An Introduction to Tessellation Shaders 6

Philip Rideout and Dirk Van Gelder

6.1 Introduction

Tessellation shaders open new doors for real-time graphics programming. GPU-based tessellation was possible in the past only through trickery, relying on multiple passes and misappropriation of existing shader units.

OpenGL 4.0 finally provides first-class support for GPU tessellation, but the new shading stages can seem nonintuitive at first. This chapter explains the distinct roles of those stages in the new pipeline and gives an overview of some common rendering techniques that leverage them.

GPUs tend to be better at "streamable" amplification; rather than storing an entire post-subdivided mesh in memory, tessellation shaders allow vertex data to be amplified on the fly, discarding the data when they reach the rasterizer. The system never bothers to store a highly-refined vertex buffer, which would have an impractical memory footprint for a GPU.

Pretessellation graphics hardware was already quite good at rendering huge meshes, and CPU-side refinement was often perfectly acceptable for static meshes. So why move tessellation to the GPU?

The gains are obvious for animation. On a per-frame basis, only the control points get sent to the GPU, greatly alleviating bandwidth requirements for high-density surfaces.

Animation isn't the only killer application of subdivision surfaces. *Displacement mapping* allows for staggering geometric level-of-detail. Previous GPU techniques required multiple passes over the geometry shader, proving awkward and slow. Tessellation shaders allow displacement mapping to occur in a single pass [Castaño 08].

Tessellation shaders can also compute geometric level-of-detail on the fly, which we'll explore later in the chapter. Previous techniques required the CPU to resubmit new vertex buffers when changing the level-of-detail.

6.1.1 Subdivision Surfaces

One of the most compelling uses of GPU tessellation is efficiently rendering Catmull-Clark subdivision surfaces. Most of these techniques use tessellation shaders to evaluate a parametric approximation of the limit surface rather than performing iterative subdivision. Iterative subdivision can still be done on the GPU but is often better suited for CUDA or OpenCL.

Parametric approximation of Catmull-Clark surfaces (ACC) arose from Charles Loop's research at Microsoft in 2008 [Loop and Schaefer 08], and was subsequently enhanced to support creases [Kovacs et al. 09]. An excellent overview of the state of the art can be found in [Ni et al. 09]. This includes a report from Valve, the first major game developer to use tessellation shaders in this way.

6.1.2 Smoothing Polygonal Data

Catmull-Clark surfaces are not the only way to make good use of tessellation shaders; game developers may find other surface definitions more attractive. For example, tessellation can be used to simply "smooth out" traditional polygonal mesh data. PN triangles are a popular example of this. An even simpler application is *Phong tessellation*, the geometric analogue of a Phong lighting.

6.1.3 GPU Compute

OpenCL or CUDA can be used in conjunction with tessellation shaders for various techniques. The compute API can be used for simulation, e.g., hair physics, [Yuksel and Tariq 10], or it can be used to perform a small number of iterative subdivisions to "clean up" the input mesh, removing extraordinary vertices before submitting the data to the OpenGL pipeline [Loop 10].

6.1.4 Curves, Hair, and Grass

Tessellation shaders can also be applied to lines with isoline tessellation, which opens up several possibilities for data amplification. One is tessellating a series of line segments into a smooth cubic curve. In this way, application code works only with a small number of points. Smooth curves are generated entirely on the GPU, either for 3D applications like hair or rope or for 2D applications such as Bézier curves from a drafting tool. Isoline tessellation can also be used to generate multiple curves from a single curve.

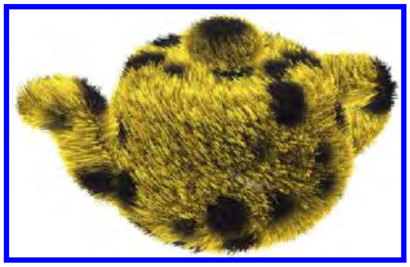

Figure 6.1. Hairy teapot; lines grown from patches.

Geometry shaders can be used in conjunction with isoline tessellation, which can be useful for applications such as grass and hair. Figure 6.1 is a screenshot from the accompanying sample code in which a surface is tessellated into many small polygons, then extruded into hairs using a geometry shader.

6.1.5 Other Uses

There are also many less obvious uses for tessellation. If a post-tessellated mesh is sufficiently refined, its geometry can be deformed to simulate lens distortion. These effects include pincushion warping and panoramic projection. Because GPU rasterizers can only perform linear interpolation, traditional techniques relying on post-processing often result in poor sampling.

