

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. This lab also introduces you to the fascinating field of terrain rendering through an example that generates a terrain model using tessellation shaders.

Note: Tessellation stages are available only in OpenGL 4.0 and later versions. Your program may generate run-time errors on systems with older versions of OpenGL.

I. CubePatches.cpp:

In this exercise, we will use tessellation shaders to subdivide the faces of a cube and convert them to Bezier patches to generate a triangle mesh as shown below (Fig. 1).

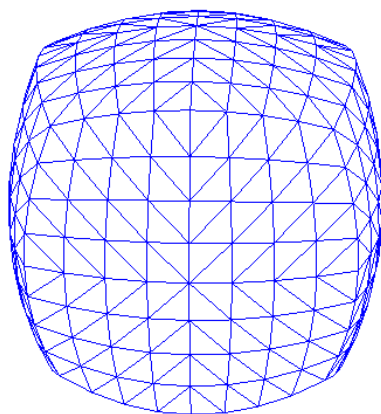
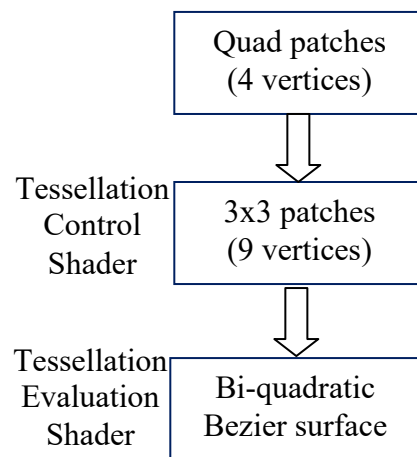


Fig. 1



- (1) The file `CubePatches.h` contains the coordinates of 8 vertices of a cube, and the definitions of 6 faces using vertex indices (a total of 24 elements). The program `CubePatches.cpp` generates the display of the wireframe model of the cube (Fig. 2). The cube is continuously rotated about the y -axis using a timer callback. The vertex shader transforms vertices to clip coordinates, and the fragment shader outputs a single colour value.

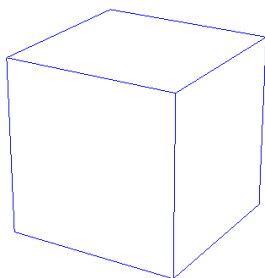


Fig. 2

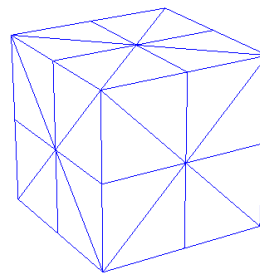


Fig. 3

- (2) A pass-thru tessellation control shader (CubePatches.cont) that sets all tessellation levels to 2 is provided. Also provided is an evaluation shader (CubePatches.eval). Please add the tessellation shaders to the program object by uncommenting the two lines in the "Load Shaders" section of the initialize() function. Also uncomment the two lines in the "Attach and link shaders" section.

If a tessellation shader stage is active, the input primitive must be of type GL_PATCHES. In the display() function, modify the glDrawElements() function argument to output patches instead of quads:

```
glDrawElements(GL_PATCHES, ...)
```

Also, specify the number of vertices in each patch, inside the initialize() function:

```
glPatchParameteri(GL_PATCH_VERTICES, 4);
```

A patch is not a renderable primitive. Therefore, we should not convert the patch coordinates to clip space. Edit the vertex shader and remove the multiplication of coordinates by the model view projection matrix. The shader is now a simple pass-through shader.

The evaluation shader (CubePatches.eval) receives the (u, v) coordinates of the tessellated mesh in the built-in variable gl_TessCoord. It also receives the 4 patch vertices in the array gl_in[i].gl_Position, $i=0..3$. Use these vertices and the (u, v) coordinates to generate a bi-linear mapping shown on Lec11-Slide 4. (Complete the expression for the variable "posn"). Note that the output is a vertex of a triangle on the patch, which is multiplied by mvpMatrix to transform it to the clip coordinate space.

