Fixed
Wont Fix

EUROGRAPHICS 2016 /

Volume 35 (2016), Number 0

A Discussion of Curved Point-Normal Triangles

A very good draft.

R. van Veen[†1] and L. E. N. Baakman[†1]

[1]Computing Science Department of the University of Groningen

Figure 1: From left to right, the input triangulation, the model, and the curved point-normal model, colored according to its normals. The model was taken from Vlachos et al. [Vla+01].

Abstract

Write some abstract, this may be at most 3 inches (7.62cm) long.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Shape modeling—Parametric curve and surface modeling

1. Introduction

Vlachos et al. [Vla+01] introduced curved point-normal triangles as a method to improve the visual quality of triangular meshes in real-time entertainment. A major advantage of this method at the time of introduction (2001) is that it is a fast method to improve the visual quality of the rendered objects.

Define and then use

If PN triangles are used, each triangle of the mesh is replaced by a curved shape, that can be subdivided into a user selected number of flat smaller triangles. Each point-normal triangle consists of a geometric and a normal component. These components are respectively defined as a cubic and a quadratic Bézier patch. The coefficients of the patches are based on the points and the normals of the corresponding triangle in the mesh.

vertices

Although curved point-normal triangles improve the visual appeal of a rendered mesh via"smoother silhouettes and better shading" [Vla+01] and are fast enough for real-time rendering, their application domain is restricted to entertainment. They should not be used in, for example computer-aided design. Since the rendering gives the model a visual smoothness that is not present in the actual mesh.

Section 2 introduces the problems we discuss, after that we treat the methods used to analyze the introduced problems in Section 3. Section 4 reviews our implementation of the discussed methods. We present and discuss our results in Section 5. Finally Section 6 presents our conclusions.

2. Problem Definition

We consider a number of sub-problems associated with point-normal triangles.

(i) In the introduction we have mentioned that point-normal triangles are not suitable for rendering in computer aided

† These authors contributed equally to this work.

designed software, due to the 'fake' normal field, as the normal component of the PN triangle does not accurately represent the actual normals. Adapting the PN triangle algorithm to use 'real' normals, expands its application domain to include rendering for CAD software. However it is possible to compute the 'real' normals of the vertices of the subdivided PN-triangle, using the method we present in section 3.2.2. In this paper we compare both the computational and the visual performance of the 'real' and 'fake' normals. We do not expect a large difference in visual performance when 'real' normals are used. Nor do we suppose that computing the normals based on the geometry will have a significant impact on the computational complexity of point-normal triangles.

(ii) One major advantage of PN-triangles at the time of its introduction (2001) was its speed. Since then geometry and tessellation shaders have been introduced into the OpenGL pipeline. We investigate which changes have to be made to the point-normal triangles as proposed by Vlachos et al. to fit into a recent OpenGL pipeline (version 4.1). In this paper we discuss the changes between the CPU implementation proposed by Vlachos et al. and our GPU implementation. We expect that due to their locality point-normal triangles are extremely suited for computation on the GPU. Allowing us to subdivide triangular meshes faster than is possible on the CPU.

3. Method

The main goal of PN triangles is to improve the visual quality of rendered models, compare the models in Figure 1, and to do this on resource-limited hardware environments where e.g. no information about neighboring triangles can be accumulated. Or more precisely, the goal of point-normal triangles is "to soften triangle creases and improve the visual appeal by generating smoother silhouettes and better shading" [Vla+01]. Other than the visual improvements PN triangles provide, Vlachos et al. mention the following benefits:

(i) Point-normal triangle construction is *compatible* with the existing API data structures i.e., vertex arrays together with triangle index streams are used, and the triangles arrive in an unpredictable order.

(ii) The models are *backward compatible* with hardware that does not support point-normal triangles, with minimal or no changes needed to existing models.

(iii) No setup of the application, API, or hardware driver is needed. Specifically hardware should not be able to provide random access to neighboring primitives. Consequently the only possible communicated information between primitives are provided by using shared normals at the vertices. Which does restrict the models to be rendered somewhat, as discussed in section 3.1.3.

