# CPSC 314, Programming Project 1: Star-Nosed Mole

**Out: Wed 20 Jan 2016. Due: Tue 2 Feb 2016, 23:59. Value: 8% of final grade**

In this project, you will create a star-nosed mole. There are three required parts (100 points), and up to 5 extra credit points can be earned. The best work will be posted on the course web site in the Hall of Fame, and shown in class.
Read through this entire writeup carefully before starting, there are many hints in the Suggested Strategy section!

**Modelling**: [44 points total] Model a mole out of transformed cubes, using only 4x4 matrices to position and deform them.

- (44 pts) Model your mole out of transformed cubes, using the provided drawCube() function that draws a unit cube. You should not include any additional WebGL *geometry* commands in your program, but you may call WebGL *transformation* commands anywhere. Remember that you can do nonuniform scaling to create long skinny boxes.

  Orient your mole so that you see a side view from the default camera position. Create your mole using a hierarchical scene graph structure. That is, instead of just using an absolute transformation from the world origin for each individual part, you should use a relative transformation between an object and its parent in the hierarchy. For example, the head should be placed relative to the body coordinate system and the nose should be placed relative to the head. This hierarchical structure is absolutely critical in the long run for both modelling and animation, even if it might seem like extra work at first! Your beast should have at least the following parts:

  - (-) Body (provided)
  - (4 pts) Head, Nose
  - (22 pts) 22 Nose Tentacles (11 on each side, 9 large and 2 small)
  - (16 pts) 4 Limbs: each with 1 Paw, 5 Claws (4 pts per limb)
  - (2 pts) Tail
  - (extra credit: up to 2 pts) Add color, and/or add more geometric detail to your beast. And/or make an interesting environment for your beast, again using only cubes and 4x4 matrices.

**Animation**: [47 points total] Animate the joints of your beast. Specifically

- (2 pts) Body tilt up ('u'). The whole body should tilt upwards, with all body parts of course moving along with it. When the key is hit again, tilt back downwards return to the flat position. (The first half of this motion is provided within the template.)
- (2 pts) Body tilt down ('d'). As above, but moving down not up.
- (-) Head move right ('h'). The tail should rotate to the right, pivoting at the place where the tail meets the body. When the key is hit again, the tail returns to pointing straight back from the body.
- (2 pts) Head move left ('g'). As above, but moving left rather than right.
- (2 pts) Tail move right ('t'). The tail should rotate to the right, pivoting at the place where the tail meets the body. When the key is hit again, the tail returns to pointing straight back from the body.
- (2 pts) Tail move left ('v'). As above, but moving left rather than right.
- (11 pts) Nose tentacles fan out ('n'). The tentacles should pivot outwards radially from where they intersect the nose at their base. When the key is hit again, they rotate back in to their original position.
- (14 pts) Swim ('s'). When swimming, the star-nosed mole paddles first with one front foot, then the other, extending an opposing foot synchronously.[1] The tail and head also move along with the limbs. When the key is first hit, the front left paw should move forward along with the back right paw, the tail should move all the way to the left, the head should move sideways to the right, and the nose tentacles should fan out. When the key is hit again, those legs should move back to their original position and the opposite legs should move forward, the tail should move all the way to the right, and the head should move all the way to the left, and the nose tentatcles stay out. When the key is hit a third time, everything should move back to the original position: limbs, head, tail, and nose tentacles.
- (12 pts) Dig ('d'): The front paws both rotate down, with claws rotating even further downward. When the key is hit again, the paws and claws move back to their original position.

[1] http://www.esf.edu/aec/adks/mammals/starnosed_mole.htm

- Smooth transitions: for each animation above, you get half the points for implementing a simple jumpcut. You get the other half of the points for implementing smooth transitions. That is, draw many frames in succession where each movement is small. You should linearly interpolate between the old position and the new one. You should request an animation frame between each step rather than taking control away from the main event-handler by doing the transition in your own loop.

- (extra credit: up to 3 pts) Add more motions. For instance, support different kinds of wiggles by creating a body with more articulation points. Or have your beast dive into water, or dance, or do kung-fu. Or new camera positions or trajectories. Or make sure the animation runs at the same wall-clock speed no matter how fast or slow the computer can draw. Or whatever strikes your fancy!

**Interaction**: [9 points total] Interactively control your beast.

- (9 pts) Keys: Add the following keyboard bindings for the animation: 'u' for body tilt up, 'd' for body tilt down, 't' for tail move right, 'v' for tail move left, 'h' for head move right, 'g' for head move left, 'n' for nose tentacles fanout, 'd' for dig, 's' for swim. These keys should act as toggles in jumpcut mode: when the user hits the key, move from rest position to new position, or from the new position back to the rest position. Have the spacebar toggle between jumpcut mode and smooth transition mode. In smooth transition mode, hitting a key should cause a single smooth transition: either from the rest position to the new position, or from the new position to the rest position the next time the key is hit. If you create extra credit motions, pick unused letters to trigger them and document these in your README writeup.

