometry shaders) are mutually exclusive. The data structures of uniforms and buffers to be used with task and mesh shaders can be freely defined. It is common practice to prepare auxiliary information about a meshlet's bounds and normals distribution, and evaluate that information in task shaders for visibility culling. The standard usage conventions and restrictions regarding possible input geometry clusters (meshlets) are defined by the corresponding Khronos conventions [Kub18b; KB19]. Similar mechanisms and rule sets are set to become adapted by competing hardware vendors in the near future.

With Turing, Nvidia also introduced GPU acceleration structures and real-time ray tracing pipelines (meanwhile standardized and defined by the Khronos conventions [Koc20a]). While task and mesh shaders are strictly limited to rasterization-based graphics pipelines and hence not usable with ray tracing pipelines, the usage of acceleration structures (enabled for usage in any shader stage through Khronos convention [Koc20b]) might appear as a possible option. Current real-time graphics APIs, however, only support ray-based access to the hardware-accelerated ray tracing data structures [Koc21] and hence do not suit rasterization.

Although meshlets as input for the hardware rendering pipeline have only been recently introduced as a topic for computer graphics, similar concepts have been proposed previously. The idea of clustering geometry is a fundamental theme in high-performance rendering of complex scenes, along with possible opportunities for optimization of their visualization at runtime [Ura19; HA15; SBOT08]. Before the introduction of mesh shaders, Kerbl et al. documented the exact rule sets used by modern GPUs to partition triangle meshes into batches before processing them in bulk [KKI*18]. In addition to rendering, the generation of meshlets ahead of time is a relevant topic in itself. Common solutions include the application of mesh optimizers, which can be easily adapted to produce basic meshlets instead [LY06; HS17; Hop99; Kap21; Wal21]. While these can account for basic qualities of meshlets, such as high connectivity and spatial compactness, they cannot ensure specialized target criteria, such as tight bounds under animation, which is one of the targeted characteristics in this paper. The idea of using geometry clusters to perform efficient visibility culling has been previously pursued, though usually with a clear focus on static geometry and applications in industry [AHA15; HC11]. Position bounds for view frustum and occlusion culling are trivially derived from the extrema found in the set of vertices in each meshlet. For backface culling, a normal cone is required, which can easily be built from the set of surface normals present in the meshlet, although finding the optimal cone is a more challenging task. Recently, Wihlidal proposed methods for generating static geometry clusters to maximize their likelihood of getting culled for visibility [Wih16]. In this paper, we will expand on this concept to enable high-performance visibility culling for meshlets of animated models.

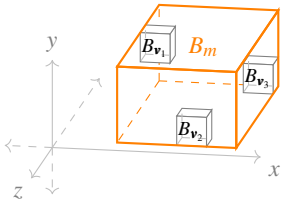


Figure 3: *Individual per-vertex AABBs $B_{\boldsymbol{v}_1}$, $B_{\boldsymbol{v}_2}$, and $B_{\boldsymbol{v}_3}$ are combined into a common bounding box B_m by taking the minimum and maximum coordinates from all AABBs' corners. B_m represents the conservative bounds of meshlet m.*

3. Meshlet Bounds Computation

In this section, we describe an algorithm to analytically compute conservative bounds per meshlet. At first, vertex bounds are calculated. Subsequently, the bounds of a meshlet can be easily computed by combining all its associated vertices' bounds into a common bounding box, which is exemplarily shown in Figure 3.

Vertex skinning transforms a vertex $\boldsymbol{v}$ according to its weighted assignment to the bones of an underlying skeleton. As multiple bones can have influence on $\boldsymbol{v}$, we first compute one AABB $B_{\boldsymbol{v}b_i}$ per influencing bone b_i and combine them in a second step into an AABB $B_{\boldsymbol{v}}$ which represents **conservative spatio-temporal bounds** for $\boldsymbol{v}$ under LBS. The temporal aspect of $B_{\boldsymbol{v}}$ refers to the animation time interval that we compute it for. One natural choice for the animation time interval is the span between two keyframes. We assume the transformation between two successive keyframes to be specified with a triple of translation, rotation, and scaling values which are interpolated in-between. The skeleton depicted in Figures 4a and 4b shall serve as an example for two different keyframe times. In Figure 4b, Bone 4 has rotated 45° clockwise (CW) w.r.t. its parent bone, and Bone 5 has rotated 45° counter-clockwise (CCW) w.r.t. Bone 4 compared to the state in Figure 4a. All child bones of Bone 5 inherit their parent's transformation and therefore change their *global* position, too. Their *local* transformations, however, stay constant between the two keyframes. Furthermore, Figure 4 shows meshlet m which has several vertices assigned to it. Each vertex has weighted assignments to one or multiple bones. Whenever the skeleton is animated into a certain position, the vertices get transformed accordingly, leading to different shapes of m in Figures 4a and 4b.

We propose to compute and store the bounding box B_m of meshlet m in the space of the most influential bone. We find this bone on a per-meshlet basis by summing all the normalized weightings of all vertices that are assigned to m per bone. The bone with the highest sum of normalized weights is deemed to be the most influential bone and thus we assign it to m as its **principal bone**. The reasoning behind this approach is that B_m can be assumed to be of minimal extent if it is computed in the space that has the most influence on the assigned vertices. Relative to the principal bone's space (PBS), the bounds of the majority of the associated vertices can be assumed to be smaller than in other spaces. This is illustrated in Figure 4 where it can be observed that the bounds in m's PBS (depicted in Figure 4d) are smaller than the bounds computed relative to mesh space or world space (depicted in Figure 4b).

3.1. Vertex Bounds Computation

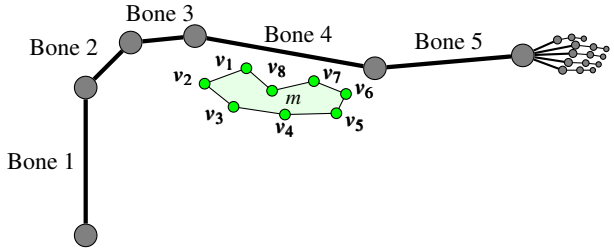
The most computation-intensive part of our algorithm is conservative vertex bounds calculation in a given target bone space—referring to the PBS of a meshlet, which is the most influential bone's coordinate system as described in Section 3. Our algorithm consists of the following major steps for computing the bounds of a specific vertex $\boldsymbol{v}$ in the target bone's coordinate system (referred to by bone b_t) between two animation times t_1 and t_2. Animation times must not stretch over keyframe time boundaries but must be limited to keyframe bounds, otherwise sudden changes in transformations—which must be expected for subsequent keyframed intervals—run the risk of missing extreme positions and thus, fail to remain conservative. For sub-keyframe intervals, t_1 and t_2 can be chosen arbitrarily narrow. Furthermore, $\boldsymbol{v}$ has a list of associated bones $I_{\boldsymbol{v}} = \{b_0, \ldots, b_n\}$, where each one of these mappings has a respective weight $W_{\boldsymbol{v}} = \{w_0, \ldots, w_n\}$ assigned. They satisfy the conditions $w_i \in [0,1]$ and $\sum_{w \in W_{\boldsymbol{v}}} w = 1$.

1. For each bone $b_i \in I_{\boldsymbol{v}}$, compute $\boldsymbol{v}$'s conservative spatio-temporal bounding box $B_{\boldsymbol{v}b_i}$ between t_1 and t_2 with full weight (i.e., as if $w_i = 1$) in the coordinate system of b_t.
   In more detail, for each bone $b_i \neq b_t$ the procedure is like follows:
   a. Transform $\boldsymbol{v}$ into the coordinate system of b_i and apply b_i's *local* scale, rotation, and translation transformations at animation time t_1. Initialize $B_{\boldsymbol{v}b_i}$ by setting its minimum and maximum coordinates to the result, yielding an AABB with zero volume in the local space of b_i.
   b. Extend $B_{\boldsymbol{v}b_i}$ by the b_i-*local* scale, rotation, and translation transformation *differences* between t_1 and t_2.
   c. Traverse to node b_j which is one node closer towards b_t from b_i. Break if b_t has been reached, otherwise loop as follows:
      i. Transform every corner of $B_{\boldsymbol{v}b_i}$ into the coordinate system of b_j, and construct a new AABB $B_{\boldsymbol{v}b_j}$ there.
      ii. Apply b_j's *local* scale, rotation, and translation transformations at animation time t_1 to every corner of $B_{\boldsymbol{v}b_j}$. Use the transformed corners to construct a new AABB $B'_{\boldsymbol{v}b_j}$.
      iii. Extend $B'_{\boldsymbol{v}b_j}$ by the b_j-*local* scale, rotation, and translation transformation *differences* between t_1 and t_2.
      iv. Assign $b_i = b_j$ and $B_{\boldsymbol{v}b_i} = B'_{\boldsymbol{v}b_j}$, let b_j refer to the next node which is one step closer towards b_t. Break if b_t has been reached, otherwise loop.
2. Combine all $B_{\boldsymbol{v}b_i}$ based on their respective weights w_i into the vertex' conservative spatio-temporal bounding box $B_{\boldsymbol{v}}$ that represents all positions that $\boldsymbol{v}$ can occupy between t_1 and t_2.