(iv) Point-normal triangles are applicable to *arbitrary topology*.

(v) PN-triangle rendering is done *fast* and via *simple implementation* in hardware on the CPU in 2001. At the time

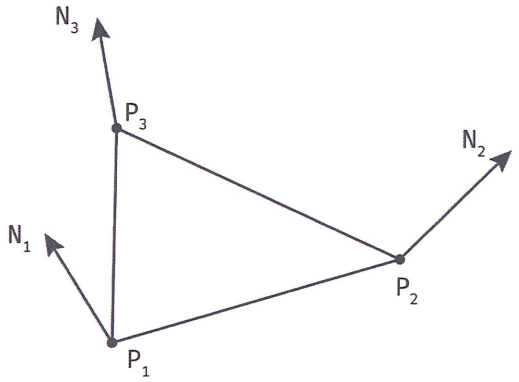


Figure 2: An input primitive as used by point-normal triangle consists of three vertices and three vertex normals. Note that only the information from a single primitive is used during the construction of both the geometric and the normal component. The labels of the vertices and normals coincide with those used in the equations in this section.

of writing even faster executing is possible by using programmable tessellation shaders on the GPU.

In the remaining part of this section we discuss the construction of point-normal triangles conceptually as well as mathematically. As mentioned in the introduction, A PN triangle is split in two different components: we refer to section 3.1 for a discussion on the geometric component and to section 3.2 for the review of the normal component.

3.1. Geometric component

Let us emphasize again that the the only available information for the construction of the geometric and normal component are the three vertices that define a single primitive. The vertices contain the vertex xyz-coordinates together with a unique shared normal per vertex i.e. the normals of each vertex is the same for every primitive using that vertex. Figure 2 contains an illustration of an input primitive, which is the start point of the construction of every point-normal triangle.

In the next subsection we discuss the definition of the geometry of a point-normal triangle. In subsection 3.1.2 we discuss the construction of the control points that define the geometry.

3.1.1. Basic form

The geometric component of a point-normal triangle is defined as a triangular cubic Bézier patch. Such a patch b is defined as follows:

$$b: \quad R^2 \mapsto R^3, \quad \text{for } w = 1-u-v, \quad u,v,w \geq 0$$

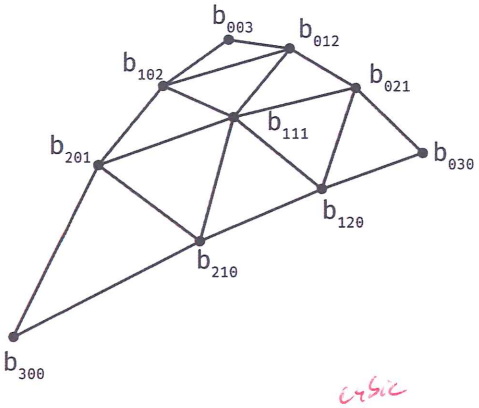


Figure 3: The geometric component of a point-normal triangle, i.e., the control net of a triangular Bézier patch. Note that the names of the vertices used in this image coincide with those used in the equations in this section.

$$
\begin{aligned}
b(u,v) &= \sum_{i+j+k=3} b_{ijk} \frac{3!}{i!j!k!} u^i v^j w^k \\
&= b_{300}w^3 + b_{030}u^3 + b_{003}v^3 \\
&\quad + b_{210}3w^3 + b_{120}3wu^2 + b_{201}3w^2v \\
&\quad + b_{021}3u^2v + b_{102}2wv^2 + b_{012}3uv^2 \\
&\quad + b_{111}6wuv.
\end{aligned} \tag{1}
$$

The b_{ijk} variables in equation 1 are the control points of the patch, also called coefficients. In Figure 3 the visualization of the network of these control points is shown. We group the coefficients in three different groups, as their construction differs:

vertex coefficients: b_{300}, b_{030}, b_{003}

tangent coefficients: b_{210}, b_{120}, b_{021}, b_{012}, b_{102}, b_{201}

center coefficient: b_{111}

The formula in equation (1) can be used to interpolate any point, parameterized by the barycentric coordinates u and v on the patch. This is used in the sub-triangulation stage of the rendering process. This is the stage where the cubic Bézier surface is approximated by using a number of smaller flat triangles; these flat triangles are actually rendered. The number of sub-triangles is determined by the level of detail (lod). For the original sub-triangulation we refer the reader to Vlachos et al. [Vla+01], as this is where the implementation presented here deviates from the original. Details about this are provided in section 4.