  (Some keys are already specified the template: 'z' to show/hide a floor grid that will help you see how things are oriented, and '0' to reset the camera to a side view (the snout should be on the left, and the tail on the right.)

**Suggested Strategy**

- Interleave the modelling, animation, and interaction by building up your animal gradually. Do **not** model the whole animal and then try to animate it! Instead, build up incrementally, where as you add each new part into your model you also implement the animation of that part - and thus you'll also need to implement the animation controls for that action in order to test it. Implement the smooth animation for a part right after the jump-cut animation, don't wait until the end to do all the smooth ones at once. Obviously, there are dependencies here: if you don't model a part, you can't get credit for animating that part.

- Think in advance about what transformation hierarchy makes the most sense for your animal, and begin your implementation with the root of that tree. If you don't interleave the modelling and animation, you might spend a lot of time creating an animal with the wrong kind of hierarchical structure to move correctly.

  For example, think about adding a limb to the body. You would want to first add the paw, and then implement the jumpcut animation of that paw, and then make sure the smooth animation also works properly. Then move on by adding a claw to the bottom of that paw, and test that. Then add the other four claws. Think about how to write reasonably modular code, rather than copying and pasting to create large blocks of nearly identical code.

- Build your beast in a 'rest' pose standing on all fours. Consider the rest pose a starting place where you define the rotation angle of each joint to be 0, and to move the joints for the animation you will be changing that angle. You might find it easier to debug your code if you use a separate transformation for the joint animation than the one you use for the modelling, but that's up to you.

- Think about what parts of the animation can be reused, for example how you might write functions to control the legs in a general way that handle all the required movements. Also think about what complex animations can be built up by reusing some or all of simpler ones.

- Think about what data structures to create for supporting animations. Keep in mind that if there are any numbers you need to use, it's much better style to store them in a data structure rather than scattering magic numbers all over your code! Some of the state that you might want to keep track of is

  - What animation mode are you in: jumpcut or smooth?
  - How many frames do you want to use in the smooth animation?
  - How far through a given animation are you: that is, what is the current frame number? Think about whether this number can be globally specified or must be tracked separately for each animation.
  - What are the parameters that control the position of the relevant body part(s) at the start and end of each animation? For example, for a given animation, what needs to change - just rotation? Both rotation and translation? Is scaling ever necessary?

  - Which direction is the animation going for this part? For example, during the nod, is the head moving from up to down, or from down to up?

- You may not use the Three.js commands `rotation`, `translatation`, and `scale`, you must manipulate the matrix directly yourself. Remember that it's not safe to interpolate the individual elements of a $4x4$ matrix!

- It's OK if your animal intersects with the floor or with self-intersects with itself when moving. While it's not physically realistic, you will not lose any points for that behavior on this assignment. It's also OK if there are biologically unrealistic gaps between segments of your animal - your animal can look like a cartoon.

- While you're debugging, don't forget to try moving the camera as described below to check whether things are placed correctly. Sometimes a view from one side can look right, but you can see from the top or front that it's in the wrong spot along one of the other axes.

  Also consider how to do visual debugging, since now you're a graphics programmer! We have provided a ground plane for orientation in the template code. You might choose to add more. For example, you might color each face of your cube differently, to help you understand its orientation - you would do this by changing the drawCube function from its default. You could have colored lines sticking out of each face of the cube for debugging purposes. You can turn your cubes from solid to wireframe to help you see whether one is hidden inside another, by changing the WebGL geometry type.

  Try moving physical objects or experimenting with demos to help you think about transformations.

- Do not implement smooth transitions with a loop where you take control away from the standard event-handling code; instead, use the requestAnimationFrame() function to request that the display function is called to work within the event handling architecture.

  In your smooth transition, change should happen gradually over a certain number of frames, after the animation is triggered by a key click. The straightforward way to do this is to pick a certain number X for the frame rate, and just redraw X frames. Then on each redraw, `param += (new-old)*X`. You would get full credit for this approach. A more complex way would be to make sure that the animation always takes the same amount of time no matter what the per-frame drawing speed of the computer that it runs on; you could think about how to accomplish this for extra credit.

- Finish doing all required functionality before starting on any extra credit.

- I recommend using version control to make sure that you do not lose any of your work. It's a bad idea to just keep overwriting the same file again and again. Save off versions often, for example after you get one thing to work and before you start on the next piece, or just before you do something drastic. There are many ways to do this: the least sophisticated way is to keep commenting out blocks of code, you'll quickly lose track. The best way is to use version control software: then it's easy to browse your previous work and revert if needed. For maximum benefit, use meaningful comments to describe what you did when you check in a new version (for example: *started on tail*, *fixed head breakoff bug*, *leg code compiles but doesn't run yet*). Version control is useful when you're working alone and even more crucial if you work in teams. Git is the most popular system these days, and svn is an alternative.

