CSE 528 COMPUTER GRAPHICS (FALL 2024)

HOMEWORK TWO

For this homework, there are two separate parts: (1) Non-programming part; and (2) Programming part. Please refer to the TA Help Page for general requirements, and take note that all deadlines are **strictly enforced**. To prevent late submissions caused by network issues, **avoid** submitting at the last minute! Please also note that your credits for this homework will be scaled (i.e., from $100+200+100$ to $50+100+50$) on Brightspace to match the overall grading schemes.

Non-programming Part

1. (**60 points**) Please define and explain the following commonly-used terms in computer graphics modeling and rendering: (1) Perspective projection; (2) Normal of a plane; (3) 3D graphics viewing pipeline; (4) Mid-point algorithm; (5) Oblique parallel projection; (6) Conic sections; (7) Solid models and CSG operations; (8) BSP; (9) Octree representation; (10) Trilinear interpolation; (11) Surfaces of revolution; (12) Subdivision curves; (13) Quadric surfaces; (14) Sweeping objects; (15) Developable surfaces; (16) Local illumination models; (17) Ray tracing; (18) Phong shading; (19) Environment mapping; (20) Procedural modeling; (21) L-systems; (22) Z-buffer; (23) Free-form deformation; (24) Physics-based modeling; (25) Rational Bezier spline; (26) Hidden surface removal; (27) Sketch-based interface; (28) Bicubic spline polynomials; (29) Global illumination models; and (30) Texture mapping.

2. (**40 points, 8 points for each part**) (a) What are the fundamental differences among linear, affine, and projective transformations in 2D and 3D graphics, to simplify the discussions, you can simply focus on 2D transformations now. (b) Besides discussing their differences, please also document their properties for each transformation, and when such transformation becomes useful in 2D graphics. (c) In practice, consider Figure 1 as an example, we are applying 2D transformations to image warping and deformation, how do you recover the linear, affine, and projective transformations if you believe that $T(x, y)$ is one of them? In particular, how can you recover all relevant parameters? Your answer should be as formal as possible with a mathematical proof. (d) Could you please extend your method and your discussions to handle their possible combination and its reconstruction as far as $T(x, y)$ is concerned? (e) What if $T(x, y)$ involves some kinds of non-linear transformation, what are your thoughts on handling and recovering non-linear transformation? For this part only, you do NOT need to provide any proof, a clear description and justification of ideas would suffice for this part!

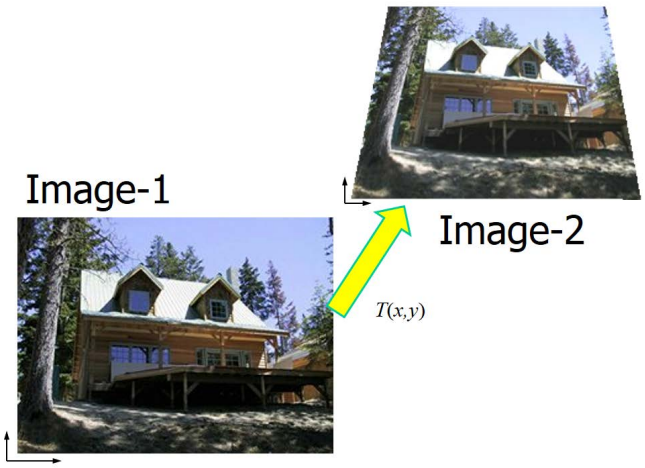

Figure 1: Illustration of a 2D image transformation (from Image-1 to Image-2). Certain types of transformation and/or their possible combination occur from Image-1 to Image-2. We are interested in recovering the right transformation $T(x, y)$.

Programming Part

The main objective of this programming component is to develop an interactive curve editor with which you will be familiar with the properties of cubic Bezier curves (serving as the basic building block) and how cubic Bezier curves can be used for data interpolation and fitting. Once again, we will be focusing on 2D computer graphics for this assignment. Even though our goal is to develop a simple interactive curve editor, we will achieve this goal through a step-by-step approach.

Starting from this homework, you will be granted complete access to modern OpenGL. Please feel free to explore different OpenGL commands, shaders, and other functionalities, including those not mentioned in our lectures.

However, please keep in mind that you are still required to use OpenGL in the **core-profile** mode (modern OpenGL). Deprecated functions in immediate mode (legacy OpenGL) are **not** permitted. Furthermore, you are required to draw all parametric curves in this programming part with **OpenGL's tessellation shaders**. Do **not** rasterize them manually in the main program and flush huge vertex buffers to the rendering pipeline. (Such an implementation violates industrial practice, and has poor performance.)

1. (**20 points**) Write a simple program that can display one segment of cubic Bezier curve defined by four control points, once again, we are only focusing on 2D graphics for this assignment (except the bonus part)!

   User Interface. The user will press key **1** to enter this part. The user will left-click the mouse at the first three locations to specify the first three control nodes, and right-click the mouse somewhere to specify the last control node. When the user moves the mouse, there should be a preview of a polyline connecting all control nodes ("bounding polyline", with control nodes highlighted with larger marks, e.g., `GL_POINTS` with `glPointSize(9.0f)`. When the user right-clicks the mouse, the preview should be finalized, and the corresponding Bezier curve should be displayed. For your convenience (mainly for part 2), you could assume that the user will only draw one spline in this part. If the user wants to draw another spline, he or she

will switch to another part, and then switch back to this part. This should reset everything in this part, and the user will restart from zero.

