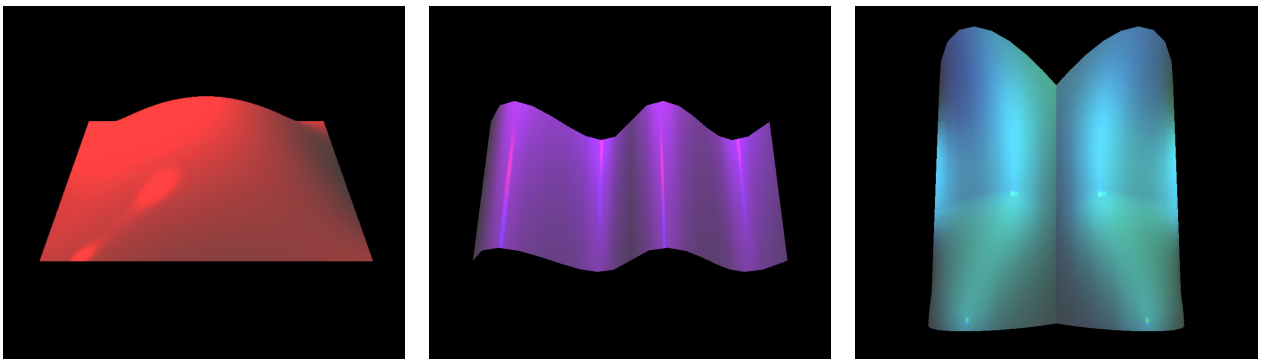
Middle East Technical University | Department of Computer Engineering

CENG 469

Computer Graphics II

Spring 2022-2023
Assignment 1
Surface Rendering
(v1.1)
v1.1: Added/edited text for clarification, added supplementary figures.

Due date: April 2, 2023, Sunday, 23:59



(a) input1.txt (b) input2.txt (c) input3.txt

Figure 1: Output visuals of the provided input files.

1 Objectives

In this assignment, you are going to render smooth surface visuals in OpenGL by using the data provided in the input file. The expected end product of this assignment is an OpenGL program which:

- Renders the surface whose information resides in the input text file that is provided as the only input argument to the program,
- Shows the diffuse and specular shading effects of the given point light(s) on the surface,
- Includes user interaction capabilities with keyboard buttons.

The specifications are explained in Section 2. Note that you are free to implement your own design choices as long as you obey the requirements detailed in this assignment text.

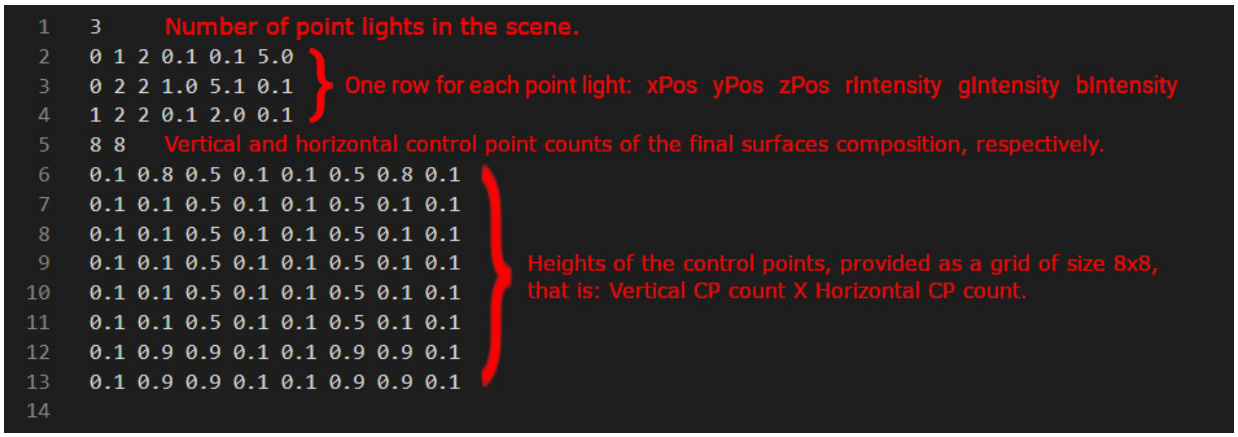


Figure 2: Input file format detailed line by line. In this example, the number of vertical and horizontal Bezier surfaces are both $8/4 = 2$, totalling to $2\text{x}2 = 4$ surfaces in the final composition. Grid values indicate the heights of each control point. Control point heights are read once when the program is launched, and do not change during the runtime. For more explanation on the grid structure of the control point heights, see Figure 3.

