Excellent report.
9,5

Volume 35 (2016), Number 0

A Discussion of Curved Point-Normal Triangles

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

[1]Computing Science Department of the University of Groningen

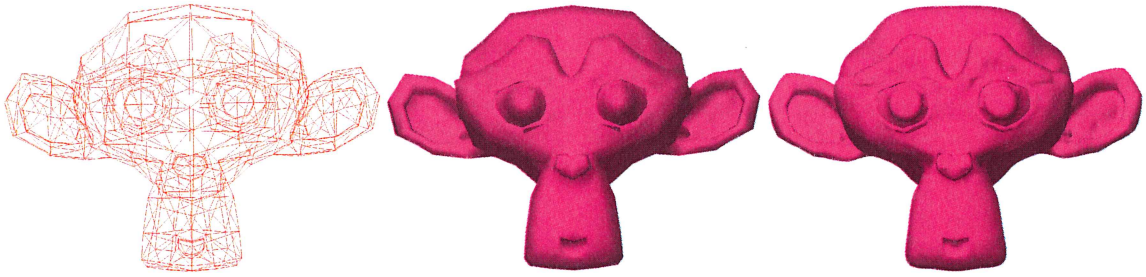


Figure 1: From left to right: the triangular input mesh, the Phong shaded mesh, and the Phong shaded mesh using point normal triangles and an inner and outer tessellation level of 12.0.

Abstract

Vlachos et al. introduced point-normal triangles in 2001 to improve the visual quality of triangle meshes. The method was designed to be a fast, without the need to change the in original mesh. Since then a lot of changes have been made to the OpenGL pipeline, making it possible to render with point-normal triangles evaluated on the GPU.

We discuss how one can implemented point-normal triangles on the GPU and discuss the differences with the original CPU implementation. We have found that there are probably no differences.

Further more we also consider the use of normals based on the geometry, 'real' normals, instead of the quadratically varying normals, 'fake' normals, proposed by Vlachos et al. We have concluded that although the resulting normal fields are the same the 'fake' normals are computationally less expensive, and thus preferred.

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 appeal of triangular meshes in real-time entertainment. A major advantage of this technique at the time of introduction (2001) was that it is a fast method to improve the visual quality of the rendered objects.

If point-normal (PN) triangles are used each triangle of the mesh is replaced by a curved shape. This shape is tessellated into a, user selected, number of smaller flat 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 control points of the patches are based on the vertices and the normals of the corresponding triangle in the input mesh.

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 entertain-

† These authors contributed equally to this work.

ment. They should not be used in, for example computer-aided design. Since rendering with PN triangles gives models 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 Section 6 presents our conclusions.

2. Problem Definition

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

(i) In the introduction we have mentioned that point-normal triangles use a 'fake' normal field. Consequently the normal field of a PN triangle does not accurately represent the actual normals. One might wonder why Vlachos et al. chose to use a 'fake' normal field, instead of the real normals. Since it is possible to compute the 'real' normals of the vertices of the tessellated 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. Note that adapting the PN triangle algorithm to use 'real' normals does not expand its application domain to include rendering for CAD software. As the smoothness of the rendering, a result of the added geometry, is not present in the actual mesh.

(ii) One major advantage of point-normal triangles at the time of its introduction (2001) was that it was fast when computed on the CPU. Since then geometry and tessellation shaders have been introduced into the OpenGL pipeline. Making it possible to compute PN triangles on the GPU. 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 tessellate triangular meshes faster than is possible on the CPU.

3. Method

The main goal of point-normal triangles is to improve the visual quality of rendered models, especially on resource-limited hardware environments where e.g. no information about neighboring triangles can be accumulated. We ask the reader to have a look at Figure 1, with the following question in mind: Which rendering do you prefer? The center and right images both show a rendering using the phong shading and illumination model. The only difference being, that the right image is rendered using poit-normal triangles, with an inner and outer tessellation level of 12.0. The influence of the tessellation levels is discussed in Section 4. Vlachos et al. formulated the goal of point-normal triangles as: "soften triangle creases and improve the visual appeal by generating smoother silhouettes and better shading". Vlachos et al. mention, besides the visual improvements, the following benefits of using PN triangles:

