COMP_SCI 351-1: **Project C: Better Lights & Materials**
Instructor: Jack Tumblin **Fall 2022**

GOALS:

A) Your goal in Project C is to create good interactive lighting and materials in WebGL in a visually interesting animated interactive ‘virtual world.’

B) As before, users can move and explore 3D animated assemblies of rigid 3D parts pivoting on hinges and placed on a ground- plane grid or floor pattern that stretches to the horizon in the x, y directions. Unlike Projects A and B, this ‘virtual world’ will no longer use cartoonish fixed colors at each vertex. You will instead compute colors in your vertex and fragment shader programs by simulating how a light source illuminates an oriented surface whose material reflects some fraction of that light towards the camera.

C) Your rigid 3D parts now *MUST* include a surface-normal attribute for each vertex, and also a material descriptor that chooses a set of RGB reflectance values. Your GLSL shader programs will combine these reflectance values with RGB illumination values from 3D light sources in the virtual world to compute colors displayed on-screen.

D) Your program wmust use several vertex/fragment shader pairs (instead of one giant shader pair cluttered with conditionals) to compute the Phong and Blinn-Phong lighting model with both
   a. ‘Gouraud’ shading (yields faceted appearance) and
   b. ‘Phong’ shading (for smooth-looking, facet-free surfaces with nicely rounded specular highlights).

Requirements:

Project Demo Day (and due date): Wed Nov 30, 2022

A)-- In-Class Demo: As with Projects A & B, on the due date (Wed Nov 30) you will demonstrate your completed program to everyone in the class, and to see and discuss everyone else’s project too! You will then have several days to apply what you learned to create your final version to turn in for grading.
As a courtesy, please wear a face-mask for Demo Day (as we just resumed ‘in-person’ demos after 2 years on ZOOM) because you will confer closely with every other student in the class: that’s thousands of virus-transfer opportunities in 1 hour! You will show your own work for 15-minutes, then view others for 30 minutes.

B) --Submit your finalized project to CMS/Canvas no later than (Mon Dec 5, 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) 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 our browsers by simply uncompressing your ZIP file, and double-clicking an HTML file found inside, in the same directory as your project report. Include the ‘lib’ directory, and test your ZIP file’s contents before submission.

---**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.
--**IMPORTANT:** Use only POSIX-compliant chars(LINK) in all file names and all directory names, please! No spaces, no commas, no carets, etc. -- only letters(a-z, A-Z) digits(0-9), period(.), underscore(_) or hyphen(-).
---To submit your work, upload your ZIP file to Canvas→Assignments. DO NOT e-mail! (rejects executables).
---BEWARE! Late penalties can add up quickly! (see Canvas→Assignments, or the Syllabus/Schedule).

Project C consists of:

1) -- Report: A short written, illustrated report, submitted as a printable PDF file as part of your final version (not needed for Demo Day). Length: >1 page, and typically <5 pages, but you should decide how much is sufficient. A complete report consists of these parts:

A)--your name, netID, and a descriptive title for your project
(e.g. “Project C: Jeweled Crabs Scuttle over Sparkly Sand,” not just “My Proj 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.* “Mouse-drags change camera aiming (pitch,yaw) while WASD keys move fwd/back and strafe left/right; Q/E keys strafe vertically +/-z.”) 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 correctly-drawn sketch of your program’s **scene graph (required!)** (‘tree of transformations’: unsure? lecture notes, 2022.05.VectorMathPart2_DualitySceneGraphs_VanDamm. Remember:

- root node is always the CVV (a group node; an oval);
- transform nodes (rectangle) always have only 1 parent and only 1 child node, (insert ‘group’ nodes if you need more children).

- a set of vertices for one 3D ‘**part**’ is always a leaf node (triangle); ***no*** children (none!).
- *Only* group nodes can have multiple children – no others can (not transforms, not parts)!

