CSCI 3260 Principles of Computer Graphics

Assignment 4: Bezier Curve and Surface

Due Time: 11:59 pm, Nov 30, 2023 (Thursday)

Fail the course if you copy

I. Introduction

In this assignment, you will explore topics on geometric modeling covered in lecture. You will build Bezier curves and surfaces using de Casteljau algorithm. For possible extra credit, you can optionally implement the vertex normal for smooth shading.

In computer graphics, Bezier curves and surfaces are frequently used to model smooth and infinitely scalable curves and surfaces, such as in Adobe Illustrator (a curve drawing program) and in Blender (a surface modeling program). A Bezier curve of degree n is defined by (n+1) control points. It is a parametric curve based on a single parameter t, ranging between 0 and 1. Similarly a Bezier surface of degree (n,m) is defined by (n+1)x(m+1) control points. It is a parametric surface based on two parameters u and v, both ranging between 0 and 1.

Part I: Bezier Curves with 1D de Casteljau Subdivision.

In Part I of this assignment, you will use de Casteljau algorithm to evaluate Bezier curves and surfaces for any given set of control points and parameters, such as the Bezier curve below.

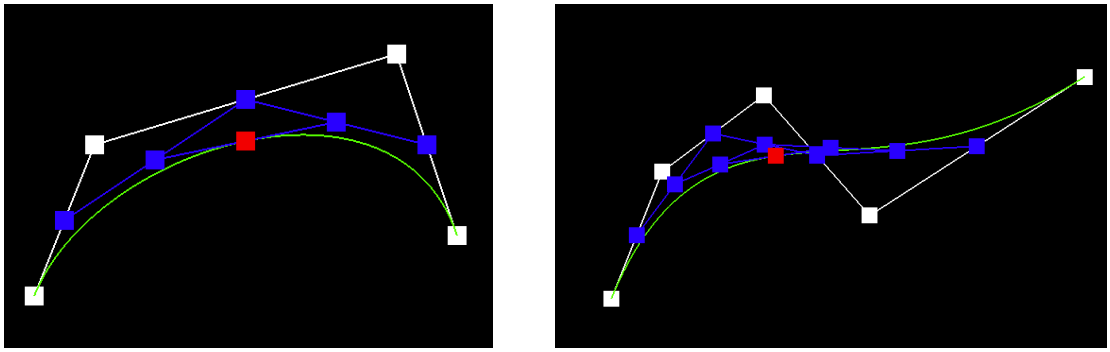

Fig. 1: The Bezier curves of four and five control points, respectively

In the image, white squares are the given control points, blue squares are the intermediate control points evaluated at the given parameters by de Casteljau algorithm, and the red square is the final evaluated point that lies on the Bezier curve.

In Part 1, you will implement Bezier curves. To get started, take a look at bezierCurve.h and examine the member variables defined in the class. Specifically, you will primarily be working with the following:

- std::vector<glm::vec2> controlPoints: A std::vector of original control points that define Bezier curve, initialized from input Bezier curve file (.bzc)
- float t: A parameter at which to evaluate the Bezier curve, ranging between 0 to 1.

Recall from lecture that de Casteljau algorithm gives us the recursive step below.

Given n (possibly intermediate) control points $p_1, \ldots, p_n$ and the parameter t, we can use linear interpolation to compute n-1 intermediate control points at the parameter t in the next subdivision level, $p_1' \ldots p'_{n-1}$, where

$$p'_i = lerp(p_i, p_{i+1}, t) = (1-t)p_i + t\,p_{i+1}$$

By applying this step recursively, we eventually arrive at a final, single point and this point, in fact, lies on the Bezier curve at the given parameter t.

You need to implement this recursive step within BezierCurve::evaluteStep(…) in student_code.cpp. This function takes as input a std::vector of 2D points and outputs a std::vector of intermediate control points at the parameter t in the next subdivision level. Note that the parameter t is a member variable of the BezierCurve class, which you have access to within the function. Each call to this function should performs only one step of the algorithm, i.e., one level of subdivision.

To check if your implementation is likely correct, you can run hw4 with a Bezier curve file (.bzc) and view the generated Bezier curve on screen. For example, you can run the following command in command line tools (assuming that the executable file is named as hw4):

./hw4 ../curve1.bzc

For this command, it consists of two arguments where the first one is the path to your executable file, and second one is the path to your Bezier file, and they might vary on different computers. Here, curve1.bzc is a cubic Bezier curve, and the provided curve2.bzc is a degree-4 Bezier curve. Feel free to create your own .bzc files and explore other Bezier curves.

Functions to modify: (Part I)

- BezierCurve::evaluateStep()

For your write-up: (Part I)

- Briefly explain de Casteljau's algorithm and how you implemented it in order to evaluate Bezier curves.
- Take a look at the provided .bzc files and create your own Bezier curve with 6 control points of your choosing. Use this Bezier curve for your screenshots below.
- Show screenshots of each step/level of the evaluation from the original control points down to the final evaluated point. Press E to step through. Toggle C to show the completed Bezier curve as well.
- Show a screenshot of a slightly different Bezier curve by moving the original control points around and modifying the parameter t via mouse scrolling.

