EECS 351-1: Intro to Computer Graphics   Proj C: Better Lights & Materials

Instructor: Jack Tumblin   **Winter 2015**

Your mission in Project C is to create realistic interactive lighting and materials in WebGL in a 'virtual world.' As before, users 'fly' to explore 3D animated solid objects placed on a patterned 'floor' plane that stretches to the horizon in the x, y directions. Unlike Project B, the objects in this 'virtual world' are made from different materials, each with individually-specified emissive, ambient, diffuse, specular parameters. The world also contains several smoothly-movable user-adjustable light sources, each with individually specified position, ambient, diffuse, and specular parameters. From these, your program computes the Phong lighting model with Phong shading to yield smooth-looking, facet-free surfaces with nicely rounded specular highlights. To do this, your program and its shaders must compute lighting from 3D vectors for each fragment and not just for each vertex.

Requirements: **Project Demo Day (and due date): Wed Mar 11, 2015**

1) In-Class Demo Day: when you will demonstrate your completed program to the class, and two other students will each evaluate it by filling out a 'Grading Sheet' for you. You then have an additional 72 hours to revise and improve your project based on what you learned from Demo Day, before submitting the final version of your project. Your grade will combine grading sheet scores and your improvements in the final version.

Submit your finalized project to CMS/Canvas no more than 72 hours later (Sat Mar 14 11:59PM) to avoid late penalties. Submit just one single compressed folder (ZIP file) that contains:
1) your written project report as a PDF file, and
2) just one sub-folder that holds all your Javascript source code, libraries, HTML, and any support files you made: we must be able to read your report run your program in the Chrome browser with nothing more than the files you sent us.
---IMPORTANT: name of your ZIP file and name of the directory inside: **Familyname**Personalname_ProjC
For example, my project C file would be: TumblinJack_ProjC.zip, and it would contain my report namedTumblinJack_ProjC.pdf, and my project/source-code folder named TumblinJack_ProjC
---To submit your work, use Canvas→Assignments. DO NOT e-mail your project!
---BEWARE! SEVERE LATE PENALTIES! (see Canvas→Syllabus→'Course Details')

Project C consists of:
1)—Report: A short written, illustrated report, submitted as a printable PDF file.
Length: >1 page, and typically <5 pages, but you should decide how much is sufficient.
A complete report consists of these 3 sections:

a)--your name, netID, and a descriptive title for your project
(e.g. Project B: Forest of Trees Fly-through, not just "Project C")

b)—a brief 'User's Guide' that starts with a paragraph that explains your goals, and includes instructions on how to make your on-screen animated images run/stop (animation) and change.
(*e.g*. "A,a,F,f keys to pan your view left/right; S,s,D,d to translate your viewpoint left/right; arrow UP/DN keys to climb/dive; PgUp, PgDn keys to speed up/slow down; slowing down past zero lets you fly in reverse") Your classmates should be able to read ONLY this report and easily run and understand your project without your help!

c)—a brief, illustrated 'Results' section that shows **at least 4 still pictures** of your program in action (use screen captures; no need for video capture), with figure captions and text explanation of what the pictures are showing us, and how they help your project meet your goals.

2)—User Instructions: When your program starts and/or when users press the F1 (help) key, print a brief set of user instructions, either in the console window, the 'canvas' or in the HTML page that contains your canvas (e.g. 'arrow keys move skateboard, left mouse drag steers, right mouse drag aims flame-thrower').

3)---'Ground Plane' Grid: Your program must clearly depict a horizontal 'ground plane' that extends to the horizon: -- a very large, repetitious pattern of repeated lines, triangles, or any other shape that repeats to form a vast, flat, fixed 'floor' of your 3D world. You MUST position your camera in the **x,y plane** (z=0) of your 'world-space' coordinate system. This grid should make any and all camera movements obvious on-screen, and form a reliable 'horizon line' when viewed with a perspective camera. Hint: you can make plausible terrain from your ground plane if you a) assign modestly randomized terrain-like materials at each vertex, and b) displace the vertices with the sum of a few sine-waves at different non-harmonic wavelengths, or by fractal/subdivision methods (See: http://www.gameprogrammer.com/fractal.html) .

4)---At least 3 solid (not wireframe) separately-located, jointed, 3D animated objects with sensible surface normals at each vertex (otherwise your Phong Lighting results won't make sense). These jointed objects must continually and smoothly change their joint angles, without requiring any user input to continue moving. Place each jointed object at a different location on the 'ground plane'. For example, could you make a tree that waves in the wind (from cylinders)?

5)---A single-viewport displayed image that completely fills a user-resizable window. No matter how tall or how wide the window, your code must fill the entire window with the output of a perspective camera whose vertical field of view is always 40 degrees from top edge to bottom edge, and whose aspect ratio matches the display window; your window cannot contain any 'blank' areas, and must not distort any displayed objects: circles must remain circles for any window height and any window width. (Make 'ground plane' object large enough to ensure it always fills the entire display.)