(i) Point-normal triangle construction is *compatible* with the existing graphics API data structures, i.e., vertex arrays together with triangle index streams, where the triangles arrive in 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 is not required to provide random access to neighboring primitives. Consequently the only possible communicated information between primitives are provided by using shared normals at the vertices. This does restrict the models to be rendered somewhat, as discussed in Section 3.1.3.
(iv) Point-normal triangles are applicable to meshes with *arbitrary topology*.
(v) PN-triangle rendering is *fast* and done via *simple implementation* in hardware on the CPU in 2001. At the time of writing point-normal triangles can be rendered via 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 only a single primitve, i.e. three vertices and their associated normals, is used for the construction of the geometric and normal component. The vertices contain the three-dimensional vertex location together with a shared normal per vertex, i.e., the normal of each vertex is the same for every primitive using that vertex. Figure 2 shows an illustration of an input primitive, based on such input primitives the geometric component of the point-normal triangle is constructed.

In the next section we discuss the definition of the geometry of a point-normal triangle. In Section 3.1.2 we discuss the construction of the control points that define the geometry. Section 3.1.3 reviews the properties of the geometric component of a point-normal triangle.

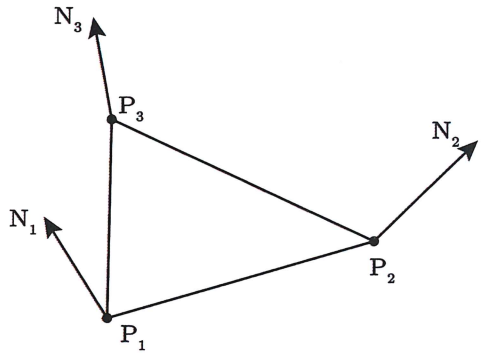


Figure 2: An input primitive based on which a point-normal triangle can be computed. The input primitive consists of three vertices, P_1, P_2 and P_3, and three vertex normals, N_1, N_2, N_3. Note that only the information of a single primitive is used during the construction of both the geometric and the normal component, i.e. no information about neighboring primitives is used.

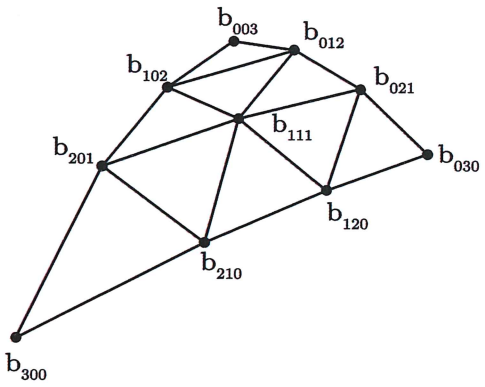


Figure 3: The geometric component of a point-normal triangle, i.e. the control net of a cubic Bézier triangle.

3.1.1. Basic form

The geometric component of a point-normal triangle is defined as a triangular cubic Bézier patch. Such a patch $b(u,v)$ is defined as follows:

$$
\begin{aligned}
b(u,v): \quad & R^2 \mapsto R^3, \quad \text{for } w = 1-u-v, \quad u,v,w \geq 0 \\
b(u,v) &= \sum_{i+j+k=3} b_{ijk} \frac{3!}{i!\,j!\,k!} u^i v^j w^k \\
&= b_{300}w^3 + b_{030}u^3 + b_{003}v^3 \\
&\quad + b_{210}3w^3 + b_{120}3wu^2 + b_{201}3w^2v \\
&\quad + b_{021}3u^2v + b_{102}2wv^2 + b_{012}3uv^2 \\
&\quad + b_{111}6wuv.
\end{aligned} \tag{1}
$$

The b_{ijk} parameters 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 distinguish three different groups of control points, each with their own method of construction:

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}

Equation 1 can be used to evaluate any point on the patch that is parameterized by the barycentric coordinates (u,v,w). This evaluation is done in the triangulation stage of the rendering process. In this stage the cubic Bézier surface is approximated by using a number of smaller flat triangles; these are eventually rendered. The number of sub-triangles is determined by the level of detail (lod). For the original subtriangulation scheme we refer the reader to Vlachos et al. [Vla+01], as this is where the implementation presented in this paper deviates from the one discussed by Vlachos et al. For a more exhaustive discussion of these differences we refer the reader to Section 4.

