Phong Tessellation for Quads

Jonathan Dupuy[1,2] *
[1]LIRIS, Université Lyon 1 [2]LIGUM, Dept. I.R.O., Université de Montréal

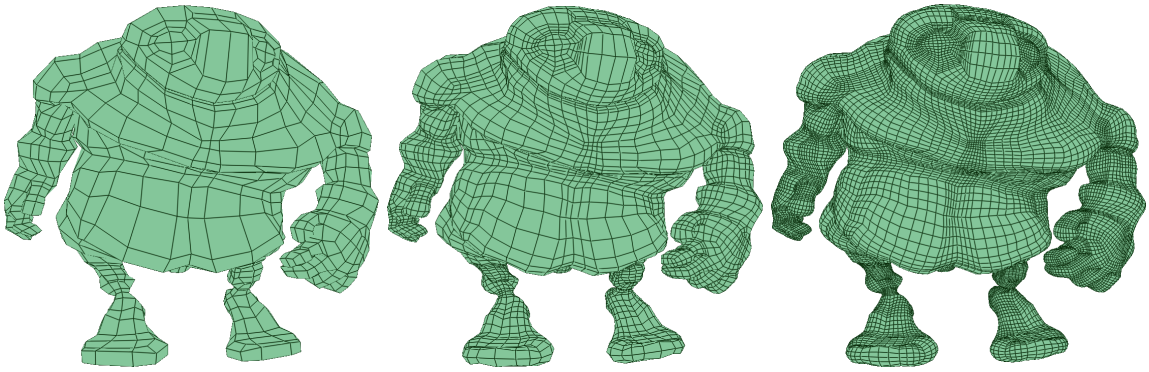

Figure 1: First 3 levels of Phong essellation on a quad mesh (the leftmost surface is equivalent to the input mesh).

Abstract

Phong tessellation is a cheap solution to produce smooth surfaces from coarse meshes using tessellation. Although originally derived for triangles, it can easily be extended to quads. This paper shows how, and additionally derives the formulas to compute the tangents and bitangents of the surface at any location. A complete GLSL program is also provided.

Phong Tessellation for Quads

Phong tessellation provides a projection operator which, once interpolated across a polygon using its barycentric coordinates, produces a smooth surface. Given a quad with vertices p_0, p_1, p_2, p_3 and respective normals n_0, n_1, n_2, n_3, the smooth surface P at barycentric position (u, v) is given by

$$\begin{aligned} P(u,v) \stackrel{\text{def}}{=} &(1-u)\,(1-v)\,\boldsymbol{\pi}(q(u,v),p_0,n_0) \\ &+ u\,(1-v)\,\boldsymbol{\pi}(q(u,v),p_1,n_1) \\ &+ v\,(1-u)\,\boldsymbol{\pi}(q(u,v),p_3,n_3) \\ &+ uv\,\boldsymbol{\pi}(q(u,v),p_2,n_2). \end{aligned} \tag{1}$$

The term $\boldsymbol{\pi}$ is the projection operator as defined originally by Boubekeur and Alex [1]

$$\boldsymbol{\pi}(q(u,v),p,n) = q(u,v) - ((q(u,v)-p)\cdot n)\,n, \tag{2}$$

where $q(u, v)$ is the barycentric interpolation of each vertex

$$q(u,v) = (1-u)\,(1-v)\,p_0 + u\,(1-v)\,p_1 + v\,(1-u)\,p_3 + uv\,p_2. \tag{3}$$

The results are illustrated in Figure 1.

