CS535: Assignment #3 - Bump it!
Normal Mapping + Shadow Mapping + XX

Out: February 24, 2026
Back/Due: March 12, 2026

Objective:
The objective of this assignment is helping you understand GPU programming of shaders through bump mapping / normal mapping, shadow mapping, and more. First you will learn how to introduce more details to the existing lighting system using normal mapping, which increases the complexity of the illusion on the surface; then you will have a feeling of how shadows improve the sense of depth and immersion to the scene.

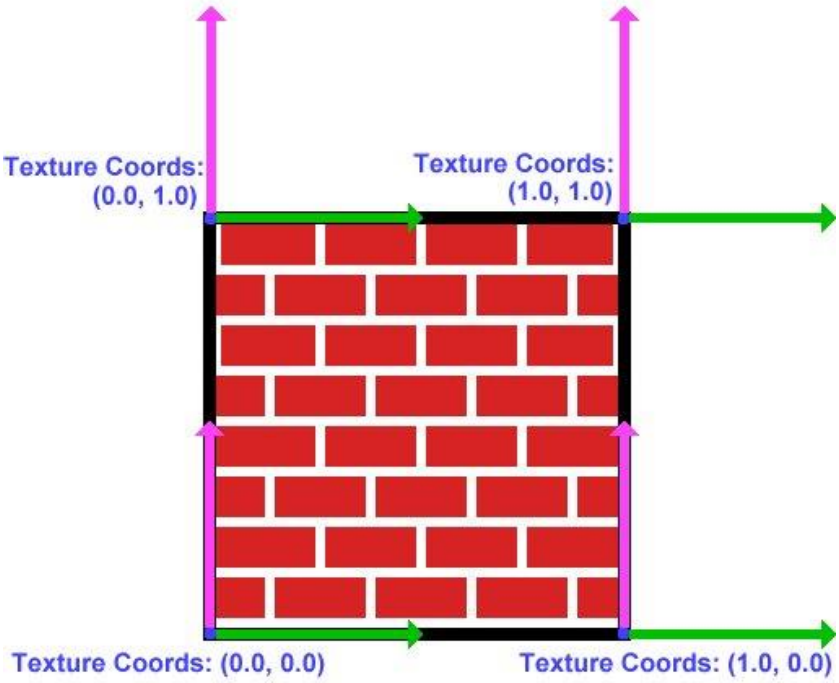
Summary:
In this assignment, you are provided with a scene with several cubes placed on a plane. For simplicity there is only one point light source. Your first task is adding normal mapping (also called bump mapping) to the illumination system, for which you need to first calculate tangent/bitangent vectors and then, in the shader, construct the TBN coordinate system and convert the computation of lighting from world space to tangent space. In the second task, the scene will be rendered twice. In the first pass it is rendered from the light's perspective to get the depth map as texture, in the second pass the scene is just rendered as normal. So you have to first calculate the matrix to convert the scene from world space to the light's viewing space (the goal is to get the depth map which will be used as texture in the second pass), then in the shader you need to implement a function to decide whether a fragment is in shadow.

Specifics:

1. Start with the template from the course website. The template comes with keyboard and mouse controls that allow you to rotate the view, move and rotate the light sources, change the viewing modes, and alter the light and material properties. Read the command-line output for an overview of the provided controls. Upon starting the program, a config file `config.txt` is read that gives default values for the abovementioned properties. Feel free to modify this file when testing your code. You may also provide an alternative configuration file as a command-line argument. The textures (also the normal maps) are put under *textures/*.
2. **Model loading and setup.** The models used in this assignment are *cube.obj* and *plane.obj*, whose format is more complex than the models in the previous assignment, since texture coordinates are involved. For the specification of OBJ files, you can refer to the instructions here. In config.txt, rotation matrices and translation vectors are also included to give the objects an initial position, and their format is the same as in