2)—Your Complete WebGL Program, which must include:

a)---User Instructions: When your program runs, it must explain itself to users. How? You decide! Perhaps print a brief set of user instructions below the HTML-5 Canvas object? Or print ‘press F1 for help’? Create a pop-up window? Perhaps within the ‘canvas’ element using the ‘HUD’ method in the book, or in the JavaScript ‘console’ window (in Google ‘Chrome’ browser), etc. Your program should never puzzle its users, or require your presence to explain, find, or use any of its features.

b)---‘Ground Grid’ Surface: Your program must clearly depict a ‘ground plane’ that extends to the horizon: a very large, repetitious pattern that forms a ‘floor’ to your 3D world that helps users sense camera aiming and movement. You can use repeated crossed lines, a 2D (or 3D) pattern of triangles, or any other shape that repeats to form a vast, flat or mostly-flat floor. You MUST set your grid in the **x,y plane** (z=0) of your ‘world-space’ coordinate system; *do not* use +y’ as ‘up’! **HINT:** you can make plausible terrain from a ground plane made of triangles 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: **https://en.wikipedia.org/wiki/Fractal_landscape**).

c)---At least 3 solid (not wireframe) separately-located, jointed, 3D animated assemblies with sensible surface normals at each vertex (otherwise your lighting results will look strange/wrong). Assemblies must change their joint angles continually and smoothly, without requiring any user input to continue moving. For example, could you make a tree that waves in the wind (from cylinders)? Place each jointed assembly at a different location on the ‘ground plane’. You may re-use Project A & B shapes, but you must ‘light’ them, which will require each vertex of each 3D part to include its own, separately-specified surface-normal vector attribute.

d)---Scene must contain a large, slowly-spinning sphere at the world-space origin (0,0,0) location that we can view and light from any direction to let users visually confirm that all forms of shading (e.g. Phong, Gouraud, flat…) and lighting work correctly. (e.g. Phong, Blinn-Phong, Cook-Torrance, etc.)

e)---One Viewport in re-sizeable Webpage: Your program must depict its 3-D scene using a perspective camera with a 30-degree vertical field-of-view, whose viewport and HTML-5 canvas element fills all the width of your browser window and the top 2/3rds (66%) of its height. Browser window re-sizing to any height or width should never create scroll-bars, empty gaps above or beside the canvas, or any image distortions (stretch or squash).
HINT: to avoid 'scroll bars', create a ~16-20 pixel border around the canvas; see 'resize' starter code.

f) —View Control: smoothly & independently control 3D Camera positions and aiming direction. (Same as Project B) Your code must enable users to explore the 3D scene via user interaction. I recommend that you use arrow keys, W/A/S/D, mouse-dragging, or other widely-used key combinations to steer and move through the scene. You may design and use your own camera-movement system, but for full credit your system must allow complete 3D freedom of movement:

1. at any 3D location, your camera *MUST* be able to smoothly pivot its viewing direction without any change in 3D position (if you pretend that your head is the camera, you must be able to turn your head without moving your body), and:
2. your camera *MUST* be able to move from its current 3D location to any other 3D location in a straight line, moving in the camera's aiming direction. During that travel, the camera aiming direction MUST NOT CHANGE (will not rotate towards origin, etc.)
   You *MUST NOT* require users to adjust controls that move the camera eye-point in world coord. x, y, z directions only, and/or adjust the camera aim-point in world coord. x,y,z directions, as found in our textbook's starter code.

HINT: Make global variables for eye-point, up-vector, the compass-heading angle 'theta', and z-axis difference 'deltaZ' between camera and aim-point; compute the aim-point as needed from these stored values.

g)---Assign obviously different-looking Phong materials to at least 3 rigid parts shown on-screen. Each of these 3 parts must use (or select) a different visually-distinct material, an easy-to-access set of **ALL 13** parameters or more that describes material response to light in 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 ($0.0 \le R,G,B \le 1.0$ (often zero, as few materials 'glow'), and
- the 'shinyness' coefficient n_{shiny} that sets the size of the specular highlight seen on a surface. Remember: smaller highlight→larger shininess component n_{shiny}. (Starter code helps!)

h)---Create at least one non-directional light source that will illuminate your assemblies using the Phong lighting model. Set position of this light in 'world' coordinates, at a user-adjustable 3D position (OPTIONAL: keep a second light fixed to the camera position (a 'headlight'). To earn credit for more than one light source, your program must be able to 'turn on' all lights at the same time. As shown in starter code (`2022.11.18.Phong+Structs.zip`) use a 'struct' data type in GLSL to hold all parameters that describe materials (see `materials_Ayerdi04.js`), and another 'struct' data type to hold all parameters for a light source. The light source struct should include the 3D position of the light source plus RGB values for ambient, diffuse, and specular illumination values (I_a, I_d, I_s).