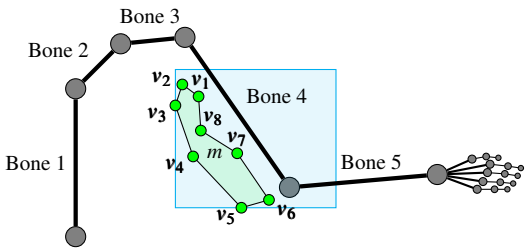
Traversing the bone hierarchy step-wise is a crucial property of our algorithm and a necessity for computing conservative vertex bounds for an animation interval. Decomposition of a global, affine bone matrix would not be a viable solution since extreme positions and orientations from intermediate steps could be missed. Care must be taken regarding the skeleton traversal direction when moving bone-by-bone towards b_t as illustrated in Figure 5. If Equation (1)

$$\boldsymbol{v}' = \boldsymbol{P}_{b_i}\, \boldsymbol{T}\, \boldsymbol{R}\, \boldsymbol{S}\, \boldsymbol{v} \tag{1}$$

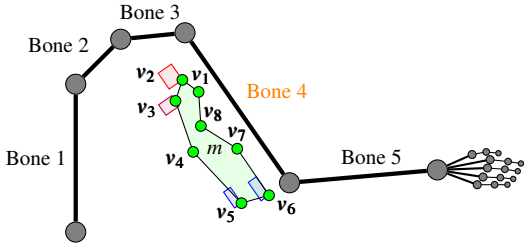
transforms from a child's bone space into its parent's bone space (with $\boldsymbol{T}$ referring to the translation, $\boldsymbol{R}$ to the rotation, $\boldsymbol{S}$ to the scaling, and $\boldsymbol{P}_{b_i}$ being a constant matrix that positions bone b_i relative to



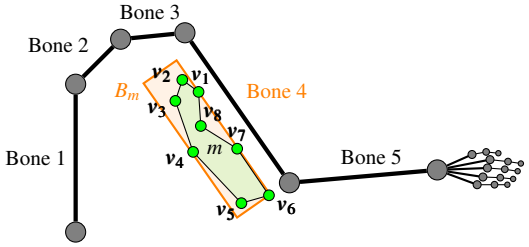
(a) *The vertices of a skinned mesh are assigned to one or multiple bones with different individual weightings each. Vertices* $\boldsymbol{v_1} \ldots \boldsymbol{v_8}$ *are combined to form meshlet m.*



(b) *Bones are transformed by an animation (Bone 4 rotated by* 45° *CW, Bone 5 by* 45° *CCW w.r.t. 4a), associated vertices are transformed according to their weights. As vertex weights vary, the shape of m changes. The cyan box encompasses all vertex positions between the poses in 4a and 4b.*



(c) *Each vertex' bounding box encompasses all vertex positions between the two different poses, relative to the coordinate system defined by Bone 4. The boxes of vertices* $\boldsymbol{v_1}$, $\boldsymbol{v_4}$, $\boldsymbol{v_7}$, *and* $\boldsymbol{v_8}$ *are infinitely small, which indicates that their weight w.r.t. Bone 4 is one. Vertices* $\boldsymbol{v_5}$ *and* $\boldsymbol{v_6}$ *are influenced by Bones 4 and 5 (blue bounding boxes). Bone 3 influences the position of* $\boldsymbol{v_2}$ *and* $\boldsymbol{v_3}$*, and Bone 2 has minor influence on* $\boldsymbol{v_3}$ *(red and purple boxes).*



(d) *Combining all vertex bounding boxes which are axis-aligned w.r.t. the coordinate system of Bone 4 to a common bounding box can be trivially computed like described in Figure 3. Performing this procedure yields conservative spatio-temporal bounds* B_m *of meshlet m for the animation between the two different skeleton poses when animated from 4a to 4b.*

Figure 4: *In this example, meshlet m represents a part of a skinned mesh's skin. It consists of vertices* $\boldsymbol{v_1} \ldots \boldsymbol{v_8}$ *each of which has one or multiple weighted bone assignments. According to those, vertices are moved when the skeleton's pose changes between the state in Figure 4a and the state in Figures 4b to 4d. The final bounding box* B_m *is computed relative to the coordinate system of the bone which has the highest combined influence on the vertices—which is Bone 4 in this case, the principal bone of meshlet m.*

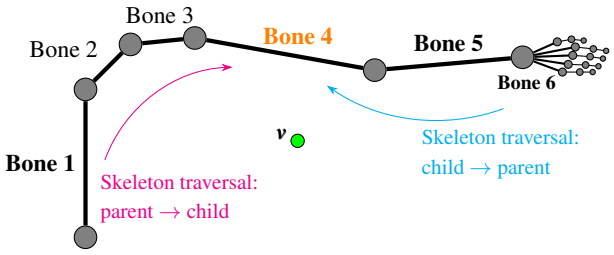


Figure 5: *Assume a vertex* $\boldsymbol{v}$ *with non-zero weights w.r.t. four bones: Bone 1, Bone 4, Bone 5, and Bone 6. To compute the combined bounds of* $\boldsymbol{v}$ *w.r.t. a given target bone (Bone 4), we compute a bounding box for each bone that influences* $\boldsymbol{v}$ *during our algorithm's first step, accumulating transformations along the bone hierarchy towards the target bone. To compute bounds w.r.t. a single bone, we start at that bone and traverse the skeleton until we reach the target. When computing the bounds of Bone 1, we must regard the transformations of Bones 2 and 3 in the given animation interval, even if they have no direct influence on* $\boldsymbol{v}$*. Similarly, when computing the bounds of Bone 6, we must include the transformations of Bone 5. Even though Bone 5 is contained in the path from Bone 6 to Bone 4, separate bounding boxes for Bone 5 and Bone 6 must be computed.*

its parent bone), the transformation in the inverse direction must be reformulated as stated in Equation (2)

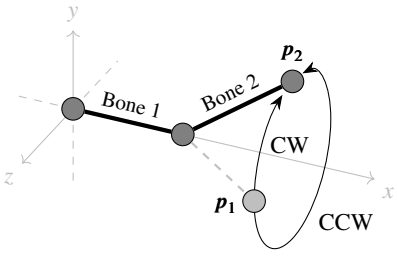
$$\boldsymbol{v'} = (\boldsymbol{P}_{b_i}\, \boldsymbol{T}\, \boldsymbol{R}\, \boldsymbol{S})^{-1}\, \boldsymbol{v} = \boldsymbol{S}^{-1}\, \boldsymbol{R}^{-1}\, \boldsymbol{T}^{-1}\, \boldsymbol{P}_{b_i}^{-1}\, \boldsymbol{v} \tag{2}$$

to correctly apply the separate transformations step-wise. We chose matrix notation for the sake of brevity and clarity, but different forms are practicable as well—most notably using unit quaternions for applying rotations.

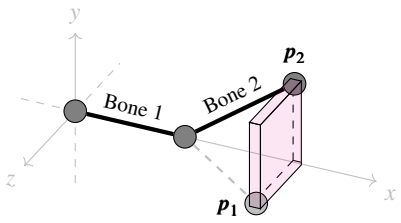
If an AABB $B_{\boldsymbol{v}b_j}$ is represented by two vectors—one for its minimum coordinates and the second for its maximum coordinates—it can be transformed conservatively as follows:

- $B_{\boldsymbol{v}b_j}$ is translated by adding the translation vector to its minimum and maximum coordinates.
- $B_{\boldsymbol{v}b_j}$ is rotated by rotating each of its corners and constructing a new AABB $B'_{\boldsymbol{v}b_j}$ from the results.
- $B_{\boldsymbol{v}b_j}$ is scaled by component-wise multiplication of its minimum and maximum coordinates with the scaling vector.
- $B_{\boldsymbol{v}b_j}$ is transformed by a matrix by constructing a new AABB $B'_{\boldsymbol{v}b_j}$ from the matrix-transformed corners.