As stated above the geometry of a point-normal triangle is a cubic Bézier patch. This degree 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 [BA08]. 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 a higher degree is needed, and that therefore they settled on the form of $b(u,v)$ as presented in Equation 1.

3.1.2. Coefficients

This section discusses how the 'curved' control net geometry, see Figure 3, is calculated from the flat input primitive shown in Figure 2. The input primitive provides 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 according to

$$b_{ijk} = \frac{(iP_1 + jP_2 + kP_3)}{3}.$$

(ii) The intermediate positions of the vertex coefficients are the same as their final positions, see Equation 2. Their intermediate position places them at the vertices of the input triangle, and this is where they are required to stay to keep the mesh watertight.

(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 step is illustrated in Figure 4.

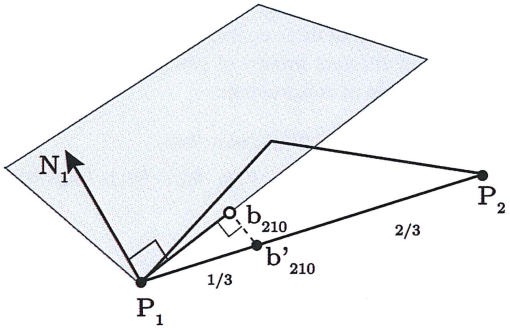


Figure 4: Projection of the initial tangent coefficient b'_{210} to the tangent plane of the closest vertex P_1, which results in the final tangent coefficient b_{210}. The initial position is represented with a filled circle, the final position with the outline of a circle. Illustration adapted from Vlachos et al. [Vla+01].

(iv) The center coefficient is moved to the average of the tangent coefficients plus $^1/_2$ times 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. The *vertex coefficients* are defined as:

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

The tangent coefficients are given by the projection of a point Q onto the 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 positions of the *tangent coefficients* are defined as:

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

where

$$w_{ij} = (P_j - P_i)\cdot N_i \in \mathbb{R}.$$

The center coefficient is, as stated before, moved to the average of the previous computed tangent coefficients plus $^1/_2$ times the distance it traveled from its intermediate location to that average position. The *center coefficient*, is computed as:

$$b_{111} = E + (E - V)/2, \tag{4}$$

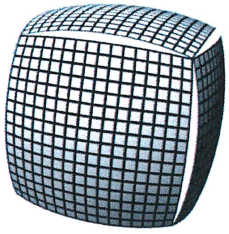


Figure 5: An illustration of what would happen if one renders a model using point-normal triangles where vertices have different normals, depending on the associated faces. Image taken from [MK10].

where

$$E = \frac{b_{210} + b_{120} + b_{021} + b_{012} + b_{102} + b_{201}}{6}$$
$$V = \frac{(P_1 + P_2 + P_3)}{3}.$$

Combining Equation 2 through 4 computes the control net shown in Figure 3, based on the input primitive presented in Figure 2.

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 it ensures that the shape of the model is preserved and that adjacent triangles do not interfere with each other.

By demanding that normals are shared, i.e., one unique normal per vertex, the boundary between two point-normal triangles is generated with the same algorithm, consequently the surface is water tight. Except at the corners, PN triangles do not usually join with tangent continuity [Vla+01]. If the normals are not shared, cracks will form as illustrated in Figure 5.

3.2. Normal component

This section discusses the parametrization of both the 'fake' and 'real' normals in respectively Section 3.2.1 and 3.2.2. What we call 'fake' normals, is the normal field proposed by Vlachos et al. Whereas the 'real' normals are calculated based on the surface defined by the geometric component.