6)---'5-DOF' camera control (tilt up/down, pan left/right, move forwards/backward in direction of gaze, sideways). Your program must create a perspective camera that moves smoothly in response to user inputs, without 'jumps' or jerkiness. You must use **`setLookAt()`** or equivalents to do this, and you must supply the user with 3 user controls that adjust only the world-space position the camera without changing the camera's aiming direction, and another, different set of 2 controls to aim the camera without changing the camera's world-space position.
For example, you could use the left/right arrow keys to turn the camera left/right; PgUp and PgDn keys to tilt camera up/down; up/down arrow keys to move camera forward/backward; shift-up/down keys to move camera sideways. You may keep the 'up' vector at the fixed value (0,0,1), where 'up' is +z, and z==0 at the 'ground' of our groundplane. HINT: to move the camera without changing its aiming direction, increment BOTH the 'eye'point and the 'lookAt' point by exactly the same amounts. To change the camera's aiming direction without moving it, keep the 'eye' point unchanged, and rotate the 'lookAt' point around that 'eye' point, as if moving on the surface of a cylinder centered on the 'eye' point.

7)---Assign obviously different-looking Phong materials descriptions to each object shown on-screen. **E**ach of your 3 (or more) objects must include a materials-describing object; an easy-to-access set of 13 parameters or more that describe its response to light according to the Phong lighting model. The parameters are: 9 floating-point color-reflectance values $0.0 \le R,G,B \le 1.0$, for ambient, diffuse, and specular reflectance (K_a, K_d, K_s); 3 floating-point color-emittance values K_e ($0.0 \le R,G,B \le 1.0$, but usually zero; these are the 'glow-in-the-dark' values for the material), and an integer 'shinyness' coefficient that sets the size of the specular highlight seen on a surface: smaller highlight$\rightarrow$larger shininess component n_{shiny}.

8)---Create at least two point-like, non-directional light sources that will illuminate your objects using the Phong lighting model. Keep one light fixed to the camera position (a 'headlight'), and place the other light in 'world' coordinates, at user-adjustable 3D position. Your program will need to create a light-describing object for each light: an easy-to-access set of parameters that describe how the light source will affect the appearance of nearby materials, including: light-source 3D position coordinates, and Phong light-source strengths ($0.0 \le R,G,B \le 1.0$) for ambient, diffuse, and specular illumination (I_a, I_d, I_s).

--For this project, you may safely ignore the distances between the light source and the surfaces it illuminates; light source position determines only the direction from the light source to a point on a surface, and ***not*** the illumination intensity.
--For this project, you must ALSO allow users to switch on/off each light-source component independently and separately for each light source: switch ambient term on/off, switch diffuse term on/off, switch specular term on/off.

9)---Write your own vertex shader and fragment shader to implement
Phong lighting with Phong Shading.
Combining the Phong lighting model (ambient, diffuse, specular, emissive) with Phong shading dramatically improves the appearance of realism and smooth surfaces in openGL, because the specular highlights are round, move smoothly, and do not have the faceted appearance of Gouraud Shading.
Your Phong shading program must interpolate surface normal, and possibly other vectors needed for lighting. It will supply them to your Fragment Shader via GLSL 'varying' variables, and will use these per-pixel normal values to compute pixel colors. This method dramatically improves the quality of specular highlights; the surface appears smooth and the highlights appear rounded instead of the faceted highlights of Gouraud shading.

10)---EXTRA CREDIT: Texture Mapping. Apply image textures to one or more objects in your scene. Please note that this DOES NOT replace the requirements for Phong-lit materials for this project; you must still meet all of those requirements as well. Note that online, in our Lengyel reading, and in the latter parts of our WebGL book you can find information on 'bump mapping', a commonly used method in modern computer grames to apply image values as displacements to local surface normals. This method greatly boosts the apparent complexity and richness of shapes.

11)—EXTRA CREDIT: GLSL Geometry Distortions.
Adjust all shapes in non-linear ways that no matrix transform can duplicate. You can try at least two different distortion methods: change vertex positions as a function of vertex position; change surface normal as a function of vertex position.
For example, you may 'twist' vertices around an object's own z-axis by applying a different z-axis rotation to each vertex. The shader can accept objects' vertices in their unmodified 'model' or 'local' coordinates, and rotate each one around the z axis by the z-dependent amount (z*twist) degrees before applying your own version of the GL_MODELVIEW matrix.
Or perhaps you want to impart a local 'wavyness' to 3D space? You could make a 3D sinusoidal displacement field: a function of x,y,z that provides a value between -1 and +1 that varies smoothly with position. Evaluate the displacement field at the position of the current vertex, and add it to the vertex's position, like the time-varying 'wavyness' demonstrated in the starter code. But be careful -- YOUR 'wavyness' must be distinctly and noticeably different from the starter-code waviness; no credit if unchanged! Better yet, devise your own local nonlinear spatial distortions.
NOTE: by 'local' geometry, I mean these distortions must be applied in the coordinate system axes of the individual objects, and not in shared 'world', 'eye' or other coordinates. The distortions must not change when you move an object to a different 3D location, and they must not change as you move or aim the camera differently.

Sources & Plagiarism Rules:
You are welcome to use any, all, or none of the book's example code, the 'starter code' I supply or others, from websites, or any other useful resource you can find, but you must credit their work properly—no plagiarism!
Share what you find with other students, too -- list the URLs on CMS/Canvas discussion board, etc.
Get inspired by what you find on the web, but don't copy it. Read it, learn from it, then put it aside.
Then sit down and write your own. TurnItIn checks for plagiarism.
(If I find plagiarism evidence, the University requires me to report it to the Dean of Students for investigation).