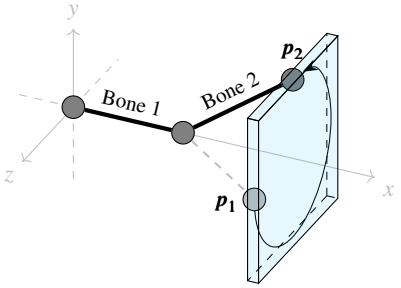
Extending the bounds by translation, rotation, and scale values as required in steps 1.b. and 1.(c.)iii. of our algorithm can be trivially computed for translation and scaling, but not for rotations:



(a) *When rotating a certain point from an initial position p_1 to its target position p_2, the rotation can be performed either in CW direction or in CCW direction about the axis along Bone 1.*

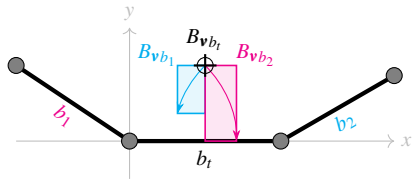


(b) *To compute a conservative bounding box that encompasses all possible positions that the rotated point can take along the rotation path, it is insufficient to consider its initial/final positions.*

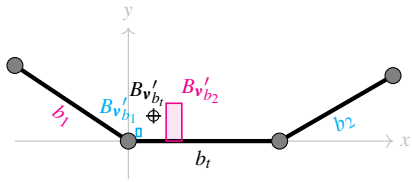


(c) *If an animation is defined s.t. p_1 is rotated to p_2 in CCW rotation direction, its actual conservative spatio-temporal bounding box is much larger than the incorrect one depicted in Figure 6b.*

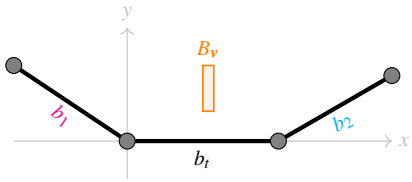
Figure 6: *These illustrations show that it is essential to regard the actual rotation paths for computing conservative spatio-temporal AABBs. It is insufficient to only consider initial and final positions since the maximum rotation extents might occur elsewhere.*



(a) *All AABBs have been transformed into the coordinate system of bone b_t. The arrows indicate that the bounding boxes were created based on a certain rotation relative to b_t. $B_{\mathbf{v}b_t}$ is an AABB with zero volume.*



(b) *Based on bone weights $w_1 = \frac{1}{2}$, $w_t = \frac{1}{3}$, and $w_2 = \frac{1}{6}$, intermediate AABBs $B'_{\mathbf{v}b_1}$, $B'_{\mathbf{v}b_t}$, and $B'_{\mathbf{v}b_2}$ are calculated by multiplying each original AABB's minimum and maximum coordinates by its respective weight.*



(c) *For an AABB that represents the combined conservative spatio-temporal bounds $B_{\mathbf{v}}$ for vertex $\mathbf{v}$ under LBS, all the intermediate bounds' ($B'_{\mathbf{v}b_1}$, $B'_{\mathbf{v}b_t}$, and $B'_{\mathbf{v}b_2}$) respective minimum and maximum coordinates are summed up.*

Figure 7: *Three bone-specific spatio-temporal AABBs are shown which were created by our algorithm to include all possible positions vertex $\mathbf{v}$ can occupy between two animation times. Bone-specific vertex AABBs $B_{\mathbf{v}b_1}$, $B_{\mathbf{v}b_t}$, and $B_{\mathbf{v}b_2}$ are combined into AABB $B_{\mathbf{v}}$ by taking the bone weights of vertex $\mathbf{v}$ into account and following the steps described in 7a, 7b, and 7c in that order. $B_{\mathbf{v}}$ represents conservative bounds for LBS.*

- $B_{\mathbf{v}b_j}$ is extended by translation through extending the bounding box by both, the translated minimum and maximum coordinates.
- $B_{\mathbf{v}b_j}$ is extended by scaling through extending the bounding box by both, the scaled minimum and maximum coordinates.
- $B_{\mathbf{v}b_j}$ is extended by rotation through the method described in Section 3.2.

3.2. Computing the Maximum Rotation Extents

For rotations, we need to take situations like depicted in Figure 2b into consideration and prevent missing any possible location that a rotated point could occupy. Figure 6 illustrates this problem with a different example: Rotating Bone 2 about the axis described by its parent Bone 1 lets a rotated point p_1 end up in a certain end position p_2. Spanning a bounding box only over the initial and final positions, p_1 and p_2, can lead to incorrect boundaries. Figure 6c shows the correct bounding box for the given scenario which can only be computed by regarding all possible locations along the circular segment described by the rotation, or—in the case of axis-aligned data structures—by considering those particular rotations that lead to maximum extents in each of the principal axes' directions.

To find and efficiently compute the maximum extents of a given rotation, we present a solution based on *Rodrigues' rotation formula* [Rod40], which computes the result $\mathbf{v}'$ of rotating a vector $\mathbf{v}$ by a given angle θ about a given (normalized) axis of rotation $\mathbf{n}$. Rotation transforms within a skeleton are often specified via unit quaternions which can be converted into angle-axis representation [Sho85]. Thus, Rodrigues' rotation formula is applicable. It is stated in Equation (3):

$$\mathbf{v}' = \mathbf{v}\cos\theta + (\mathbf{n}\times\mathbf{v})\sin\theta + \mathbf{n}(\mathbf{n}\cdot\mathbf{v})(1-\cos\theta). \tag{3}$$

We use its first-order derivative by θ to find those angles that lead to maximum extents in each of the principal axes' directions. Setting that first-order derivative of Equation (3) by θ to zero in order to find the extrema results in Equation (4)

$$\mathbf{x}_\theta = -\tan^{-1}\frac{\mathbf{n}\times\mathbf{v}}{\mathbf{n}(\mathbf{n}\cdot\mathbf{v})-\mathbf{v}}, \tag{4}$$

which yields a vector of angles $\mathbf{x}_\theta$ in radians that represents the rotation angles which lead to maximum extents in each principal axis direction. Please note that the operations in Equation (4) mean component-wise application of the division and $\tan^{-1}$.

An AABB $B_{\mathbf{v}}$ can be extended to encompass all the possible positions of $\mathbf{v}$ rotated by angle θ about axis $\mathbf{n}$ by calculating nine rotation angles:

- $\phi_1 = \text{clamp}(\boldsymbol{x}_{\theta_x}\ \ \ \ , \min(\theta,0), \max(0,\theta))$
- $\phi_2 = \text{clamp}(\boldsymbol{x}_{\theta_x} - \pi, \min(\theta,0), \max(0,\theta))$
- $\phi_3 = \text{clamp}(\boldsymbol{x}_{\theta_x} + \pi, \min(\theta,0), \max(0,\theta))$
- $\phi_4 = \text{clamp}(\boldsymbol{x}_{\theta_y}\ \ \ \ , \min(\theta,0), \max(0,\theta))$
- $\phi_5 = \text{clamp}(\boldsymbol{x}_{\theta_y} - \pi, \min(\theta,0), \max(0,\theta))$
- $\phi_6 = \text{clamp}(\boldsymbol{x}_{\theta_y} + \pi, \min(\theta,0), \max(0,\theta))$
- $\phi_7 = \text{clamp}(\boldsymbol{x}_{\theta_z}\ \ \ \ , \min(\theta,0), \max(0,\theta))$
- $\phi_8 = \text{clamp}(\boldsymbol{x}_{\theta_z} - \pi, \min(\theta,0), \max(0,\theta))$
- $\phi_9 = \text{clamp}(\boldsymbol{x}_{\theta_z} + \pi, \min(\theta,0), \max(0,\theta))$

and extending $B_{\boldsymbol{v}}$ by the results of Equation (3), calculated with vector $\boldsymbol{v}$, (normalized) axis of rotation $\boldsymbol{n}$, and each of these nine rotation angles. The angles are clamped to the range $[\theta, 0]$ or $[0, \theta]$, depending on the sign of θ, to keep the resulting bounding box as tight as possible—yet conservative—around the original position $\boldsymbol{v}$.

3.3. Vertex Bounds Combination for LBS

The steps described in Section 3.1 yield a number of conservative spatio-temporal vertex bounds $B_{\boldsymbol{v}b_i} \dots B_{\boldsymbol{v}b_n}$, each representing all possible positions between two animation times t_1 and t_2 as if the respective bone was the single bone of influence (i.e., if it had a weighting of 1) on $\boldsymbol{v}$. Each $B_{\boldsymbol{v}b_i}$ is given in the same space, namely the coordinate system of a uniformly selected principal bone for all the vertices associated to meshlet m, which we called PBS.

