COSC363 Computer Graphics

# Lab10: OpenGL-4 Tessellation

## Aim:

This lab introduces tessellation control and evaluation shaders of the OpenGL-4 pipeline, and demonstrates their applications in mesh tessellation and modelling.

## I. Quad_Patches.cpp:

In this exercise, we will use tessellation shaders to model a smooth curved surface segment in three-dimensional space. The program specifies the coordinates of 9 patch vertices in the array "patchVerts[]" (line 98). The relative positions of the patch vertices are shown in Fig. 1.

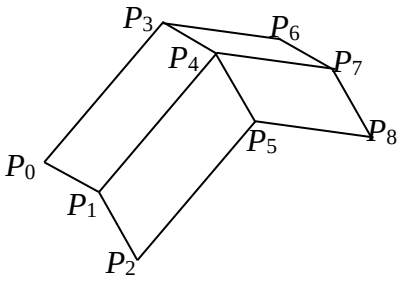



Fig. 1

Please note the following:

- Patches (primitives of type GL_PATCHES) contain only point information. Therefore, only one VBO is needed to specify the data for a patch.
- The number of patch vertices must be specified using `glPatchParameteri()` function. The program includes this statement (line 125).
- When the tessellation shader stage is active, the `glDrawArrays()` function must specify the primitive type as GL_PATCHES (see display function).
- A patch is not a renderable primitive.

The vertex shader `QuadPatches.vert` has its simplest structure in the form of a pass-through shader. It directly outputs the patch vertices without any transformation, to the next shader, the tessellation control shader.

The control shader `QuadPatches.cont` sets the outer tessellation levels to 6 and inner levels to 4. It also transfers the patch vertices received from the vertex shader directly to the evaluation shader.

The evaluation shader `QuadPatches.eval` processes the vertices of triangles generated by the primitive generator. The vertices are defined using tessellation coordinates (`gl_TessCoord.x, gl_TessCoord.y`). The evaluation shader repositions these vertices in three dimensional space by combining the patch vertices using blending functions. In the current form, this equation uses only the four corner vertices of the patch $P_0$, $P_2$, $P_6$, $P_8$ and bi-linear blending functions (see Lec[10] slide 16), and produces a tessellated quad that passes through the four points (Fig. 2a). Please note that the evaluation shader outputs the repositioned vertices ("posn") in clip coordinates.

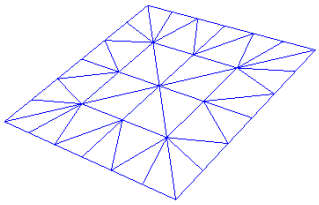


Fig. 2a

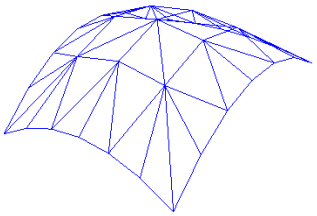


Fig. 2b

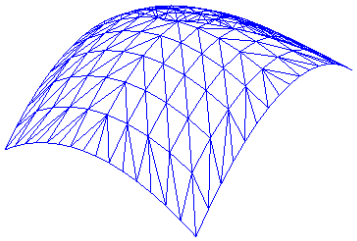


Fig. 2c

Replace the equation in the evaluation shader with the equation in Fig. 3 that uses bi-quadratic blending functions and all 9 patch vertices (see Lec [10] slide 19).

$$\begin{aligned} \boldsymbol{Q} = \quad & (1-v)^2 \left\{ (1-u)^2 P_0 + 2u(1-u) P_1 + u^2 P_2 \right\} \\ & + 2v(1-v) \left\{ (1-u)^2 P_3 + 2u(1-u) P_4 + u^2 P_5 \right\} \\ & + v^2 \left\{ (1-u)^2 P_6 + 2u(1-u) P_7 + u^2 P_8 \right\} \end{aligned}$$

Fig. 3

The program should produce an output shown in Fig. 2b. The surface model can be rotated about the y-axis using the left and right arrow keys. Increase the outer tessellation level to 15 and the inner level to 10 to get an output shown in Fig. 2c. The models generated are bi-quadratic Bezier surfaces. The patch vertices act as control points for the surface. Experiment with variations in the positions of patch vertices and changes in tessellation levels.