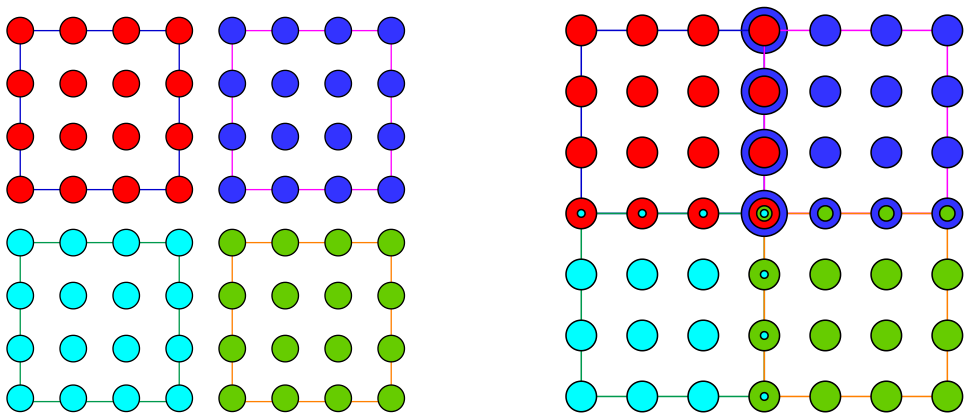
2 Specifications

1. Vertex and fragment shaders will be implemented in this homework, and the Bezier surface equation should be used to produce the surface visual.

2. On launch, the program should read an input text file that includes point light definitions of the scene and the heights of the Control Points (CPs) of one or more Bezier surfaces. Those values do not change during the runtime. The format of the input file is discussed in Figure 2 and Figure 3, and it is detailed as follows:

   - There will be at least 1 and at most 5 point lights defined.
     - The RGB intensity values of each point light should be attenuated in the lighting calculations by applying the inverse square law.
   - There will be at least 4 and at most 24 CPs defined in each vertical and horizontal axes.
     - As each Bezier surface uses 4 CPs, the input file will include at least 1 Bezier surface, and at most $(24/4)\text{x}(24/4) = 36$ Bezier surfaces. It is also possible to have a non-square surface layout such as 1x6 many Bezier surfaces, meaning a horizontally long surface composition in the final 3D visual.

3. The vertex shader should implement the Bezier surface equation. The vertices that are to be used in OpenGL rendering will be placed to their 3D world locations as a uniform grid with their appropriate heights queried from the Bezier surface equation whose CP heights are given. Suppose the user-interactable parameter **samples** is 10, then 10x10 vertices should be sampled that uniformly covers that surface equation including the corners, for each different surface equation (i.e. if multiple Bezier surfaces' CP heights are defined in the input file).



(a) (b)

Figure 3: When multiple Bezier surfaces are defined in the input file, the locations of the Control Points that reside in the edges of those different surface equations overlap in the 3D world, as seen in (b). However, the input file specifies them separately while indicating their heights, as shown in (a). Notice the color matchings between (a) and (b). Circle sizes differ just for visual clarity.

The user should be able to increase/decrease the value of this parameter using keyboard buttons during runtime as follows:

- The parameter's initial value should be 10.
- Tapping **W** button should increase its value by 2 up to 80.
- Tapping **S** button should decrease its value by 2, down to a minimum value of 2.

4. OpenGL should draw triangles using the sampled vertices. Wireframe visuals in Supplementary section can be inspected for a better understanding.

5. The normals of the vertices should be calculated from the surface equation. The normals will then be used during the lighting and color calculations in the fragment shader.

6. The fragment shader should implement ambient lighting, Lambertian diffuse, and Blinn-Phong specular shadings. All point lights that are defined in the input text file should be included in the lighting calculations in the fragment shader. As mentioned previously, the RGB point light intensities should be attenuated by applying the inverse square law.

7. The user-interactable parameter **coordMultiplier** is used to extend or shrink the X and Y coordinates the whole surface starts and ends at. Its effect can be seen in the example runtime video, where we center the whole surface to the origin if the vertical and horizontal CP counts are given as the same in the input file (i.e. as a square grid such as defined in Figure 2). The parameter should be configured as follows:

   - The parameter's initial value should be 1.0.
   - Tapping **E** button should increase its value by 0.1. There is no upper value limit.
   - Tapping **D** button should decrease its value by 0.1, down to a minimum value of 0.1.

