Assignment 2:
Assignment 2 : Geometric Modeling

NAME: LIU FANGXUN
STUDENT NUMBER: 2020533047
EMAIL: LIUFX@SHANGHAITECH.EDU.CN

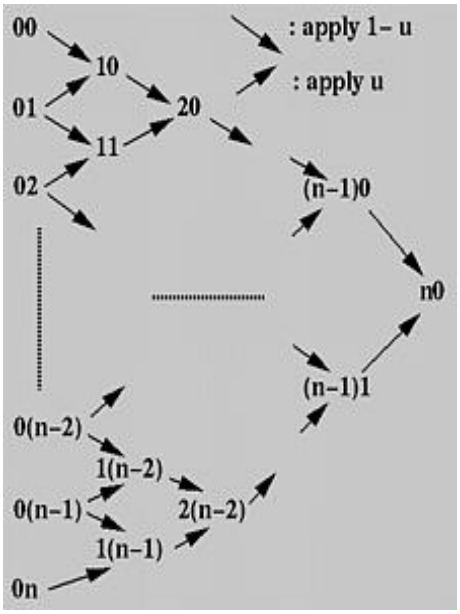

Fig. 1. figure 1: de Casteljau's algorithm

1 INTRODUCTION

- Task 1 (Implementation of the basic iterative de Casteljau Bézier vertex evaluation algorithm) has been done.
- Task 2 (Construction of Bézier Surface with normal evaluation at each mesh vertex) has been done.
- Task 3 (Rendering a Bézier Surface in a OpenGL window based on vertex array and vertex normal) has been done.
- Task 4 (Stitching multiple Bézier surface patches together to create a complex meshes) has been done.

2 IMPLEMENTATION DETAILS

2.1 Implementation of the basic iterative de Casteljau Bézier vertex evaluation algorithm

For this part, I mainly implement the **Vertex BezierCurve::evaluate (std::vector<vec3>& control_points, float t)** function in **bezier.cpp**. The basic idea of de Casteljau's algorithm is shown in figure 1.

The **00, 01, ..., 0n** points are the original $n+1$ control points, then let **1i = (1-u)*0i + u*0(i+1)**, where $u \in [0,1]$, for $i = 0, 1, ..., n-1$, we can get n points **10, 11, ..., 1(n-1)**, which forms a new polyline of $n-1$ line segments. Apply this procedure again to this new polyline, we can get a second polyline of $n-1$ points **20, 21, ..., 2(n-2)**, which again forms a new polyline of $n-2$ line segments. Continuingly apply this procedure to the new polyline until we only get 1 point **n0**, and this is the point $C(u)$ on the curve that corresponds to u. Thus we have evaluated the point on Bézier Curve given a certain u.

The following algorithm shows how we implement it.

The first **for** loop is to save the control points to our working array

Algorithm 1: de Casteljau's algorithm

Input: array P[0:n] of n+1 points and real number u in [0,1]
Output: point on curve, C(u)

```
Working_array Q[0:n];
for i := 0 to n do
    Q[i] := P[i];
end
for k := 1 to n do
    for i := 0 to n - k do
        Q[i] := (1 - u)Q[i] + u Q[i + 1];
    end
end
return Q[0];
```

for further computation. Then the nested **for** loop does the procedure we discussed above, and the final point is saved in $Q[0]$.

Another requirement in this task is to calculate the tangent of the final point, and this can be obtain by **(n-1)0, (n-1)1**. We can get this two points in the above procedure, and calculate the tangent by subtraction between them and then normalize. The tangent is temporarily saved in **Vertex.normal**.

2.2 Construction of Bézier Surface with normal evaluation at each mesh vertex

For this part, I mainly implement the **Vertex BezierSurface::evaluate (std::vector<std::vector<vec3»&control_points, float u, float v)** function in **bezier.cpp**. Now we have finished the Bézier Curve evaluation, doing Bézier Surface evaluation is just to apply Bézier Curve evaluation twice, one based on another. We can first construct m intermediate Bézier curves based on the corresponding n control points and evaluate m points $q_0(v), q_1(v), ..., q_{m-1}(v)$ at parameter v, and then construct another Bézier curve based on these m points and evaluate the point $p(u,v)$ at u. The following algorithm shows how we implement it.

