CompSci 351-1: Intro to Computer Graphics **Spr 2022**
Instructor: Jack Tumblin

PROJECT A: Moving, Jointed 3D Assemblies

Goals:

A) **Make Parts**. Build a collection of several obviously-different, visually-distinctive rigid 3D ‘**parts**’ stored sequentially in one VBO on the GPU. Each ‘part’ is a fixed set of colorful vertices that form a closed rigid 3D volume. Your program must make every ‘part’ from a list of vertices it transfers to one VBO in the GPU (just once; don’t send them for each new animation frame!), and then it must draw parts as needed using gl.drawArray()’ commands that specify the starting vertex and vertex count (please don’t use drawElements() yet). Draw each required part using WebGL triangle primitives, but you are welcome to add further decorations using WebGL points and lines primitives if you wish.

B) **Make Assemblies**. Use sequences of 3D transformations, matrix save/retrieve operations (stack push/pop) and drawing calls to link together various ‘3D parts’ into several kinds of animated, jointed 3D ‘**assemblies**’. These assemblies move independently in visually-interesting ways – they might fly, walk, flex, pivot, sway, dance, swim, or more, but all parts of the assembly stay connected together. Each ‘**assembly**’ consists of several ‘**parts**’ connected only by fixed or hinged joints. A ‘fixed’ joint locks one part rigidly to another to form a new rigid part. A ‘hinged’ joint links one fixed point (or fixed axis) on the surface of one 3D part to another fixed point (or fixed axis) on the surface of another 3D part. The only allowable rotations are around that point (or that axis), and hinge points (or axes) are fixed to the 3D parts (*e.g.* a robot’s elbow joint won’t move up and down its upper arm!).
You will build each ‘assembly’ by using a push-down stack of matrices (e.g. ‘modelMatrix’) to construct all the various transformations needed to draw each ‘part’ of each ‘assembly’ as it moves on-screen. Confused? Read through the ‘robot arm’ starter code …

C) **Interactive Animation**: these assemblies and their flexible joints should then move dramatically, smoothly, and continuously without any user input (animation), but they should also respond to users’ mouse & keyboard inputs and GUI controls.

Learn enough WebGL to make an *interactive* drawing that *you* find interesting and compelling.
You may choose to draw ANYTHING with multiple joints: an octopus? Fractal trees that grow and wave in the wind? An N-legged walking creature whose legs consist of 2 or more segments? (Google/Bing: ‘DaintyWalker’, ‘Strandbeest’, ‘Catmull hand video’, ‘Jointed Rigid Assemblies”)? Bicycle? bird? helicopter?

Requirements: **Project Demo Day (and due date): Wed April 20, 2022**

A)-- In-Class Demo: on the Project’s due date (Wed Apr 20) 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 from the Demo Day discussions to create your final version to turn in for grading.
As a courtesy, please wear a face-mask for this year’s Demo Day (our first ‘in-person’ demo day in 2 years!) because you will confer closely with every other student in the class: that’s 8,190 (=91*90) virus-transfer opportunities! You get 15 minute session to show your work, and 30 minutes (2 sessions) to see others’ work.

B) – Final Version: Submit your final project code and report on CMS/Canvas **by 11:59PM Mon April 25)** (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 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. Include the ‘lib’ directory, and test your ZIP file’s contents before submission.

C) -- ***IMPORTANT:*** **Name your ZIP file and the directory** inside as: **Familyname**Personalname_ProjA
For example, my project A file would be: TumblinJack_ProjA.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 A consists of:

1)—Report: A short written, illustrated report, submitted as a printable PDF file as part of your final version (not 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 (3-4 letters, 3-4 digits: my netID is jet861), and a descriptive title for your project (e.g. ‘Project A: Alligators Swim Through Planetary Gears,’ not just ‘Project A’

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.* “A,a,F,f keys rotate outer ring forwards/backwards; S,s,D,d keys rotate inner ring forwards/backwards; webpage shows alligator velocity in km/hr”) Your classmates and I should be able to read ONLY this report and on-screen instructions to 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 on CANVAS…).
Remember:

- root node is always the CVV (a group node; an oval);
- transform nodes (box-shaped) always have only 1 parent and only 1 child node, (use ‘group’ nodes if you need more children):
- a set of vertices for one ‘**part**’ is always a leaf node (triangle), 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 On-screen: 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 enable any of its features.

b)— At least two (2) different 3D ‘parts’ that you designed yourself—no copying! The part you design must enclose a 3D volume, must be more complex than a rectangle or a cube, and must show 12 or more vertices at distinct and different 3D locations that are stored in the Vertex Buffer Object (VBO).
In step d) below, you connect these parts to make moving, jointed assemblies.

c)— Smoothly-varying Per-Vertex Colors. All 3D parts must vary their colors between their vertices, and at least one triangle in each ‘part’ must use 3 obviously-different vertex colors (not just 2!) to show on-screen blending by proper use of ‘varying’ variables in shaders. Every vertex must have its own RGB color attributes in the VBO (see ‘multiple attributes’ in your book; be sure you understand the proper use of ‘stride’ and ‘offset’ as described in Chapter 5 and demonstrated in starter code). Of course, do not use any HTML-5 'canvas' drawing primitives (e.g. context.filledRect()); you must use vertex buffer objects (VBOs) for sets of vertices as demonstrated in WebGL Programming Guide, Chapter 3,4).