8. The user-interactable float parameter **rotationAngle** is used to rotate the surface around the horizontal axis so that the surface can be inspected thoroughly. The angle is defined in degrees. Its effect can be seen in the example runtime video, and it should be configured as follows:

    - The parameter's initial value should be -30.0.
    - Tapping **R** button should increase its value by 10. There is no upper value limit.
    - Tapping **F** button should decrease its value by 10. There is no lower value limit.

9. Figure 1 results are produced artificially for better showcasing (rotated to an artificial angle manually to show the forms of the surfaces and the specular light effects in a better angle, etc.). Therefore, in Supplementary section, we provide the exact expected visuals when your code is launched with the provided input files. The following setup is to be used to produce the visuals given in the Supplementary section:

    - The initial width and height of the window is 800x600, respectively.
    - Camera is located at (0, 0, 2), and looking at -Z direction (into the scene).
    - Perspective projection with 45 degree Fovy angle is used. Near and Far are set as 1.0 and 100.0, respectively.
    - The surface composition, which includes 1 or many Bezier surfaces, lays on the X-Y plane with equidistant CPs and equidistant OpenGL vertices, and it can be rotated around the horizontal axis. Positive X is towards the right of the figure, and positive Y is towards the up. The whole surface fits into the X-Y plane's coordinates from (-0.5, -0.5) to (0.5, 0.5) if the input file defines a square-like grid (e.g. 1 Bezier surface, or 2x2, or 3x3 surfaces, etc.). If a non-square grid is given in the input file such as 1x2 Bezier surfaces (as in Figure1b), the longest side of the final surface composition fits to that axis' [-0.5, 0.5] range starting from -0.5. Then, the other axis is filled until that axis's half-way (from -0.5 to 0), due to ratio of the given vertical and horizontal surface counts of 1x2. **coordMultiplier** parameter is multiplied with the total of start and end coordinates of -0.5 and 0.5, with is 1.0. Therefore, when that parameter becomes 1.2, for example, the coordinate range becomes [-0.6, 0.6].
    - As was discussed in the **coordMultiplier** part, it is set as 1.0 initially.
    - As was discussed in the **rotationAngle** part, -30.0 degree rotation is applied initially.
    - As was discussed in the **samples** part, 10x10 vertices per Bezier surface exist initially.
    - During the lighting calculations in the fragment shader:
        - Ambient light radiance: (0.8, 0.8, 0.8).
        - Ambient reflectance coefficient: (0.3, 0.3, 0.3).
        - Diffuse reflectance coefficient: (0.8, 0.8, 0.8).
        - Specular reflectance coefficient: (0.8, 0.8, 0.8).
        - Phong exponent: 400.

10. You can inspect the Matlab sample codes (**curves_and_surfaces_matlab_codes.zip** file) provided to you on ODTUClass for the Bezier surface equation part.

11. There is no frames-per-second (FPS) constraint in the homework, but the user interaction should feel continuous and smooth. Even though all the hard work such as querying the surface equation is done in the vertex shader (for each vertex, in parallel, instead of calculating on CPU sequentially) in this homework, as you increase the number number of samples, you can see a small slow-down in your application.

12. **Hints:**

    - gl_VertexID variable can be used in the vertex shader in conjunction with glDrawElements call, and the vertices' X and Y locations can be set in the vertex shader directly via some mathematical operations to their appropriate locations in the final surface visual, without needing to pass X-Y vertex location data from CPU to GPU. Their heights are sampled from the Bezier equation in the vertex shader.
    - Any additional necessary information can also be passed to the shaders as uniforms.

13. **Blog Post:** You should write a blog post that shows some output visuals from your work, and explains the difficulties you experienced, and design choices you made during the implementation phase. You can also write about:

    - An analysis of **samples** vs **FPS** (frames-per-second) of your code (maybe as a table).
    - If you want to implement any other vertex normal calculation method other than querying the Bezier surface equation, a discussion on how you have implemented it, or why you couldn't.
    - Some interesting design choices you made during implementation, etc.

    The deadline for the blog post is 3-days later than the deadline for the homework. You can submit your code before the deadline, and finalize the blog post during the 3-day late submission period without losing late days. You consume your late days only if you submit your "code" late. However, submitting the blog post more than three days later will incur grade deductions.