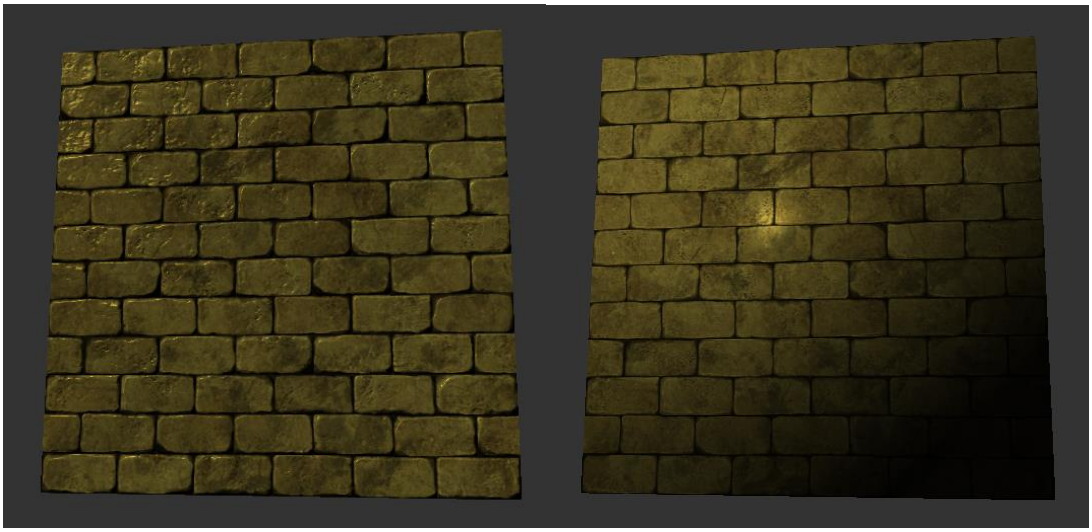
assignment #1. The scene only has one positional light. You can use a keyboard/mouse to change the position of the light/objects.

3. **Normal mapping (60%).** The normals in normal maps are in their local coordinate systems (i.e. tangent space/TBN space). To ensure correct lighting, you have to obtain the TBN system for each face and then convert all related light vectors to this system and compute the shading.
    - **Tangents and bitangents (20%).** In `mesh.cpp`, you need to complete the `load` function. For each triangle of the mesh, given the positions and the texture coordinates of the three vertices, you need to calculate tangent and bitangent vectors, and then store them to *vertices*, so that they can be passed to shaders. From the lecture slides you can find the procedure to get them.
    - **Tangent space (20%).** It is a space local to the surface of the model. It consists of normal, tangent and bitangent vectors. Normal is already given as face normal. In the image below, normal is pointing out of the image, the pink arrows are bitangents and the green ones are tangents. In the `vertex shader`, you need to set up the TBN system and then convert the related lighting vectors (light position, viewer vector and fragment position) to tangent space, before passing them to the fragment shader.



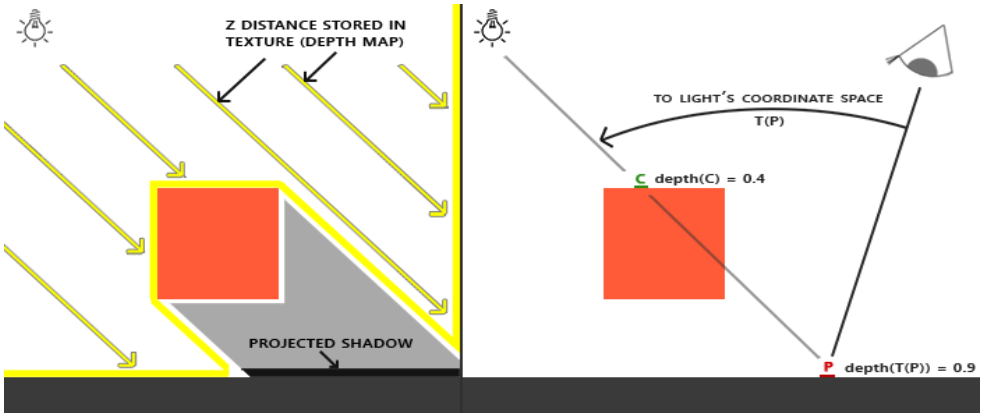
    - **Lighting calculation (20%).** The normal map is given as `texCubeNorm`. There are three places in the `fragment shader` where you need to get the normal, the light vector and the view vector (all in tangent space). If you calculate them correctly, you will see the result below (when you turn on normal mapping mode by pressing *m/M*):

Turning on normal mapping v.s. Turning off:



