Comp Sci 351-1: Intro to Computer Graphics **Project B: 3D Views**

Instructor: Jack Tumblin **Fall 2021**

GOALS:

A) Create a large, animated 3D 'world' that users view and explore with an interactive movable 3D camera. One simple set of GUI controls (keyboard, possibly mouse) will aim the camera in any direction by adjusting compass-heading (rotate left/right) and the camera's up/down rotation or 'tilt'. Another set of GUI controls (probably arrow keys or WASD keys or similar set of 4) move the camera the forward or backward in the camera's aiming direction, and 'strafe' horizontally, moving side-to-side without changing the camera's aiming direction.

B) Your program will automatically re-size its HTML canvas object to fill the **full width** of your browser window and at exactly (70%) of its height. The 'canvas' object will show two camera images side-by-side; the right half will show the camera's image made with an orthographic projection matrix or 'lens' (use Matrix4 'ortho()' function), and the left half will show the camera's image made with a perspective projection matrix or 'lens' (use the Matrix4 'perspective()' or 'frustum()' function).

C) The 3D world you explore will have patterned, grid-like 'floor' plane that stretches out to the horizon in the x,y directions. I require that **world-space +z points 'up'** to the sky, ***unlike*** the book starter code 7.07b.JT.LookAtScene….html). Arranged on this vast floor, you will place several animated, jointed assemblies made of solid 3D parts (not wireframe; not lines) that you can explore by interactively moving the camera among them.

You may build Project B [illegible] make an entirely new program. As with Project A, I wan[illegible] eaningful and/or compelling, and use any and all inspir[illegible] interactive NxN Rubik's cube (default is 3x3)? A steer[illegible] ping its wings, or a mechanical ornithopter? A helicopter with spinning rotors? A forest [illegible] g fractal/graftal/L-system trees and bushes (a 'tree of transformations' made visible[illegible] hicles, legged animals or machines, a trapeze, or perhaps a 3-wheeled car?

Requirements: **Project Demo Day (and due date): Wed Nov 10, 2021**

A)-- In-Class Demo: on the Project's due date (Wed Nov 10) you will demonstrate your completed program to the class. At least two other students may evaluate your work (perhaps on a 'Grading Sheet'), as may Professor Tumblin and assistants. You then have a few days to revise your project to apply what you learned on Demo Day. This year's hybrid in-person/remote class will try something different: we will use a large number of ZOOM breakout rooms that students can visit to see students' work demonstrated. You will have a 15 minute session to demonstrate your work to others, and 30 minutes (2 sessions) to visit and see the works of others.

You will then have several days to revise your project to apply what you learned from the Demo Day discussions to create your final version you will turn in for grading.

B) -- Submit a new demo video and your final project code to CMS/Canvas by 11:59PM Mon Nov 15) to avoid late penalties. Submit your code as 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 the Firefox browser by simply uncompressing your ZIP file, and double-clicking an HTML file found inside, in the same directory as your project report.

C) -- *IMPORTANT:* Name your ZIP file and the directory inside as: **Familyname**Personalname_ProjB
For example, my Project B file would be: TumblinJack_ProjB.zip. It would contain sub-directories such as 'lib' and files such as TumblinJack_ProjA.pdf (a report), TumblinJack_ProjA.html, TumblinJack_ProjA.js ,etc.
---To submit your work, upload your ZIP file to Canvas→Assignments. DO NOT e-mail projects! (ignored!).
---BEWARE! Late penalties can add up quickly! (see Canvas→Assignments, or the Syllabus/Schedule).

Project B 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 (3 letters, ~3 digits: my netID is jet861), and a descriptive title for your project (e.g. "Project B: Flying Through a Forest of Flexing Trees", not just "Project B")

B) a brief 'User's Guide' that explains your goals, and then gives user instructions on how to control the project as it runs. (*e.g.* "WASD keys aim the camera without moving it: A/D keys rotate view left/right; W,S keys tilt up/down. Arrow keys move the camera without rotating it: up/down arrow keys move forwards/backwards in direction-of-gaze; left/right arrow 'strafes' camera left/right at current altitude). HUD text gives speed: km/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), with figure captions and text explanations.
NOTE --You' f your program's scene graph
(the 'tree of tr art2_DualitySceneGraphs…).
Remember:

- root n
- transform nodes always have only 1 p de, (use 'group' nodes if you need more c
- a set of vertices for one '**part**' is alwa hildren (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 browser debug tools), etc. Your program should never puzzle its users, or require your presence to explain, find, or use any of its features.
Your program should never 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 crossed lines, a 2D (or 3D) pattern of triangles, or any other shape that repeats to form a vast, flat or mostly-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, *do not* use +y' as 'up'! This grid should make any and all camera movements obviously visible on-screen, and form a reliable 'horizon line' when viewed with a perspective camera.

c)—Animated, adjustable, assembly made of rigid 3D parts arranged with at least 3 flexing sequential joints. Your code must show **at least one smoothly-animated assembly** that connects at least four (4) sequentially-jointed rigid 3D parts. These three (3, not two!!) sequential joints at different locations (one MORE joint than required for Project A). Animate those joints, and enable users adjust those joint angles smoothly by some sort of user interactions; mouse, webpage controls, keyboard, etc.

