Graphics Programming HW 3

Due: 4/18 11:59 PM

Assignment Overview (Prob. 1, 2)

In this assignment you will implement several splines and Bézier surface that have been covered in the class. One thing to note is that we will use recent OpenGL techniques, geometry shader and tessellation shader. Compared to traditional approaches, these new techniques have several advantages. You can perform adaptive subdivision and adjust visual quality to the required level of detail (LOD). You can also display a fine-resolution model even the provided input is in a coarse resolution (geometric compression). Hope you can learn geometry shader and tessellation shader doing this homework.

Prob 1. Splines with Geometry Shader (65 pt)

A spline interpolation refers to a piece-wise smooth curve that consists of low-degree polynomials. In computer graphics, cubic polynomial is usually used, because it is the lowest-degree polynomials that can represent 3D space curve with C^1 continuity. Given a set of control points, our goal is to smoothly interpolate or approximate them. Mathematically, cubic function can be represented with a set of monomial basis (t^3, t^2, t^1, t^0),

$$x(t) = a_3t^3 + a_2t^2 + a_1t + a_3.$$

A Bézier curve is a parametric curve that uses Bernstein polynomial as basis. For the cubic case, four control points are used. The curve is defined for $t \in [0, 1]$ as follows,

$$\begin{aligned} p(t) &= b_0B_0^3(t) + b_1B_1^3(t) + b_2B_2^3(t) + b_3B_3^3(t) \\ &= (1-t)^3b_0 + 3t(1-t)^2b_1 + 3t^2(1-t)b_1 + t^3b_3 \end{aligned}$$

where b_0, b_1, b_2, b_3 are control points. The above equation can be represented as a matrix form,

$$p(t) = T \cdot M_B \cdot G \tag{1}$$

$$= \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}. \tag{2}$$

A Catmull-Rom spline is a polynomial interpolation without tangent condition. It approximates tangent condition parallel to line of adjacent two points. B-Spline achieves both C^2-continuity and local controllability, but it no longer interpolates control points; it only approximates them. Because details were covered

in the class, we will skip the rest. Anyway, aforementioned three splines can be represented as a succinct matrix form, $p(t) = T \cdot M \cdot G$, similar to Equation 2.

In this problem, you will implement these three splines using geometry shader. You have to do followings.

- Implement simple curves for three splines (Fig. 1 (a)). In this case, there are only 4 control points. The data can be found in `spline_simple.txt`.
- Bézier curve is widely used for fonts, such as TrueType or PostScript. Here you will render alphabet 'u' using Bézier curve (Fig. 1 (b)). Control points are in `spline_u.txt`.
- Finally, render a smooth loop with more than 4 control points (Fig. 1 (c)). Bézier curve cannot be used for this case, and implement it for the other two cases. Control points are in `spline_complex.txt`.
- Add another simple shader to render the outer line of the control points. It is drawn as white line in Fig. 1. Make it possible to toggle whether to render the outer line by pressing the key 9.

Here are a few more things that you have to notice.

- Consider only cubic cases, not quadratic cases.
- You have to use a single shader to render splines. Do not make shader for each spline. You can render different types of splines with only a single shader.
- Of course it is impossible to render completely continuous spline, so divide spline into small line segments. Set line segment number to a constant number, e.g. 20.
- (Hint) You have to use different drawing modes for individual splines. For Catmull-Rom spline and B-spline, we have to reuse the three closest control points from the previous spline. So use `GL_LINE_STRIP_ADJACENCY`. On the other hand, for Bezier spline, we do not have to reuse control points. So just use `GL_LINE_ADJACENCY`.
- There is no restriction on splines' position, but please arrange them neatly.

For each data file, N control points are written in the following format. For loop spline, you do not have to do anything to make it loop. First three points are already added at the last.

$$
\begin{aligned}
&N \\
&x_1\ y_1\ z_1 \\
&x_2\ y_2\ z_2 \\
&\cdots \\
&x_N\ y_N\ z_N
\end{aligned}
$$

Prob 2. Bézier Surfaces with Tessellation Shader [1](20 pt)

Bézier spline can be easily expanded to the surface.

$$p(u,v) = \sum_{j=0}^{n} \sum_{k=0}^{m} b_{jk} B_j^n(u) B_j^m(v)$$

[1]We will not cover Tessellation shader deeply in lab 3. Please refer to this site for details.