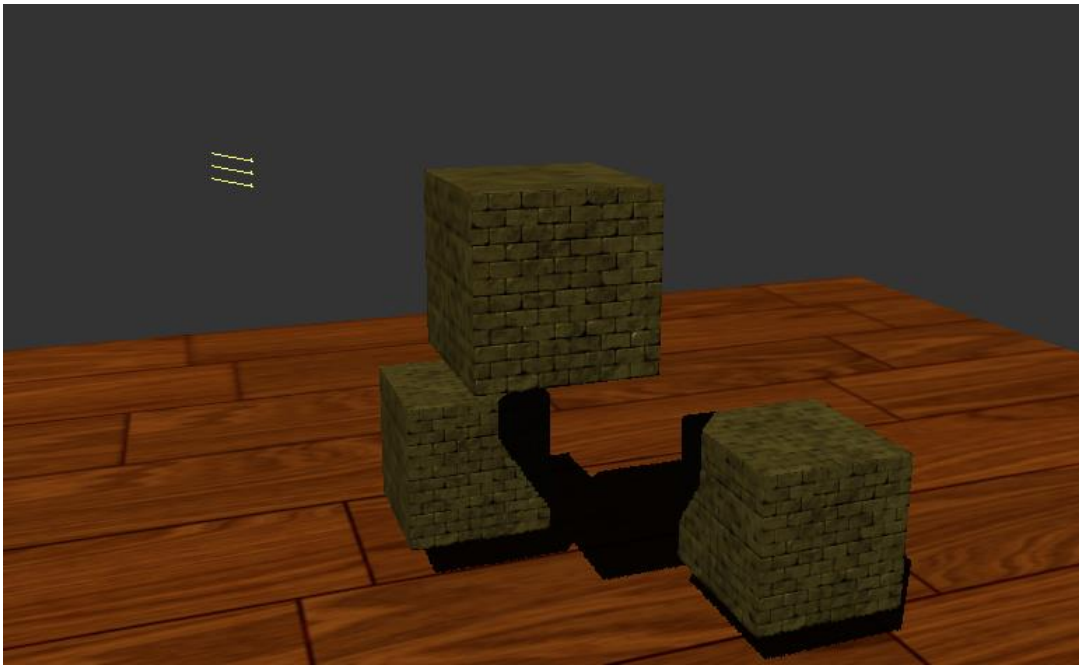
4. **Shadow Mapping (40%).** In the implementation of shadow mapping, there will be two passes.
   In the first pass, to decide what is visible and what is invisible (thus in shadow), you will render the scene from the light's perspective. To generate the depth map/shadow map, this stage requires two very simple shaders (called *depth_v.glsl* and *depth_f.glsl*, already provided).
   - **Render from light (20%).** You need to finish the `paintGL` function of `glstate.cpp`. Your task is calculating `lightSpaceMat`, which is the transform matrix to convert the scene to the light's space. To do that, you may first get `lightProj` and `lightView` matrices and then use them to obtain

`lightSpaceMat`. Then it is passed into *depth_v.glsl* to transform the scene to the light's viewing space.



Before the second pass, the depth map/shadow map is generated as texture and will be used for testing shadows. And this time the scene is rendered from the camera's space.

- **Shadow testing (20%).** The shadow map is given as `shadowMap` in the fragment shader. Your job is to implement the function `calculateShadow` in the shader. In this function, you will first read the closest depth value from the depth map, then get the depth of the current fragment, and compare them to decide whether this fragment is in shadow. The current parameter `light_frag_pos` is the fragment position in light's space. Feel free to add other arguments to the function if you think they are necessary. Finally apply shadow to the diffuse term (shadow should have no effect on the ambient term). You shall see the result as below:



5. **Extra credit (15%). Beyond Bump Mapping.** As extra credit, consider doing one of parallax mapping, steep parallax mapping, parallax occlusion mapping, or other similar enhancement. These enhancements require a depth map, in addition to the normal map. In the textures directory, is a "cube1_depth.png" that was created using the marvels of current neural networks, so treat it with a grain of salt, but it should be good for your use; else you can improvise as well or change the texture altogether, for example.

   For parallax mapping (and other approaches), you basically must ensure the depth map is loaded and then perform additional computations in the fragment shader.

Turn-in:
To give in the assignment, please use Brightspace. Give in

- **a zip file with your complete project (project files, source code, and precompiled executable).**
- **a short video of your program working**

It is your responsibility to make sure the assignment is delivered/dated before it is due.

If you implement the extra credit, please ensure instructions are given on how to use it --- put such in the GUI or have clear instructions printout – do not assume we will read your code and decipher how to use your extra credit.

Good luck!