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

Instructor: Jack Tumblin **Winter 2016**

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, not just for each vertex.

Requirements: **Project Demo Day (and due date): Mon Mar 07, 2016**

A)-- In-Class Demo: As with Projects A & B, on the due date (Mon Mar 07) you demonstrate your completed program to the class. Two other students each evaluate your work on a 'Grading Sheet', as may Tumblin and assistants. Based on Demo Day advice, you then have ≥72 hours to revise and improve your project before submitting the final version for grading. Your grade will mix Demo Day grading sheets + your improvements.

B) --Submit your finalized project to CMS/Canvas no more than 72 hours later (Thurs Mar 10 11:59PM) to avoid late penalties. Submit just one single compressed folder (ZIP file) that contains:
2) one folder that holds sub-folders with all Javascript source code, libraries, HTML, etc. (mimic the 'starter code' ZIP-file organization). We must be able to read your report & run your program in the Chrome browser by simply uncompressing your ZIP file, and double-clicking an HTML file found inside, in the same directory as your project report.
---**IMPORTANT:** Name your ZIP file and the directory inside as: **Familyname**Personalname_ProjC
For example, my project C file would be: TumblinJack_ProjC.zip. It would contain sub-directories such as 'lib' and files such as TumblinJack_ProjC.pdf (a report), TumblinJack_ProjC.html, TumblinJack_ProjC.js ,etc.
---To submit your work, upload your ZIP file to Canvas→Assignments. DO NOT e-mailed projects (deleted!).
---BEWARE! SEVERE LATE PENALTIES! (see Canvas→Assignments, or the Syllabus/Schedule).

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 C: Jeweled Starfish and Crabs Scuttle on Sparkling Sand, not just "Project C")

b)—a brief 'User's Guide'. Begin with a paragraph that explains your goals, then give user instructions on how to run and control the project. (*e.g.* "A,a,F,f keys rotate outer ring forwards/backwards; S,s,D,d keys rotate inner ring forwards/backwards; HUD text shows velocity in kilometers/hour.") 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 or gifs), with figure captions and text explanations. Your figure(s) also must include a sketch of your program's scene graph
(the 'tree of transformations': unsure? See lecture notes 2016.01.13.DualitySceneGraphs_VanDamm04).

2)—Your Complete WebGL Program, which must include:
a)---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').

b)---'Ground Plane' Surface: 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. The ground must span the **x,y plane** (z= ~0) of your 'world-space' coordinate system; *do not* use +y' as 'up'; use +z! This ground 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 by +/-z with the sum of a few 2D sine-waves at different non-harmonic wavelengths, or by fractal/subdivision methods (See: http://www.gameprogrammer.com/fractal.html) .

c)---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. For example, could you make a tree that waves in the wind (from cylinders)?P lace each jointed object at a different location on the 'ground plane'. You may re-use shapes from Project A & B, but 'shade' them.

d)---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. While you may add a fixed-height window region to hold HTML buttons & text, otherwise your window cannot contain any variable-sized 'blank' areas, and re-sizing the window must not distort any displayed objects: circles must remain circles for any window height and any window width. (Be sure you make the 'ground plane' surface large enough to ensure that it always fills the entire browser window).

e)---'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 ground plane. HINT: use the 'glass cylinder' method of Project B.

f)---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→larger shininess component $n_{shiny.}$

g)---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.

h)---Write your own vertex shaders and fragment shaders to implement 4 selectable shading methods. Your program must allow users to switch between these lighting and shading interactively (via keyboard, mouse, or HTML buttons, etc.), without stopping or disrupting the program or its on-screen display. User must be able to select between at least these 4 methods:

a) Phong lighting with Phong Shading, (no half-angles; uses true reflection angle)
b) Blinn-Phong lighting with Phong Shading (requires 'half-angle', not reflection angle)
c) Phong lighting with Gouraud Shading (computes colors per vertex; interpolates color only)
d) Blinn-Phong lighting with Gouraud Shading.

Combining the Phong lighting model (ambient, diffuse, specular, emissive) with Phong or Blinn-Phong shading dramatically improves the appearance of realism and smooth surfaces in OpenGL/WebGL, because the specular highlights are round, move smoothly, and do not have the faceted appearance of Gouraud Shading. By selecting between those methods using an interactive 'uniform' variable, users can change shading/lighting as their program runs and see exactly how they differ on-screen.
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.

j)—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 model and view 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. 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.

k)---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. You can also find information on render-buffer and texture buffer operations such as 'render-to-texture' that permit you to render mirrors that show other views of the scene.

Sources & Plagiarism Rules:
Simple: *never* submit the work of others as your own. You are welcome to begin with the book's example code and the 'starter code' I supply; you can keep or modify any of it as you wish without citing its source. I strongly encourage you to always start with a basic graphics program (hence 'starter code') that already works correctly, and incrementally improve it; test, correct, and save a new version at each step. Also, please learn

from websites, tutorials and friends anywhere (e.g. .gitHub, openGL.org, etc), please share what you find on Canvas Discussion board (but NEVER post code you will turn in for grading—only ideas or examples), but you must always properly credit the works of others in your graded work—**no plagiarism!**

Plagiarism rules for writing essays apply equally well to writing software. You would never cut-and-paste paragraphs or whole sentences written by others and submit it as your own writing: and the same is true for whole functions, blocks and statements. Never try to disguise it by rearrangement and renaming (TurnItIn won't be fooled). Instead, study good code to grasp its best ideas, learn them, and make your own version in your own style. Take the ideas alone, not the code: make sure your comments properly name your sources. (And, Ugh, if I find plagiarism evidence, the University requires me to report it to the Dean of Students for investigation).