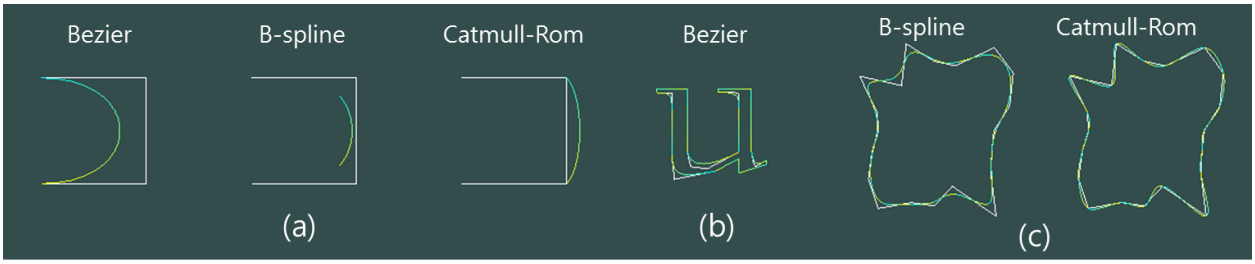


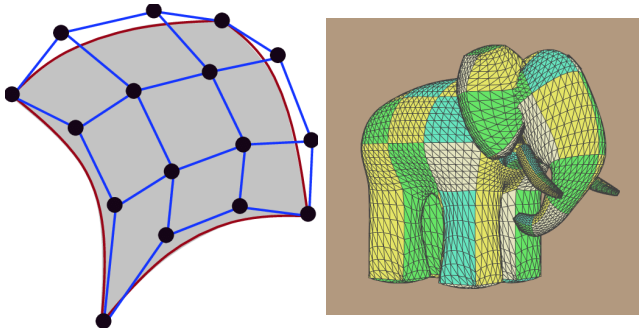
Figure 1: Example of rendered scene for problem 1.

The most common use of Bézier surfaces is as nets of bicubic patches (where n = m = 3). The mesh is divided into several bicubic patches which are completely defined by 16 control points (Fig. 2). Again, we can represent this in a similar matrix form (more precisely, it is the tensor product),

$$
\begin{aligned}
p(u,v) &= \sum_{j=0}^{n}\sum_{k=0}^{m} b_{jk} B_j^n(u) B_j^m(v) \\
&= U^T \cdot M \cdot G \cdot M^T \cdot V \\
&= \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix}
\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}
\begin{bmatrix} b_{00} & b_{10} & b_{20} & b_{30} \\ b_{01} & b_{11} & b_{21} & b_{31} \\ b_{02} & b_{12} & b_{22} & b_{32} \\ b_{03} & b_{13} & b_{23} & b_{33} \end{bmatrix}
\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}
\begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix}.
\end{aligned}
$$

In this problem, You have to render several models that consist of Bézier surface patches using Tessellation shader, which is supported from OpenGL 4.0. The example is on Fig. 3. You also have to implement Level of Detail (LOD). When the camera is far from an object, only a coarse model is enough, while a finer model is required as the camera moves closer. You can achieve this by changing `gl_TessLevelOuter` and `gl_TessLevelInner` of Tessellation Control shader, depending on the distance from the camera. To determine these values, you also have to consider object's scale. Bigger object needs more detailed patch. There is no given rule to determine `gl_TessLevelOuter` and `gl_TessLevelInner`. Do it by your own way. Just satisfy that mesh should be coarser as the camera moves far from the object. Render objects in `GL_LINE` mode, so that we can check patches more explicitly. Allow us to use key 0 to toggle between `GL_LINE` and `GL_FILL`.

Object's color and position are not constrained. Do it in your own way. Just make each patch distinguishable. Please render more than three different models. Data files are in `bezier_surface_data` folder. For each data file, N patches with 16 control points are written in the following format:



(a) (b)

Figure 2: (a) Bicubic Bézier surface (b) Ed Catmull's "Gumbo" model, composed from patches