Figure 6.2 depicts an example of cylindrical warping using tessellation shaders applied to a cubescape. The vertex buffer sent to the GPU is extremely light because each cube face is a 4-vertex patch.

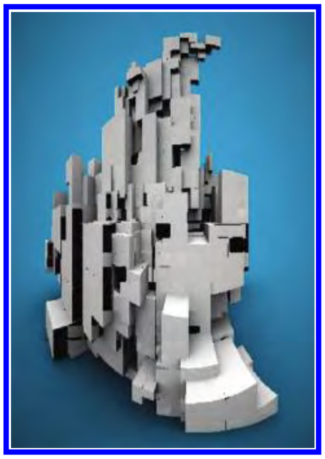

Figure 6.2. Cylindrical distortion using tessellation shaders.

6.2 The New Shading Pipeline

Figure 6.3 depicts a simplified view of the OpenGL shading pipeline, highlighting the new OpenGL 4.0 stages. The pipeline has two new shader stages and one new fixed-function stage.

Those who come across Direct3D literature should be aware that the control shader is there known as the *hull shader*; the evaluation shader is known as the *domain shader*.

To start off, OpenGL 4.0 introduces a new primitive type, `GL_PATCHES`, that must be used to leverage tessellation functionality. Unlike every other OpenGL primitive, patches have a user-defined number of vertices per primitive, configured like so:

```
glPatchParameteri(GL_PATCH_VERTICES, 16);
```

The tessellator can be configured in one of three domains: `isolines`, `quads`, and `triangles`. Later in the chapter, we'll examine each of these modes in detail.

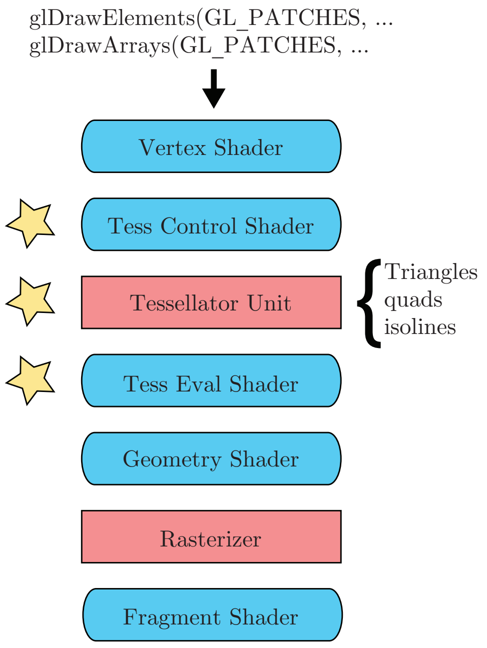

Figure 6.3. The new tessellation stages in OpenGL 4.0.

6.2.1 Life of a Patch

Although vertex data always starts off arranged into patch primitives, it gets transformed as it progresses through the pipe, as depicted in Figure 6.4.

If desired, the tessellation control shader can perform some of the same transformations that were traditionally done in the vertex shader. However, unlike the vertex shader, the tessellation control shader has access to all data within the local patch as well as a patch-relative *invocation identifier*. It also acts as a configurator for the fixed-function tessellator stage, telling it how to split up the patch into triangles or lines. The tessellation control shader can be thought of as a "control point shader" because it operates on the original, pretessellated vertices.

Next, the tessellator stage inserts new vertices into the vertex stream according to tessellation levels stipulated by the control shader and the tessellation mode stipulated by the evaluation shader.

The evaluation shader then transforms vertices in the expanded stream, making them able to read data from any vertex within the local patch.

After this point in the OpenGL pipeline, vertices are finally arranged into triangles or lines, and the patch concept is effectively discarded.

