COMP 2659 Course Project – Stage 4: Renderer

Given: Wednesday, February 6, 2019
Target Completion Date: Friday, February 15, 2019

Overview

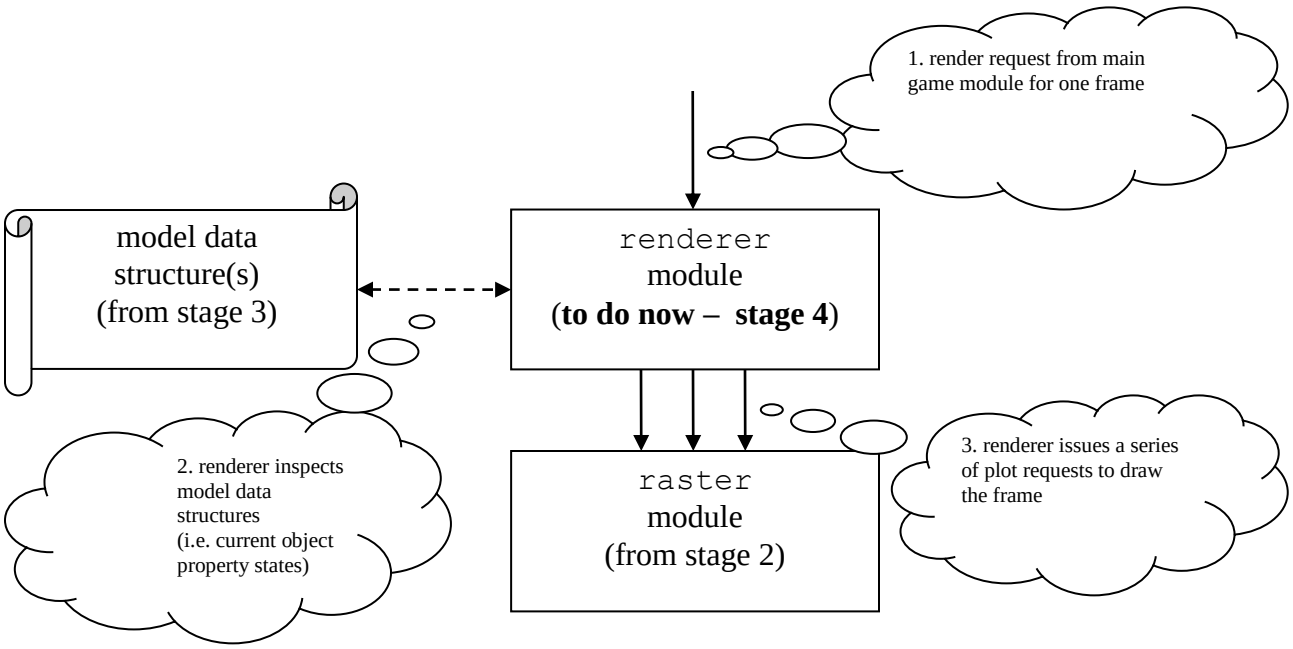
In this stage, you will write a module which links the model with the low-level graphics library, so that it is possible to render an individual, static frame of animation based on a static snapshot of the world.

Before starting on this stage it is critical that the model's data structures are complete, well designed and implemented, and also that the raster module contains a reliable plotting routine for each type of graphics primitive (e.g. bitmap, shape, line, etc.) required by your game.

Once stage 4 is complete, stage 5 will involve creating a main game module which updates the model based on user input and clock tick events, and that repeatedly renders frames of animation to achieve an animated, evolving game world.

Background: The Design of Video Games, Part 1 Recap

Recall that "rendering" is the process of generating an image from a model.



Note that the main game module (which will be written in a later stage) will eventually invoke the renderer module once for each frame of animation. In this stage, we are concerned only with rendering a single frame (i.e. a static, full-screen image) based on a static model (i.e. a snapshot of the model data structure(s) at a single point in time).

Requirement: `renderer` module

Provide a "render object" function for each type of object which can appear on-screen. For example, Pong would include "render ball" and "render paddle" functions.

