CPSC 314, Programming Project 1: Rambunctious Rabbit

Out: Mon 14 Jan 2013. Due: Fri 25 Jan 2013, 5pm. Value: 8% of final grade

In this project, you will create a rambunctious rabbit. 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: [41 points total] Model a rabbit out of transformed cubes, using only 4x4 matrices to position and deform them.

- (1 pt) Create a drawCube() function that draws a unit cube. This is the only place in your program that you should call OpenGL *geometry* commands (but you may call OpenGL *transformation* commands anywhere!).
- (40 pts) Model your rabbit out of transformed cubes. Remember that you can do nonuniform scaling to create long skinny boxes. You should orient your rabbit so that you see a side view from the default camera position. You should create your rabbit 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 ear 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:
  - (9 pts) Body in three parts: front, middle, and back
  - (12 pts) Neck, Head, Eyes, Ears
  - (4 pts each, 16 total) 4 Limbs: each with Leg, Paw
  - (3 pt) Tail
  - (extra credit: up to 2 pts) Add color and/or add more geometric detail to your beast, such as claws or whiskers. And/or make an interesting environment for your beast, again using only cubes and 4x4 matrices.

Animation: [49 points total] Animate the joints of your beast. Specifically

- (4 pts) Head nod: Move the neck/head down vertically with respect to the upper body, the eyes/ears must of course move along with the head. When the key is hit again, return to the previous position.
- (8 pts, 2 pts per leg) Leg raise: Rotate a limb up and forward with respect to the body. The paw must of course move with the leg. When the key is hit again, return to the previous position.
- (4 pts, 2 per ear) Ear wiggle: wiggle each ear up or down with respect to the head. If the ear starts pointing down it should end pointing up, or vice versa.
- (8 pts) Rear: The rabbit should rear up on its hind legs, moving from the rest position on all fours to an upright position with the body vertical to the ground. When the key is hit again, return to the previous position.
- (10 pts) Body curl: Curl the body up as if the rabbit is asleep, with the front and back segments of the body moving towards each other so that the legs come closer to each other. When the key is hit again, return to the previous position.
- (15 pts) Jump: When the rabbit is on the ground, the jump command should make the whole animal move up into the air. The body should curl out in the opposite direction from the body curl, so that the back is arched and the legs move away from each other. The second time the key is pressed, the rabbit should move back down from the air to the ground, and uncurl back into the rest position.
- 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 a redisplay event from GLUT between each step rather than taking control away from the main event-handler by doing the transition in your own loop. For full credit, you should properly handle the case where a second transition begins while another transition is already happening, and you should also ensure that if your program does not redraw unnecessarily and burn CPU cycles when transitions are not happening.
- (extra credit: up to 3 pts) Add more motions. For instance, a more complex jumping motion, or different kinds of wiggles by creating a body with more articulation points. Or have your beast hop forward, 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: [10 points total] Interactively control your beast.

- (10 pts) Keys: Add the following GLUT key bindings for the animation: 'h' for head nod, 'l' for front left leg raise, 'm' for front right leg raise, 'n' for rear left leg raise, 'o' for rear right leg raise, 'e' for right ear wiggle, 'w' for left ear wiggle, 'j' for jump, 'c' for body curl, 't' for rear. 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.

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. (Hint: it's one of the body segments - think about which one!) 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 upper leg part, and then implement the jump-cut animation of that leg, and then make sure the smooth animation also works properly. Once that all works properly, then move on by adding the paw to the bottom of that leg, and test that.

- 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 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 use the OpenGL commands `glRotate`, `glTranslate`, and `glScale`. Note that it is safe to interpolate the parameters to these commands linearly, even though it's not safe to interpolate the individual elements of a $4x4$ matrix!

- You may use `glutSolidCube` to build your cube, or build your own out of six square faces. If you build your own, make sure to specify the vertices in each face in counterclockwise order. It's your choice where to have the origin of the cube. One option is the center of the object, another option is to have it at a corner.

- It's OK if your animal intersects with the floor or with self-intersects with itself when curling up. 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. You can get far with three camera placements: default side, top, and front.