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

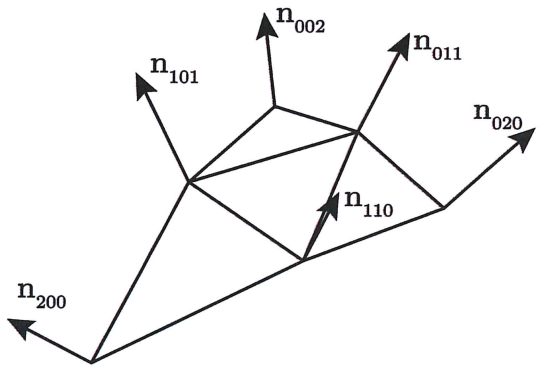


Figure 6: The normal field of the point-normal triangle.

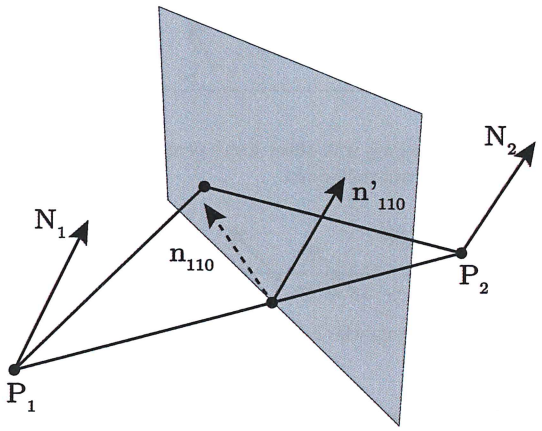


Figure 7: Construction of the mid-edge normal coefficient, used for quadratically varying normals. The two input normals are averaged, which results in n'_{110}, and then reflected across the plane perpendicular to the edge. The resulting edge coefficient, n_{100}, is represented by the dashed vector. Illustration adapted from Vlachos et al. [Vla+01].

3.2.1. Fake normals

Vlachos et al. define the 'fake' normals using a quadratic function $n(u,v)$:

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

The coefficients of this quadratic 'patch' are the normals shown in Figure 6. These vectors are computed for a point, halfway on every edge, i.e., $(u,v,w) = (\frac{1}{2},\frac{1}{2},0)$, $(0,\frac{1}{2},\frac{1}{2})$, $(\frac{1}{2},0,\frac{1}{2})$.

Using the function $n(u,v)$ defined in (5), the normal of any point, parametrized by the barycentric coordinates (u,v), can be calculated given the control net. We distinguish two types of coefficients:

vertex coefficients: n_{200}, n_{020}, n_{002}
edge coefficients: n_{110}, n_{011}, n_{101}

A vertex coefficient is simply the normal provided by the input primitive, i.e.

$$n_{200} = N_1,\ n_{020} = N_2,\ b_{002} = N_3. \tag{6}$$

An edge coefficient is calculated by taking the average of the two input vertex normals of the vertices of the edge. This results the normals presented in Figure 8b. To capture the inflection point the average normal is then reflected across the plane perpendicular to the edge, as illustrated in Figure 7. When a inflection point exists this will give the correct mid-edge normal, as shown in Figure 8c. Consequently the edge coefficients are defined as:

$$\begin{aligned} n_{110} &= \frac{h_{110}}{\|h_{110}\|} \quad h_{110} = N_1 + N_2 - v_{12}(P_2 - P_1) \\ n_{011} &= \frac{h_{011}}{\|h_{011}\|} \quad h_{011} = N_2 + N_3 - v_{23}(P_3 - P_2) \\ n_{101} &= \frac{h_{101}}{\|h_{101}\|} \quad h_{101} = N_3 + N_1 - v_{31}(P_1 - P_3) \end{aligned} \tag{7}$$

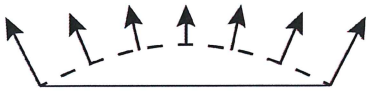
where

$$v_{ij} = \frac{(P_j - P_i)\cdot(N_i + N_j)}{(P_j - P_i)\cdot(P_j - P_i)} \in \mathbb{R}.$$

When no inflection point exists the normal field is not altered, this is illustrated in Figure 8a, where once can see that the reflection of the mid-edge normal of the curve is the mid-edge normal.

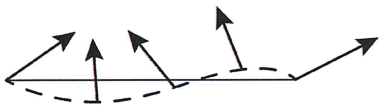
The quadratically varying normals ensure that the normal component of the PN triangles captures the inflection points, that are possible due to the cubic geometric component.