The output should now look similar to that given in Fig. 3. We use this output only to verify that the tessellation shaders are active and are receiving proper inputs (and generating proper outputs). Increase the tessellation levels specified in the control shader (CubePatches.cont) and observe the corresponding changes in the output.

- (3) The tessellation control shader also allows us to modify the structure of the input patch. For example, if the input patch has just 4 vertices forming a quad, we can add additional points inside the tessellation control shader so that the patch becomes a set of 3x3 vertices or 4x4 vertices suitable for generating Bezier approximations inside the evaluation shader.

We will now modify the structure of the input patch from a quad to a 3x3 grid of points by creating additional patch vertices inside the control shader. The control shader receives all 4 vertices of each patch in the array gl_in[].gl_Position. The vertices will have the same order as generated by the application. Inside the control shader, specify the number of output patch vertices as 9:

```
layout(vertices = 9) out;
```

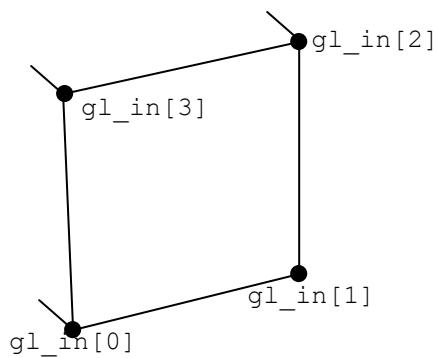


Fig. 4(a)

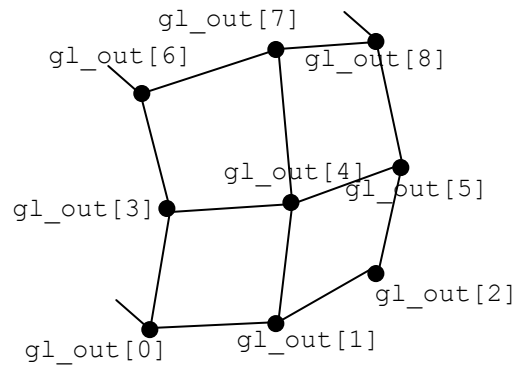


Fig. 4(b)

Fig. 4(a) shows the configuration of input vertices for one of the patches. With the number of output vertices specified as 9, the control shader will execute 9 times, with `gl_InvocationID` varying from 0 to 8, producing an output vertex position in `gl_out[gl_InvocationID].gl_Position` on each execution of the shader. These output positions are defined as follows.

The corner vertices are simply copied to the corresponding location in the `gl_out` array: For example, the corner points on the top edge of Fig. 4(b) are specified as

```
if(gl_InvocationID == 6)
    gl_out[gl_InvocationID].gl_Position = gl_in[3].gl_Position;

if(gl_InvocationID == 8)
    gl_out[gl_InvocationID].gl_Position = gl_in[2].gl_Position;
```

Add the statements for the remaining two corner vertices.

The midpoints on the edges are obtained by taking the average of two corner vertices and moving that point out of the plane by scaling its coordinates by a factor 1.4. Since averaging requires a division by 2, we minimize the arithmetic operations by adding the corner vertices and scaling by 0.7. Example:

```
if(gl_InvocationID == 5)
    gl_out[gl_InvocationID].gl_Position =
    vec4((gl_in[1].gl_Position.xyz+gl_in[2].gl_Position.xyz)*0.7
        , 1) ;
```

[Why wouldn't the following expression work?

```
gl_out[gl_InvocationID].gl_Position =
    (gl_in[1].gl_Position + gl_in[2].gl_Position)*0.7
]
```

Add similar expressions for the remaining edge points.