In addition, provide a master "render" function which renders a whole frame based on the current state of the model. Of course, it delegates to the various "render object" sub-functions.

Each of these functions must take a frame buffer "base" pointer as input, as well as function-specific input. The base pointer points to the buffer into which the plotting must be performed. The function-specific input is the relevant portion of the model data structure(s).

For example:

```
void render(const struct Model *model, UINT8 *base);
void render_ball(const struct *Ball, UINT8 *base);
/* ... etc ... */
```

Place all the functions for this stage in a file module named `renderer`, with a corresponding header file.

Requirement: Test Driver

Write one or more test driver programs which help verify your renderer implementation. It is suggested that you construct various static models, and then render each one. This will allow you to inspect each frame individually. Because you construct the model on which the frame is based, you know exactly what the frame should look like.

Stage 4b: Minimizing Screen Plotting

In the first version of your renderer (call it "stage 4a"), it is acceptable to plot an entire frame each time the master render routine is invoked. Later, an easy way to animate the game (in stage 5) is to clear the screen and then completely re-render the entire model, for each frame. Of course, the disadvantage of this approach is that it is slow. It is therefore difficult to achieve smooth, high-quality graphics with a high frame rate.

Once stage 4a is working, you may wish to consider improving the renderer so that it only re-plots the aspects of the model that have changed since a previous invocation. There is more than one way to accomplish this – working out the details is your responsibility.

Completing stage 4b is not mandatory before starting stage 5. However, it is strongly recommended that you start thinking about how to do it. By the end of stage 6, it is likely that your game's graphics will suffer very noticeably if it naively and inefficiently over-plots.

Other Requirements

Your code must be highly readable and self-documenting, including proper indentation and spacing, descriptive variable and function names, etc. No one function should be longer than approximately 25-30 lines of code – decompose as necessary. Code which is unnecessarily complex or hard to read will be severely penalized.

For each function you develop, write a short header block comment which specifies:

1. its purpose, from the caller's perspective (if not perfectly clear from the name);
2. the purpose of each input parameter (if not perfectly clear from the name);
3. the purpose of each output parameter and return value (if not perfectly clear from the name);
4. any assumptions, limitations or known bugs.

At the top of each source file, write a short block comment which summarizes the common purpose of the data structures and functions in the file.

You should add inline comments if you need to explain an algorithm or clarify a particularly tricky block of code, but keep this to a minimum.

Background: Plotting Text using TOS's Bitmapped Font Table

For the plotting of text, it is possible to define your own "glyph" bitmaps – one for each character that may need to be plotted. These bitmaps can then be collected into an array of glyphs indexed by ASCII value. Assuming appropriate bitmap plotting routines exist, plotting strings of text is straightforward.

Alternatively, the Atari ST's operating system, TOS, has a bitmapped font table that you can use if you like. The font table is made up of 256 8×16 bitmaps – one bitmap per character in the extended ASCII character set. The following code shows how to access the table:

```
#include <linea.h>
#include <osbind.h>

...

UINT8 *base = Physbase();
UINT8 *fontTable;
char ch = 'a';                      /* example character to plot   */

linea0();                           /* required TOS system call    */
fontTable = (UINT8 *)V_FNT_AD;      /* start address of font table */
```

Treat `font` as an array of bytes. Indexing it at the ASCII value of `ch` will give a byte. This byte is the first row of `ch`'s bitmap! *The second row is 256 bytes further down the table*, and so on…

For example, the following code plots `a` in the upper-left of the screen:

```
*base          = *(fontTable + ch);
*(base + 80)   = *(fontTable + ch + 256);
*(base + 160)  = *(fontTable + ch + 512);
...
```

Of course, this can be accomplished more compactly using a 16-iteration loop.

Depending on how you have coded your bitmap plotting routines, it is likely that you expect bitmap rows to be contiguous, not 256 bytes apart! If you would rather that glyph rows be contiguous, write a program that "rips" the TOS font table and produces a modified font table with the glyphs stored as desired. In fact, this will be mandatory eventually, since a later stage will eliminate reliance on TOS altogether. If you are printing strings, It is actually more convenient to leave the font table in its original form.