## II TeapotPatches.cpp:

The previous exercise used bi-quadratic blending functions and 9 patch vertices to construct a Bezier surface patch. A bi-cubic Bezier surface is similarly constructed using a 4x4 control patch containing 16 patch vertices (Fig. 4). The basic teapot model consists of only 32 such control patches, with a total of 32x16 = 512 patch vertices. The program `TeapotPatches.cpp` reads the data file "PatchVerts_Teapot.txt" that contains the coordinates of all patch vertices.

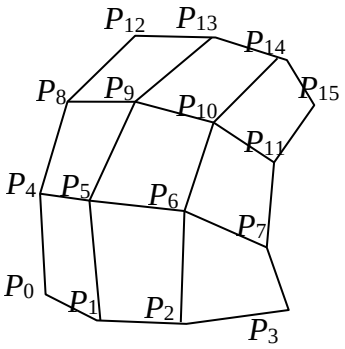



Fig. 4

The vertex shader `TeapotPatch.vert` is a simple pass-thru shader.

Note that the program does not use a tessellation control shader. A control shader is needed only if either patch vertices or tessellation levels are required to be computed separately for each patch. All patches of the teapot are assigned a constant tessellation level in the application itself (TeapotPatches.cpp). The tessellation levels are specified inside the display() function using the `glPatchParameterfv()` function.

The evaluation shader `TeapotPatch.eval` receives 16 patch vertices in the built-in array `gl_in[]`. These vertices can be combined with the tessellation coordinates *u*, *v* using bi-cubic polynomials (Fig. 5) to obtain vertices of a Bezier patch. The structure of `TeapotPatches.eval` is very similar to that of `QuadPatches.eval`.

$$
\begin{aligned}
\boldsymbol{Q} = \quad & (1-v)^3 \left\{(1-u)^3 P_0 + 3u(1-u)^2 P_1 + 3u^2(1-u) P_2 + u^3 P_3\right\} \\
& + 3v(1-v)^2 \left\{(1-u)^3 P_4 + 3u(1-u)^2 P_5 + 3u^2(1-u) P_6 + u^3 P_7\right\} \\
& + 3v^2(1-v) \left\{(1-u)^3 P_8 + 3u(1-u)^2 P_9 + 3u^2(1-u) P_{10} + u^3 P_{11}\right\} \\
& + v^3 \left\{(1-u)^3 P_{12} + 3u(1-u)^2 P_{13} + 3u^2(1-u) P_{14} + u^3 P_{15}\right\}
\end{aligned}
$$

Fig. 5

Please use the above equation to compute the position of the current vertex (`posn`) in the evaluation shader. The output of the program is shown in Fig. 6.

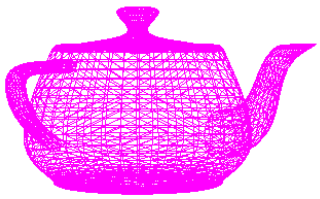
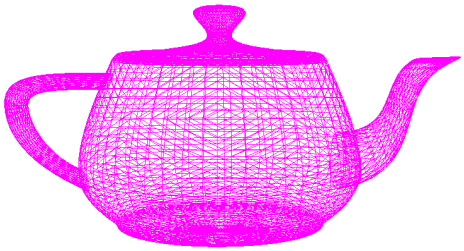


Fig. 6.

The program allows the movement of the camera towards and away from the teapot using up/down arrow keys. Inside the callback function "special()", the camera distance is clamped between 10 units and 50 units.

Take-home exercise: Implement a simple level of detail (LOD) method in the display() function to decrease the tessellation levels of the teapot (variable `level`) as the distance from the camera (`camDist`) is increased. Set the minimum value of the tessellation level at 3 (at max camera distance 50), and the maximum tessellation level at 13 (at minimum camera distance 10). The corresponding outputs are shown below (Fig. 7).



CamDist = 10 CamDist = 50

Fig. 7.

Ref:

[10] COSC363 Lecture slides: Lec10_Tessellation.pdf

## III. Quizzes

**Quiz-09 closes this week: 5pm, 26-May-2023**

Quiz-10 closes on 2-June-2023 (This is the last quiz!)