Finally, the midpoint of the patch (`gl_out[4].gl_Position`) is got by averaging the four corner points and scaling it by a factor of 2 (i.e., scaling the sum by 0.5).

```
if(gl_InvocationID == 4)
gl_out[gl_InvocationID].gl_Position =
    vec4( (gl_in[3].gl_Position.xyz
          + gl_in[1].gl_Position.xyz
          + gl_in[2].gl_Position.xyz
          +gl_in[0].gl_Position.xyz )*0.5,  1) ;
```

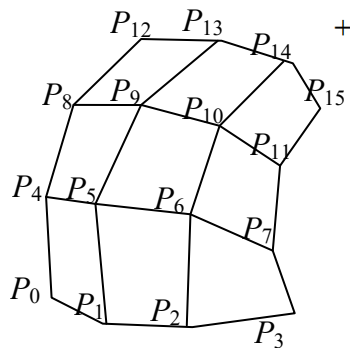
Also in the control shader, change all tessellation levels to 8.

- (4) The tessellation evaluation shader now receives 9 vertices per patch. We can combine these 9 vertices using the tessellation coordinates u, v to form a bi-quadratic Bezier patch as shown in Lec11-Slide 5. Update the mapping equation in the evaluation shader using this formula. If correctly implemented, each patch (side) of the cube will get replaced with a Bezier surface as shown earlier in Fig. 1.

II TeapotPatches.cpp:

Bi-cubic Bezier surfaces are constructed using 4×4 control patches containing 16 vertices (see Lec10-Slide 22). The teapot model consists of 32 such control patches, with a total of $32 \times 16 = 512$ patch vertices. The program `TeapotPatches.cpp` reads the data file "PatchVerts_Teapot.txt" that contains the coordinates of all patch vertices.

The vertex shader `TeapotPatch.vert` is a simple pass-thru shader. 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 the Bezier patch (see also Lec10-Slide23, Lec11-Slide26).



$$\begin{aligned} \mathbf{Q} = & (1-u)^3 \left\{ (1-v)^3 \mathbf{P}_0 + 3(1-v)^2 v \mathbf{P}_1 + 3(1-v)v^2 \mathbf{P}_2 + v^3 \mathbf{P}_3 \right\} \\ & + 3(1-u)^2 u \left\{ (1-v)^3 \mathbf{P}_4 + 3(1-v)^2 v \mathbf{P}_5 + 3(1-v)v^2 \mathbf{P}_6 + v^3 \mathbf{P}_7 \right\} \\ & + 3(1-u)u^2 \left\{ (1-v)^3 \mathbf{P}_8 + 3(1-v)^2 v \mathbf{P}_9 + 3(1-v)v^2 \mathbf{P}_{10} + v^3 \mathbf{P}_{11} \right\} \\ & + u^3 \left\{ (1-v)^3 \mathbf{P}_{12} + 3(1-v)^2 v \mathbf{P}_{13} + 3(1-v)v^2 \mathbf{P}_{14} + v^3 \mathbf{P}_{15} \right\} \end{aligned}$$

Fig. 5

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

or modified while rendering an object. All patches of the teapot are assigned a constant tessellation level. The tessellation levels are specified inside the `display()` function using the `glPatchParameterfv()` function.

Please implement the above bi-cubic mapping in the evaluation shader to compute the position of the mapped vertex on the Bezier surface. The evaluation shader must also transform the coordinates to the clip coordinate space. The output of the program is shown in Fig. 6.

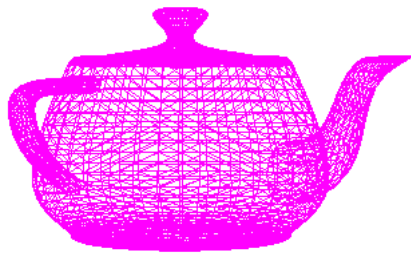


Fig. 6.

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 minimum camera distance is clamped at 10 units and the maximum distance at 50 units.

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).