We propose the approach depicted in Figure 7 for computing the combined, weighted vertex bounds $B_{\boldsymbol{v}}$ that are suitable for LBS:

1. For each vertex' AABB $B_{\boldsymbol{v}b_i}$, multiply its minimum and maximum coordinates with its corresponding bone-weighting w_i.
2. Add the resulting minimum and maximum coordinates of all $B_{\boldsymbol{v}b_i} \dots B_{\boldsymbol{v}b_n}$ as computed in step 1., which yields $B_{\boldsymbol{v}}$.

The final step in our algorithm for computing conservative spatio-temporal meshlet bounds B_m is the combination of its associated vertices' AABBs $B_{\boldsymbol{v}_1} \dots B_{\boldsymbol{v}_n}$ as illustrated in Figure 3.

4. Normals Distribution of Meshlets

In the previous Sections 3.1 to 3.3, we have addressed the computation of conservative spatio-temporal meshlet bounds which can be used to enable view frustum culling. In this section, we describe how these bounds can be utilized to compute a conservative estimation for a meshlet's normals distribution to ultimately enable backface culling. Our algorithm consists of the following steps:

1. Determine an initial normal $\boldsymbol{n}_m$ and an initial normals distribution angle α w.r.t. $\boldsymbol{n}_m$ for meshlet m.
2. For each triangle associated to m, consider its vertices' conservative spatio-temporal AABBs $B_{\boldsymbol{v}_1}$, $B_{\boldsymbol{v}_2}$, and $B_{\boldsymbol{v}_3}$.
3. Optionally: Test if a plane can be found which divides space s.t. $B_{\boldsymbol{v}_i}$ and the respective other two AABBs lie on opposite sides of it. If such a plane cannot be found, abort normals distribution calculation for m, otherwise continue.
4. For each combination of corners $c_{\boldsymbol{v}_1} \in B_{\boldsymbol{v}_1}$, $c_{\boldsymbol{v}_2} \in B_{\boldsymbol{v}_2}$, $c_{\boldsymbol{v}_3} \in B_{\boldsymbol{v}_3}$:
   a. Compute $\boldsymbol{n}_i = (c_{\boldsymbol{v}_2 i} - c_{\boldsymbol{v}_1 i}) \times (c_{\boldsymbol{v}_3 i} - c_{\boldsymbol{v}_2 i})$
   b. Compute the angle between $\boldsymbol{n}_i$ and $\boldsymbol{n}_m$, and store the maximum angle α_{max} from all combinations of corners.
5. Store α_{max} and use it for backface culling computations for m.

Initial $\boldsymbol{n}_m$ and α values (step 1.) are computed from a resting or configuration pose. By taking the maximum angle during step 4.b., a conservative normals distribution is calculated for a given animation interval. As an optional, potentially performance-improving step 3., we propose the approach outlined in Figure 8: If a set of $B_{\boldsymbol{v}_1}$, $B_{\boldsymbol{v}_2}$, and $B_{\boldsymbol{v}_3}$ does not fulfill the requirement described in Figure 8a, their normals distribution might encompass the whole sphere of normals. If the requirement is fulfilled, we can be sure that a useful α_{max} can be found like illustrated in Figure 8d.

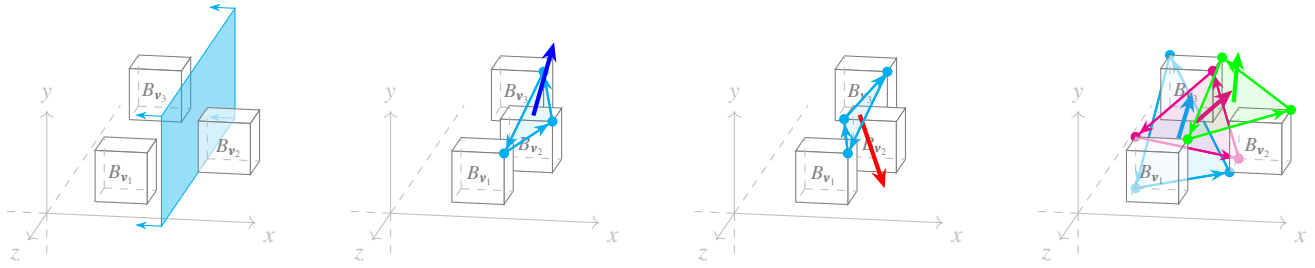
5. Rendering Strategies Using Meshlet Bounds for Culling

Computing meshlet bounds as described in the previous sections is a computationally elaborate task that requires computation times in the order of milliseconds to seconds per time interval for models of similar detail to our test models. It is parallelizable well and suitable for multithreaded CPU implementations as well as GPU implementations. The following list shows the average computation times for computing AABBs for all meshlets for a given time interval. The computations were performed with 24 parallel threads on an AMD Zen 2 CPU at 3.8 GHz. Times are reported in seconds for the creation of vertex and normal bounds:

Model	Spatial Extents	Orientations	Total Time
Gawain	0.25s	0.19s	0.44s
Giant Worm	0.37s	0.28s	0.65s
Butcher	0.19s	0.40s	0.59s
Wyvern	0.56s	0.45s	1.00s

Hence, we propose using a precomputation step for computing the meshlet bounds. However, another essential question is how the precomputed bounds (defined per meshlet, animation clip, and time interval) shall be used during rendering. One possible application is storing the AABBs in GPU buffers and computing the correct lookup index in a task shader. Taking the current animation time into account, we may then read the AABB from the buffer and evaluate the bounds against the current view frustum. While this might appear like a feasible idea at first glance, it is a strategy that we advise against for several reasons: Animated models can contain a large number of keyframes, easily ranging in the hundreds or thousands. Since memory consumption would be in linear relation with the number of keyframes, this would lead to extensive memory usage and incur additional delays in task shader executions for the memory transfer.

As a better strategy, we propose to use a precomputation step to answer a simple question for each meshlet, namely: "Across the entirety of an animation clip, what is the maximum deviation of a meshlet's bounds w.r.t. certain predefined reference bounds?". This approach is illustrated in Figure 9 for spatial bounds. We choose the bounding sphere that encompasses all of a meshlet's vertices in its initial "bind pose" or "T-pose" as the fixed reference per meshlet. When bone-animated, the sphere's center and radius are transformed with the principal bone's transformation matrix, which in general can lead to states where the transformed bounding sphere no longer encompasses all its assigned vertices as illustrated in Figure 9b. Based on our conservative spatio-temporal AABB B_m for that specific animation state (computed as described in Figure 4), we can compute a factor by how much the transformed reference bounding



(a) *We can compute a useful normals distribution if, for each AABB, we can find a plane so that it lies on one side of the plane, and the other two AABBs lie on the other side.*

(b) *If vertex AABBs are positioned in an unfavorable way w.r.t. each other, we cannot find a useful normals distribution (compare with Figure 8c).*

(c) *For suboptimally positioned bounding boxes, different combinations of $B_{\mathbf{v}_1}$'s, $B_{\mathbf{v}_2}$'s, and $B_{\mathbf{v}_3}$'s corners lead to normals pointing in opposite directions.*

(d) *If AABBs are positioned favorably, we can use the 8^3 combinations of $B_{\mathbf{v}_1}$'s, $B_{\mathbf{v}_2}$'s, and $B_{\mathbf{v}_3}$'s corners to compute extreme normal deviations for conservative meshlet bounds.*

Figure 8: *A conservative normals distribution of a meshlet can be found by analyzing each of its triangles separately. For each triangle, a total number of 8^3 possible normal directions are created by computing the face normals of each triangle that can be constructed from a combination of the bounding box corners from the vertices that describe the triangle, as illustrated in Figure 8d. In this way, extreme normal deviations are computed and put in relation to a reference normal $\boldsymbol{n}_m$. The triangle-specific order among $B_{\mathbf{v}_1}$, $B_{\mathbf{v}_2}$, and $B_{\mathbf{v}_3}$ must be maintained for these computations. A conservative test of whether a useful normals distribution can be calculated is presented in Figure 8a. If the vertices' AABBs are positioned in an unfavorable manner w.r.t. each other, the normals distribution might encompass the whole sphere of directions as described in Figures 8b and 8c.*

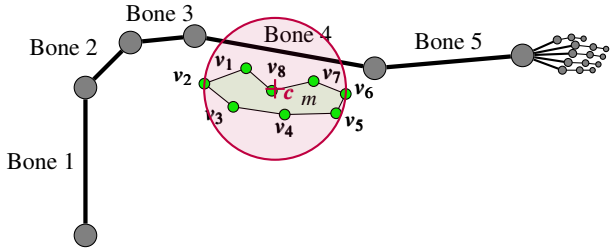
sphere's radius has to be extended in order to also encompass all positions of B_m. Having computed the maximum required radius across all time intervals of an animation clip, it suffices to store a meshlet's initial bounding sphere's center point $\boldsymbol{c}$ and the extended radius per meshlet.