As stated above the definition of the geometry of a point-normal triangle is a cubic Bézier patch. This order of patches is a trade-off between simplicity, visual performance, and computational cost. Quadratic patches do not provide the same modeling range of a surface as a cubic patch. For example a cubic representation is necessary to capture inflections implied by the triangle position and normal data. Vlachos et al. state that there is no additional data to suggest an higher degree is needed, and that therefore they settled on the form of b as presented in equation 1.

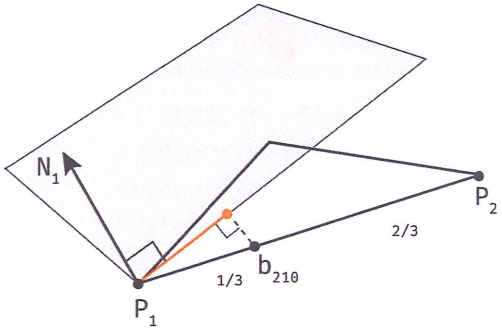


Figure 4: Projection of a tangent coefficient b_{210} to the tangent plane of the closes corner P_1.

3.1.2. Coefficients

This section discusses how the 'curved' control net geometry, see Figure 2, is calculated, from the flat input primitive shown in Figure 2. The input primitive provides the the positions $P_1, P_2, P_3 \in \mathbb{R}^3$ and normals $N_1, N_2, N_3 \in \mathbb{R}^3$. The coefficients b_{ijk} are computed as follows:

(i) Initially the coefficients b_{ijk} are spread uniformly i.e., the intermediate position of b_{ijk} is calculated using the formula $(iP_i + jP_2 + kP_3)/3$.

(ii) The intermediate positions of the vertex coefficients are the same as their final positions, see equation (2). Since their intermediate position places them at the vertices of the input triangle, which is where they are required to stay to keep the mesh water-tight.

(iii) The tangent coefficients are placed at their final position by projecting the intermediate position into the tangent plane of the closest corner, see equation (3). This is illustrated in Figure 4.

(iv) The center coefficient is moved to the average of the tangent coefficients plus $1/2$ the distance it had to travel from its intermediate position, see equation (4).

The following set of formulas describe how the positions of the coefficients are calculated. For clarity, we group together the formulas in the same way as the coefficients. This gives the following formulas for the *vertex coefficients*:

$$b_{300} = P_1,\ b_{030} = P_2,\ b_{003} = P_3. \tag{2}$$

The tangent coefficients are given by the projection of a

point Q onto a plane defined by the normal N of the point P. The projected point Q' is then given by: $Q' = Q - wN$, where $w = (Q-P)\cdot N$. Using this the following set of formulas give the positions of the *tangent coefficients*:

$$\begin{aligned} w_{ij} &= (P_j - P_i)\cdot N_i \in \mathbb{R} \\ b_{210} &= (2P_1 + P_2 - w_{12}N_1)/3, \\ b_{120} &= (2P_2 + P_1 - w_{21}N_2)/3, \\ b_{021} &= (2P_2 + P_3 - w_{23}N_2)/3, \\ b_{012} &= (2P_3 + P_2 - w_{32}N_3)/3, \\ b_{102} &= (2P_3 + P_1 - w_{31}N_3)/3, \\ b_{201} &= (2P_1 + P_3 - w_{13}N_1)/3. \end{aligned} \tag{3}$$

The last coefficient is the center coefficient which is moved to the average of the previous computed tangent coefficients plus 1/2 the distance it traveled from its intermediate location to that average position. The *center coefficient*, is thus defined as:

$$\begin{aligned} E &= (b_{210} + b_{120} + b_{021} \\ &\quad + b_{012} + b_{102} + b_{201})/6, \\ V &= (P_1 + P_2 + P_3)/3, \\ b_{111} &= E + (E - V)/2. \end{aligned} \tag{4}$$

Combining (2) through (4) results in the control net shown in Figure 3.