d)—At least two (2) different *Kinds* of jointed, moving assemblies, both appearing on-screen at the same time. Each different 'kind' of assembly must be distinct, unrelated, and able to move entirely independently – a 'kind' of butterfly and kind of spider; a 'kind' of ornithopter and a 'kind' of helicopter; but not a 'kind' of robot arm and a 'kind' of robot leg (because both are just parts of the same robot assembly—the legs will never walk away from the arms!). EACH of these 2 (or more) different 'kinds' of assemblies should:

- each require a differently-shaped 'scene graph' to describe its jointed parts (a different sequence of graph nodes; a differently-shaped 'tree of transformations'; a different arrangement of joints), and
- get assembled from at least **3 or more 'parts'**, connected at clearly-different hinge-point locations, and
- move fully independently; one 'kind' of assembly cannot depend on position, size, orientation, etc. of another. CAUTION! Don't use just one 'current angle' variable; make several to enable independent movements require DIFFERENT angles that vary at DIFFERENT rates and reverse direction at DIFFERENT times and/or angles.
- contain **two or more sequential hinge joints.** 'Sequential' means we have a hinge-joint that, when rotated to a new joint angle, moves a rigid 'part' around its hinge point, and that 1st part in turn has another part attached at at different hinge-point at a different location. This 2nd hinge joint connects this 1st moving part to a 2nd also-moving part. Changing the 1st joint-angle will move BOTH parts, but changing only the 2nd joint angle will move only the 2nd part.
- I strongly recommend that you make a separate JS function to draw each 'kind' of assembly (e.g. drawRobot(), drawHelicopter()) using the current contents of your modelMatrix. You can then easily draw many robots and many helicopters anywhere, at any on-screen size, by just setting the modelMatrix and calling the function to draw the assembly. I also recommend that your assembly-drawing functions use hinge-joint angles as arguments (e.g. drawRobot(shoulder, elbow, wrist, fingers)) so that you can easily change angles for every assembly-drawing made.
- **HINT**: Remember, a well-designed jointed assembly will always require you to 'push' and 'pop' matrices from your model matrix stack a few times as you draw it, as demonstrated in the Week 3 'Stretched Robot' starter code.
  'Two sequential joints' means that your scene graph has at least one group node that is the child of the 1st hinge-transform node, and that group node must have at least two child nodes: one that draws the movable 1st part, and another whose descendants will include the 2nd hinge-transform, which rotates around a point fixed elsewhere on the 1st part. That 2nd transform node then has as one of its descendants a node to draw the 2nd part.

  For example, 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 stick-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 angle has no effect on any other part. Its scene graph holds a body transform followed by a group node with 5 children; one for each arm-angle transform, and no arm part is the descendant of another arm part.

Your JavaScript program **must** create modelMatrix-like concatenations of 4x4 matrices to transform all the vertices of your parts to position, scale, and orient/rotate them in pleasing ways. Construct your 'model' matrix in Javascript using the **`cuon-matrix-quat03.js`** or **`glMatrix.js`** library, then transfer its values to the GPU as a 'uniform' variable where your vertex shader applies it to the contents of vertex-buffer objects (VBOs).

e)—Interactive Animation: Like all projects in this course, your program must show a picture in a browser that moves and changes, both by itself (animation) and in response to user inputs (interaction) from mouse or keyboard. Users must be able to pause/unpause the animation, and add/remove/modify easily-visible movements from user controls. Controls may be keyboard, mouse, HTML buttons/sliders/edit boxes, etc.

Test your code to ensure ALL your user controls are visually obvious – if I can't see the effect easily after ~3 repetitions (repeated key-press or mouse-click) or modest mouse-drag, then that control is not suitably 'obvious' and 'usable' – and thus it might get overlooked for suitable credit during grading.

d)—Smooth movement only: As your assemblies move due to animation and/or user inputs they must travel smoothly, continuously; animations shouldn't have any large jerky 'jumps' from one pose to another.

c)—Event Handlers: your program and its shaders should make proper use of registered event handlers for **keyboard, mouse and display**. You have many choices here, including the simple methods demonstrated in Chapter 3 and in the 'starter code' posted (e.g. ControlMulti and the GUImess zip file).
Event handlers let your programs respond to the mouse, respond to changes in the display window size, respond to keyboard inputs, and more. You are also welcome to use better, external libraries for user-interfaces in HTML/JavaScript, such as basic CSS controls or Google's dat.GUI: https://github.com/Pixelshaped/dat-gui
(look online for dat.gui tutorials too!)

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; 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. MDN, GitHub, StackOverflow, learnWebGL, 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 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 source of good ideas that inspired you, and then write your own, better code in your own better style; write tall code; 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 and 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!