  Also consider how to do visual debugging, since now you're a graphics programmer! We have provided an axis and 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 (if you build your own instead of using the built-in GLUT cube). 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 OpenGL geometry type.

  Try moving physical objects to help you think about transformations, just like I do during lecture. You might also play with the scenegraph applets from Brown that I show as demos in class.

- Do not implement smooth transitions with a loop where you take control away from the standard event-handling code; instead, use GLUT idle functionality to request that the display function is called to work within the event handling architecture. Your program should redraw exactly and only when necessary: when nothing is moving, the view should not be continually redrawing. In response to a keystroke trigger a jumpcut change, the program should redraw once. When a smooth transition is triggered, the view should redraw many times until the transition has ended, and then redrawing should stop. Remember that you can turn the idle function on and off, as shown in the `glut4.cpp` example code discussed in lecture.

  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 that you use some form of 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. A slightly better way would be to save off the file under the a new name. 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 along for these first three projects, and it will be even more crucial if you work in teams for the final one. You have many choices: RCS, CVS, git, svn. See the course home page for svn quickstart documentation!

  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/#p1` There are separate template packages and instructions for unpacking for Linux, Mac, and Windows. The course home page has links to setup instructions for thes three platforms. The Linux package is set up for Eclipse use. The Mac package is set up for XCode use. The Windows package is set up for Visual Studio use. You are welcome to set up your own development platform in any way that you like; for instance, you could install cygwin on Windows in order to use `g++`. Remember that you must demo your code on the Linux lab machines, so make sure to doublecheck that everything works properly there before you submit your final version.

The template code allows you to change the viewpoint to look at the central object from any of six directions. Consider the beast to be at the center of a cube. In the default you're looking at it from one face of the cube, for a side view. You can move the camera so that you can see the beast from any of the other 5 faces of the cube: other side, front, back, over, under. Trigger this action with the 'p', 'f', 'b', 'a', 'u', respectively. The 'r' key resets to the original view. You should construct your beast so that the default view is indeed the side view. The 'q' key exits the program. Each time you hit the 'i' key a new image is saved in the `ppm` directory. When you hit 'd', the image counter will be reset and program will dump out a new image at each redraw until you hit 'd' again to stop the dump. Hitting any key will trigger a redraw.

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 lab machines. If you worked on this assignment elsewhere, 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 worked individually on this assignment and that I am the only author of this code. I have listed all external resoures (web pages, books) used below. I have listed all people with whom I 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/Vjan2013/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. The template code allows you to save frames as PPM image files. You can browse the images that you created with the `display` command. 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.

  You can make animated GIFs easily using the `convert` command: `convert -delay 20 -loop 0 img*.ppm anim.gif`

  You can make an MPEG animation file (uses much less space, but won't play directly on a web page) using the `ffmpeg` command: `ffmpeg -i img%03d.ppm -r 24 a1.mp4`

and play your movie using `ffplay a1.mp4`

If you will dump a large number of images, consuming a lot of disk space, create a temporary directory `/tmp/foo` where `foo` is your user name. Edit the appropriate line in the dumpPPM() function call in order to ensure that image files get written to this directory. Hitting 'p' when running your code in animate mode will results in a large number of PPM files being written to this directory of the form imgNNN.ppm, where NNN is the frame number. Dont forget to delete all your PPM files once your movie has been created to save some disk space. If you want to keep your movies reasonably small in size, use a small window when dumping your frames. For example, a 500 x 300 video will use only 25% the disk space of a 1000 x 600 video.

Handin Create a root directory for our course in your account, called `cs314`. All the assignment handin subdirectories should be put in this directory. For project 1, create a subdirectory of `cs314` called `p1` and copy to there all the files you want to hand in: README, source files ending in `.cpp` and `.h`, image files ending in `.png` or `.jpg` or `.gif`, and movie files ending in `.mp4`. You should also include a makefile. If you're using Eclipse, it's fine to simply copy the entire project folder from Eclipse, make sure that there is a Release subdirectory that includes the auto-generated makefiles. However, 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.