Assignment 2:

NAME: ZHANG SIYUAN
STUDENT NUMBER: 2019533240
EMAIL: ZHANGSY3@SHANGHAITECH.EDU.CN

1 INTRODUCTION

This assignment implements:

1) the basic iterative de Casteljau Bézier vertex evaluation algorithm.
2) constructing Bézier surfaces with the normal evaluation at each mesh vertex.
3) rendering the Bézier surfaces based on the vertex array.
4) creating more complex meshes by stitching multiple Bézier surface patches together.
5) constructing the B-Spline surfaces.
6) the adaptive mesh construction based on the curvature estimation.
7) calculating the curvature on each vertex.

2 IMPLEMENTATION DETAILS

2.1 de Casteljau Bézier Algorithm and Normal

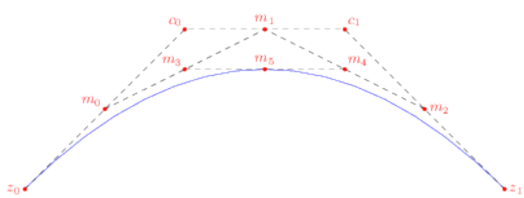

Fig. 1. example for drawing a bezier curve

A Bézier curve B (of degree n, with control points $\beta_0, \ldots, \beta_n$) can be written in Bernstein form as follows

$$B(t) = \sum_{i=0}^{n} \beta_i b_{i,n}(t)$$

where b is a Bernstein basis polynomial

$$b_{i,n}(t) = \binom{n}{i} (1-t)^{n-i} t^i$$

2.1.1 Simple Implication. So we can imply the algorithm with a double loop over the control points:

```
position de_casteljau(t, ControlPoints):
    beta = ControlPoints;
    n = beta.size;
    for j = (0..n-1):
        for k = (0..n - j - 1):
            beta[k] = beta[k] * (1 - t) +
              beta[k + 1] * t;
    return beta[0];
```

2.1.2 How about Normal. So as to get the normal of each vertex, we must get the tangent of the vertex on both the x, y direction. Seeing that on a curve, the tangent of the vertex is the direction of the last iterated two vertices. A recursive function to get the tangent and the position as follows:

```
vertex evaluate(control_points, t):
    if (control_points.size() == 2)
      x.position = (1 - t) * control_points[0] +
                 t * control_points[1];
        x.tangent = control_points[1] -
                control_points[0];
        return x;
    else:
        vector<position> next_control_points;
        first_vertex = control_points[0];
        for i = (1..control_points.size() - 1):
            second_vertex = control_points[i];
            next_control_points.push_back(
        (1 - t) * first_vertex +
                t * second_vertex);
            first_vertex = second_vertex;
        return evaluate(next_control_points, t);
```

So the normal of the vertex can be calculated as follows:

```
normal = normalize(cross(
  evaluate(control_points_on_x_direction, u).tangent,
  evaluate(control_points_on_y_direction, v).tangent));
```

2.2 K^{th} Order Derivatives and Curvature

2.2.1 K^{th} Order Derivatives. The K^{th} order derivative of a Bessel curve as follows:

$$C^{(k)}(t) = n(n-1)\ldots(n-k+1) \sum_{i=0}^{n-k} B_{i,n-k} p_i^{(k)}$$

$$p_i^{(k)} = p_{i+1}^{(k-1)} - p_i^{(k-1)}$$

So we can calculated any derivative of any order as follows:

```
vec3 at(control_points, t, derivative_order):
    beta = control_points;
    n = beta.size;
    prefix = 1;
    for i = (0..derivative_order - 1):
```


Computer Graphics I Course Assignment

```
        prefix *= n - i;
    for k = (0..derivative_order - 1):
        for i = (0..n - derivative_order):
            beta[i] = beta[i + 1] - beta[i];

    for i = (0..n - derivative_order - 1):
        for j = (0..n - derivative_order - i - 1):
            beta[j] = (1 - t) * beta[j] +
                      t * beta[j + 1];
    }
    return prefix * beta[0];
}
```

2.2.2 curvature. The curvature of a point on a three-dimensional parametric curve can be expressed as follows:

$$r(t) = (x(t), y(t), z(t))$$

$$\kappa(t) = \frac{|r''(t) \times r''(t)|}{|r'(t)|^3}$$

So any vertex on the Bézier curve can be calculated as follows:

```
float curvature(control_points, t):
    v_d1 = at(control_points, t, 1);
    v_d2 = at(control_points, t, 2);
    return length(cross(v_d1, v_d2)) \
           power(length(v_d1), 3)
```

2.3 Ordinary Mesh Construction

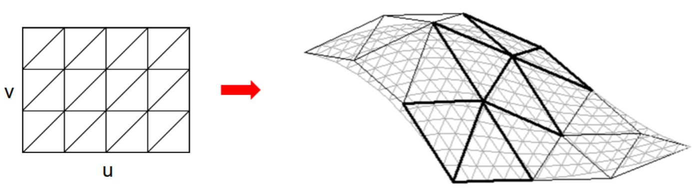

Fig. 2. triangulation in u,v parameter space

Just evaluate (u, v) by uniform distribution, which may lost accuracy at some points (such as high curvature points or between very far neighbor control points).

2.4 Adaption Mesh Construction

My aim is to solve two problems:

1) high curvature points needs more vertex to render
2) neighbor evaluated vertex may be too far between due to far neighbor control points.