*jdupuy@liris.cnrs.fr

Surface Tangents and Bitangents

Providing a tangent frame on a surface may be useful for a variety of applications such as anisotropic shading for instance. The tangent and bitangent of a Phong Tessellated surface, respectively denoted as $T(u,v)$ and $B(u,v)$, can be retrieved from the gradients of P

$$T(u,v) \stackrel{\text{def}}{=} (\nabla P)_u = \left(\frac{\partial x_P}{\partial u}, \frac{\partial y_P}{\partial u}, \frac{\partial z_P}{\partial u}\right)^t \tag{4}$$

$$B(u,v) \stackrel{\text{def}}{=} (\nabla P)_v = \left(\frac{\partial x_P}{\partial v}, \frac{\partial y_P}{\partial v}, \frac{\partial z_P}{\partial v}\right)^t. \tag{5}$$

A first developement yields (using the identity $(fg)' = f'g + g'f$)

$$\begin{aligned}
T(u,v) &= (v-1)\,\boldsymbol{\pi}(q(u,v),p_0,n_0) + (1-u)\,(1-v)\,(\nabla\boldsymbol{\pi})_u(q(u,v),p_0,n_0)\\
&\quad + (1-v)\,\boldsymbol{\pi}(q(u,v),p_1,n_1) + u\,(1-v)\,(\nabla\boldsymbol{\pi})_u(q(u,v),p_1,n_1)\\
&\quad - v\,\boldsymbol{\pi}(q(u,v),p_3,n_3) + v\,(1-u)\,(\nabla\boldsymbol{\pi})_u(q(u,v),p_3,n_3)\\
&\quad + v\,\boldsymbol{\pi}(q(u,v),p_2,n_2) + uv\,(\nabla\boldsymbol{\pi})_u(q(u,v),p_2,n_2)\\
B(u,v) &= (u-1)\,\boldsymbol{\pi}(q(u,v),p_0,n_0) + (1-u)\,(1-v)\,(\nabla\boldsymbol{\pi})_v(q(u,v),p_0,n_0)\\
&\quad - u\,\boldsymbol{\pi}(q(u,v),p_1,n_1) + u\,(1-v)\,(\nabla\boldsymbol{\pi})_v(q(u,v),p_1,n_1)\\
&\quad + (1-u)\,\boldsymbol{\pi}(q(u,v),p_3,n_3) + v\,(1-u)\,(\nabla\boldsymbol{\pi})_v(q(u,v),p_3,n_3)\\
&\quad + u\,\boldsymbol{\pi}(q(u,v),p_2,n_2) + uv\,(\nabla\boldsymbol{\pi})_v(q(u,v),p_2,n_2).
\end{aligned}$$

The only terms left to solve are the gradients of the projection operator. Using the Chain Rule we find

$$(\nabla\boldsymbol{\pi})_u = J_{\boldsymbol{\pi}}\,(\nabla q)_u \tag{6}$$

$$(\nabla\boldsymbol{\pi})_v = J_{\boldsymbol{\pi}}\,(\nabla q)_v \tag{7}$$

where $J_{\boldsymbol{\pi}}$ is the jacobian of $\boldsymbol{\pi}$ with respect to the components of $q = (x_q, y_q, z_q)$

$$J_{\boldsymbol{\pi}} = \begin{pmatrix} \partial x_{\boldsymbol{\pi}}/\partial x_q & \partial x_{\boldsymbol{\pi}}/\partial y_q & \partial x_{\boldsymbol{\pi}}/\partial z_q \\ \partial y_{\boldsymbol{\pi}}/\partial x_q & \partial y_{\boldsymbol{\pi}}/\partial y_q & \partial y_{\boldsymbol{\pi}}/\partial z_q \\ \partial z_{\boldsymbol{\pi}}/\partial x_q & \partial z_{\boldsymbol{\pi}}/\partial y_q & \partial z_{\boldsymbol{\pi}}/\partial z_q \end{pmatrix} \tag{8}$$

$$= \begin{pmatrix} 1-x_n^2 & -x_n y_n & -x_n z_n \\ -x_n y_n & 1-y_n^2 & -y_n z_n \\ -x_n z_n & -y_n z_n & 1-z_n^2 \end{pmatrix}. \tag{9}$$

Finding the gradients of q is also straightforward

$$(\nabla q)_u = (v-1)\,p_0 + (1-v)\,p_1 - v\,p_3 + v\,p_2 \tag{10}$$

$$(\nabla q)_v = (u-1)\,p_0 - u\,p_1 + (1-u)\,p_3 + u\,p_2. \tag{11}$$

And so Equations (4,5) can be solved completely. As an illustration, Figure 2 shows the cross products of the tangent and bitangent (e.g. the true normals of the surface) of a Phong Tessellated quad mesh. The noticeable discontinuities on the surface are generated at the edges of each patch because Phong tessellation generates C1 surfaces only [2]. In order to avoid this issue, C2 surfaces such as the ones generated by Catmull-Clark subdivision should be used.

