Comp Sci 351-1: Intro to Computer Graphics

Project B: 3D Views

Instructor: Jack Tumblin

Spr 2022

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) will move the camera forward or backward in the camera's aiming direction, and will 'strafe' horizontally, moving side-to-side without changing the camera's aiming direction or height above the ground-plane.

B) Your program will automatically re-size its HTML canvas object to fill the **full width** of your browser window and at exactly two-thirds (66%) 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' (from the 'ortho()' function), and the left half will show the camera's image made with a perspective projection matrix or 'lens' (use either the Matrix4 'perspective()' or 'frustum()' function).

C) The 3D world you explore will have some sort of 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 using your Project A results or any portion of it, or make an entirely new program. As with Project A, I want you to depict something *you* find interesting, meaningful and/or compelling, and use any and all inspiration sources. Perhaps some clockwork gears? A city of giant self-adjusting Rubik's cubes? A steerable butterfly that flies in 3D by flapping its wings (or a mechanical ornithopter or a helicopter with spinning rotors)? A forest scene made from waving fractal/graftal/L-system trees and bushes (a 'tree of transformations' made visible)? Scattered wheeled vehicles, legged animals or machines, a trapeze, or perhaps a 3-wheeled car? A parade of marching robots, cats, dogs, and dinosaurs in formation?

Requirements: **Project Demo Day (and due date): Wed May 11, 2022**

A)-- In-Class Demo: on the Project's due date (Wed May 11) 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 this year's 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 6,480 (=81*80) virus-transfer opportunities! You will show your own work for 15-minutes, then view others for 30 minutes.

B) -- Final Version: Submit your final project code and report on CMS/Canvas **by 11:59PM Mon May 16)** (please 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 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_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_ProjB.pdf (a report), TumblinJack_ProjB.html, TumblinJack_ProjB.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), dots(.), underlines(_), or hyphens(-).
---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 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"). 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'll earn extra credit if you include a correct sketch of your program's scene graph (the 'tree of transforms'. Unsure? See lecture notes VectorMathPart2_DualitySceneGraphs…).
Remember:

- root node is always the CVV (a group node; an oval);
- transform nodes always have only 1 parent and only 1 child node, (use 'group' nodes if you need more children):
- a set of vertices for one 3D '**part**' is always a leaf node, with ***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' to open a pop-up window using CSS? Or 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.

b) 'Ground Plane' Grid: Your program must clearly depict a 'ground plane' that extends to the horizon: a very large, repetitious pattern. 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, fixed 'floor' of your 3D world. You MUST set your grid in the **x,y plane** (z=0) of your 'world-space' coordinate system; *do not* use +y' as 'up'! This grid will help 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 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.

HINT: 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-based Mouse-Drag Rotation of 3D Object placed on Ground-Plane Grid. Create and draw a colorful 3D object positioned on your ground-plane grid that users can rotate intuitively and interactively by dragging the mouse on the HTML-5 canvas. Mouse-dragging should always give sensible, track-ball-like rotation results, exactly as seen in the starter code 2021.10.15.Quaternions→QuaternionStarter→ControlQuaternion. Unlike that code, your Project B mouse-drag rotations must also work correctly with your movable 3D camera. Regardless of camera position, if users can see the 3D object on-screen, then they should be able to rotate it by dragging the mouse, and the rotation axis should appear to the user as perpendicular to the mouse-drag direction. Think carefully about 'drawing axes' & camera axes – you can probably figure it out, and if not, ask about it in class and in Peer Mentor office hours…

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 two-thirds (66%) 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 **HINT:** user resizing will usually change the camera aspect ratio. 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)?), and will give you a sensible 'ortho' camera result. The 'perspective' camera' vertical FOV is fixed at 35º (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 W/A/S/D keys to move your camera, and arrow keys or mouse-dragging to aim your camera, but you are welcome to make other choices. You may design and use your own camera-movement system, but for full credit your system *must* allow sensible, intuitive, and 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 one straight line. During that travel, the camera aiming direction MUST NOT CHANGE (will not rotate towards origin, etc.)
3. 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.

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.

Aim-directed movement is quite intuitive, and quite important; use it!

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 world-space x,y,z directions because the 4x4x4 grid cube-of-streets slowly tumbles around an ever-changing axis. Your camera controls should allow users to 'drive' down those streets easily at any time, without any awkward zig-zagging. Your system DOES NOT meet project requirements if the camera can only move along x,y,z directions, or if it 'orbits' around a separately adjustable aim point, or if it moves the camera location when users change the aiming directions.

BIG HINTS: Use `LookAt()` to create your 'view' matrix. Make global variables for eye-point, up-vector, the horizontal aiming angle 'theta' (or 'compass angle'), and just the z-coordinate of the camera's aim-point. (or use a tilt angle phi). Compute the rest of the aim point as needed.

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 that already works correctly ('starter code'), 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, whatever you can find), 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, code-blocks and statements.

Take their good ideas, but not their code add a gracious comment that recommends and shares the source of these good ideas, and then write your own, better code in your own better style. Stay compact, yet complete; create a clear, 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!