3.1.3. Properties

Vlachos et al. have shown that point-normal triangles do not deviate too much from the original triangle. This is an important property since this ensures that the shape of the model is preserved and adjacent triangles do not interfere with each other.

By demanding shared normals i.e. one unique normal per vertex, the boundary between two point-normal triangles is generated by the same algorithm, thus the surface is water tight. Except at the corners, PN triangles do not usually join with tangent continuity. Figure 5 illustrates what happens if normals are not shared.

3.2. Normal component

This section discusses the parametrization of the normals. This section distinguishes between 'fake' normals and 'real' normals, in respectively section 3.2.1 and 3.2.2. The fake normals are interpolated based on a quadratic patch, whereas the real normals are calculated based on the geometry of the point-normal triangle.

It should be noted that just as the geometric component, the normal component of the point-normal triangle is computed based on solely the input primitive, i.e. the vertex positions and their normals, as shown in Figure 2.

Figure 5: Illustration of what would happen if one renders a model using point-normal triangles where vertices have different normals, depending on the associated faces. Image taken from [MK10].

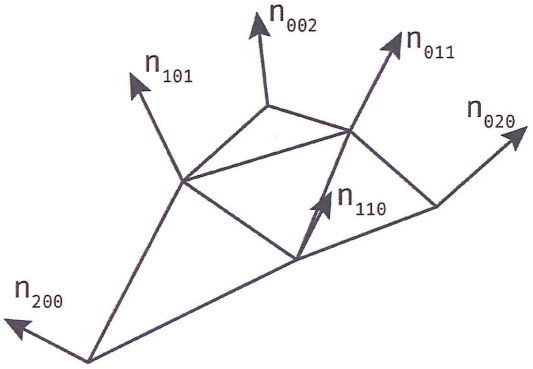


Figure 6: The normal field of the point-normal triangle.

3.2.1. Fake normals

Vlachos et al. defines the 'fake' normals with quadratic function n:

$$\begin{aligned} n: \quad R^2 \mapsto R^3, \quad & \text{for } w = 1-u-v, \quad u,v,w \geq 0 \\ n(u,v) &= \sum_{i+j+k=2} n_{ijk} u^i v^j w^k \\ &= n_{200}w^2 + n_{020}u^2 + n_{002}v^2 \\ &\quad + n_{110}wu + n_{011}uv + n_{101}wv \end{aligned} \tag{5}$$

The coefficients of this quadratic 'patch' are the normals shown in Figure 6. The normals are computed for a point, halfway on every edge. Point-normal triangles use quadrati-

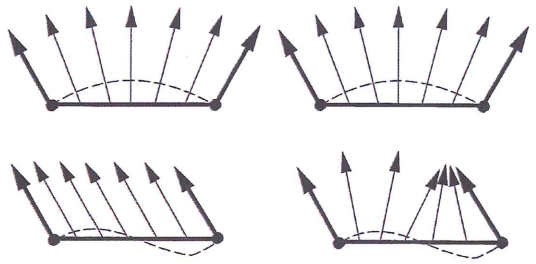


Figure 7: The upper row shows a parabolic curve with linearly varying normals on the left and quadratically varying normals on the right. The lower row compares the normals of a cubic curve using linearly varying normals on the left and quadratically varying normals on the right. The illustration was taken form Overveld and Wyvill [OW97].

cally varying normals to be able capture the inflection points that are possible due to the cubic geometric component.

Figure 7 illustrates why one needs quadratic patches to capture these inflection points.

two cases: the top two images show a parabolic curve wGeehere both linear and quadratically varying normals perform the same. The more interesting case is the one illustrated by the two bottom images that show a cubic curve. We see that the linear varying normals do not capture the correct normal for this curve, but the quadratic varying normals do.