Figure 8 illustrates why one needs quadratic patches to capture these inflection points. Figure 8a shows that when the normals point in a different direction the curve is parabolic, and no inflection point exists. In this case both linear and quadratically varying normals are able to capture the surface. Figures 8b and 8c show the normals of a curve with an inflection point, approximated with linear and quadratic interpolation respectively. Observe that the linear varying normals in Figure 8b do not capture the inflection point, i.e., the approximated normals are not sensible when one considers the curve. The quadratically varying normals



(a) Vector averaging with either linear or quadratic interpolation over a curve without inflections.

(b) Vector averaging with linear interpolation over a curve with an inflection.



(c) Vector averaging with quadratic interpolation over a curve with an inflection.

Figure 8: Examples of normal vector averaging over an edge. The dashed lines indicate the profile of the surface that should be approximated. (a) The normals of curves without inflections are correctly approximated by both linear and quadratic interpolation. The normals of a curve with inflections are incorrectly approximated with (b) linear interpolation and correctly with (c) interpolation. Figure 8a was adapted from Overveld and Wyvill [OW97], Figures 8b and 8c from Vlachos et al. [Vla+01].

in Figure 8c give sensible normals, i.e, they capture the inflection point.

3.2.2. Real normals

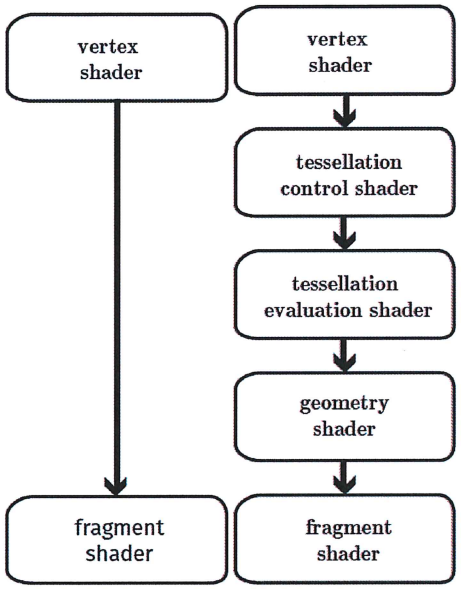
The real normals are computed based on the geometric component of the point-normal triangle. This is done by taking the cross product of the the partial derivatives with respect to u and v. The partial derivative of $b(u,v)$ with respect to u and v are respectively [Far14]:

$$\begin{aligned}\frac{\partial b(u,v)}{\partial u} &= \sum_{i+j+k=2} b_{i+1,j,k}\frac{2!}{i!j!k!}u^i v^j w^k \\ &= w^2 b_{102} + v^2 b_{120} + u^2 b_{300} + \\ &\quad 2vwb_{111} + 2uwb_{201} + 2uvb_{210},\end{aligned} \tag{8}$$

$$\begin{aligned}\frac{\partial b(u,v)}{\partial v} &= \sum_{i+j+k=2} b_{i,j+1,k}\frac{2!}{i!j!k!}u^i v^j w^k \\ &= w^2 b_{012} + v^2 b_{030} + u^2 b_{210} + \\ &\quad 2vwb_{021} + 2uwb_{111} + 2uvb_{120}.\end{aligned} \tag{9}$$

The partial derivatives define two tangent vectors in the u and v directions, at some point (u,v) on the surface. The normal, as defined by the geometric component of the PN triangle, at some point (u,v), n_{uv}, is defined as the crossproduct of the tangent vectors:

$$n_{uv} = \frac{\partial b(u,v)}{\partial u} \times \frac{\partial b(u,v)}{\partial v}. \tag{10}$$



(a) OpenGL version 1.3 (b) OpenGL version 4.1

Figure 9: 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.

4. Implementation

A schematic overview of both the OpenGL pipeline at the time of the publication of the paper and the current pipeline is presented in Figure 9. These diagrams show that since 2001 three new shaders have been introduced into the pipeline, namely the tessellation control (TC), tessellation evaluation (TES) 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 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 10. The tessellation control shader is invoked once for each vertex in the input primitive, see Figure 2 for an illustration of our input primitives. 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 10 the TCS passes additional information about that patch along to the tessellation evaluation shader (TES) via the output patch. The 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, which ensures a 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 10.