As you learned in Project A, a well-designed jointed assembly may require you to 'push' or 'pop' matrices from your model matrix stack just before you draw. In a scene-graph, this means you have at least three sequential transform nodes; one node is a 'descendant' of another node, which in turn is a descendant of a 3rd node, and you will draw a transformed 3D part before and after you visit each of these transform nodes. A robot arm-and-hand satisfies this requirement: torso (part 1) attaches to displaced upper arm (part 2) via hinge-like shoulder (joint 1); the upper arm then attaches to displaced lower arm (part 3) through a hinge-like elbow (joint 2); lower arm then attaches to displaced hand (part 4) through a hinge-like wrist joint (joint 3). Torso movement moves all the sequentially attached parts. Conversely, you will *not* satisfy this requirement with a straight-legged starfish. If made of a pentagonal body and 5 hinged but joint-free single-segment arms, it has 6 parts and 5 joints, but 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 descendant of another arm.

d)—Four (4) or More Additional, Separate, jointed assemblies, also made of rigid 3D parts. 'Separate' means individually positioned, spatially separate, distinct, differently-shaped assemblies that move differently. (For example, the top parts of a robot and the bottom parts together make up just one assembly, as you wouldn't move them to opposite sides of the screen). The assemblies don't have to travel, but they do have to be distinct and fundamentally different, unrelated, animated, obviously moving, and spatially-separated. 'Multi-color' means at least one rigid 3D part contains 1 or more triangles in which each of the 3 vertices have different color attributes, and WebGL must blend between these vertex colors to make smoothly varying pixel colors for the triangle(s).

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. (HINT: see quaternion starter code—it does some R, G, B axis-drawing)

f)—Quaternion-bas **Ground-Plane Grid.** Create and draw a colorful 3D objec an rotate intuitively and interactively by dragging the hould always give sensible, track-ball-like rotation results, exactly as seen in the starter c 2021.10.22.Quaternions→QuaternionStarter→ControlQuater your Project B mouse-drag rotations must also work correctly with your movable 3D ca era position, if users can see the 3D object on-screen, then they should be able to rotat use, and the rotation axis should appear to the user as perpendicular to the mouse-drag direction. Think carefully about 'drawing axes' & camera axes!

g)—Two Side-by-Side Viewports in a Re-sizable Webpage: Your program must depict its 3-D scene twice, in two side-by-side viewports that together fill all the width of your browser window and 70% of its height. Resizing browser window to any height or width should never create scroll-bars, empty on-screen gaps, or any distortion (stretch or squash). You can achieve this with new variables that let you compute the matched viewport and camera settings. The left viewport shows image from a 3D perspective camera; the right shows image from an orthographic camera.

h)—Perspective Camera AND orthographic Camera: Both cameras must use exactly the same eye point, look-at point, up vector, z-near, and z-far values (you may need to experiment a bit to find sensible near and far values that look good on-screen; (1,100)? (1,500)? (1,1000)?). The 'perspective' camera' vertical FOV is fixed at 35^{o} (horizontal FOV depends on browser window size), and the 'orthographic' camera's width and height must match the perspective camera's view-frustum size at the plane where -z = (far-near)/3.

i)—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, arrow keys, W/A/S/D, 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 to any 3D location from any other 3D location in one straight line, WITHOUT changing its viewing direction during travel. You *MUST NOT* require users to adjust controls that move only in world coord. x, y, z directions, or only in circles of varying radius! (I strongly recommend: move forward/back in viewing direction, and 'strafe' left/ right, where 'strafe' means to move horizontally, perpendicular to viewing direction, at fixed height).

   For example: imagine a scene of 64 colorful cubes placed in a 4x4x4 grid above the 'ground plane' for a city of floating buildings and flying cars (see **http://youtu.be/IJhlD6q71YA?t=29s**). However, these streets don't follow the x,y,z directions –the 4x4x4 grid was rotated to place two opposite corners on the z axis: its streets align with vectors (1,1,1) , (-1,1,1), and (1,-1,1), and the cube-of-streets slowly tumbles; it rotates at 30 degrees/hour around an axis whose orientation also changes very slowly). Your camera controls should allow users to 'drive' down those streets easily; to thread themselves through all the streets and around an irregular, branching grid of buildings, without any awkward zig-zagging. Your system DOES NOT meet project requirements if the camera as it moves (e.g. 'orbits' a fixed point) or if it moves the camera location when users change the aiming directions.

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 aim-point or 'look-at' point, and vary them independently. In class we described the 'glass-cylinder' model for camera movement that easily achieves all the Project B goals. If you do this, make global variables for eye-point, up-vector, the horizontal aiming angle 'theta', and just the z-coordinate of the camera's aim-point; compute the aim-point's x,y coordinates from eye-point and theta as needed.

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

Sources & Plagiari

Simple: *never* submit the wor

You are welcome to begin with ; you can keep or modify any of it as you wish without citing its source. I strongly encourage you t graphics program that already works correctly ('starter code'), and incrementally improv a new version at each step.

I ***want*** you to explore -- to learn from websites, tutorial (e.g. GitHub, StackOverflow, MDN, CodeAcademy, OpenGL.org ou 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. (And, ugh, if I find plagiarism evidence, the University requires me to report it to the Dean of Students. Ugly.)

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!