References

[1] Tamy Boubekeur and Marc Alexa. Phong Tessellation. *ACM Trans. Graphics (Proc. SIGGRAPH Asia)*, 27, 2008.

[2] Thomas J. Cashman. Beyond Catmull-Clark? A Survey of Advances in Subdivision Surface Methods. *Comput. Graph. Forum*, 31(1):42–61, February 2012.

Figure 2: Left: Phong tessellation *true* normals. Right: Perspective interpolation of the normals provided to the patches before tessellation.

GLSL Program

Below is a complete GLSL program to produce Phong Tessellated surfaces with an OpenGL4+ GPU. The shaders were tested on both Nvidia and AMD cards. The code puts emphasis on readability rather than peformance and is naturally far from optimal.

Listing 1: Vertex Shader

```
// Vertex shader
#version 420
in vec3 i_position;
in vec3 i_normal;
layout(location = 0) out vec3 o_normal;

void main() {
  gl_Position.xyz = i_position;
  o_normal = i_normal;
}
```

Listing 2: Tessellation Control Shader

```
// Tessellation Control shader
#version 420
layout(vertices = 4) out;
layout(location = 0) in vec3 i_normal[];
layout(location = 0) out vec3 o_normal[];

uniform vec2 u_inner;
uniform vec4 u_outer;

void main() {
  // copy position and normal
  gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
  o_normal[gl_InvocationID] = i_normal[gl_InvocationID];

  // set inner tess levels
  gl_TessLevelInner[0] = u_inner.x;
  gl_TessLevelInner[1] = u_inner.y;

  // set outer tess levels
  gl_TessLevelOuter[0] = u_outer.x;
  gl_TessLevelOuter[1] = u_outer.y;
  gl_TessLevelOuter[2] = u_outer.z;
  gl_TessLevelOuter[3] = u_outer.w;
}
```

Listing 3: Tessellation Evaluation Shader

```
// Tessellation Evaluation shader
#version 420
layout(quads, equal_spacing, ccw) in;
layout(location = 0) in vec3 i_normal[];
layout(location = 0) out vec3 o_tangent;
layout(location = 1) out vec3 o_bitangent;

uniform mat4 u_mvp;

void main() {
  // barycentric coordinates
  float u = gl_TessCoord.x;
  float v = gl_TessCoord.y;

  // patch vertices
  vec3 p0 = gl_in[0].gl_Position.xyz;
  vec3 p1 = gl_in[1].gl_Position.xyz;
  vec3 p2 = gl_in[2].gl_Position.xyz;
  vec3 p3 = gl_in[3].gl_Position.xyz;

  // patch normals
  vec3 n0 = i_normal[0];
  vec3 n1 = i_normal[1];
  vec3 n2 = i_normal[2];
  vec3 n3 = i_normal[3];

  // get interpolated position
  vec3 q = pt_q(p0,p1,p2,p3,u,v);

  // smooth surface position
  vec3 p = (1.0-u) * (1.0-v) * pt_pi(q, p0, n0)
         + u * (1.0-v) * pt_pi(q, p1, n1)
         + (1.0-u) * v * pt_pi(q, p3, n3)
         + u * v * pt_pi(q, p2, n2);

  // smooth surface tangent
  vec3 dqdu = pt_dqdu(p0, p1, p2, p3, v);
  vec3 t = (v-1.0) * pt_pi(q, p0, n0) + (1.0-u) * (1.0-v) * pt_dpidu(dqdu, p0, n0)
         + (1.0-v) * pt_pi(q, p1, n1) + u * (1.0-v) * pt_dpidu(dqdu, p1, n1)
         - v * pt_pi(q, p3, n3) + v * (1.0-u) * pt_dpidu(dqdu, p3, n3)
         + v * pt_pi(q, p2, n2) + u * v * pt_dpidu(dqdu, p2, n2);

  // smooth surface bitangent
  vec3 dqdv = pt_dqdv(p0, p1, p2, p3, u);
  vec3 b = (u-1.0) * pt_pi(q, p0, n0) + (1.0-u) * (1.0-v) * pt_dpidv(dqdv, p0, n0)
         - u * pt_pi(q, p1, n1) + u * (1.0-v) * pt_dpidv(dqdv, p1, n1)
         + (1.0-u) * pt_pi(q, p3, n3) + v * (1.0-u) * pt_dpidv(dqdv, p3, n3)
         + u * pt_pi(q, p2, n2) + u * v * pt_dpidv(dqdv, p2, n2);

  // set varyings
  o_tangent = normalize(t);
  o_bitangent = normalize(b);

  // project vertex
  gl_Position = u_mvp * vec4(p, 1.0);
}
```