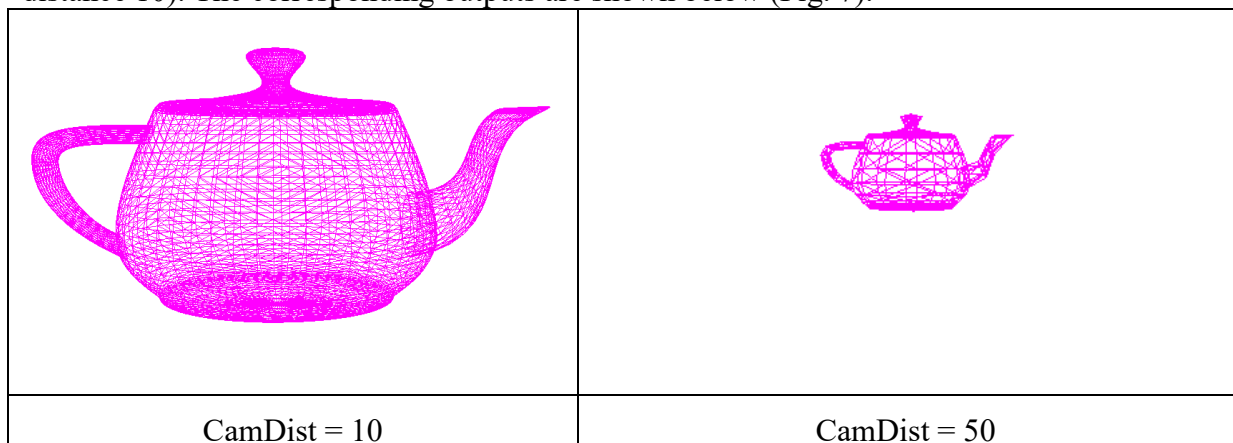


Fig. 7.

III. TerrainPatches.cpp:

- (1) The program `TerrainPatches.cpp` draws a set of quads arranged in a rectangular 10x10 grid containing 100 vertices (Fig. 8). The quads represent the terrain's initial ground plane.

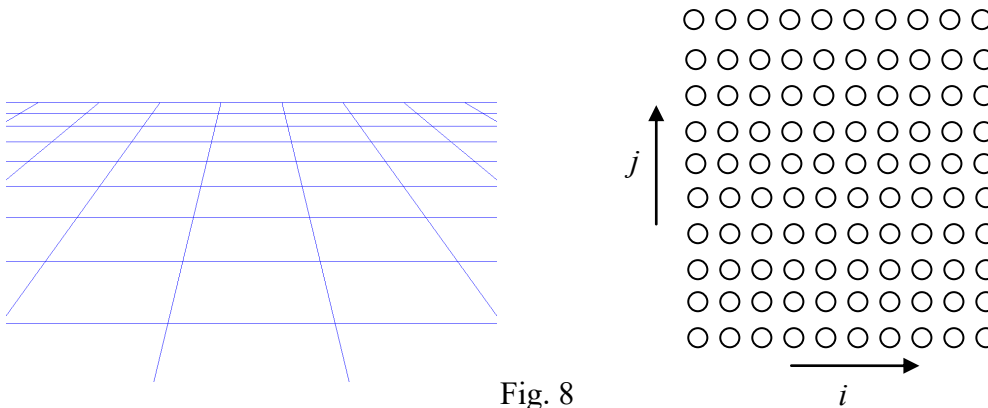


Fig. 8

(2) Similar to Exercise I (CubePatches.cpp),

- In `display()`, replace the primitive type in `glDrawElements()` with `GL_PATCHES`
 - In `initialise()`, load and attach control and evaluation shaders, and also specify the number of patch vertices as 4.
 - Convert the vertex shader to a pass-thru shader by removing the multiplication by `mvpMatrix`.
 - In the control shader, set all tessellation levels to 6.
 - In the evaluation shader, define a bi-linear mapping of mesh vertices (u, v).
- The program produces a tessellated floor as shown below (Fig. 9).

