EECS 351-1: Intro to Computer Graphics — Project B: 3D Views & Shading

Instructor: Jack Tumblin — **Winter 2015**

CHANGES FROM VERSION 5 HIGHLIGHTED IN BLUE

Your mission in Project B is to fill the Canonical View Volume (CVV) of WebGL with the view seen by a 3D camera that users fly like an airplane, free to turn, dive, climb and go anywhere to explore a gigantic 3D 'virtual world'. Your program will automatically re-size its 3D graphics to fill the full width of your browser window, to show two re-sized camera views side-by-side (unlike the top/bottom starter code program 7.11.JTHelloCube_Resize.html), with an orthographic view on the right, and perspective view on the left. The 3D world you explore will have patterned, grid-like 'floor' plane that stretches out to the horizon in the x,y directions (+z points 'up' to the sky). Arranged on this vast floor you will place several animated, jointed solid objects (not wireframe) that you can explore by 'flying' around, between and behind them. Each vertex of each object must include its own individually-specified surface normal attributes, and these enable your shaders to compute a very basic overhead-lighting effect for diffuse/Lambertian materials.

You may build on your Project A results, or make a wholly new program. As with ProjectA, depict something *you* find interesting, meaningful and/or compelling, and use any and all inspiration sources. How about some clockwork gears? An interactive NxN Rubik's cube (default is 3x3)? A steerable butterfly that flies a random path in 3D by flapping its wings, or a mechanical ornithopter? A helicopter with spinning rotors? A forest scene made from waving fractal/graftal/L-system trees and bushes (a 'tree of transformations' you can see)? Scattered wheeled vehicles, legged animals or machines, a trapeze or perhaps a unicycle?

Requirements: **Project Demo Day (and due date): Mon Feb 23, 2015**

A)-- In-Class Demo Day: you will demonstrate your completed program to the class, and to two other students that evaluate it on a 'Grading Sheet' for you, as will Professor 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 project grade combines your grading sheet scores and all your improvements since Demo Day.

B)-- Submit your finalized project to CMS/Canvas no more than 72 hours later (Thurs Feb 26 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 all support files. We must be able to read your report and run your program in the Chrome browser with only these files you sent us.

---**IMPORTANT:** name of your ZIP file and name of the directory inside: **Familyname**Personalname_ProjB

For example, my project B file would be: TumblinJack_ProjB.zip, and it would contain my report namedTumblinJack_ProjC.pdf, and my project/source-code folder named TumblinJack_ProjB

---To submit your work, upload your ZIP file to Canvas→Assignments. DO NOT e-mail your project!

---BEWARE! SEVERE LATE PENALTIES! (see Canvas→Syllabus→'Course Details')

Project B 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 (3 letters, 3 numbers: my netID is jet861), and a descriptive title for your project (e.g. Project B: Forest of Trees Fly-through, not just "Project B")

b)—a brief 'User's Guide' that starts with a paragraph that explains your goals, and includes complete instructions on how to run and control the program (*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 to fly in backwards") 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 explanations. Your figure(s) also must include a sketch of your program's scene graph (the 'tree of transformations': unsure? See lecture notes 2015.01.14.2008VanDamm03TransformTrees).

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

a)—User Instructions: When your program runs and/or when users press the F1 (help) key, print a brief set of user instructions onscreen somewhere. How? You decide! Perhaps put them in the webpage outside of the 'canvas' element, or using the 'HUD' method in the book, or in the browser's JavaScript 'console' window (in Google 'Chrome' browser), etc. Your program must not puzzle its users, or require your presence to explain, find or use any of its features.

b)---'Ground Plane' Grid: Your program must clearly depict a '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.

c)—Animated, adjustable, 3-Jointed, 4-segment 3D Shape: Your code must show **at least one smoothly-animated jointed 3D object** with at least four (4) sequentially-jointed segments connected by three(3) sequential joints at different locations (one MORE sequential joint than required for Project A). Enable users adjust those joint angles smoothly by interactions with the mouse and/or keyboard.

Remember, sequential joints require you to 'push' and 'pop' matrices from your gl.modelView matrix stack as you draw the jointed object. Each and every joint in your jointed objects must rotate, yet stay pinned together as if connected by hinges. In a scene-graph, this means you have at least three sequential transform nodes; one node is a 'child' of the other node, which is a child of the 3rd, and you draw a displaced part before and after you visit each node. Each of my arms will satisfy this requirement: my torso (part 1) attaches to my displaced upper arm (part 2) through my hinge-like shoulder (joint 1); my upper arm then attaches to my displaced lower arm (part 3) through a hinge-like elbow (joint 2); my lower arm then attached to my displaced hand (part 4) through my hinge-like wrist joint. Moving my torso moves all the sequentially attached parts. Adjusting my shoulder affects all the subsequent arm parts. Conversely, I cannot not satisfy this requirement with a stick-legged starfish. If made of a pentagonal body and 5 hinged but joint-free single-segment arms, it has no sequential joints. Adjusting one arm joint has no effect on any other part. Its scene graph holds a body transform followed by 5 children, one for each arm-angle transform, and no arm is the child of another arm.