2. (**120 points**) Extend the above part to implement a simple interactive curve editor for C^2 interpolating splines using piecewise cubic Bezier curves (i.e., having many pieces of cubic Bezier curves, and at the data point where two cubic Bezier curves meet the curve should be C^2). Your program should afford users the following functionalities: (1) Interactively specify the location (i.e., (x, y) components) of all the interpolation points using the mouse in a sequential order; (2) After the curve creation, interactively select any interpolation point of the curve and move this point to any location on 2D with mouse; (3) Insert a new interpolation point after any selected point; (4) Delete a selected interpolation point; (5) Read curve interpolation points from your data file; and (6) Save curve points into your data file. Please feel free to add any necessary features in order to make the program interface user-friendly as much as you could.

   User Interface. This part is an extension of part 1, and there's no need to "switch" from part 1 to here. The user will use four sequential mouse left clicks to specify the first segment of the piecewise C^2 cubic Bezier spline. Please note that, the piecewise-C^2 constraint implicitly determines the first three control nodes for the next segment. The user will use several left clicks to add more segments to the spline, one segment at a time. The user will add the last segment and finish drawing with a right click. Like part 1, there should be a preview of the "bounding polyline" of the spline. After the user right-clicks the mouse, the complete spline should be displayed. For your convenience, you could assume that from now on, the user will only perform the following modifications to the existing spline. If he or she wants to draw a new spline from zero, he or she needs to switch to another part, and then switch back to this part.

   Control Node Repositioning. After the spline is finalized, the user will left-click an existing control node to select it, and the program should display the node in a different color as visual feedback. The user should then be able to drag the selected node to somewhere else, and the spline should be updated. For your convenience, we do not require a preview when the user drags the node.

   Due to the piecewise-C^2 constraint, this dragging will update more than one control node. You are free to determine which nodes should be updated, as long as the dragged node remains in its new location, and the whole spline remains piecewise-C^2.

   Control Node Insertion. The user should be able to select an existing control node, and add a new control node after the selected node, by left-clicking the mouse while pressing and holding the **Insert** key. The whole spline would need to be updated to fit the new set of control nodes. You are still free to determine which nodes should be updated, as long as the new node remains in its original location, and the whole spline remains piecewise-C^2.

   Control Node Deletion. The user should be able to select an existing control node and press the **Delete** key to delete it. Again, the whole spline would need to be updated to fit the new set of control nodes. You are still free to determine which nodes should be updated, as long as the neighbors of the deleted node remain in their original locations, and the whole spline remains piecewise-C^2.

   From And to Files. The user should be able to save the existing spline to the configuration file (by pressing combo keys **Ctrl + S**), or load and display another spline from the configuration file (**Ctrl + L**). The configuration file `sbuid_hw2/etc/config.txt` has $N + 1$ lines. The first

line contains three space-separated integers, separately denoting: (1) Whether this configuration is 2D or 3D (see the BONUS part), 2 for 2D and 3 for 3D; (2) Whether this configuration denotes a piecewise-C^2 spline or a piecewise-C^1 spline, 2 for C^2 and 1 for C^1; and (3) The number of control nodes, N. The succeeding N lines will contain the coordinates of the control nodes, e.g., for 2D scenes, 2 space-separated floating numbers denoting the x and y screen-space coordinates of the node. The screen-space coordinate system is that, the origin is the bottom-left corner, with the positive x-axis points toward the right, and the positive y-axis points toward the top.

3. (**60 points**) As discussed in detail in our lectures, the above piecewise representation indeed achieves C^2 continuity at data interpolation points (i.e., junction points which are shared by two consecutive cubic Bezier curves), however, through your programming efforts you have made all the way to this point, you should have realized that the above scheme is far from being ideal because this representation does not afford local control of the graphics modelers/animators. So we are hoping to continue to improve the above implementation. The idea is to give up C^2 continuity requirement, but only require C^1 continuity at all the data interpolation points, with a goal of local control. This scheme is also called Catmull-Rom spline, and your program should implement a Catmull-Rom spline which is essentially a piecewise cubic Bezier representation, but only requires C^1 continuity, so please modify the previous implementation to arrive at a Catmull-Rom spline curve editor. At the same time, you should also have the above functionalities (1-6) explained in the previous phase.

   User Interface. The user will press key **3** to enter this part. All other GUI interactions remain the same as before. The only difference is that piecewise-C^1 splines allow more interpolation points at each segment (compared to piecewise-C^2 cubic Bezier curves). Please modify your program to fit this new scenario.

4. (**BONUS, 100 points**) Extend the above implementation to 3D graphics, i.e., you should be able to interactively specify (x, y, z) coordinates using mouse. This includes three parts: (a) Extend one 2D Bezier curve to a 3D setting; (b) Extend your piecewise C^2 spline curve editor to the 3D environment; and (c) Extend your Catmull-Rom curve editor to 3D. One viable approach is to map a 2D point to a 3D point, and the virtual trackball technique could help you with this purpose (detailed in the lectures).

   User Interface. The user will press key **4** to enter this part. All other GUI interactions remain the same as before. Please modify your program to fit this new scenario.