During rendering, very little computational overhead is required: $\boldsymbol{c}$ and its extended radius are transformed by the meshlet's principal bone matrix for the current animation state, further transformed into the same space of the view frustum's planes, and tested against the frustum planes. We might also use AABBs for spatial bounds during rendering, but spheres incur significantly less computational overhead when culling against a view frustum. Using spheres, only six plane-to-point distance computations (one for each of the six view frustum planes) have to be performed in a task shader, whereas bounding boxes demand computation of 6×8 plane-to-point distances.

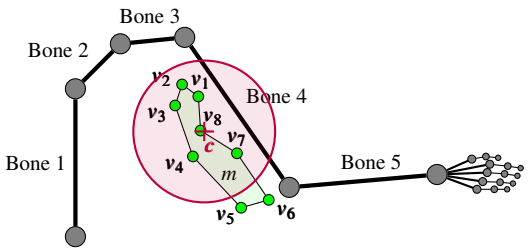
The quality of the radius extension factor can be further improved by analyzing ever-smaller time intervals. The largest possible time interval to evaluate is from one keyframe time t_1 within a certain animation clip to its subsequent keyframe time t_2, because we may not jump over diverging transformations in order to remain conservative. If we determine that the spatio-temporal bounds between the two keyframes do not satisfy our quality requirements, we can start to adaptively subdivide the time interval for bounds computation. In general, we can assume that if a meshlet contains vertices that are influenced by bones other than the meshlet's principal bone, subdividing the time intervals between t_1 and t_2 leads to smaller vertex bounds and consequently to tighter meshlet bounds. The maxim of our algorithm is to compute *conservative* bounds, which is why we cannot disregard the temporal influence of an animation on the bounds. The subdivision approach allows to close in on the theoretical minimum bounds, while constantly staying conservative. This way, the generation of vertex bounds naturally adapts to the shape of the position function of seemingly arbitrarily moving vertices, such as the one illustrated in Figure 2b, placing more emphasis on ranges with a strong variation.

For the normals distribution (as described in Section 4), a similar strategy can be employed: The mean normal direction of an initial "bind pose" or "T-pose" serves as the reference normal $\boldsymbol{n}_m$ for meshlet m. In the same vein of finding a maximum radius extension factor, for the normals distribution a maximum angle-deviation α_{max} can be determined by analyzing all keyframe or sub-keyframe intervals. Smaller animation time intervals generally lead to smaller values for α_{max}. During rendering, the computation of whether or not m can be conservatively backface culled can be evaluated using $\boldsymbol{c}$, the extended radius, $\boldsymbol{n}_m$, α_{max}, and the camera's position.

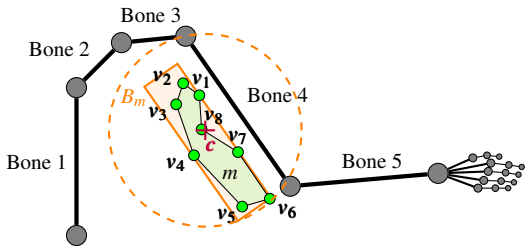
A variation of the strategy described above is to not store individual values for the extended radius and α_{max} per meshlet but to define constant values for them, and determine in a precomputation step which meshlets satisfy these quality requirements. I.e., this strategy would answer the questions: "Which meshlets satisfy the requirement that their bounding sphere's radius does not have to be extended by more than a constant scaling factor s_r s.t. all of its vertices stay within the extended bounding sphere?", and regarding the normals distribution: "Which meshlets satisfy the requirement that their α_{max} is smaller than a constant α_T?" This approach has the advantage that constant values are used for s_r and α_T and do not have to be read per meshlet, thus helping to reduce the required memory bandwidth. Low-overhead shaders can be used to render those meshlets which do not fulfill any of the two criteria. Meshlets that fulfill one criterion can be rendered with the appropriate culling code. Shaders including both, BFC and VFC code, can be used for meshlets that fulfill both requirements. The subdivision approach described above can be used to determine more meshlets as being of sufficient quality for both or either of the criteria. We employed this approach for the setup of our benchmarks presented in Section 7.



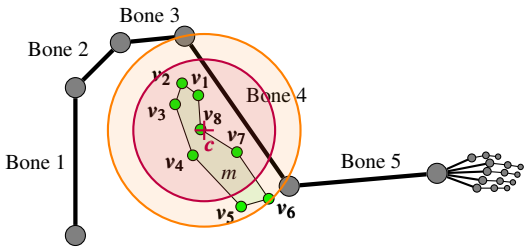
(a) *An initial bounding sphere (with center* $\boldsymbol{c}$ *and radius r) is computed s.t. it encompasses all vertices that are assigned to meshlet m and represents the bounds for one specific initial pose only.*



(b) *Transforming* $\boldsymbol{c}$ *and r with m's principal bone (Bone 4) at a certain animation state reveals that the transformed initial bounding sphere not necessarily encompasses all of m's vertices any longer.*



(c) *By using m's conservative spatio-temporal bounds* B_m*—computed as described in Figure 4—we compute a factor by how much r must be extended to also encompass the entirety of* B_m*.*



(d) *By computing m's extended radii for all animation intervals and storing the maximum radius together with the original* $\boldsymbol{c}$*, a conservative spatio-temporal bounding sphere can be computed for m.*

Figure 9: *In order to minimize computational overhead during rendering, we propose to use a meshlet m's spatio-temporal AABB* B_m *for computing a scaling factor which tells how much* B_m*'s extents w.r.t. an initially created reference bounding sphere has grown or shrunk. The maximum scaling factor across all possible poses represents the conservative bounding sphere of meshlet m.*

6. Implementation Details

We have implemented the algorithms presented in the previous sections using C++ and Vulkan. Computing conservative vertex bounds, combining them into conservative meshlet bounds, and computing conservative normal bounds per meshlet are implemented in a CPU-based precomputation step. Our current implementation uses the same amount of parallel threads for this step as the number of logical processors reported by the operating system.

During our precomputation step, we evaluate each meshlet m_i against predefined maximum values s_r (referring to a maximum scaling factor w.r.t. m_i's initial radius r_i as described in Figure 9) and α_T (referring to a maximal threshold for m_i's α_{max}, which is computed as described in Section 4). Based on this evaluation, we assign m_i to one of the following three categories:

- m_i is both, view-frustum cullable and backface cullable, if it satisfies both limits for all animation intervals of interest.
- m_i is view-frustum cullable but not backface cullable, if it satisfies the requirements w.r.t. s_r for all animation intervals of interest, but not the requirements w.r.t. α_T.
- m_i is neither view-frustum cullable nor backface cullable if it does not fulfill at least the requirement w.r.t. s_r.

Based on these categorizations, we issue a total number of three draw calls using different pipeline configurations for each of the three categories: Meshlets that are suitable for culling are rendered with pipelines that include culling code. The meshlets that have been deemed to not be cullable are rendered with a pipeline that does not include culling code, thus not suffering from the potential computational overhead caused by the additional culling instructions. In trying to minimize runtime overhead of our culling code we chose to go for the approach with constant values for s_r and α_T instead of storing and evaluating individual thresholds per meshlet.

Our GPU implementation is based on Vulkan [Kub18b] and GLSL [KB19]. Both types of culling are performed in task shaders. Task shaders operate in groups of 32 threads per warp [NVI18], where each of these threads tests a different meshlet in parallel. We use *ballot* shader instructions to synchronize the threads before passing on the information of how many and which meshlets have not been culled and are to be further processed by later shader stages. In the subsequent mesh shader stage, vertex skinning is performed for the meshlets that have survived culling. The vertices assigned to such meshlets are transformed with 32 parallel threads per meshlet. The mesh shader constitutes the final geometry processing stage, which means that its output is forwarded to the fixed-function rasterizer stage in graphics pipelines for further processing.