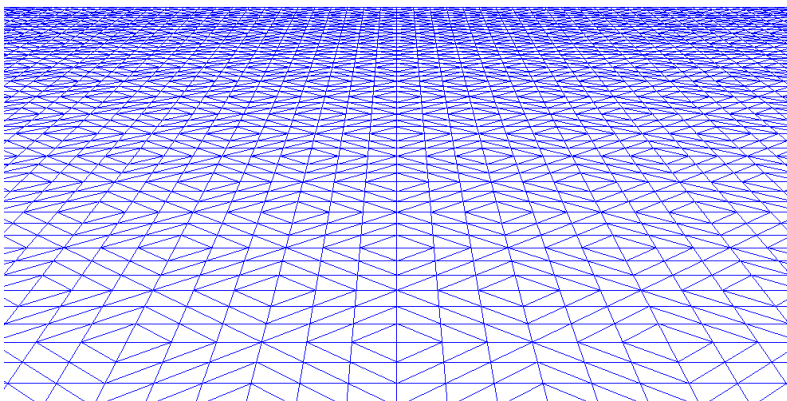
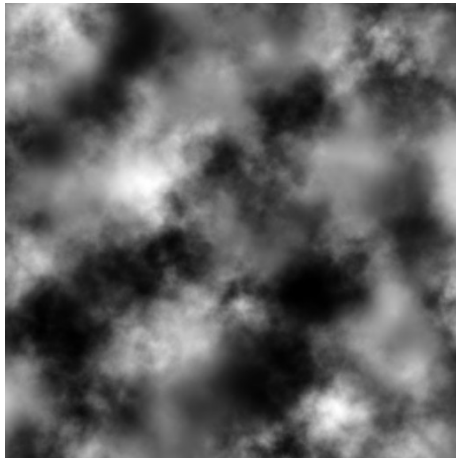


Fig. 9.

- (3). The program includes a function to load a texture image "Terrain_hm_01.tga". Note also that the vertex array object includes a buffer object containing texture

coordinates at the vertices of the 10x10 grid. The texture represents a gray-level height- map of a terrain (Fig. 10)



(Fig. 10) Terrain height map

- (4) The vertex shader outputs the texture coordinates of each patch vertex using the out variable "vert_texCoord". The control shader can be made to receive the texture coordinates of *all* 4 patch vertices in an array, by using the declaration

```
in vec2 vert_texCoord[];
```

The control shader should pass these values on to the evaluation shader using an 'out' variable array:

```
out vec2 cont_texCoord[];  
cont_texCoord[gl_InvocationID] =  
    vert_texCoord[gl_InvocationID];
```

Please add the above statements in the control shader.

- (5) Modify the evaluation shader to receive the above texture coordinates, by adding the statement.

```
in vec2 cont_texCoord[];
```

Also declare a Sampler2D object to access the texture:

```
uniform sampler2D heightMap;
```

The array "cont_texCoord[]" holds the values of texture coordinates at patch vertices. We can obtain the texture coordinates at a tessellated mesh vertex, using bi-linear interpolation, in exactly the same way we obtained the positions of the mesh vertices (see Lec11-Slide 33). Include the statement to compute the interpolated texture coordinates:

```
tcoord = (1-u) * (1-v) * cont_texCoord[0]  
        + u * (1-v) * cont_texCoord[1]  
        + u * v * cont_texCoord[2]  
        + (1-u) * v * cont_texCoord[3];
```

Use the above texture coordinate with the `Sampler2D` object to get a colour value from the height map. Since the height map is a gray-level image, any of its colour components will give the height of the terrain in the range `[0, 1]`:

```
float height = texture(heightMap, tcoord).r;
```

Scale this value by 10, and assign it to the *y*-coordinate of the mesh vertex:

```
posn.y = height * 10.0;
```

As usual, the final position is multiplied by the model-view-projection matrix, and output by the evaluation shader.

The program will output a terrain model as shown in Fig. 11.