  There are many graphical file comparison tools that allow you to easily compare different versions of your code. On the lab Linux machines, there is `xfdiff4` for side by side comparison and `xwdiff` for in-place comparison with crossouts. On Windows, try `windiff` (downloadable from `http://keithdevens.com/files/windiff`). On the Macs, try FileMerge, at `/Developer/Applications/Utilities` if you have installed XCode.

**Template** Download the template from the links at `http://www.ugrad.cs.ubc.ca/~cs314/Vjan2016/#p1`
The template code allows you to change the viewpoint. Click and draft with the left mouse button to rotate, with the right mouse to pan, and use the scroll wheel to zoom in and out. The arrow keys also control precision panning. Consider the beast to be at the center of a cube. In the default that results from resetting with the '0' key, you're looking at it from one face of the cube, for a side view, with the tail on the right.

**Style** All of the above breakdown of marks was based on correct performance. You also need to produce clean code. You can lose marks for poor style up to a maximum of 15% of the assignment grade. The most important style issue is to have reasonable modular structure: avoid duplicate or near-duplicate code. For example, parameterized functions should do similar things rather than a lot of cut-and-paste code with slight alterations to handle different cases. Note that global variables are necessary in event-driven programming, we do not consider them to be a style problem. Your code should be readable, with well-chosen variable and function names and enough comments to explain what is happening. Your rule of thumb for comments should be: what and how should you document to help somebody who has to fix a bug in this code two years from now? Your comments should help that person understand the structure of the code quickly, and explain anything tricky or non-obvious.

**Grading** This project will be graded with face-to-face demos: you will demo your program for the grader. We will post the signup link for a 10-minute demo slot on Piazza. You should be in the 005 lab machine at least 5 minutes before your scheduled session. The grader will spend part of the time with you, doing a mix of looking at your code, running your demo, and discussing the code with you. The grader will spend some time alone, looking at the code further and writing up notes. The signup link will be posted on Piazza the week before grading begins. If you do not sign up, or you sign up for a slot and do not show up, you will be penalized 10% of the mark - don't make us hunt you down!

You *must* ensure before submitting the assignment that your program compiles and runs on the machine you intend to demo it in, either your laptop or a lab machine. If you worked on this assignment elsewhere but will demo it on a lab machine, it is your responsibility to test it in the lab. Plan ahead: ensure your code runs correctly on the lab machines before submitting it, both in terms of compile/run, and parameter settings for animations so that your transitions look good (the lab computers may be slower or faster than your home machine). Note that you cannot ssh in to the lab machines (`linXX.ugrad.cs.ubc.ca`, where XX is 01 through 24) remotely from outside the CS department, but you can get to them by first logging into a CS server (such as `remote.ugrad.cs.ubc.ca`). Be considerate about using the graphics on a machine remotely, as it could drastically slow down the machine for the person sitting at the console - if you're making the machine slow to a crawl, the person at the console might choose to reboot it!

The face to face grading time slots are short, you will not have time to do any quick fixes. If, nevertheless, you somehow discover some critical problem at the last minute, do **not** just edit the original file! Instead, copy the submitted code to a new file and change only that new file. Then the grader can quickly verify that you only made a trivial change by running `diff` to compare the two files.

**Documentation**

- **README.txt (required)**: Your README.txt file should include your name, student number, 4-character username, and the statement:

  > By submitting this file, I hereby declare that I (or our team of two) worked individually on this assignment and wrote all of this code. I/we have listed all external resoures (web pages, books) used below. I/we have listed all people with whom I/we have had significant discussions about the project below.

  You do not need to list the course web pages, textbooks, the template code from the course web site, or discussions with the TAs. Do list everything else, as directed at in the collaboration/citation policy for this course at

  `http://www.ugrad.cs.ubc.ca/~cs314/Vjan2016/cheat.html`

  Including this statement in your README is your official declaration that you have read and understood this policy.

  In your README, state what functionality you have successfully implemented. If you don't complete all the requirements, please state clearly what you have tried, what problems you are having, and what you think might be promising solutions. If you did extra credit work, say what you did and how many extra credit points you think the work is worth. Please be clear and concise.

- **2 images (required)**: You must include at least two images of your beast, front and side views are a good choice. We'll post the best of these images in the Hall of Fame. You may submit some extra images.

- **movie (optional)**: You may include a movie, especially nice if you'd like to show off a cool extra-credit animation of your beast in action in the Hall of Fame.

**Handin** Transfer your files to a UBC CS machine and login to it. All the assignment handin subdirectories should be put the `cs314` directory of your root directory, which you created for project 0. For this project, create a subdirectory of `cs314` called `p1` and copy to there all the files you want to hand in: README.txt, source files ending in `.html` and `.js`, image files ending in `.png` or `.jpg` or `.gif`, and movie files ending in `.mp4`. Do not include multiple revisions of your code, just the final version.

The assignment should be handed in with the exact command: `handin cs314 p1`. For more information about the `handin` command, see `man handin`.

You can run handin as many times as you want, you don't need to wait until the very last minute and then submit in a rush. Consider handing in intermediate versions to be on the safe side.