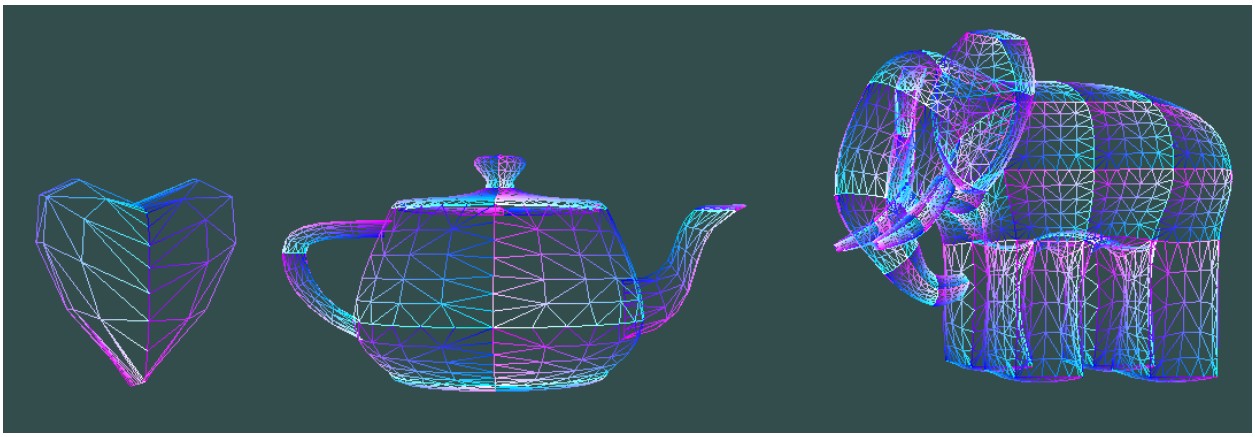


Figure 3: Example of rendered scene for problem 2.

N

3 3

$x_{1,0,0}$ $y_{1,0,0}$ $z_{1,0,0}$

$x_{1,0,1}$ $y_{1,0,1}$ $z_{1,0,1}$

...

$x_{1,3,3}$ $y_{1,3,3}$ $z_{1,3,3}$

...

3 3

$x_{N,0,0}$ $y_{N,0,0}$ $z_{N,0,0}$

$x_{N,0,1}$ $y_{N,0,1}$ $z_{N,0,1}$

...

$x_{N,3,3}$ $y_{N,3,3}$ $z_{N,3,3}$

Prob 3. Catmull-Clark Subdivision (65 pt)

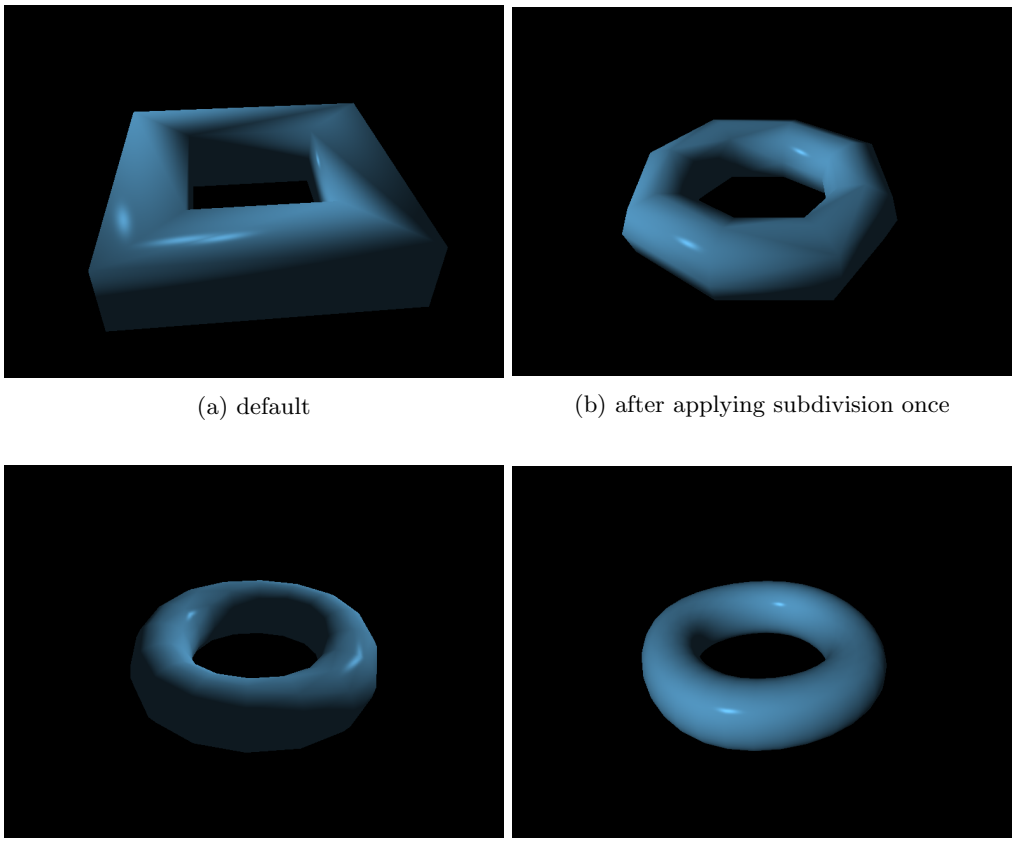
You will implement the Catmull-Clark Subdivision algorithm in this problem. Specifically, you should write the code of 6 functions which are **is_holeEdge, is_holeVertex, face_point, edge_point, vertex_point,** and **catmull_clark**. The details of the functions are following.