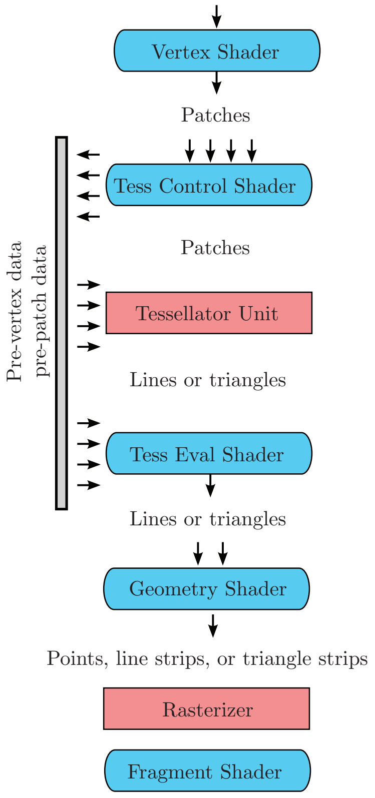

Figure 6.4. Tessellation data flow; GLSL arrays are depicted by multiple incident arrows.

6.2.2 Threading Model

Table 6.1 shows how the new programable stages are invocated, relative to the number of elements in the vertex buffer.

The threading model of the control shader is unique in that the relative order of multiple invocations is somewhat controllable, and it can access a shared read/write area for each patch.

Control shaders allow developers to specify a synchronization point where all invocations for a patch must wait for other threads to reach the same point. Such a synchronization point can be defined using the built-in `barrier()` function.

Unit	Invocation Scheme
Vertex Shader	one invocation per input vertex
Tess Control Shader	one invocation per output vertex
Tess Eval Shader	one invocation per post-tessellated vertex

Table 6.1. Threading in the new OpenGL shading pipeline.

The function `barrier()` is different from `memoryBarrier()`; the latter was introducted in OpenGL 4.2 and can be used from *any* shader unit.

Control shaders can access per-patch shared memory by qualifying a set of `out` variables with the new `patch` keyword. If multiple invocations within a patch write different values into the same `patch` variable, the results are undefined.

6.2.3 Inputs and Outputs

Table 6.2 enumerates all the built-in variables available to the two tessellation shader stages.

The built-in arrays of struct, `gl_in` and `gl_out`, provide access to vertex position, point size, and clipping distance. These are the same variables that can be output from the vertex shader and processed by the geometry shader.

In addition to the built-ins in Table 6.2, tessellation shaders can declare a set of custom `in` and `out` variables as usual. Per-vertex data must always be declared as

Identifier	Shader Unit(s)	Access
`gl_PatchVerticesIn`	Control and Eval	in
`gl_PrimitiveID`	Control and Eval	in
`gl_InvocationID`	Control Shader	in
`gl_TessLevelOuter[4]`	Control Shader	out
`gl_TessLevelInner[2]`	Control Shader	out
`gl_TessLevelOuter[4]`	Evaluation Shader	in
`gl_TessLevelInner[2]`	Evaluation Shader	in
`gl_in[`n`].gl_Position`	Control and Eval	in
`gl_in[`n`].gl_PointSize`	Control and Eval	in
`gl_in[`n`].gl_ClipDistance[`m`]`	Control and Eval	in
`gl_out[`n`].gl_Position`	Control and Eval	out
`gl_out[`n`].gl_PointSize`	Control and Eval	out
`gl_out[`n`].gl_ClipDistance[`m`]`	Control and Eval	out
`gl_TessCoord`	Evaluation	in

Table 6.2. Built-in GLSL variables for tessellation.

an array, where each element of the array corresponds to a single element within the patch. Per-patch data, qualified with `patch`, is not arrayed over the patch.

Tessellation shaders have read-only access to `gl_PatchVerticesIn`, which represents the number of vertices in a patch. In the evaluation shader, this number can vary between patches.

The read-only `gl_PrimitiveID` variable is also available to both tessellation shaders. This describes the index of the patch within the current draw call.

As with any other shader stage in OpenGL, tessellation shaders can also read from uniforms, uniform buffers, and textures.

6.2.4 Tessellation Control Shaders

This stage is well suited for change-of-basis transformations and deciding on level-of-detail. Control shaders can also be used to achieve early rejection by culling patches when all corners are outside the viewing frustum, although this gets tricky if the evaluation shader performs displacement.

Listing 6.1 presents a template for a tessellation control shader.

```
layout(vertices = output_patch_size) out;

// Declare inputs from vertex shader
in float vFoo[];

// Declare per-vertex outputs
out float tcFoo[];

// Declare per-patch outputs
patch out float tcSharedFoo;

void main()
{
  bool cull = ...;
  if (cull)
  {
    gl_TessLevelOuter[0] = 0.0;
    gl_TessLevelOuter[1] = 0.0;
    gl_TessLevelOuter[2] = 0.0;
    gl_TessLevelOuter[3] = 0.0;
  }
  else
  {
    // Compute gl_TessLevelInner...
    // Compute gl_TessLevelOuter...
  }
  // Write per-patch data...
  // Write per-vertex data...
}
```

Listing 6.1. Tessellation control shader template.