--Each light source should allow users to switch on/off each light-source component independently and separately (*e.g.* ambient light on/off, diffuse light on/off, specular light on/off); this aids in debugging.

--For this project, you may safely ignore light attenuation by the distances between the light source and each illuminated surface; light source position determines only the direction from the light source to a point on a

surface, and ***not*** the incident illumination intensity.
NOTE: If you implement distance dependent attenuation, please give users easy ways to adjust or disable it!

j)---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 methods interactively (via keyboard, mouse, or HTML buttons, etc.), without stopping or disrupting the program or it's on-screen display.   Users 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 (computes colors per vertex; interpolates color only)

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

Note that your GLSL Phong *shading* program and GLSL Gouraud *shading* program will differ substantially.
I strongly recommend that you write them as separate GLSL shaders (*e.g.* use a different VBObox for each).

For Gouraud shading, the vertex shader does most of the lighting calculations.
The vertex shader determines color at each vertex, and sends it as 'varying' color to the fragment shader for display on-screen. These 'per-vertex' shading calculations are simpler, but don't look good – specular highlights appear faceted and rough.

For Phong shading, the fragment shader does most of the lighting calculations.
The fragment shader receives interpolated vectors (surface normal N, light vector L, view vector V, *etc*.) as 'varying' values sent from the vertex shader, and 'uniform' values such as light-source values (Ia, Id, Is) and materials reflectances (Ka, Kd, Ks, Se). The fragment shader must re-normalize any interpolated unit-length vectors, and then compute the Phong lighting using the interpolated 'varying' vectors it receives.
These 'per-pixel' shading calculations are more complex and time-consuming, but they dramatically improve the on-screen appearance. Phong-shaded surfaces appear smooth with nicely-rounded highlights instead of the faceted highlights of Gouraud shading.

k)—EXTRA CREDIT: GLSL Geometry Distortions applied to just one 3D part or assembly.
Nonlinear shape adjustments in Vertex Shader (possibly animated) that no matrix transform can duplicate.
HINT: note similarity to 'Activity 4: GLSL Essentials.'
You already know several different distortion methods: change vertex positions as a function of vertex position; change surface normal as a function of vertex position; make at least one of them time-varying. For example, you may 'twist' vertices around an objects's own z-axis by applying a different z-axis rotation to each vertex. The shader receives 3D part vertices in their unmodified 'model' or 'local' coordinates; you would then rotate each one around the z axis by the z-dependent amount (z*twist) degrees before applying the model, view, & projection matrices.
Better yet, devise your own local, time-varying nonlinear spatial distortions!
NOTE: Use 'local' geometry; apply distortions in the coordinate system axes of the individual 3D part or assembly, and not in the shared 'world', 'eye' or other coordinates. This ensures the distortions will not change with an assembly's joint angles, nor change with different 3D locations of an assembly.

l)---EXTRA CREDIT: Texture Mapping.
Apply image textures in a visually obvious way to one or more 3D parts 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.
You can also find information on 'bump mapping' online, in our Lengyel reading, and in the latter parts of our WebGL book. Bump-mapping is commonly used method in modern computer games to apply image values as

directional offsets to local surface normals, yielding rich visual complexity to rigid 3D parts without a high vertex count. 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. Try it!

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.

I ***want*** you to explore -- to learn from websites, tutorials and friends anywhere (e.g. GitHub, StackOverflow, MDN, CodeAcademy, OpenGL.org, etc), and to apply what you learn in your Projects. Please share what you find with other students, too -- list the URLs on CMS/Canvas discussion board, etc. and list in the comments the sources of ideas that helped you write your code.

BUT always, ALWAYS credit the works of others— ***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.
Take their good ideas, but not their code add a gracious comment that recommends the inspiring source of those good ideas, and then write your own, better code in your own better style; stay compact, yet complete, create an easy-to-read, easy-to-understand style.

Don’t waste time trying to disguise plagiarized code by rearrangement and renaming (MOSS 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.

Also, please note that I apply the ‘MOSS’ system from Stanford (https://theory.stanford.edu/~aiken/moss/) and if I find any plagiarism evidence (sigh), the University requires me to report it to the Dean of Students for investigation. It’s a defeat for all involved: when they find misconduct they’re very strict and very punitive. It’s happened before; don’t let it happen again – it’s a hugely tragic waste for everyone involved!