Listing 4: Fragment Shader

```
// Fragment shader
#version 420
layout(location = 0) in vec3 i_tangent;
layout(location = 1) in vec3 o_bitangent;
layout(location = 0) out vec3 o_colour;

void main() {
  vec3 t = normalize(i_tangent);
  vec3 b = normalize(i_bitangent);

  // show Phong tessellation true normal
  o_colour = cross(-t,b);
}
```

Listing 5: Phong Tessellation Functions

```
// q (interpolated position)
vec3 pt_q( in vec3 p0, in vec3 p1, in vec3 p2, in vec3 p3, in float u, in float v ) {
  return (1.0-u) * (1.0-v) * p0 + u * (1.0-v) * p1 + (1.0-u) * v * p3 + u * v * p2;
}

// interpolated normal (same as interpolated position)
vec3 pt_n( in vec3 n0, in vec3 n1, in vec3 n2, in vec3 n3, in float u, in float v ) {
  return pt_q(n0, n1, n2, n3, u, v);
}

// dq / dv
vec3 pt_dqdu(in vec3 p0, in vec3 p1, in vec3 p2, in vec3 p3, in float v ) {
  return (v-1.0) * p0 + (1.0-v) * p1 - v * p3 + v * p2;
}

// dq / dv
vec3 pt_dqdv(in vec3 p0, in vec3 p1, in vec3 p2, in vec3 p3, in float u ) {
  return (u-1.0) * p0 - u * p1 + (1.0-u) * p3 + u * p2;
}

// pi (projection operator)
vec3 pt_pi( in vec3 q, in vec3 p, in vec3 n ) {
  return q - dot(q - p, n) * n;
}

// dpi / du (gradient of the projection operator with respect to u)
vec3 pt_dpidu( in vec3 dqdu, in vec3 p, in vec3 n ) {
  vec3 gradx = vec3(1.0 - n.x * n.x, -n.y * n.x, -n.z * n.x);
  vec3 grady = vec3(-n.x * n.y, 1.0 - n.y * n.y, -n.z * n.y);
  vec3 gradz = vec3(-n.x * n.z, -n.y * n.z, 1.0 - n.z * n.z);
  return mat3(gradx, grady, gradz) * dqdu;

}

// dpi / dv (gradient of the projection operator with respect to v)
vec3 pt_dpidv( in vec3 dqdv, in vec3 p, in vec3 n ) {
  vec3 gradx = vec3(1.0 - n.x * n.x, -n.y * n.x, -n.z * n.x);
  vec3 grady = vec3(-n.x * n.y, 1.0 - n.y * n.y, -n.z * n.y);
  vec3 gradz = vec3(-n.x * n.z, -n.y * n.z, 1.0 - n.z * n.z);
  return mat3(gradx, grady, gradz) * dqdv;
}
```