The layout declaration at the top of the shader defines not only the size of the output patch but also the number of invocations of the control shader for a given input patch. All custom `out` variables must be declared as arrays that are either explicitly sized to match this count or implicitly using empty square brackets.

The size of the input patch is defined at the API level using `glPatchParameteri`, but the size of the output patch is defined at the shader level. In many cases, we want these two sizes to be the same. Heavy insertion of new elements into the vertex stream is best done by the fixed-function tessellator unit and not by the control shader. The implementation-defined maximum for both sizes can be queried at the API level using `GL_MAX_PATCH_VERTICES`. At the time of this writing, 32 is the common maximum.

The application code can determine the output patch size defined by the active shader:

```
GLuint patchSize;
glGetIntegerv(GL_TESS_CONTROL_OUTPUT_VERTICES, &patchSize);
```

Tessellation modes. The tessellation mode (known as *domain* in Direct3D parlance) is configured using a layout declaration in the evaluation shader. There are three modes available in OpenGL 4.0:

- `triangles`. Subdivides a triangle into triangles.
- `quads`. Subdivides a quadrilateral into triangles.
- `isolines`. Subdivides a quadrilateral into a collection of line strips.

The array `gl_OuterTessLevel[]` always has four elements, and `gl_InnerTessLevel` always has two elements, but only a subset of each array is used depending on the tessellation mode. Similarly, `gl_TessCoord` is always a `vec3`, but its z component is ignored for isolines and quads. Table 6.3 summarizes how domain affects built-in variables.

Domain	Outer	Inner	TessCoord
`triangles`	3	1	3D (Barycentric)
`quads`	4	2	2D (Cartesian)
`isolines`	2	0	2D (Cartesian)

Table 6.3. The effective sizes of the tess level arrays and the `gl_TessCoord` vector.

Fractional tessellation levels. The inner and outer tessellation levels control the number of subdivisions along various edges. All tessellation levels are floating points, not integers. The fractional part can have a different meaning depending on the *spacing* (known as *partitioning* in Direct3D parlance). Spacing is configured in the evaluation shader using a `layout` declaration. For example,

```
layout(quads, equal_spacing) in;
```

The three spacing schemes are

- `equal_spacing`. Clamp the tess level to [1,*max*]; then round up to the nearest integer. Every new segment has equal length.
- `fractional_even_spacing`. Clamp the tess level to [2,*max*]; then round up to the nearest even integer. Every new segment has equal length, except for the two segments at either end, whose size is proportional to the fractional part of the clamped tess level.
- `fractional_odd_spacing`. Clamp the tess level to [1,*max*-1]; then round up to the nearest odd integer. Every new segment has equal length, except for the two segments at either end, whose size is proportional to the fractional part of the clamped tess level.

In the above descriptions, *max* refers to the value returned by

```
GLuint maxLevel;
glGetIntegerv(GL_MAX_TESS_GEN_LEVEL, &maxLevel);
```

If we're computing tessellation levels on the fly, the two fractional spacing modes can be used to create a smooth transition between levels, resulting in a diminished popping effect. See Figure 6.5 for how fractional tessellation levels can affect edge subdivision.

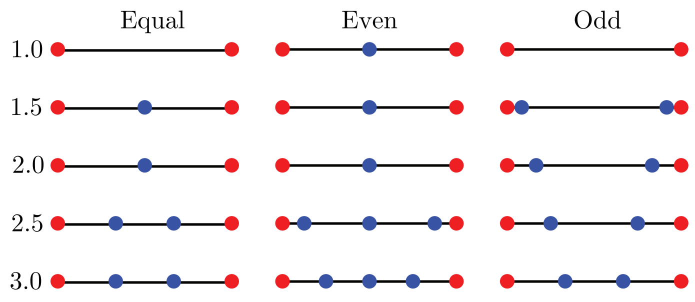

Figure 6.5. Fractional tessellation levels.