Another requirement in this task is to calculate the normal of the

Algorithm 2: Bézier Surface evaluation

Input: a m+1 rows and n+1 columns of control points and (u,v)
Output: point on surface p(u,v)

```
for i := 0 to m do
    Apply de Casteljau's algorithm to the i-th row of control points with v;
    Let the point obtained be qi(v);
end
Apply de Casteljau's algorithm to q0(v), q1(v), ..., qm(v) with u;
The point obtained is p(u,v);
return p(u,v);
```

final point, which can be calculated by cross-product of two tangents along two dimensions on the final point. As the above procedure can only give one tangent, so we need to apply the procedure again in the other order, namely apply de Casteljau's algorithm on m control points first to get n points at parameter u, then evaluate the point $p(u, v)$ at v based on these n points. Now we get the two tangents and can calculate the normal by their cross-product and then normalize.

2.3 Rendering a Bézier Surface in a OpenGL window based on vertex array and vertex normal

For this part, first we need to implement the **void Object::init()** in **object.cpp**. I initialize the VAO, VBO and EBO and configure the two attributes of **Vertex**, namely **position** and **normal**. This is discussed in detail in assignment 1, so I just mention it briefly here.
Next I implement the draw functions. As the implemention of the four draw functions are similar and I use the shader and VBO to draw, I'll take the implemention of the **void Object::drawArrays (const Shader& shader)** function as an example. First use the mumber function **use()** of **shader** to use the shader, then bind VAO and draw triangles with VBO.
Then I implement the **Object BezierSurface::generateObject()** function in **bezier.cpp** to draw an object based on a Bézier Surface. **Object BezierCurve::generateObject()** is similar, so I'll focus on the surface case. First we need to iterate u, v from 0 to 1 respectively, and I set the iteration interval to 0.1. I call the **BezierSurface::evaluate(...)** function to calculate the surface points, and I construct a intermediate 2D array to save these points. Then I arrange the points three by three to be vertices of triangles, and save them to **object.vertices**. Last I call **object.init()** to initialize the object by the vertices and return the object.
Now that all the functions for rendering the Bézier surfaces have been implemented, now we can use them in **main.cpp**. I put drawing single bézier surface in case 0. First I assign a $4 * 4$ array of control points, and pass it into bézier surface by **void BezierSurface::setControlPoint(...)**. Then I call **Object BezierCurve::generateObject()** to form the object. Lastly, in the render loop, I first set the uniform variables of the shader, and then call the **void Object::drawArrays(...)** to render the bézier surface. Now the rendering is done.

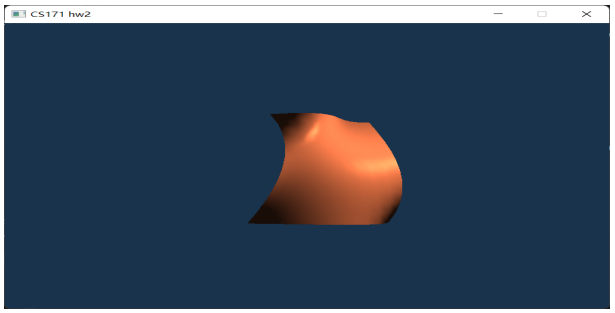

Fig. 2. result 1

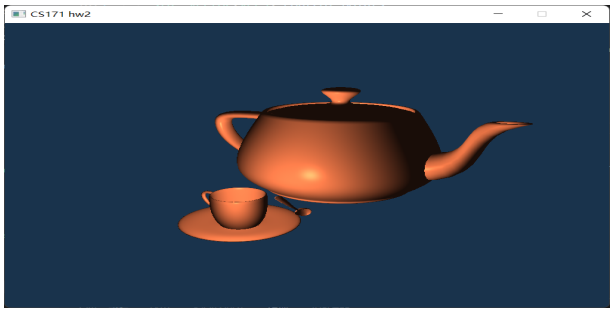

Fig. 3. result 2

2.4 Stitching multiple Bézier surface patches together to create a complex meshes

For this part, first I implement the **std::vector<BezierSurface> read(const std::string &path)** in **bezier.cpp**. I save the control points' indices and control points respectively, and then initialize the bezier surfaces by calling **void BezierSurface::setControlPoint(...)** to save the control points into the bezier surfaces and save the bezier surfaces into an array.
Next I call this function in case 1 in **main.cpp**, by reading in the **.bzs** file and call **Object BezierCurve::generateObject()** to every bezier surface in the array, I successfully render the tea party.

3 RESULTS

As in result 1 we can see we have rendered a single bézier surface, and in result 2 we can see we have rendered the tea party.