The tessellation evaluation shader 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 10. Based on this information the tessellation evaluation shader computes the position of the current vertex, [Wol13]. The computed data are passed to the next shader in the pipeline, either the geometry shader or the fragment shader, see Figure 9b.

4.2. Tessellation Control Shader

The TCS receives both the positions of the vertices and the normals of the input patch, i.e. the input primitive shown in Figure 2. Based on its input this shader computes the control net that defines the geometry of the point-normal triangle according to Equation 2 through 4. If 'fake' normals are used, their control net is computed as described in Section 3.2.1. The 'real' normals need no computations in the tessellation control shader, since they are based on the geometry of the point-normal triangle.

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` = 1, 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 receive the first vertex of an input patch to finish. Another downside of this method is that since you cannot guarantee that all executions in a warp have the same `invocation_ID`, the branching impacts the performance negatively [Fun+07]. Alternatively, one could compute all vertices of the output patch, independent of the `invocation_ID`. This results in a lot of unnecessary computations, but removes the branching penalty that occurs when one only computes 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 level 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.

Although the next shader, the tessellation primitive generator, is not programmable, we can control its output via some parameters that are set in the TES, most notably the tessellation levels and the edge tessellation spacing algorithm. We have left the selection of the tessellation levels to the user. This means that the tessellation levels of a rendered object can be changed on the fly, consequently it is important that the transition between different tessellation levels is smooth. We have thus opted to set the tessellation algorithm to fractional odd spacing, which gives a smooth transition [CR12].

4.3. Tessellation Primitive Generator

The TPG tessellates an input triangle, see Figure 11a, according to these steps [CR12]:

(i) The patch is divided into the same number of concentric triangles as the inner tessellation level, see Figure 11b.
(ii) The spaces between the concentric triangles, except the outer triangle, are subdivided intro triangles, see Figure 11c.
(iii) The outer edges of the triangles are decomposed into smaller edges according to the outer tessellation level, see Figure 11d.
(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 11e.

A triangle that is tessellated with fractional spacing, and an inner and outer tessellation level of 4.0, is presented in Figure 12. It should be noted that the resulting tessellation is different than the subdivision of triangles proposed by Vlachos et al.

4.4. Tessellation Evaluation Shader

Due to our parallelization the first step in the TES is to extract the control points from the input. Once we have the control nets, we determine the positions of the vertices of the flat triangles that will form the final patch, according to Equation 1. We compute the normals according to either Equation 5 or Equation 10.

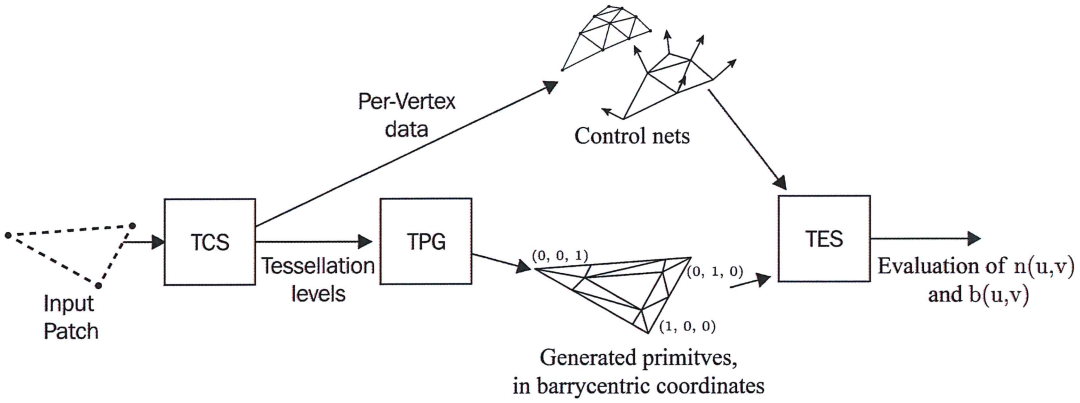
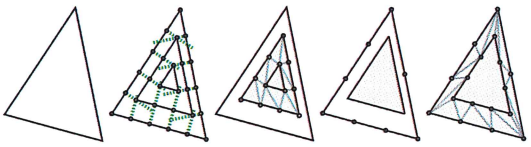