d)—Additional Separate Multi-colored Objects: 4 or more. 'Separate' means individually positioned, spatially separate, distinct, different-shaped objects. The top parts of a robot and the bottom parts together make up just one object, as you wouldn't move them to opposite sides of the screen! They don't have to move, but they do have to be distinct and fundamentally different, unrelated, spatially-separated various objects. 'Multi-color' means an object uses at least 3 different vertex colors, and WebGL must blend between these vertex colors to make smoothly varying pixel colors.

e)—Show 3D World Axes, and some 3D Model Axes: Draw one set fixed at the origin of 'world space' coordinates, and at least two others to show other coordinate systems within your jointed object. I recommend that you create a 'drawAxes()' function that draws a 3 unit-length lines: bright red for x axis, bright green for y axis, bright blue for z axis.

f)--Demonstrate simple diffuse overhead shading on at least one moving/rotating 3D shape. Each vertex of the 3D shape must contain attributes for both 'color' and 'surface-normal' vectors. Set these 'normal' vectors to unit-length, aimed outwards and perpendicular to the shape at each vertex. (HINT: sphere normals are easy to compute). Then to compute the color of each vertex, your vertex shader must: 1) compute the dot-product of the normal vector and the upwards (+z) vector in the 'world' coordinate system, 2) create its non-negative 'clamped' result (if <0, set to zero; see GLSL-ES clamp() function) and 3) compute the vertex on-screen as: 'color'*(0.3 + 0.7*clamped_result). As it moves and rotates, the bottom side of your object should always look dim (0.3*color), and the top should always look bright for any orientation.

CAUTION! Don't use explicitly-defined lights and non-diffuse materials, etc. in Project B.
Why? Because we're saving them for projects C.

g)—Two Side-by-Side Viewports in a Re-sizeable Webpage: Your program must depict its 3-D scene twice, in two side-by-side viewport that together fill the entire width of your browser window, without gaps and without any distortion of the images within, even after window re-sizing makes the webpage taller or wider (e.g. an 'anamorphic'combination of viewport and camera settings). The left viewport will show a 3D perspective image; the right will show an orthographic image, each from the same camera location and viewing direction.

h)—View Control: smoothly & independently control 3D Camera positions and aiming direction. Both, together! Your code must enable users to explore the 3D scene via user interaction. I recommend that you use mouse dragging and arrow keys 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:

-- your camera MUST be able to move to any 3D location from any other 3D location in a straight line: do not require users to move in only the x,y,z directions, or only in circles of varying radius. Do not require users to keep the camera aimed at the world-space origin or some other fixed point as they move.

-- from any 3D location, your camera MUST be able to smoothly pivot its viewing direction without any change in 3D position (if you pretend that you are the camera, you must be able to turn your head without moving your body).

-- from any orientation, your camera MUST be able to move smoothly in a straight line to any other desired 3D location without changing its orientation (if you pretend that you are the camera, you must be able to move your body in any desired direction without turning your head).

For example: imagine a scene of 64 colorful cubes placed in a 4x4x4 grid above the 'ground plane' to form a city of floating buildings and flying cars (e.g. **http://youtu.be/IJhlD6q71YA?t=23s**). Your camera must be able to show the city from the viewpoint of any of those cars as they drive in, around, and through the 3D grid of buildings; you must invent a 3D Google StreetView-like camera. If your system cannot easily position the camera to 'drive around the block' in 3D, if your system always aims the camera at the origin, or if it can only move the camera in a circle around the origin, it does not meet the project requirements.

BIG BIG HINTS: If you use **`LookAt()`** to create your 'view' matrix, your user controls must modify BOTH the camera position (VRP or 'eye') AND the target point or 'look-at' point, and vary them independently. In class we described the 'glass-cylinder' model for camera movement that easily achieves these goals.

3)—Note all the opportunities for extra credit by adding more features to your project; see Grading Sheet.

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, testing and correcting at each step. Don't make massive untested changes all at once; you may never find all the flaws hidden in all the new code, and it's never easy to test it thoroughly. Instead, break down your big changes into small easy-to-test steps; if one small step ruins the program, search only your small set of most-recent changes and you will always find the fatal flaws. Also, please learn from websites, tutorials and friends anywhere (e.g. .gitHub, openGL.org, etc), but you must always properly credit the works of others—**no plagiarism!**

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 that helped you write your code.

Plagiarism rules for writing essays apply equally well to writing software. You would never cut-and-paste paragraphs or whole sentences written by others unless it's a clearly marked quote with cited sources. The same is true for whole functions, blocks and statements written by others. Never cut-and-paste others' code without crediting them, and never try to disguise it by rearrangement and renaming (TurnItIn won't be fooled). Instead, write your own code to learn the principles well and to earn your own grades: study their good code carefully, learn its best ideas and methods, and then close the book or website. Take the good ideas, but not the code: add gracious comments recommending the inspiring source of the ideas, and then write your own, better code in your own better style.

Also, please note that I use the 'TurnItIn' anti-plagiarism software:
https://canvas.northwestern.edu/courses/1580/pages/turnitin-in-canvas
(If I find plagiarism evidence, the University requires me to report it to the Dean of Students for investigation).