Computing the tessellation levels. Writing to `gl_TessLevelInner` and `gl_TessLevelOuter` is optional; if they are not set by the control shader, OpenGL falls back to the API-defined defaults. Initially, these defaults are filled with 1.0, but they can be changed like so:

```
GLfloat inner[2] = { ... };
GLfloat outer[4] = { ... };
glPatchParameterfv(GL_PATCH_DEFAULT_INNER_LEVEL, inner);
glPatchParameterfv(GL_PATCH_DEFAULT_OUTER_LEVEL, outer);
```

At the time of this writing, the latest drivers do not always honor the defaults, so the tessellation levels should always be set from the shader. In practical applications, we often need to compute this dynamically anyway, which is known as *adaptive tessellation*. One approach for computing the level-of-detail is based on screen-space edge lengths:

```
uniform float GlobalQuality;

float ComputeTessFactor(vec2 ssPosition0, vec2 ssPosition1)
{
  float d = distance(ssPosition0, ssPosition1);
  return clamp(d * GlobalQuality, 0.0, 1.0);
}
```

The `GlobalQuality` constant may be computed in application code using the following heuristic:

$$\texttt{GlobalQuality} = 1.0/(\texttt{TargetEdgeSize} * \texttt{MaxTessFactor}).$$

Another adaptive scheme uses the orientation of the patch relative to the viewing angle, leading to higher tessellation along silhouettes. This technique requires an edge normal, which can be obtained by averaging the normals at the two endpoints:

```
uniform vec3 ViewVector;
uniform float Epsilon;

float ComputeTessFactor(vec3 osNormal0, vec3 osNormal1)
{
  float n = normalize(mix(0.5, osNormal0, osNormal1));
  float s = 1.0 - abs(dot(n, ViewVector));
  s = (s - Epsilon) / (1.0 - Epsilon);
  return clamp(s, 0.0, 1.0);
}
```

For more on dynamic level-of-detail, see Chapter 10.

6.2.5 Tessellation Evaluation Shaders

The evaluation stage is well suited for parametric evaluation of patches and computation of smooth normal vectors.

```
layout(quads, fractional_even_spacing, cw) out;

// Declare inputs from tess control shader
in float tcFoo[];

// Declare per-patch inputs
patch in float tcSharedFoo;

// Declare per-vertex outputs
out float teFoo;

void main()
{
  vec3 tc = gl_TessCoord;
  teFoo = ... ;
  gl_Position = ... ;
}
```

Listing 6.2. Tessellation evaluation shader template.

Listing 6.2 presents a template for a tessellation evaluation shader. Unlike the control shader, the outputs are not arrayed over the patch.

For a visualization of `gl_TessCoord` in `quads` mode, see Figure 6.6. The meaning of `gl_TessCoord` varies according to the tessellation mode. For example, in `triangles` mode, it's a Barycentric coordinate; see Table 6.3.

By default, the progression of `gl_TessCoord` is counter-clockwise for every triangle. This is consistent with OpenGL's default definition of front-facing polygons. If desired, the `layout` declaration can flip this behavior using the `cw` token.

By default, the evaluation shader generates triangles for `quads` and `triangles` domains and lines for the `isolines` domain. However, any domain can be overridden to generate point primitives by adding the `point_mode` token to the layout declaration.

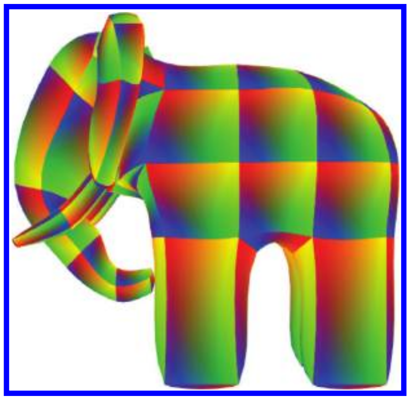

Figure 6.6. Gumbo's bicubic patches and their `gl_TessCoord` parameterizations.

6.2.6 Primitive Generation Using quads

The procedure for tessellation in the quads domain is described next (follow along with Figure 6.7):

1. The edges of a rectangular input patch are fed into the tessellator.
2. The patch is first divided into quads according to the two inner tessellation levels.
3. All of the quads produced by Step 1 except the bordering quads are decomposed into triangle pairs.
4. The outer edges of the patch are then subdivided according to the four outer tessellation levels.
5. The outer ring is then filled with triangles by connecting the points from Step 2 with the points from Step 4. The algorithm for this step is implementation-dependent.