Discuss parametrization of 'quadratic' patch

Using the function n, see (5), the normal of any point parametrized by the barycentric coordinates u and v can be calculated. To do this we first need to define a control net. We consider two different types of coefficients: the vertex normals, n_{200}, n_{020}, and n_{002}; and the edge normals, n_{110}, n_{011}, and n_{101}.

Discuss the construction of the control points for the 'quadratic' patch

3.2.2. Real normals

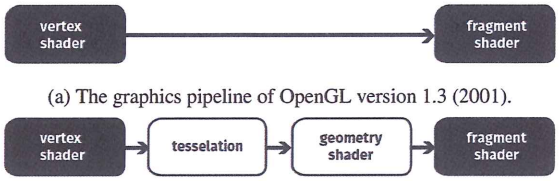
Discuss how to compute the real normals given the geometric component

The real normals are computed based on the geometric component of the point-normal triangle. This is done by taking the cross product of the the partial derivatives with respect to u and v.

Expand explanation, add math

4. Implementation

A schematic overview of both the OpenGL pipeline at the time of the publication of the paper and the current pipeline



(a) The graphics pipeline of OpenGL version 1.3 (2001).

(b) The graphics pipeline of OpenGL version 4.1 (2010).

Figure 8: A schematic overview of the programmable parts of the (a) 1.3 and (b) 4.1 OpenGL graphics pipelines. Each rounded rectangle represents one programmable shader.

are presented in Figure 8. These images show that since 2001, three new shaders have been introduced into the pipeline, namely the tessellation control, tessellation evaluation and the geometry shader.

Since only the tessellation shaders are relevant for point-normal triangles, we discuss those in more detail in section 4.1. Section 4.2 through section 4.4 present our implementation of point-normal triangles per discussed tessellation stage.

4.1. Tessellation in OpenGL

A schematic overview of tessellation in OpenGL is presented in Figure 9. The tessellation control shader (TCS) is invoked once for each vertex in the input patch, which is a triangle in our case. One of the functions of the tessellation control shader is to set the inner and the outer tessellation levels. Furthermore the TCS can compute additional information about the vertices in the input patch. As can been seen in Figure 9 the TCS passes additional information about that patch along to the tessellation evaluation shader (TES) via the output patch. The set inner and outer tessellation levels are sent to the tessellation primitive generator (TPG) [Wol13].

The TPG generates a number of new primitives based on the tessellation levels. How these primitives are generated depends on the selected edge tessellation spacing algorithm. Equal spacing results in evenly divided edges, whereas fractional spacing allows for two shorter edges in the subdivision for more stable interpolation under changing tessellation levels [Wol13; SA10]. In the case of triangles, the barycentric coordinates of the vertices of the newly generated primitives are passed to the TES, as shown in Figure 9.

The tessellation evaluation shader (TES) is executed for each vertex that is generated by the TPG. It receives the coordinates of this vertex in parameter space and the output of the TCS, as is illustrated in Figure 9. Based on this information it computes the position of the current vertex. The computed data are passed to the next shader in the pipeline, either the geometry shader or the fragment shader, see Figure 8b [Wol13].

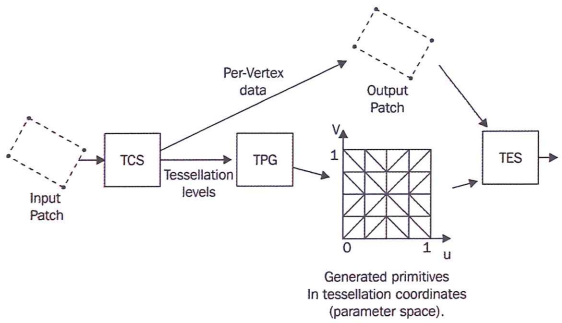
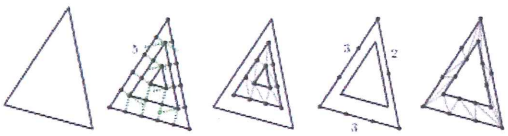


Figure 9: An overview of the tessellation part of the OpenGL render pipeline. Note that in the case of point-normal triangles, the input patches, generated primitives and output patches are triangles, not quads. Illustration taken form Wolff [Wol13].