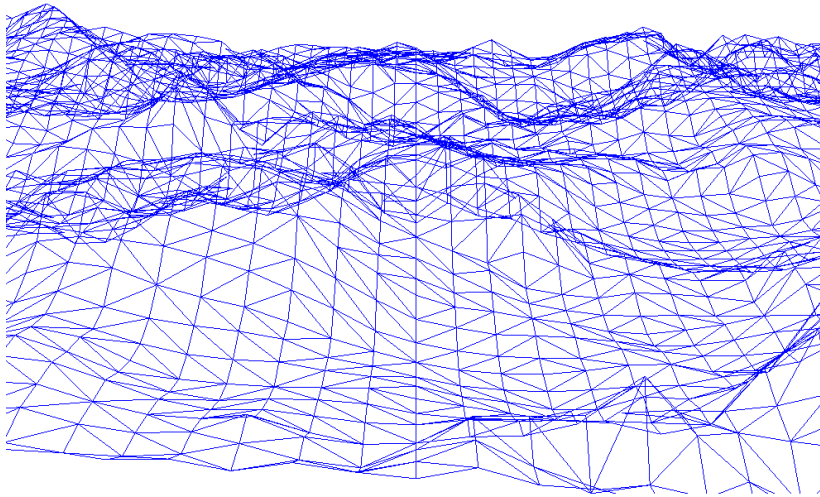


Fig. 11.

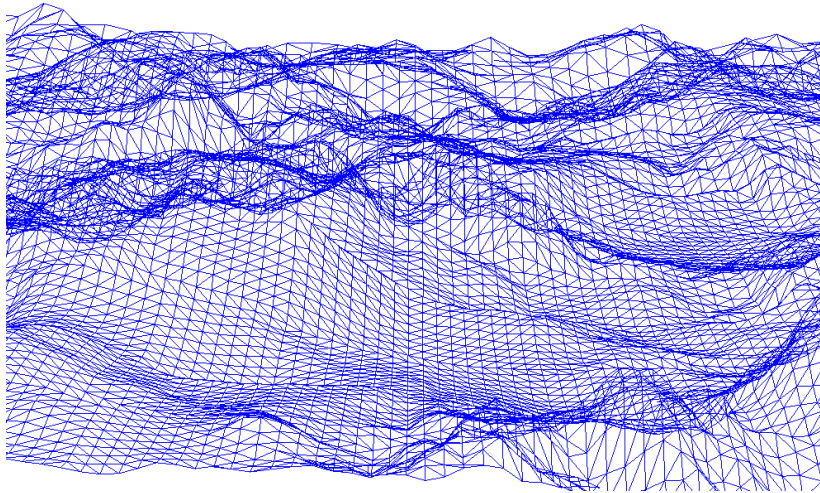
- (6) We can adjust the level of detail on the terrain based on the distance of the terrain segment from the camera. This is done by modifying the tessellation levels inside the tessellation control shader, based on the average *z* distance of the patch from the origin:

```
float dist = ( gl_in[0].gl_Position.z  
              + gl_in[1].gl_Position.z  
              + gl_in[2].gl_Position.z  
              + gl_in[3].gl_Position.z ) * 0.25;
```

The terrain's *z* values range from 0 to -100. We map these values to tessellation levels 20 to 2 (highest level of detail at *z*=0) using the formula

```
int level = int((dist+100.0)*0.18)+2;
```

Set the outer and inner tessellation levels to the value computed above. The output should now show a reduction in the tessellation levels with the distance of the patch from the origin. (Fig. 12).



(Fig. 12)

- (7) Further work: The above equation uses terrain's z - values in world coordinates. They must be replaced with z values in eye-coordinates so that they represent the distance from the camera. Also implement a keyboard interface to move the camera through the terrain so that you can see dynamically changing tessellation levels!

In order to render a texture mapped model of the terrain, you will require a geometry shader where you will have access to all three vertices of each triangle to perform lighting calculations and to assign texture coordinates to vertices. Please refer to Lec12 slides for more information. When you include a geometry shader, please remember to remove the operation of conversion of points to clip coordinates from the tessellation evaluation shader and to perform this operation in the geometry shader.

IV. Quiz-10

This is the last quiz for this course!

The quiz will remain open until **5pm, 29-May-2020**.