Figure 6.7 illustrates this procedure using the following tessellation levels:

```
gl_TessLevelInner = { 4, 5 };
gl_TessLevelOuter = { 2, 3, 2, 4 };
```

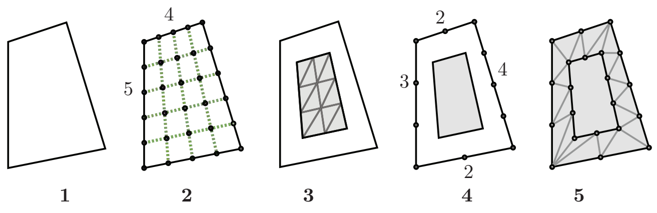

Figure 6.7. Primitive generation in the quads domain.

6.2.7 Primitive Generation Using triangles

Next, we'll describe the procedure for tessellation in the triangles domain, following along with Figure 6.8:

1. The edges of a triangular patch are fed into the tessellator.
2. The patch is first divided into concentric triangles according to the inner tessellation level.

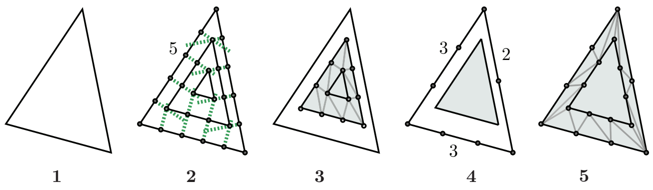

Figure 6.8. Primitive generation in the `triangles` domain.

3. The spaces between the concentric triangles, except the outer ring, are decomposed into triangles.
4. The outer edges of the triangles are then subdivided according to the three outer tessellation levels.
5. The outer ring is then filled with triangles by connecting the points from Step 2 with the points from Step 4.

The concentric triangles in Step 2 are formed from the intersections of perpendicular lines extending from the original edges.

Figure 6.8 illustrates this procedure using the following tessellation levels:

```
gl_TessLevelInner = { 5 };
gl_TessLevelOuter = { 3, 3, 2 };
```

6.3 Tessellating a Teapot

This section illustrates tessellation in the `quads` domain using simple bicubic patches. Conveniently, the famous Utah Teapot was originally modelled using bicubic patches. Tessellation levels for this demo are depicted in Figure 6.9.

Since we are not performing skinning or other deformations, we defer model-view-projection transformation until the evaluation shader; this makes our vertex shader trivial. See Listing 6.3.

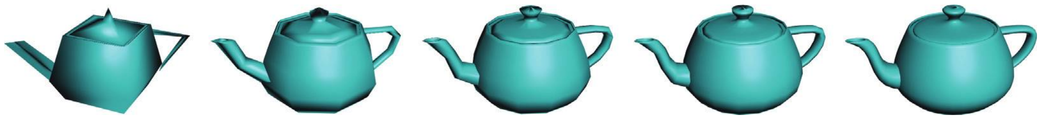

Figure 6.9. From left to right: inner and outer tess levels of 1, 2, 3, 4, and 7.

```
in vec3 Position;
out vec3 vPosition;

void main()
{
  vPosition = Position;
}
```

Listing 6.3. Teapot vertex shader.

Before diving into the control shader, a brief review of bicubic patches is in order. In its most general form, the parametric formulation of a bicubic surface uses a total of 48 coefficients:

$$\begin{aligned} x(u,v) &= a_x u^3 v^3 + b_x u^3 v^2 + c_x u^3 v + d_x u^3 + e_x u^2 v^3 + \ldots p_x, \\ y(u,v) &= a_y u^3 v^3 + b_y u^3 v^2 + c_y u^3 v + d_y u^3 + e_y u^2 v^3 + \ldots p_y, \\ z(u,v) &= a_z u^3 v^3 + b_z u^3 v^2 + c_z u^3 v + d_z u^3 + e_z u^2 v^3 + \ldots p_z. \end{aligned} \tag{6.1}$$

The (u, v) coordinates in the above formulation correspond to `gl_TessCoord` in the evaluation shader.