(a) Input (b) Step (i) (c) Step (ii) (d) Step (iii) (e) Step (iv)

Figure 10: Generation of primitive using the TPG from OpenGL. ([CR12]?)

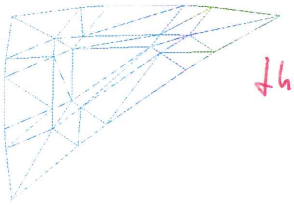


Figure 11: A single triangle tessellated with both outer and inner tessellation level set to 4.0, and edge tessellation spacing set to `fractional_odd_spacing`.

4.2. Tessellation Control Shader

The TCS receives both the normals and the positions of the vertices of the input patch, in our case, a single triangle. Using these it computes the cubic Bézier patch that defines the geometry of the point-normal triangle according to equation (1). If 'fake' normals are used, their control net is computed as well. The 'real' normals need no computations in the tessellation control shader.

We parallelize the computations of the x, y and z component of the vertices of the output patch by using the `invocation_ID` to select which component of the output patch to compute. For example, if the TCS is invoked for the second vertex of an input patch, i.e. `invocation_ID` = 2, we compute the y component of all the vertices of the output patch. This only works since the number of vertices of the input patch is equal to the dimensionality of the positions of the vertices of the output patch.

There are two alternative ways to compute the output patch. One method is by only computing the output data for e.g. the first vertex. This ensures that no computations are performed twice, however it also means that a lot of threads are waiting for the threads that happened to received the first vertex of a input patch to finish. Alternatively, one could always compute all vertices of the output patch, independent of the `invocation_ID`. This results in a lot of unnecessary computations, but removes the branching required to only compute the control net if the `invocation_ID` has a specific value. Neither of these methods are more efficient than the one we used. A downside of our method is that one has to construct the vectors that represent the vertices of the control net in the tessellation evaluation shader.

Other than computing the control nets, this shader also sets the inner and outer tessellation levels for the Tessellation Primitive Generator. The control nets are output to the tessellation evaluation shader whereas the tessellation levels are sent to the tessellation primitive generator.

4.3. Tessellation Primitive Generator

The tessellation primitive generator is not programmable, we can however control its output via some parameter, most notably the tessellation levels and the edge tessellation spacing algorithm. The TPG tessellates an input triangle, see Figure 10a, according to these steps [CR12]:

(i) The patch is divided into the same number of concentric triangles as the inner tessellation level, see Figure 10b.
(ii) The spaces between the concentric triangles, except the outer triangle, are subdivided intro triangles, see Figure 10c.
(iii) The outer edges of the triangles are decomposed into smaller edges according to the outer tessellation level, see Figure 10d.
(iv) The outer ring of triangles is subdivided into triangles by connecting the vertices from the inner triangles with those of the other triangles, see Figure 10e.

A triangle that is tessellated with fractional spacing, and an inner and outer tessellation level of 4.0, is presented in Figure 11. It should be noted that the resulting tessellation is different than the subdivision of triangles proposed by Vlachos et al.

We opted for fractional odd spacing to make changes between different tessellation levels as smooth as possible.

4.4. Tessellation Evaluation Shader

Due to our parallelization the first step in the TES is to extract the control points from the input. If we have the control nets we determine the positions of the vertices of the flat triangles that will form the final mesh, according to Equation (1). We compute the normals according to Equation (5) or depending on which normal computation is used.

reference to the at this point non existing equation.

5. Results

5.1. Fake versus real normals

Show results of PN triangles with fake and real normals, and discuss differences

Computational complexity of the fake normals and the real normals, consider one triangle.

Any unsolved problems?