7. Results

We have evaluated the performance characteristics of our implementation (as described in Section 6) with different scene compositions consisting of multiple instances of the models shown in Figure 1 and arranged according to the scheme presented in Figure 10a. For all performance benchmarks, we measure the milliseconds of the relevant draw calls with GPU timer queries for 1000 frames after a warmup phase of 100 frames. The query results of the 1000 measured frames are averaged for the results. The camera remains

(a) *Overview of our benchmark scene* **(b)** *Camera setup during our benchmarks* **(c)** *Impact of VFC* **(d)** *Impact of BFC*

Figure 10: *In the test scene that we are using for our benchmarks, we position multiple instances of our test models, three of which are duplicated along the frustum planes of our camera, which is positioned as shown in 10b. We benchmark different geometry loads according to the model duplication scheme indicated in 10a. Benchmarks with view frustum culling enabled produce effects as shown in 10c along the frustum planes. 10d shows the effects of enabled backface culling on a per-meshlet basis (same camera positioning as in 10b).*

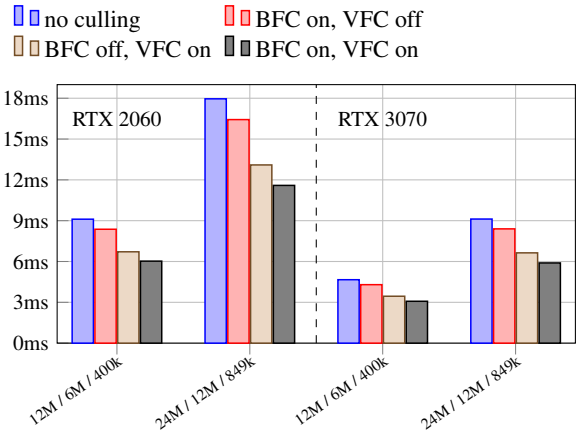


Figure 11: *GPU performance measurements (average milliseconds from GPU timer queries) are shown for two different GPUs and four different scene configurations. The x axis labels represent triples of number of vertices, number of triangles, and number of meshlets in that order. Each set of bars compares the results of a pipeline without culling code to the results of shader pipelines that include code for BFC only, VFC only, or both in their respective task shaders.*

GPU	Scene	BFC only		VFC only		BFC+VFC	
		Culled	Faster	Culled	Faster	Culled	Faster
RTX 2060	400k	11.4%	8.1%	31.3%	26.3%	39.9%	33.8%
	849k	11.5%	8.5%	31.4%	27.1%	39.7%	35.4%
RTX 3070	400k	11.4%	7.8%	31.3%	26.1%	39.9%	34.0%
	849k	11.5%	7.9%	31.4%	27.2%	39.7%	35.4%

Table 1: *This table shows the average percentage of culled meshlets (columns headed "Culled") during the benchmarked scene configurations from Figure 11 and the performance increase that resulted from culling them, which means the reduction of render time in percent (columns headed "Faster"). "400k" refers to the 12M / 6M / 400k config, and "849k" refers to the 24M / 12M / 849k config (numbers of vertices, triangles, and meshlets, respectively).*

stationary during our benchmarks at the position that Figure 10b has been captured from, while the models constantly animate, thus constantly varying positions and orientations of meshlets.

Figure 11 presents the general picture of the performance characteristics from our benchmarks, comparing culling-enabled pipelines to pipelines without culling code. Including culling code in task shaders constitutes a certain computational overhead compared to pipelines that do not include such code. The additional overhead can be more than made up for across all of our test cases and across different GPUs. Table 1 lists the average performance increases for the different benchmarks and shows the percentages of meshlets that could be culled. Assuming that the maximum reduction in render time is bounded by the percentage of culled meshlets, we can state that the pipelines implementing our technique stay within a margin of only a few percent to the theoretical optimum in our tests. With BFC and VFC enabled, we measured a culling ratio of approximately 40% for the scene described in Figure 10, achieving reductions of render times of up to 35.4%. Consistent performance patterns can be observed across both tested GPUs. To generate the data for the results presented in Figure 11, we employed a precomputation step which we configured with a run-time limit of 15 minutes per model. It gradually refines the bounds within the given time limit, starting with keyframe boundaries and subdividing them until the time limit has been reached. The resulting meshlet classification details per model are shown in Table 2. The data that we used for our benchmarks are stated under the columns to $\alpha_T = 20°$, listing the amounts of meshlets which were rendered with the "BFC on, VFC on" pipeline ("both"), with the "BFC off, VFC on" pipeline (s_r), and with a "no culling" pipeline ("none").

Performance analyses of backface culling only are presented in Table 3. The performance increase rises with the number of meshlets that are classified to be backface-cullable. If only as little as approximately 20% of meshlets are backface cullable, the additional overhead of the included culling code counteracts its potential benefits. Backface culling in task shaders has shown to have the potential of an additional reduction of render times by 11.4% in our test scenes. In Section 5 we have described that our precomputation step can be used for gradual refinement of meshlet classification. By evaluating smaller animation subintervals, steadily tighter con-

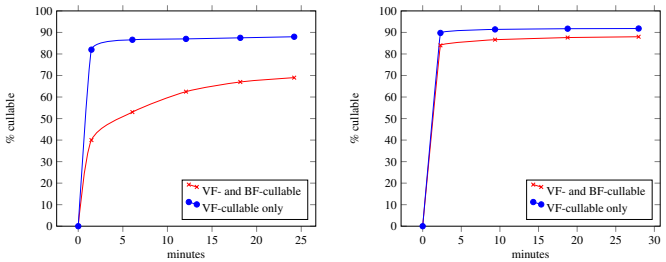
Model	$\alpha_T = 10°$ both	$\alpha_T = 10°$ s_r	$\alpha_T = 20°$ both	$\alpha_T = 20°$ s_r	$\alpha_T = 30°$ both	$\alpha_T = 30°$ s_r	none
GAWAIN	82%	17%	89%	10%	92%	7%	1%
GIANT WORM	66%	29%	71%	24%	74%	21%	5%
BUTCHER	88%	6%	90%	4%	91%	3%	6%
WYVERN	52%	42%	60%	34%	65%	29%	6%

Table 2: *Classification percentages from different models using a 15 minute time limit for each. The percentage values in columns labeled with "both" refer to meshlets that fulfill the requirements to be both, view-frustum cullable and backface cullable—i.e., stay below a radius scale factor s_r and within a normal deviation threshold of α_T. The values in "s_r" columns represent the number of meshlets that only fulfill the requirement of staying below the radius scale factor. The number of meshlets that do not fulfill the requirements are listed in column "none". Using higher values for the normal deviation threshold α_T results in more meshlets satisfying "both" requirements at the cost of less optimal backface culling performance during rendering. $s_r = 3$ was used for generating this classification.*

Cullable	BFC off	BFC on	Culled	Faster
100%	7.79ms	**6.90ms**	15.7%	11.4%
80%	7.80ms	**7.12ms**	12.5%	8.7%
60%	7.80ms	**7.35ms**	9.2%	5.7%
40%	7.79ms	**7.51ms**	7.0%	3.6%
20%	7.79ms	**7.77ms**	3.8%	0.3%
0%	**7.80ms**	8.07ms	0.0%	-3.4%

Table 3: *Average render times of scenes composed of different ratios of meshlets which are backface-cullable to meshlets that do not satisfy that requirement. For these measurements, view frustum culling was disabled. Only backface culling code is active in task shaders, producing the results listed under the "BFC on" column, while the pipeline used to create the results found under the "BFC off" column does not include any culling code. The average percentage of meshlets that were culled by BFC code is listed in the column "Culled" and the resulting reduction of render time in percent is listed under "Faster". A scene setup with 731k meshlets was used.*

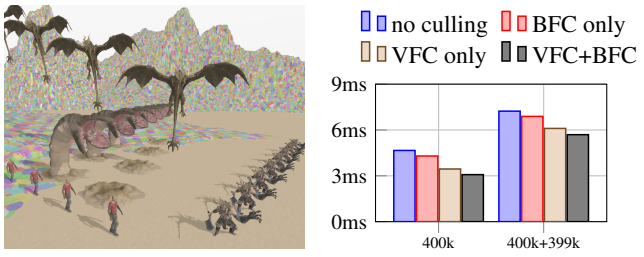
servative bounds can be found for meshlets, eventually leading to a higher number of meshlets being cullable during run time. As our results in Tables 1 and 3 attest, performance increases proportionally to the number of meshlets that could be culled in task shaders. Therefore, increasing the number of meshlets that *can* be culled benefits render times accordingly. The trade-offs between precomputation time and resulting meshlet classification percentages are shown in Figure 12. It can be seen that different outcomes must be expected from different animated models and their animation clips. While the initial classification values of the BUTCHER model show high percentages of cullable meshlets already after relatively short precomputation times (i.e., small to no subdivisions of keyframe intervals), different characteristics can be observed for the GIANT WORM model. Increased computational effort in the precomputation step leads to significantly higher numbers of backface cullable meshlets for GIANT WORM. Tighter target bounds for s_r and α_T have been chosen to emphasize the effects of gradual refinement during precomputation.