The 48 coefficients can be neatly arranged into 4×4 matrices. We can denote a_x through p_x with the matrix $\mathbf{C}_x$:

$$x(u,v) = (u^3\ u^2\ u\ 1)\ \ \mathbf{C}_x \begin{pmatrix} v^3 \\ v^2 \\ v \\ 1 \end{pmatrix}.$$

Given a set of knot points, we need to generate a set of coefficient matrices ($\mathbf{C}_x$, $\mathbf{C}_y$, $\mathbf{C}_z$). First, we select a basis matrix from a list of popular choices (e.g., Bézier, B-spline, Catmull-Rom, and Hermite) and represent it with $\mathbf{B}$. Next, we arrange the knot points into matrices ($\mathbf{P}_x$, $\mathbf{P}_y$, $\mathbf{P}_z$). The coefficient matrices can then be derived as follows:

$$\begin{aligned} \mathbf{C}_x &= \mathbf{B} * \mathbf{P}_x * \mathbf{B}^T, \\ \mathbf{C}_y &= \mathbf{B} * \mathbf{P}_y * \mathbf{B}^T, \\ \mathbf{C}_z &= \mathbf{B} * \mathbf{P}_z * \mathbf{B}^T. \end{aligned}$$

Because the coefficient matrices are constant over the patch, computing them should be done in the control shader rather than the evaluation shader. See Listing 6.4.

Listing 6.4 does not make the best use of the threading model. Listing 6.5 makes a 3 times improvement by performing the computations for each dimension (x, y, z) across separate invocations.

In some cases, the first return statement in Listing 6.5 will not improve performance due to the SIMD nature of shader execution.

```
layout(vertices = 16) out;
in vec3 vPosition[];
out vec3 tcPosition[];
patch out mat4 cx, cy, cz;
uniform mat4 B, BT;

#define ID gl_InvocationID

void main()
{
  tcPosition[ID] = vPosition[ID];

  mat4 Px, Py, Pz;
  for (int idx = 0; idx < 16; ++idx)
  {
    Px[idx / 4][idx % 4] = vPosition[idx].x;
    Py[idx / 4][idx % 4] = vPosition[idx].y;
    Pz[idx / 4][idx % 4] = vPosition[idx].z;
  }

  // Perform the change of basis:
  cx = B * Px * BT;
  cy = B * Py * BT;
  cz = B * Pz * BT;
}
```

Listing 6.4. Teapot control shader.

```
layout(vertices = 16) out;
in vec3 vPosition[];
out vec3 tcPosition[];
patch out mat4 c[3];
uniform mat4 B, BT;

#define ID gl_InvocationID

void main()
{
  tcPosition[ID] = vPosition[ID];
  tcNormal[ID] = vNormal[ID];
  if (ID > 2)
  {
    return;
  }

  mat4 P;
  for (int idx = 0; idx < 16; ++idx)
  {
    P[idx / 4][idx % 4] = vPosition[idx][ID];
  }

  // Perform the change of basis:
  c[ID] = B * P * BT;
}
```

Listing 6.5. Improved control shader.

```
layout(quads) in;
in vec3 tcPosition[];
patch in mat4 cx, cy, cz;
uniform mat4 Projection;
uniform mat4 Modelview;

void main()
{
  float u = gl_TessCoord.x, v = gl_TessCoord.y;
  vec4 U = vec4(u * u * u, u * u, u, 1);
  vec4 V = vec4(v * v * v, v * v, v, 1);
  float x = dot(cx * V, U);
  float y = dot(cy * V, U);
  float z = dot(cz * V, U);
  gl_Position = Projection * Modelview * vec4(x, y, z, 1);
}
```

Listing 6.6. Evaluation shader.

Current drivers have trouble with varying arrays of matrices; we had to replace the `c[]` array with three separate matrices.

Further gains could be achieved by removing the `for` loop and using the `barrier` instruction, but current drivers do not support the `barrier` instruction robustly.

Next, we come to the evaluation shader, which is best suited for performing the computations in Equation 6.1 and performing the model-view-projection transform. See Listing 6.6.

6.4 Isolines and Spirals

So far, we've examined the `triangles` and `quads` domains, which both decompose input patches into many tiny polygons. The remaining tessellation mode, `isolines`, changes each input patch into a series of line segments. Listing 6.7 is an excerpt from an evaluation shader that generates multiple smooth curves from a single coarse curve.