- **bool is_holeEdge(edge* e)**: if **e** belongs to only one face then return true, else return false.
- **bool is_holeVertex(vertex* v)**: if the number of faces which **v** belongs to is not equal to the number of edges which **v** belongs to then return true, else return false.
- **vertex* face_point(face* f)**: **face** structure has a **face_pt** element. if **f** already has an precalculated **face_pt**, then return it. Otherwise, create a new vertex which is the average of all points of the **f**, save it in the **face_pt** element, and return it.
- **vertex* edge_point(edge* e)**: **edge** structure has a **edge_pt** element. if **e** already has an precalculated **edge_pt**, then return it. Otherwise, there are following two cases:
  - **If e is a holeEdge**, then create a new vertex which is the mid point of the **e**, save it in the **edge_pt**, and return it.
  - **If e is not a holeEdge**, then create a new vertex which is the average of the face points of the faces which **e** belongs to and the two end points of the **e**, save it in the **edge_pt**, and return it.
- **vertex* vertex_point(vertex* v)**: **vertex** structure has a **v_new** element. if **v** already has an precalculated **v_new**, then return it. Otherwise, there are following two case:.
  - **If v is a holeVertex**, then create a new vertex which is the average of the mid points of holeEdges which **v** belongs to and the **v**. Save it in the **v_new**, and return it.
  - **If v is not a holeVertex**, then create a new vertex of which coordinates are following:

$$new\_coord = \frac{avg\_face\_points}{n} + \frac{2 * avg\_mid\_points}{n} + \frac{(n-3) * v}{n} \tag{3}$$

    n is the number of the faces which **v** belongs to. *avg_face_points* is the average point of the face points of faces which **v** belongs to. *avg_mid_points* is the average point of the mid points of edges which **v** belongs to. If the calculation completed, save the vertex in the **v_new**, and return it.
- **object* catmull_clark(object* obj)**: create a new object which has new vertices using Catmull-Clark subdivision algorithm and return it. In this problem, you are supposed to deal with two cases: triangle faces and quadrilateral faces.
  - **For triangle faces**: For each tri face in the **obj**, add 3 new faces to the new object with the new vertices. If a tri face in the **obj** consists of three vertices named a,b, and c then the newly created faces are comprised of following vertices:

    (vertex_point(a), edge_point(ab), face_point(abc), edge_point(ca))
    (vertex_point(b), edge_point(bc), face_point(abc), edge_point(ab))
    (vertex_point(c), edge_point(ca), face_point(abc), edge_point(bc)).
  - **For quadrilateral faces**: For each quad face in the **obj**, add 4 new faces to the new object with the new vertices. If a quad face in the **obj** consists of four vertices named a,b,c, and d then the newly created faces are comprised of following vertices:



(a) default (b) after applying subdivision once

(c) after applying subdivision twice (d) after applying subdivision 3 times

Figure 4: Results of the Catmull-Clark subdivision algorithm.

(vertex_point(a), edge_point(ab), face_point(abcd), edge_point(da))
(vertex_point(b), edge_point(bc), face_point(abcd), edge_point(ab))
(vertex_point(c), edge_point(cd), face_point(abcd), edge_point(bc))
(vertex_point(d), edge_point(da), face_point(abcd), edge_point(cd)).

You can easily see that newly created object by this algorithm will be comprised of quadrilateral faces. We provided the skeleton codes written in C++. If you implemented above 6 functions appropriately based on the skeleton code, you can apply the Catmull-Clark subdivision algorithm to the given mesh by pressing the 'd' key. Also change the model by pressing key 1(cube), 2(donut) and 3(star). The code for each model is already given. Figure 4 shows the results of the Catmull-Clark subdivision algorithm for the donut model.