Figure 10: An overview of the tessellation part of the OpenGL version 4.1 rendering pipeline as applied to point normal triangles. An input patch, a triangle with normals, is presented to the tessellation control shader (TCS). The tessellation primitive generator (TPG) tessellates the triangle and outputs the barycentric coordinates of the newly generated triangle to the tessellation evaluation shader (TES). Apart from these coordinates the TES also receives the control nets from the TCS. Illustration adapted from Wolff [Wol13].



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

Figure 11: Generation of a primitive for the TES from 11a an input primitive using the TPG from OpenGL. Illustration taken from [CR12].

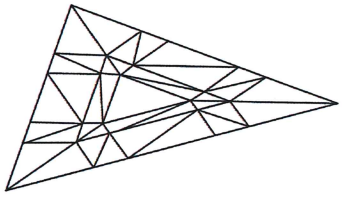
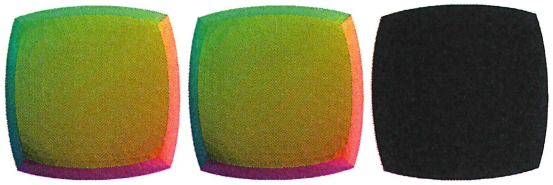


Figure 12: A single triangle tessellated with both outer and inner tessellation level set to 4.0, and edge tessellation spacing set to `fractional_odd_spacing`.

5. Results

In this section we present our results. Section 5.1 compares the computational complexity of the computation of the 'real' and 'fake' normals. Furthermore it contrasts the visual results. Section 5.2 compares the results of the implementation by Vlachos et al. on the CPU with our implementation on the GPU.



(a) 'Fake' normals (b) 'Real' normals (c) The difference between (a) and (b).

Figure 13: A cube rendered with point-normal triangles colored according to the normalized normals, scaled to the open unit interval, using (a) fake and (b) real normals. Both the inner and outer tessellation level were set to 12.0. Figure 13c shows the difference between using (a) a fake normal field or (b) a real normal field.

5.1. Fake versus real normals

To compare the computational complexity we have counted the number of floating point operations (flops) required to determine the normal of one vertex on the rendered mesh, for both 'fake' and 'real' normals. Evaluating the control net for the fake normal field for one vertex requires 23 flops. Computing the control net, which is not required for every vertex, requires 150 flops, i.e., if we consider the square roots necessary for the norm of the vectors in Equation 7 floating point operations for simplicity's sake. Consequently the number of flops required to compute q vertices based on one input primitive is $150 + 23q$ flops. If we count the floating point operations in Equation 10 we find that 59 flops are required to compute the 'real' normal of one vertex. As a result the number of flops necessary to compute q vertices based on one input primitive is $59q$. Although order of the number of

flops required to compute the normals of q vertices is linear in q, the 'fake' require less flops once a input triangle is tessellated into a shape that contains more than approximately five vertices.

Figure 13a visualizes the 'fake' and 'real' normal field of a cube. The difference image in Figure 13c shows that these two normal fields are the same. A comparison of the 'fake' and 'real' normal field on other models did not reveal any difference either. Thus we can conclude that the rendering is not influenced by the method used to compute the normals.

Incidentally the cubes in Figure 13 illustrate one disadvantage of point-normal triangles, that is otherwise outside of the scope of this paper, namely that PN triangles are not suitable for rendering sharp edges. Figures 13a and 13b show how the originally sharp edges of the cube become curved in a rendering with a high level of detail.

5.2. Pipeline

The first row in Figure 14 shows the rendering by Vlachos et al., below that, we present our rendering of, probably, the same models. The only difference between this set of models should be the tessellation algorithm. A visual inspection shows some differences between the shading of the models, but we estimate that those are due to different material and light properties. As Vlachos et al. did not report those it is near impossible to exactly replicate their results.