This shader requests that the tessellator unit generates evenly spaced isolines. The control shader needs to specify values for only two of the outer tessellation levels, and all inner levels are ignored. Specifically, `gl_TessLevelOuter[0]` describes how many curves to generate, and `gl_TessLevelOuter[1]` describes how many samples generate for each of those curves. For example, if our application needs to turn a coarsely specified curve into a single smooth curve, set `gl_TessLevelOuter[0]` to 1.0 and set `gl_TessLevelOuter[1]` to 64.0 to finely sample the output curve. Conversely, setting `gl_TessLevelOuter[0]` to 64.0 and `gl_TessLevelOuter[1]` to 4.0 causes the tessellator to generate 64 coarse curves.

Listing 6.7 performs B-spline interpolation between the four vertices of each patch, using `gl_TessCoord.x` to indicate the parametric position along the curve,

```
layout(isolines, equal_spacing, cw) in;

void main()
{
  float u = gl_TessCoord.x, v = gl_TessCoord.y;

  float B[4];
  EvalCubicBSpline(u, B); // See accompanying sample for definition

  vec4 pos = B[0] * gl_in[0].gl_Position +
             B[1] * gl_in[1].gl_Position +
             B[2] * gl_in[2].gl_Position +
             B[3] * gl_in[3].gl_Position;

  // Offset in the y coordinate using v so multiple
  // curves aren't drawn on top of each other.
  pos += vec4(0.0, v * 5.0, 0.0, 0.0);

  gl_Position = Projection * Modelview * pos;
}
```

Listing 6.7. Spirals shader.

and `gl_TessCoord.y` here is used to offset different curves generated by the tessellator unit.

In this example, a series of five "patches" are created in a spiral with four vertices each. In the first image, both outer tessellation levels are set to one, so we get a single curve. See Figure 6.10.

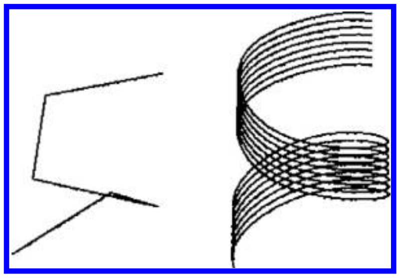

Figure 6.10. Isoline control points (left). Post-tessellated curves (right).

6.5 Incorporating Other OpenGL Features

Many types of animation and deformation are well suited to the current vertex shader. For example, skinning is still optimally done in a vertex shader; NVIDIA's Gregory patch demo is one example of this.

OpenGL's *transform feedback* functionality can be used to turn off the rasterizer and send post-tessellated data back to the CPU, possibly for verification or debugging, further processing, or to be leveraged by a CPU-side production-quality renderer.

Transform feedback could also be used to perform iterative refinement, although this is rarely done in practice due to the large memory requirements of the resulting vertex buffers. For more on transform feedback, see Chapter 17.

Bibliography

[Castaño 08] Ignacio Castaño. "Displaced Subdivision Surfaces." Presented at Gamefest: http://developer.download.nvidia.com/presentations/2008/Gamefest/Gamefest2008-DisplacedSubdivisionSurfaceTessellation-Slides.PDF, 2008.

[Kovacs et al. 09] Denis Kovacs, Jason Mitchell, Shanon Drone, and Denis Zorin. "Real-Time Creased Approximate Subdivision Surfaces." In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, *I3D '09*, pp. 155–160. New York: ACM, 2009.

[Loop and Schaefer 08] Charles Loop and Scott Schaefer. "Approximating Catmull-Clark subdivision surfaces with bicubic patches." *ACM Trans. Graph.* 27 (2008), 8:1–8:11. Available online (http://doi.acm.org/10.1145/1330511.1330519).

[Loop 10] Charles Loop. "Hardware Subdivision and Tessellation of Catmull-Clark Surfaces." Presented at GTC. http://www.nvidia.com/content/GTC-2010/pdfs/2129_GTC2010.pdf, 2010.

[Ni et al. 09] Tianyun Ni, Ignacio Castaño, Jörg Peters, Jason Mitchell, Philip Schneider, and Vivek Verma. "Efficient Substitutes for Subdivision Surfaces." In *ACM SIGGRAPH 2009 Courses*, *SIGGRAPH '09*, pp. 13:1–13:107. New York: ACM, 2009.

[Yuksel and Tariq 10] Cem Yuksel and Sarah Tariq. "Advanced Techniques in Real-Time Hair Rendering and Simulation." In *ACM SIGGRAPH 2010 Courses*, *SIGGRAPH '10*, pp. 1:1–1:168. New York: ACM, 2010. Available online (http://doi.acm.org/10.1145/1837101.1837102).