ECE198KL: Introduction to Computer Engineering II **Spring 2013**
Program 13 **due: 10 p.m. on Wednesday 1 May**

State Machines and Animation (for WALY)

This week, you will create an animation class, using class variables for images and fields for the state, thereby allowing yourself to display multiple instances of your animation simultaneously on different parts of the screen.

As always, routine details such as how to obtain the code and how to hand in your program can be found in the specification for Program 1. Instructions on targeting and making the Linux and Android versions can be found in the specification for Program 4.

The Pieces

The files are the same as last week—as mentioned in Step 0, you may want to pull in your files from Program 12, but you are not required to do so.

All work this week should be your own individual work.

Step 0: Bringing in your Own Screen

You may bring in your own screen class from Program 12, or you may use the BurstScreen class to host two or more copies of your animated class. If you want to use your screen, you should start by copying your header and source file from last week into this week's `jni` directory. You will also need to copy over (or re-edit) `Make.inc` or adjust the new one; if you copy it from last week, be sure to change `prog12main` to `prog13main` in all cases. Finally, you should copy last week's `prog12main.cpp` over this week's `prog13main.cpp`.

You will need to `svn add` your new files, of course.

Step 1: Creating a Class

Repeat Step 1 of Program 12 to create a new class to host your animation. You should add a header file and a source file, just as you did last week. Your new class should again be derived from `WALY::Frame`. You will need to add the files again to `Make.inc`. Refer to last week's specification for details, if necessary.

Step 2: Loading Image Files

We only need a single copy of a given image stored in memory, regardless of how many times we might want to copy that image to the screen for any given frame of animation to be displayed.

The `WALY::Image` class allows you to load image data (PNG files) into memory and attach it to a `Frame` object (or, in your case, to objects of a class derived from `Frame`) in order to have it drawn to the screen.

As mentioned earlier, you must create at least two copies of your animation class for simultaneous display. The images used by these copies, however, should be class variables, **not** fields. The `BurstScreen` class uses a simple model for initializing class variables. A counter (a class variable initialized to 0) records the number of `BurstScreen` objects. In the constructor, we increment the count; if the count goes from 0 to 1, we call a class function that loads files as `Image` objects and stores them in an array of `Image` pointers. In the destructor, we decrement the count of `BurstScreen` objects; if the count goes from 1 to 0, we call a class function that deletes all of the dynamically allocated `Image` objects and the dynamically allocated array.

You are encouraged to create a similar scheme. I put a bunch of the Depths hero images into the `assets/images` directory. That part of the file name is implicitly used in the `Image` constructor. You need only specify the file name starting in that directory; for example, "leftDownLeft.png" or "right2.png". If you get a file name wrong, the `Image` constructor throws an exception, and your program crashes. Make

sure that all images load successfully before going on to Step 3 (you will need to allocate an instance of your class if you used the approach just outlined).

Feel free to use your own images, of course. Simply add them (as PNG files) to the `assets/images` directory. If you have another image format, try using the `convert` tool to convert images to PNG.

Step 3: The Constructor and Use

The constructor for your class should take at least X and Y coordinates so that you can easily place your objects at different places on the screen. You can pass these directly to the `Frame` constructor: instead of just passing the parent `Frame` pointer in an initializer, pass (parent, x, y). As part of your animation, you may also want to move the object around the screen using `setX` and `setY`, but make it easy to differentiate the initial location using the constructor.

Once you have a constructor that both initializes images and allows you to place your frame, pick an image and add a call (in the constructor) to

```
void Frame::attachImage (Image* img);
```

Now if you create instances of your class as children of your screen class (or of BurstScreen), they should appear as the chosen image in the position where you choose to place them. Add at least two.

Step 4: Animation

You must define an animation state machine with at least 25 states and a function mapping those states into at least five distinct images. A counter (a cycle of states) will suffice, although you should feel free to design a more complex state machine that is affected by inputs from the user. You may also want to (optional) use `setX` and `setY` to move the image around on the screen.

The state for your state machine should be stored as fields of your new class. Each object should operate independently, so you need to use fields instead of class variables. (Images shown are just part of the output of the state machine, so they can use class variables.)

To drive the animation, you must create a class function and register your object to receive frame update callbacks to that function. By default, frame updates occur every 40 milliseconds (25 frames per second), although you can override that rate if you choose. Limits include the monitor refresh rate (∼60-70 Hz), your brain's ability to detect change (depends on the type of change, but probably not more than 200-300 Hz), and the computation speed of the machine (for tiny amounts of drawing, probably at least 1 kHz).

All event functions used with WALY have the same signature, so you can declare a handler function in your class definition as

```
static void drawEvent (Frame* f, const Event* e);
```

The `Frame` pointer is then a pointer to your class instance, and the `Event` has no content (functions such as mouse and keyboard events use the same signature and pass data through the event). In the definition of `drawEvent` (in your source file), cast `f` into a pointer to your class, then invoke a member function to handle the state machine transition. You might call it `stateTransition`, for example. In the `stateTransition` function, you can use fields of the class without prefixing them on every use.

You will also need to register for your objects to receive frame updates. To do so, set their callback function for the event to point to your handler (in your constructor):

```
setCallbackFunc (FRAME_UPDATE, drawEvent);
```

Design your machine, add the functions as above, and see what happens!

Note that your constructor must place your state machine into an initial state. For variety, that initial state may depend on the arguments passed to the constructor, but it may not be defined by "bits" (uninitialized).

Step 5: Competition

I have some leftover prizes from 391 design competitions, and permission from Microsoft to sponsor other activities for our students with them. So ... maybe we will use a combination of extra credit and prizes to encourage you to do something fun.

You can use graphics, sound, and whatever else you want to enhance your animation.

Challenge points awarded as follows...

5 for entering
5 more (10 total) for second runner up
10 more (15 total) for first runner up
15 more (20 total) for winner

We can set up something where people show them off, maybe at office hours or something similar, then have a vote. Prizes I think are Xbox games, maybe slightly dated (my last time teaching 391 was F11) OS/Office, maybe T-shirts.

Specifics

- Your code must be written in C++ and and must be contained in a header file and source file using a new name selected by you. These names must be included correctly in `Make.inc` to enable compilation. Do not change any other files.
- You must implement all steps (other than 0). All work on the animation class must be your own (you may have worked with others on part of your screen class from last week).
- You must initialize your state machine in your constructor and update it in response to frame update events. Other events can also be used, if desired (mouse, keyboard, and so forth).
- You must use class variables to hold pointers to image data, and fields to hold all data for your animation state machine.
- Your code must be well-commented. You may use either C-style (`/* can span multiple lines */`) or C++-style (`// comment to end of line`) comments, as you prefer. Follow the commenting style of the code examples provided in class and in the textbook.

Grading Rubric

Functionality (30%)
- 10% - At least two copies of your animation execute simultaneously on the screen.
- 10% - Animation uses at least five distinct images.
- 10% - Animation state machine contains at least twenty-five states.

Style (50%)
- 15% - uses class variables to hold pointers to `Image` objects
- 15% - uses fields to hold all state machine state
- 10% - compilation generates no warnings (note: any warning means 0 points here)
- 5% - only the new class' constructor and destructor are public; all else is private
- 5% - indentation and variable names are appropriate and reasonably meaningful (index variables can be single-letter)

Comments, clarity, and write-up (20%)
- 5% - introductory paragraph explaining what you did (even if it's just the required work)
- 15% - code is clear and well-commented

Note that some point categories in the rubric may depend on other categories. If your code does not compile, you may receive a score close to 0 points.