`anim`, an easy-to-use animation routine

1 Introduction

`Anim` is a utility for drawing 2D or 3D graphics that is language-agnostic and easy for students to learn and use.

The normal usage is to write a program in any language (Python, C, Fortran, COBOL, whatever...) that prints graphics commands to its standard output using the language's built-in print feature, then pipe this output to `anim` with

```
./myprogram | anim
```

or

```
python myprogram.py | anim
```

This provides a language-agnostic graphics capability that requires no additional computer science knowledge, beyond printing text, from students. It runs in Linux or Mac OS.

The program can operate in either 2D or 3D mode, and will automatically choose 3D mode if any 3D drawing commands are received.

2 Installation

2.1 Linux

You will first need to install a few packages to get OpenGL support and support for screenshots.. On an Ubuntu system, do

- `sudo apt install freeglut3-dev libglew-dev libpng-dev imagemagick libpng-dev`

Then you can do:

- `make anim`
- `make demos`
- `sudo make install` (which copies anim into `/usr/local/bin`; alternatively, copy anim into your system path somewhere else)

2.2 Mac OS

Type the following:

- `make anim-mac`
- `make examples`
- `sudo make install` (which copies anim into `/usr/local/bin`; alternatively, copy anim into your system path somewhere else)

(You will get some warnings about OpenGL 2 system calls being deprecated, but the program will still work.)

3 Usage

`Anim` is a frame-based drawing utility. Rather than displaying graphics as it receives commands, it displays graphics one complete frame at a time, allowing the creation of smooth animations. No graphics will be displayed until `anim` receives a flush frame command; at that point, the completed frame containing all objects drawn up to that point is displayed. Any subsequent graphics commands will be stored up for the next frame, and displayed once the next flush frame command is received, and so on. A typical usage is to include many drawing commands followed by a flush command within a simulation loop, drawing one frame of animation every one (or more) simulation intervals.

The input commands are as follows:

3.1 Drawing commands

- `l` *x1 y1 x2 y2* – draw a line from (x1,y1) to (x2,y2)
- `L` *x1 y1 x2 y2* – draw a line from (x1,y1) to (x2,y2) relative to the viewport (objects that should remain stationary on screen even if the viewport is scaled; see the sample program `overrelax` for an example)
- `c` *x y r* – draw a circle at (x,y) with radius r
- `C` *r g b* – change the current drawing color to the given RGB color
- `t` *x y* – Print some text at coordinates x,y. The text that will be printed is entered as the *next* line of input.
- `T` *x y* – Print some text at coordinates x,y relative to the viewport. This is useful for creating text readouts that will not move as the viewport is moved/rescaled. Again, this is a two-line command, with the text to be printed on the next line.

3.2 Drawing commands, 3D-specific

- `l3` *x1 y1 z1 x2 y2 z2* – draw a line from (x1,y1,z2) to (x2,y2,z2)
- `c3` *x y z r* – draw a sphere at (x,y,z) with radius r
- `ct3` *index x y z r* – draw a sphere at (x,y,z) with radius r, and additionally create a trail behind it. Since you might want multiple objects with trails, this command also requires a trail index. (If you want only one trail, set the index to 0.)
- `trl` *index length* – set the length of the trail with the given index
- `t3` *x y z* – Print some text at coordinates x,y,z. The text that will be printed is entered as the *next* line of input.
- `q3` *x1 y1 z1 x2 y2 z2 x3 y3 z3 x4 y4 z4* – Draw a quadrilateral with the specified endpoints (this is useful for making meshes).

3.3 Control commands

- `A` *x* – Turn on (x=1) or off (x=0) the display of axes.
- `S` *scale* – rescale the window, such that the distance from the center to the edge is *scale*, without changing the angle of view.
- `S3` *viewdist* – change the perspective, so that the subject is viewed from *viewdist* units away, without changing its apparent size
- `screenshot` – take a screenshot and save it. It will be given a default name (`anim.png` or `anim-`(number)`.png`) if you don't specify one; you can also specify a filename.
- `center3` *x y z* – center the window at coordinates x,y,z
- `!`*text* – Print *text* to standard output (so your program can still print text to the terminal, or to a log file)
- `F` – Flush the current frame to the screen
- `FG` – Flush the current frame to the screen, and also add it to the animated gif
- `endgif <name>` – Create an animated gif out of the frames flushed with `FG`; if you don't specify a name, a default will be used

The `!` command deserves special mention. The remainder of any input line prefixed by an ! will be printed to stdout, bypassing `anim`'s need to hijack the stdout of your program to receive graphics commands. Thus, you can do things like `./myprog | anim > output.txt`; all lines beginning with a ! will be written to `output.txt` (without the !'s), while all other lines will be interpreted by `anim` as graphics directives (or ignored if they make no sense to `anim`).

The commands that display text in the `anim` window (`t`, `T`, and `t3`) are all two lines long; the second line contains the text to be printed.

If you don't specify a value for the scale, framerate, etc., sensible defaults will be chosen. In the default scale the viewport runs from (-1.2, -1.2) to (1.2, 1.2), with axes enclosing the space (-1,-1) to (1,1).

3.4 Runtime control

The display of `anim` can be controlled somewhat with keyboard and mouse input into the window while it is running. The available commands are:

- WASD: move around (2D mode)
- WS/AD/QE: rotate (3D mode)
- shift-Q: quit
- shift-P: take a screenshot and save with a default name
- shift-I: invert brightness (for projectors in classrooms)
- shift-A: toggle axes
- -/=: zoom in/out (change the scale of the drawing)
- shift -/=: move the viewing position in/out, keeping the magnification fixed (changes prevalence of perspective effect)

You can also click and drag to move around and use the mousewheel to zoom in and out, although using the mousewheel may only work on Linux systems.

When `anim` is closed, it will remember the window size, viewport location and scale, 3D rotation angles, and the like.

3.5 3D mode details

The 3D mode lights the scene with a fixed light coming from the +z direction, along with a dimmer ambient light. If anyone is interested in the ability to adjust (either with interactive keyboard control in `anim` itself, or with text commands), please let me know and I'll add this.

4 Sample and demo code

To compile the demo codes that are in C, just do `make demos`.

4.1 `pendulum-basic.py`

This shows the basic structure of using `anim` to animate a swinging pendulum. In Python: just do `python pendulum-basic.py | anim` (or `python3 pendulum-basic.py | anim` if needed on your machine).

4.2 `frameskip-illustration.py`

This code shows a simple way of simulating multiple timesteps per frame of animation, as a variant on the previous; it runs just as fast, but uses a smaller timestep for more accuracy.

4.3 `kepler-problem`

This program in C models an orbit, demonstrating 3D mode and drawing motion with trails. Just run `./kepler-problem | anim`.

4.4 `modified-kepler-problem`

This program in C models an orbit with modified Newtonian gravity where the power law is not exactly inverse square, as happens in the regime where GR corrections matter. You should be able to see the orbit precess over time.

4.5 `pendulum-parallel.py`

This program in Python simulates multiple swinging pendula at once, demonstrating the amplitude-period connection, and illustrating how to build more complex 3D things in `anim`.

4.6 `string-parallel`

This program in C++ simulates multiple vibrating *strings*, showing a variety of nonlinear effects. Running `./string-parallel | anim` with no arguments will simulate the $N = 2$ normal mode with a variety of amplitudes, and also print out the order of command-line arguments if you want to change the behavior.

4.7 `membrane`

This program in C++ simulates a vibrating circular membrane. Running it with no arguments will simulate a Gaussian bump initial condition; you can also specify a Bessel function (normal mode) with `./membrane b` *(angular mode number) (radial mode number)*.