So I imply a binary search adaptive mesh construction algorithm as follows:

```
(u_v_list, vertex_list) adaptiveSub(low_v, high_v,
                                    low_position, high_position):
    mid_v = (low_v + high_v)/2
```

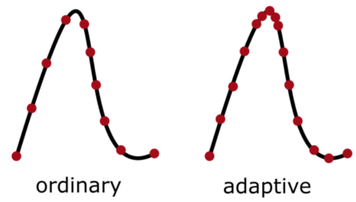

Fig. 3. ordinary vs adaptive evaluation

```
    mid_position = evaluate(u, mid_v)
    mid_curvature = curvature(u, mid_v)
    resolution = distance(low_position, high_position)
    if (resolution >= curvature_distance_epsilon) and
       (mid_curvature >= curvature_max_rate) or
       (resolution >= distance_epsilon):
       temp_u_v_list += [(u, mid_v)]
       temp_vertex_list += [mid_position]
       temp_u_v_list, temp_vertex_list +=
       adaptiveSub(low_v, mid_v,
                   low_position, mid_position)
       temp_u_v_list, temp_vertex_list +=
       adaptiveSub(mid_v, high_v,
                   mid_position, high_position)
    return (temp_u_v_list, temp_vertex_list)
```

And we can do the same on both u, v,then we get a (u, v) list and a vertex list. After triangulate the (u, v) map by Delaunator Algorithm, project the (u, v) map to the 3D area. Finally we can get a smooth enough adaptive Bézier Surface. And my algorithm's efficient is about 4 times slower then the ordinary one.

2.5 B-Spline

B-spline is defined by n order basis B-spline which can be derived by means of the Cox–de Boor recursion formula.

$$B_{i,0}(x) := \begin{cases} 1 & \text{if} \quad t_i \le x < t_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

$$B_{i,k}(x) := \frac{x - t_i}{t_{i+k} - t_i} B_{i,k-1}(x) + \frac{t_{i+k+1} - x}{t_{i+k+1} - t_{i+1}} B_{i+1,k-1}(x)$$

So we can get BasisFunctions as follows:

```
vector<float> basisFunctions(int i, float t) {
    vector<float> coeff, left, right;
    coeff.resize(degree + 1);
    left.resize(degree + 1);
    right.resize(degree + 1);
    coeff[0] = 1.0;

    for (auto j = 1; j <= degree; ++j) {
        left[j] = t - knot_vector[i + 1 - j];
        right[j] = knot_vector[i + j] - t;
        float saved = 0.0;
        for (auto r = 0; r < j; r++) {
```

```
            float tmp = coeff[r] /
                        (right[r + 1] + left[j - r]);
            coeff[r] = saved + tmp * right[r + 1];
            saved = tmp * left[j - r];
        }
        coeff[j] = saved;
    }
    return coeff;
}
```

Then we can evaluate every vertex and draw the surface.

3 RESULTS

3.1 Single Bézier Surface: Ordinary and Adaptive

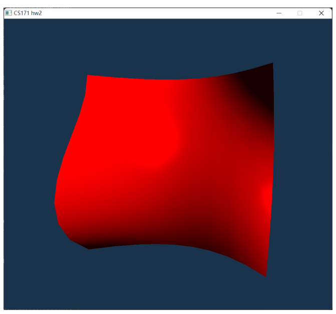

Fig. 4. ordinary evaluation

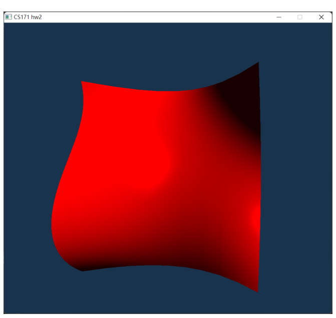

Fig. 5. adaptive evaluation

3.2 Complex Object: Teapot, Teacup and Teaspoon

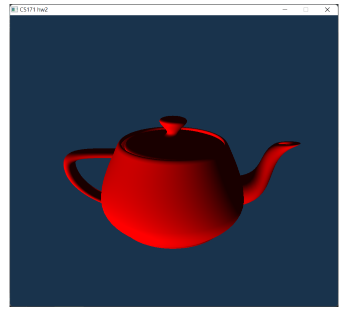

Fig. 6. Utah Teapot

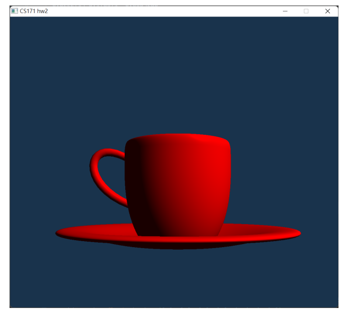

Fig. 7. Teacup

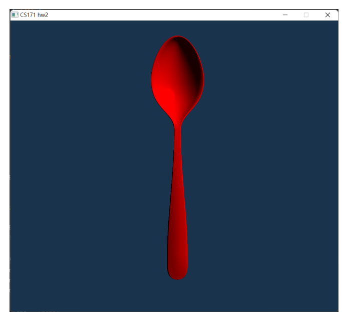

Fig. 8. Teaspoon

3.3 B-Spline

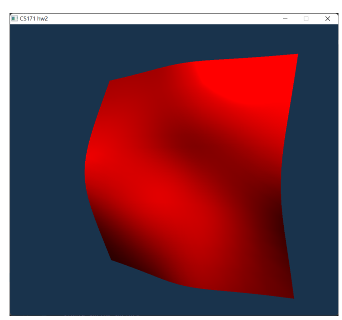

Fig. 9. B-Spline Surface