(a) *Effect of gradual bounds refinement for* GIANT WORM *clips.* **(b)** *Effect of gradual bounds refinement for* BUTCHER *clips.*

Figure 12: *Effect of different time targets for our precomputation step on the classification into meshlets that are both view-frustum cullable and backface cullable, view-frustum cullable only, or neither. $s_r = 2$ and $\alpha_T = 10°$ were chosen for these measurements.*

We further evaluate the effects of added culling for animated meshlets in a scenario that also includes static models. Figure 13 shows a scene configuration where in addition to 400k animated meshlets, 399k static-geometry meshlets are rendered. In these tests, static meshes are also drawn via task and mesh shaders and always culled with established VFC and BFC methods. The resulting performance measurements of the *combined* render times are shown in Figure 13b. The additional static meshlets raise the total render time on an RTX 3070 from 4.66ms to 7.23ms if animated meshlets are not culled. The same scene configuration with VFC and BFC enabled for animated meshlets reduces the total render time to 5.69ms, which constitutes an average reduction of combined render time by 21.3%. Culling percentages remain the same as stated in Table 1 for 400k animated meshlets. While we kept vertex processing effort at the minimum for both types of geometry, animated models still require significantly more vertex processing than static models due to skinning code in mesh shaders, highlighting the benefit of our approach for scenes with moderate to high amounts of animation.



(a) *Additional static-geometry meshlets (terrain, meshlets colored) are rendered for this benchmark.* **(b)** *Render times of 400k animated meshlets compared to render times of 400k animated + 399k static meshlets.*

Figure 13: *Mixed static/animated scenario. 13b shows average render times in milliseconds, comparing the results of the 400k measurement on RTX 3070 from Figure 11 with the measurements of the same setup, plus additional 399k static-geometry meshlets. Again, our technique leads to significant render time reductions.*

8. Discussion and Future Work

We have presented an algorithm to compute conservative spatio-temporal bounds on a per-meshlet basis. Using the spatio-temporal vertex bounds of its assigned vertices, also a conservative estimate for a meshlet's normals distribution can be computed. Bounds and normals distributions are intended to be computed during a flexible precomputation step which allows to trade tighter bounds or normal deviation angles for reduced precomputation time. In all cases, our algorithm ensures conservative results.

Adding culling to task shaders incurs some additional computational overhead of a few percent during rendering. This disadvantage can in general be more than made up for in our tests. VFC on a per-meshlet basis enables fine-grained culling of meshlets outside of the view frustum and can lead to significant reductions of render time. The benefit of including BFC in task shaders depends on the quality of meshlets insofar as many of them should be backface cullable. If the precomputation step manages to compute conservative normals distributions for close to 100% of meshlets, render time reductions of up to 11.4% are possible through BFC in graphics pipelines with very light vertex processing load and can be expected to be significantly higher with graphics pipelines that feature complex vertex processing load. The benefit of combined BFC and VFC was close to the theoretical optimum in our tests when comparing the relative reduction of render time to the percentage of culled meshlets.

Naturally, model animation is a far-reaching and complex application field. In this work, we have derived and presented a solution suitable for bounding individual animation clips. However, we note that our basic approach may easily be extended for use with a variety of techniques. For example, inverse kinematics (IK) is a common method in modern real-time animation. For pipelines that involve IK, we can reuse the same techniques presented in this paper, but instead of subdividing and bounding vertex motion across time intervals, we can instead bound a different parameter space, such as the solid angles representing ranges of possible orientations for a set of joints. Other important techniques, such as the blending of animation clips, can be addressed by not computing bounds for individual clips, but instead for the full repertoire of possible animations. If intermediate vertex states are produced from linearly blending between animations, conservative vertex bounds are then easily obtained from the union of all animations, and the bounding of the normal cone can be performed as previously described.

With the addition and ongoing development of hardware-accelerated ray-tracing, the use of already-computed spatial acceleration structures for ray-tracing might be considered as a viable, hierarchical alternative in the future. In contrast to *currently* available data structures, our approach serves to compute conservative bounds over arbitrary time intervals and does not require random access to meshlet data, as it must be expected with ray tracing. Instead, meshlet data is accessed in a strictly contiguous manner, not dissimilar to vertex attribute streaming in conventional rasterization-based graphics pipelines, hence the available data structures with logarithmic access times are unfavorable in this case. In a similar vein, hierarchical data structures such as bounding sphere trees [JP04; KŽ05a; SBT06] could represent a possible avenue for increasing the performance of our precomputation step by decreasing its computational cost from $\#vertices \times \#joints$ down to $\#meshlets \times \#joints$. However, since bounding spheres are a less accurate representation than bounding boxes, and since the meshlet-focused approach would overestimate bounds even more, we decided on sticking with the more accurate approach of computing bounding boxes per vertex.

In the future, we will investigate further options for accelerating the precomputation step and allow further tradeoff options. We also aim to support further skinning methods—such as dual-quaternion skinning—with our conservative bounds besides LBS. However, we note that the dependency on a particular skinning method is comparably small with our presented approach: the only missing piece for enabling different skinning techniques is the derivation of conservative positional bounds for a single vertex between two successive keyframes. Once derived, the corresponding methods can be supplied as a drop-in replacement for the current solution for LBS. Our approach for robust meshlet bounds can therefore work with any skinning technique for which such bounds can be found.

Acknowledgements

We thank Unity Technologies for providing the GAWAIN model through their "The Heretic: Digital Human" package. We would like to thank the anonymous reviewers of this paper and Lukas Geyer for their valuable feedback, as well as Thorsten Korpitsch for his help with 3D modelling and implementation. This work was supported by the Research Cluster "Smart Communities and Technologies (Smart CT)" at TU Wien.

References

[AA00] ABRAMS, STEVEN and ALLEN, PETER K. "Computing swept volumes". *The Journal of Visualization and Computer Animation* 11.2 (2000), 69–82 3.

[AHA15] ANDERSSON, MAGNUS, HASSELGREN, JON, and AKENINE-MÖLLER, TOMAS. "Masked Depth Culling for Graphics Hardware". *ACM Trans. Graph.* 34.6 (Oct. 2015). ISSN: 0730-0301. DOI: 10.1145/2816795.2818138 3.

[Bla20] BLAKE-DAVIES, ALEXANDER. *Next-Generation Gaming with AMD RDNA 2 and DirectX 12 Ultimate*. https://community.amd.com/t5/blogs/next-generation-gaming-with-amd-rdna-2-and-directx-12-ultimate/ba-p/427032. [Accessed 12-April-2021]. 2020 1.

[BP07] BARAN, ILYA and POPOVIĆ, JOVAN. "Automatic Rigging and Animation of 3D Characters". *ACM Trans. Graph.* 26.3 (July 2007), 72–es. ISSN: 0730-0301. DOI: 10.1145/1276377.1276467 3.

[CM04] CORDIER, FREDERIC and MAGNENAT-THALMANN, NADIA. "A Data-Driven Approach for Real-Time Clothes Simulation". *Proceedings of the Computer Graphics and Applications, 12th Pacific Conference*. PG '04. USA: IEEE Computer Society, 2004, 257–266. ISBN: 0769522343 3.

[GFSS06] GÜNTHER, JOHANNES, FRIEDRICH, HEIKO, SEIDEL, HANS-PETER, and SLUSALLEK, PHILIPP. "Interactive ray tracing of skinned animations". *The Visual Computer* 22.9 (Sept. 2006), 785–792. ISSN: 1432-2315. DOI: 10.1007/s00371-006-0063-x 3.

[HA15] HAAR, ULRICH and AALTONEN, SEBASTIAN. "GPU-Driven Rendering Pipelines". Siggraph 2015: Advances in Real-Time Rendering in Games. 2015 3.

[HC11] HILL, STEPHEN and COLLIN, DANIEL. *Practical, Dynamic Visibility for Games*. https://blog.selfshadow.com/publications/practical-visibility/. 2011 3.

[Hop99] HOPPE, HUGUES. "Optimization of Mesh Locality for Transparent Vertex Caching". *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '99. 1999, 269–276. ISBN: 0201485605. DOI: 10.1145/311535.311565 3.

[HS17] HAN, SONGFANG and SANDER, PEDRO. "Triangle Reordering for Efficient Rendering in Complex Scenes". *Journal of Computer Graphics Techniques (JCGT)* 6.3 (Sept. 2017), 38–52. ISSN: 2331-7418 3.