3 Regulations

1. **Programming Language:** C/C++ is highly recommended. You also must use gcc/g++ for the compiler if you use those programming languages. Any C++ version can be used as long as the code works on Inek machines. If you use any other programming language, be sure that the Inek machines can compile and run your code successfully and also include a simple Readme file to describe how we should run your code on Inek machines during the grading (if you don't use C/C++).

2. **Additional Libraries:** GLM, GLEW, and GLFW are typical libraries that you will need. You should not need to use any other library. But if you still want to use some other library, please first ask about it in the ODTUClass forum of the homework.

3. **Groups:** All assignments are to be done individually.

4. **Submission:** Submissions will be done via ODTUClass. You should submit your blog post link to Blog Links forum on ODTUClass, so that the URLs are publicly available for everyone's access, using the title "HW1 Blog Post". For code submission, create a "**tar.gz**" file named "hw1.tar.gz" that contains all your source code files, shader files, and a Makefile. The executable should be named as "**main**" and should be able to be run using the following commands (any error in these steps may cause a grade deduction):

   tar -xf hw1.tar.gz
   make
   ./main input.txt

5. **Late Submission:** You can submit your codes up to 3 days late. Each late day will be deduced from the total 7 credits for the homeworks of the semester. However, if you fail to submit even after 3 days, you will get 0 regardless of how many late credits you may have left. If you submit late and still get zero, you cannot claim back your late days. You must e-mail the assistant if you want your submission not to be evaluated (and therefore preserve your late day credits).

6. **Cheating: We have zero tolerance policy for cheating**. People involved in cheating will be punished according to the university regulations and will get 0 from the homework. You can discuss algorithmic choices, but sharing code between students is strictly forbidden. Please be aware that there are "very advanced tools" that detect if two codes are similar.

7. **Forum:** Any updates/corrections and discussions regarding the homework will be on ODTUClass. You should check it on a daily basis. You can ask your homework related questions on the forum of the homework on ODTUClass.

8. **Grading:** Your codes will be evaluated on Inek machines. We will not use automated grading, but evaluate your outputs visually. Implementing the homework as follows will get you 100 points:

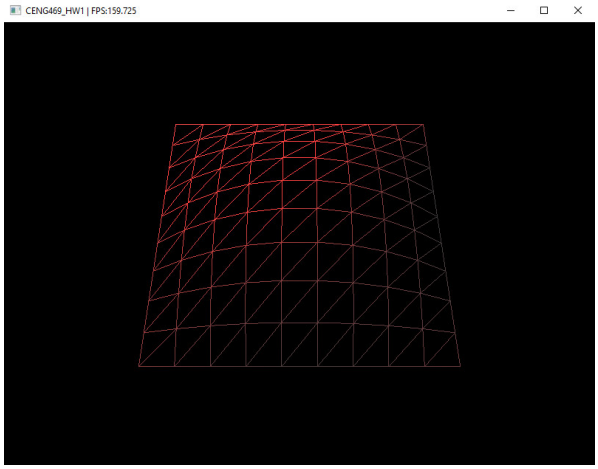
   - Correctly implements the user-interaction capabilities, and,
   - When launched with an input.txt file that is given as a launch argument, it produces the expected rendering visual with the correct surface calculations, normal calculations and lighting calculations. Example visuals for some example input text files can be seen in Supplemental section. Different input text files will be tested during grading.

   Note that you should test your code on an Inek machine before submission, as the frame rate might not be as high as your home computer if you have a powerful PC, and might cause hangs when launched. This might require reducing/optimizing per frame calculations in the code.

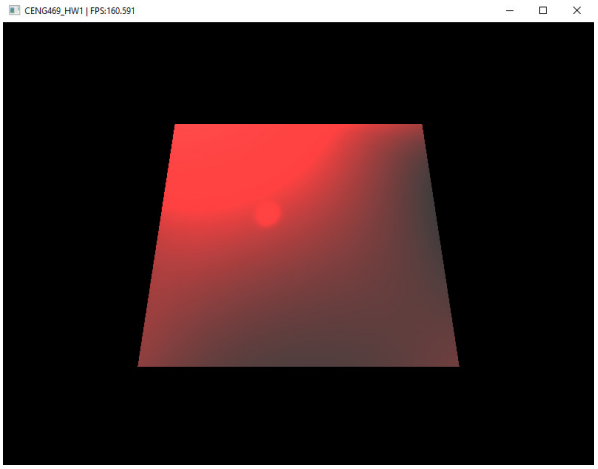
   Blog posts will be graded by focusing on the quality of the content.