Part 2. Bezier Surfaces with Separable 1D de Casteljau

In Part 2, you will adapt what you have implemented for Bezier curves to Bezier surfaces. To get started, take a look at bezierPatch.h and examine the member variables defined in the class. Specifically, you will primarily be working with the following:

- std::vector<std::vector<glm::vec3>> controlPoints: A 2D std::vector that has nxn grid of original control points. controlPoints is of size n and each inner std::vector, i.e., controlpoints[i] contains n control points at the ith-row. For example, controlPoints[0] and controlPoints[n-1] each have n control points at the bottom row and the top row, respectively.

Recall from the lecture that separable 1D de Casteljau algorithm works as the following:

The inputs to the algorithm are (1) a nxn grid of original control points, P_{ij}, where i and j are row and column index, and (2) the two parameters u and v.

We first consider that each row of n control points, $P_i^0, \dots, P_i^{n-1}$, define a Bezier curve parameterized by u. Just as in Part 1, we can use the recursive step to evaluate the final, single point P_i on this Bezier curve at the parameter u. We then consider that P_i for all n rows define a Bezier curve parameterized by v. (These n points lie roughly along a column.) We can again use the recursive step from Part 1 to evaluate the final, single point P on this Bezier curve at the parameter v. This point P, in fact, lies on the Bezier surface at the given parameter u and v.

You need to implement the following functions in student_code.cpp.

- BezierPatch::evaluateStep(…): very similar to BezierCurve::evaluateStep(…) in Part 1, this recursive function takes as inputs a std::vector of 3D points and a parameter t. It outputs a std::vector of intermediate control points at the parameter t in the next subdivision level.
- BezierPatch::evaluate1D(…): This function takes as inputs a std::vector of 3D points and a parameter t. It outputs directly the final, single point that lies on the Bezier curve at the parameter t. This function does not output intermediate control points. You may want to call BeizerPatch:;evaluateStep(…) inside this function.
- BezierPatch::evaluate(…): This function takes as inputs the parameter u and v and outputs the point that lies on the Bezier surface, defined by the nxn controlPoints, at the parameter u and v. Note that controlPoints is a member variable of the BezierPatch class, which you have access to within this function.

Sanity Check:

To check if your implementation is likely correct, you can run hw4 with a Bezier surface file (.bez) and view the generated Bezier surface on screen.

For example, you can run the following command in command line tools and you should see a teapot:

./hw4 ../teapot.bez

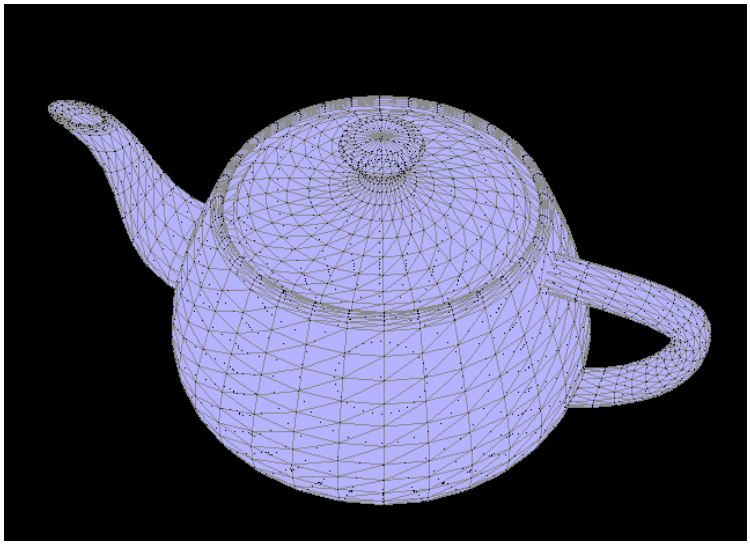

Fig. 2: The teapot made with Bezier surface

Functions to modify: Part 2.

- BezierPatch::evaluateStep(…)
- BezierPatch::evaluate1D(…)
- BezierPatch::evaluate(…)

For your write-up: Part 2

- Briefly explain how de Casteljau algorithm extends to Bezier surfaces and how you implemented it in order to evaluate Bezier surfaces.
- Show a screenshot of bez/teapot.bez and of bez/wavy_cube.bez evaluated by your implementation.
- Press "a" to increase the mesh density and press "d" to decrease the mesh density. Show screenshots of the teapot and the wavy_cube at different mesh densities.