The contours of the models is in this case the best place to look for differences. A visual inspection does not reveal any variations between the models rendered on the CPU and those rendered on the GPU. Consequently we tentatively conclude that the difference in final meshes, due to the different triangulations, does not matter.

6. Conclusion

It seems as rendering point-normal triangles on the GPU gives the same results as on the CPU, in spite of the different tessellation algorithms. However, to accurately determine the influence, if any, of the tessellation algorithm on the final rendering, one should implement point-normal triangles on both the CPU and the GPU. This would give full control over the used models and the shading parameters. This ensures that any differences between models rendered on the CPU and the GPU are due to the different tessellation algorithms.

Although 'real' normals result in the same normal as 'fake' normals it is computationally advantageous in nearly all cases to use the 'fake' normals. Only when one renders an object with point normal triangles with the level of detail set to 0, i.e. the inner and outer tessellation levels are 1.0, is it computationally advantageous to use 'real' normals.

References

[BA08] Tamy Boubekeur and Marc Alexa. "Phong tessellation". In: *ACM Transactions on Graphics (TOG)* 27.5 (2008), p. 141.

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

[Far14] Gerald Farin. "Curves and surfaces for computer-aided geometric design: a practical guide". In: Elsevier, 2014. Chap. 17, pp. 279–307.

[Fun+07] Wilson WL Fung et al. "Dynamic warp formation and scheduling for efficient GPU control flow". In: *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society. 2007, pp. 407–420.

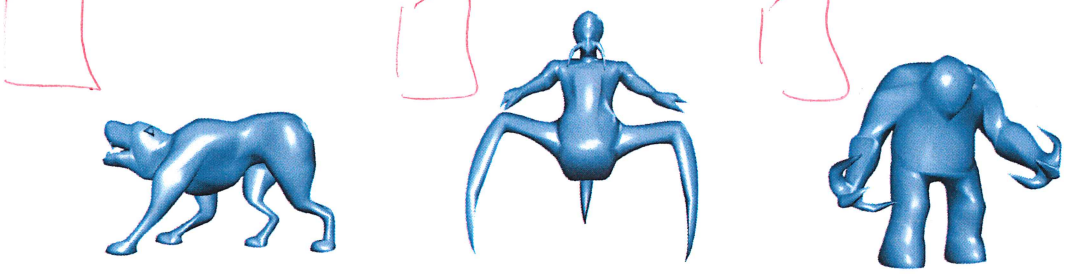
[MK10] J. McDonald and M. Kilgard. *Crack-free point-normal triangles using adjacent edge normals*. 2010.

[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.

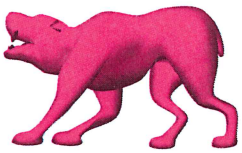
[Wol13] David Wolff. "OpenGL 4 Shading Language Cookbook". In: Packt Publishing Ltd, 2013. Chap. 6, pp. 220–225.



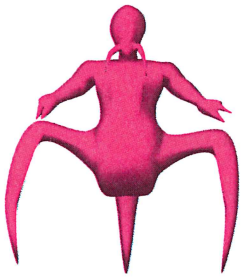
(a) Rottweiler on CPU

(b) Vore on CPU

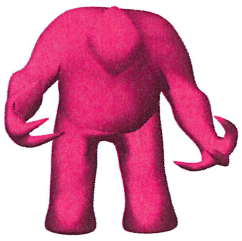
(c) Shambler on CPU



(d) Rottweiler on GPU



(e) Vore on GPU



(f) Shambler on GPU

Figure 14: A family of game characters from the game Quake 1. Figures 14a to 14c are rendered on a CPU by Vlachos et al. Figures 14d to 14f are rendered by our implementation. We rendered our models with the Phong shading and reflection model ($k_s = 0.3$, $k_d = 0.5$, $k_a = 0.4$, $\alpha = 150.0$, color = $(0.988, 0.0, 0.427)$), with a single light source ($I_s = I_d = I_a = (1.0, 1.0, 1.0)$) at $(-2.0, 5.0, -10.0)$. The inner and outer tessellation levels were set to 12.0. The eye was placed at $(2.0, 5.0, -10.0)$. Figures 14a to 14c were taken from Vlachos et al. [Vla+01].