[JP04] JAMES, DOUG L. and PAI, DINESH K. "BD-Tree: Output-Sensitive Collision Detection for Reduced Deformable Models". *ACM Transactions on Graphics (SIGGRAPH 2004)* 23.3 (Aug. 2004) 3, 12.

[Kap21] KAPOULKINE, ARSENY. *meshoptimizer, Mesh optimization library*. https://github.com/zeux/meshoptimizer. [Accessed: 13-April-2021]. 2016-2021 3.

[KB19] KUBISCH, CHRISTOPH and BROWN, PAT. *GLSL Mesh Shader Extension*. https://www.khronos.org/registry/OpenGL/extensions/NV/NV_mesh_shader.txt. [Accessed 2-June-2021]. 2019 3, 9.

[KCŽO07] KAVAN, LADISLAV, COLLINS, STEVEN, ŽÁRA, JIŘÍ, and O'SULLIVAN, CAROL. "Skinning with Dual Quaternions". *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*. I3D '07. Seattle, Washington: Association for Computing Machinery, 2007, 39–46. ISBN: 9781595936288. DOI: 10.1145/1230100.1230107 3.

[KKI*18] KERBL, BERNHARD, KENZEL, MICHAEL, IVANCHENKO, ELENA, et al. "Revisiting The Vertex Cache: Understanding and Optimizing Vertex Processing on the Modern GPU". *Proc. ACM Comput. Graph. Interact. Tech.* 1.2 (Aug. 2018). DOI: 10.1145/3233302 3.

[Koc20a] KOCH, DANIEL. *Vulkan Acceleration Structure Device Extension*. https://www.khronos.org/registry/vulkan/specs/1.2-extensions/man/html/VK_KHR_acceleration_structure.html. [Accessed 2-August-2021]. 2020 3.

[Koc20b] KOCH, DANIEL. *Vulkan Ray Query Device Extension*. https://www.khronos.org/registry/vulkan/specs/1.2-extensions/man/html/VK_KHR_ray_query.html. [Accessed 2-August-2021]. 2020 3.

[Koc21] KOCH, DANIEL. *GLSL Ray Query Extension*. https://github.com/KhronosGroup/GLSL/blob/master/extensions/ext/GLSL_EXT_ray_query.txt. [Accessed 2-August-2021]. 2021 3.

[KSW21] KARIS, BRIAN, STUBBE, RUNE, and WIHLIDAL, GRAHAM. "A Deep Dive into Nanite Virtualized Geometry". *ACM SIGGRAPH 2021 Courses, Advances in Real-Time Rendering in Games, Part 1*. https://advances.realtimerendering.com/s2021/index.html [Accessed 10-September-2021]. 2021 1.

[Kub18a] KUBISCH, CHRISTOPH. *Introduction to Turing Mesh Shaders*. https://developer.nvidia.com/blog/introduction-turing-mesh-shaders. [Accessed 12-April-2021]. 2018 1.

[Kub18b] KUBISCH, CHRISTOPH. *Vulkan Mesh Shader Device Extension*. https://www.khronos.org/registry/vulkan/specs/1.2-extensions/man/html/VK_NV_mesh_shader.html. [Accessed 2-June-2021]. 2018 3, 9.

[KVLM03] KIM, YOUNG, VARADHAN, GOKUL, LIN, MING, and MANOCHA, DINESH. "Fast swept volume approximation of complex polyhedral models". Jan. 2003, 11–22. DOI: 10.1145/781611.781613 3.

[KŽ05a] KAVAN, LADISLAV and ŽÁRA, JIŘÍ. "Fast Collision Detection for Skeletally Deformable Models". *Computer Graphics Forum* 24.3 (2005), 363–372 3, 12.

[KŽ05b] KAVAN, LADISLAV and ŽÁRA, JIŘÍ. "Spherical Blend Skinning: A Real-Time Deformation of Articulated Models". *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*. I3D '05. Washington, District of Columbia: Association for Computing Machinery, 2005, 9–16. ISBN: 1595930132. DOI: 10.1145/1053427.1053429 3.

[LH16] LE, BINH HUY and HODGINS, JESSICA K. "Real-Time Skeletal Skinning with Optimized Centers of Rotation". *ACM Trans. Graph.* 35.4 (July 2016). ISSN: 0730-0301. DOI: 10.1145/2897824.2925959 3.

[LY06] LIN, GANG and YU, THOMAS P. -Y. "An Improved Vertex Caching Scheme for 3D Mesh Rendering". *IEEE Transactions on Visualization and Computer Graphics* 12.4 (July 2006), 640–648. ISSN: 1077-2626. DOI: 10.1109/TVCG.2006.59 3.

[Mic21] MICROSOFT CORPORATION. *DirectX-Specs*. https://microsoft.github.io/DirectX-Specs. [Accessed 02-August-2021]. 2021 1.

[MLT89] MAGNENAT-THALMANN, N., LAPERRIÈRE, R., and THALMANN, D. "Joint-Dependent Local Deformations for Hand Animation and Object Grasping". *Proceedings on Graphics Interface '88*. Edmonton, Alberta, Canada: Canadian Information Processing Society, 1989, 26–33 3.

[NVI18] NVIDIA CORPORATION. *NVIDIA Turing GPU Architecture*. https://images.nvidia.com/aem-dam/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf. [Accessed 12-April-2021]. 2018 1, 3, 9.

[RLM04] REDON, S., LIN, M. C., and MANOCHA, D. "Fast Continuous Collision Detection for Articulated Models". *Solid Modeling*. Ed. by ELBER, GERSHON, PATRIKALAKIS, NICHOLAS, and BRUNET, PERE. The Eurographics Association, 2004. ISBN: 3-905673-55-X. DOI: 10.2312/sm.20041385 3.

[Rod40] RODRIGUES, OLINDE. "Des lois géométriques qui régissent les déplacements d'un système solide dans l'espace, et de la variation des coordonnées provenant de ces déplacements considérés indépendants des causes qui peuvent les produire". *Journal de Mathématiques Pures et Appliquées*. 1st ser. 5 (1840), 380–440 6.

[SBOT08] SHOPF, JEREMY, BARCZAK, JOSHUA, OAT, CHRISTOPHER, and TATARCHUK, NATALYA. "March of the Froblins: Simulation and Rendering Massive Crowds of Intelligent and Detailed Creatures on GPU". *ACM SIGGRAPH 2008 Games*. SIGGRAPH '08. Los Angeles, California: Association for Computing Machinery, 2008, 52–101. ISBN: 9781450378499. DOI: 10.1145/1404435.1404439 3.

[SBT06] SPILLMANN, J., BECKER, M., and TESCHNER, M. "Efficient Updates of Bounding Sphere Hierarchies for Geometrically Deformable Models". *Vriphys: 3rd Workshop in Virtual Realitiy, Interactions, and Physical Simulation*. Ed. by MENDOZA, CESAR and NAVAZO, ISABEL. The Eurographics Association, 2006. ISBN: 3-905673-61-4. DOI: 10.2312/PE/vriphys/vriphys06/053-060 3, 12.

[SGO09] SCHVARTZMAN, SARA C., GASCÓN, JORGE, and OTADUY, MIGUEL A. "Bounded Normal Trees for Reduced Deformations of Triangulated Surfaces". *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. SCA '09. New Orleans, Louisiana: Association for Computing Machinery, 2009, 75–82. ISBN: 9781605586106. DOI: 10.1145/1599470.1599480 3.

[Sho85] SHOEMAKE, KEN. "Animating rotation with quaternion curves". *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*. 1985, 245–254 6.

[SOG08] STEINEMANN, DENIS, OTADUY, M., and GROSS, M. "Tight and efficient surface bounds in meshless animation". *Comput. Graph.* 32 (2008), 235–245 3.

[Uni21] UNITY TECHNOLOGIES. "Unity Documentation". https://docs.unity3d.com/ScriptReference/AnimationClip-localBounds.html. Animation Clip Local Bounds. 2021 3.

[Ura19] URALSKI, YURY. *Mesh Shading: Towards Greater Efficiency of Geometry Processing, Advances in Real-time Rendering*. Siggraph Course. 2019 3.

[Wal21] WALBOURN, CHUCK. *DirectXMesh geometry processing library*. https://github.com/microsoft/DirectXMesh. [Accessed: 13-April-2021"]. 2014-2021 3.

[Wih16] WIHLIDAL, GRAHAM. "Optimizing the Graphics Pipeline with Compute". Game Developers Conference. 2016 2, 3.