Part III: Compute the surface normal for shading

In Part III of this assignment, we are going to compute the surface normal for shading. Each Bezier Patch is represented as a list of triangles. Thus we will just need to compute the normal for each of the triangles. Suppose the three vertices of a triangle is p0, p1, p2 (in anti-clock order), then the normal of the triangle is

n = cross (p1-p0, p2-p0)

n = normalize(n)

Please implement this calculation is the function: compute_triangle_normal(…) in student_code.cpp.

Sanity Check:

To check your implementation is correct, you can run the following command in command line tools:

./hw4 ../teapot.bez

And press “q” to toggle the shading mode. Press “w” to remove the wireframe. You will see the something like this picture.

Fig. 3: The shading effects with surface normal

Functions to modify: Part 3

- compute_triangle_normal()

For your write-up: Part 3

- Show screenshots of the shading mode for both teapot and wavy_cube
- Show screenshots of the shading mode for both teapot and wavy_cube at different mesh densities (press “a” and “w” to change the mesh densities)
- Discuss the effect of shading w.r.t mesh density. Discuss any remaining problems of this shading.

Bonus Points:

There are several ways to get bonus points in this assignment.

- In Part III, we assign the same normal to all the three vertices of a triangle. This is not ideal. You can implement a per-vertex normal. The normal at a given vertex can be computed as a weighted average of the surface normals of the triangles surrounded the vertex. The weight can be set as the area of each triangle. **You should show the comparison with the the previous one and tell the difference in the write-up.**
- Bezier surface allows us to compute texture coordinate analytically for each vertex as well. You can implement texture mapping for the Bezier surfaces using the provided texture images. Your result should be like in Fig. 4.

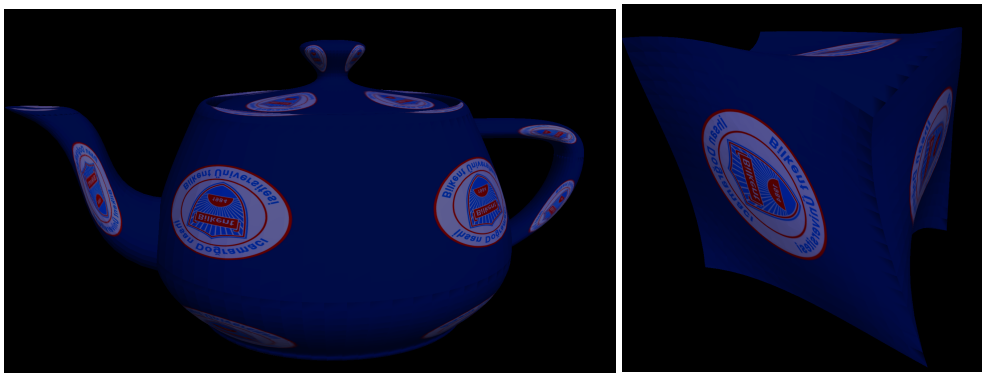

Fig. 4: Texture mapping for Bezier surfaces

II. Grading Scheme

Your assignment will be graded by the following marking scheme:

Basic (100%)	
Draw the two Bezier curves we provided.	25%
Draw a customed Bezier curve with 6 control points.	25%
Draw the teapot and wavy_cube.	25%
Correctly calculate the surface normal.	25%
Advanced (at most 20%)	
Implement a per-vertex normal.	10%
Implement texture mapping for the Bezier surfaces.	10%
Total (maximum):	**120%**

Note: no grade will be given if the program is incomplete or fails compilation.

III. Guidelines to submit programming assignments

1) You can write your programs on Windows and macOS. The **official grading platform** should be macOS. If we encounter problems when execute/compile your program, you may have to show your demo to the teaching assistant in person.

2) For the basic task, modify the provided *student_code.cpp* and write all your code in this file. For the bonus task, you can modify existing files, but it is **not recommended** to create other additional **.cpp** or **.h** files. Type your full name and student ID in *student_code.cpp*. ***Missing such essential information will lead to mark deduction (up to 10 points).***

3) We have provided a **CMakeLists.txt** file in the skeleton code. You are free to choose to use CMake (tutorial week 12) or setting up the project by yourself (tutorials in week 1). Both ways should work.

4) Zip the source code file (i.e., *student_code.cpp*), Bezier curve with 6 control points (i.e., *bezier3.bzc*), the executable file and libs (e.g., *Assignment4.exe & glew32.dll*) for windows users, the write-up (i.e., *readme.pdf*) in a .zip (see Fig. 5). **If you have changed any files to get bonus, please also put them in the zipped folder**. Name the zip with your own student id (e.g., *1155012345.zip*).

5) We may check your code. And we may deduct your marks if you have seriously wrong implementation.

6) Submit your assignment via eLearn Blackboard. (https://blackboard.cuhk.edu.hk).

7) Please submit your assignment before 11:59 p.m. of the due date. **Also check the course late policy.**

8) In case of multiple submissions, only the latest one will be considered.

9) ***Fail the course if you copy.***

a) Windows b) Mac

Fig. 5. Files to be submitted