5.2. Pipeline

Figure 12 shows the rendering by Vlachos et al. next to our rendering of, probably, the same model. The only difference between these two models should be the pattern in which the triangles of the input mesh are subdivided into flat triangles. A visual inspection shows no obvious differences between the models, leading us to suggest that the difference in triangulation does not matter much.

6. Conclusion

Future experiments, if any

How has the 2015 openGL pipeline changed the perofrmance of PN triangles compared to the 2001 pipeline? I.e. should you use a shader, or the CPU?

Contrast fake and real normals, when to use one, when the other?

References

[CR12] Patrick Cozzi and Christophe Riccio. *OpenGL Insights*. CRC press, 2012.

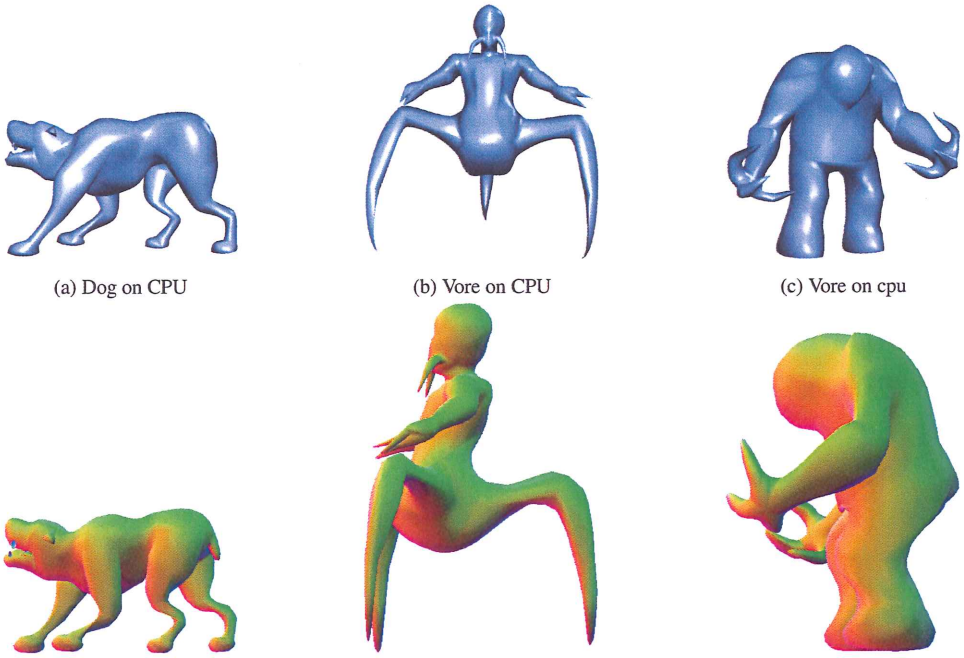
[MK10] J. McDonald and M. Kilgard. *Crack-free point-normal triangles using adjacent edge normals*. 2010.

[OW97] Cornelius W.A.M. van Overveld and Brian Wyvill. "Phong normal interpolation revisited". In: *ACM Transactions on Graphics (TOG)* 16.4 (1997), pp. 397–419.

[SA10] Mark Segal and Kurt Akeley. *The OpenGL® Graphics System: A Specification (Version 4.1 (Core Profile) - July 25, 2010)*. The Khronos Group Inc. 2010.

[Vla+01] Alex Vlachos et al. "Curved PN triangles". In: *Proceedings of the 2001 symposium on Interactive 3D graphics*. ACM. 2001, pp. 159–166.

[Wol13] David Wolff. *OpenGL 4 Shading Language Cookbook*. Packt Publishing Ltd, 2013.



(a) Dog on CPU (b) Vore on CPU (c) Vore on cpu

(d) Dog on GPU (e) Vore on GPU (f) Vore on GPU

Figure 12: A family of game characters from the game Quake 1. (a), (b) and (c) are rendered on a CPU by Vlachos et al. (d), (e), (f) are rendered by our implementation. Figures (a), (b) and (c) were taken from Vlachos et al. [Vla+01].