   Supplementary section can be found in the next page.

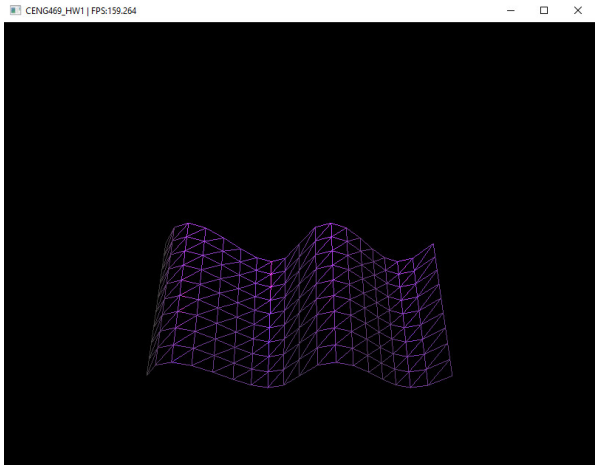
Supplementary



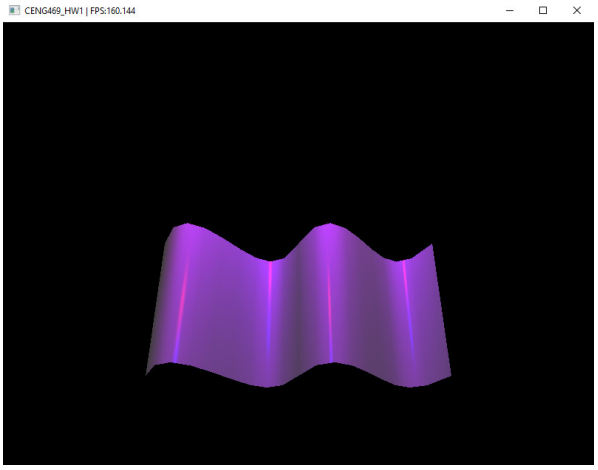
(a) Wireframe visual of input1.txt



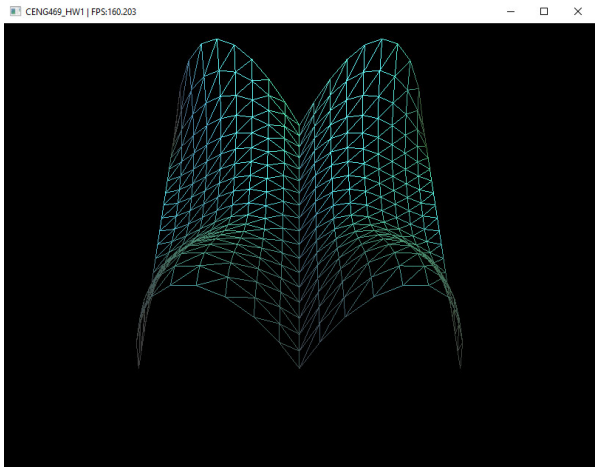
(b) Non-wireframe visual of input1.txt



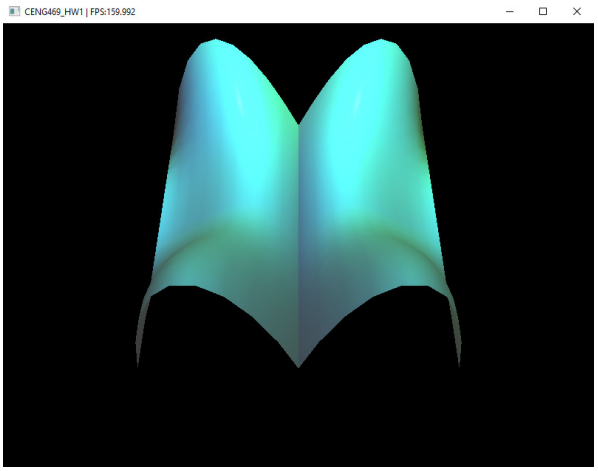
(c) Wireframe visual of input2.txt



(d) Non-wireframe visual of input2.txt



(e) Wireframe visual of input3.txt



(f) Non-wireframe visual of input3.txt

Figure 4: Non-wireframes are the expected visuals of the input files when your code launches. To experiment in wireframe mode: glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);