COMP 2659 Course Project – Stage 3: Model

Given: Friday, February 6, 2015
Target Completion Date: Sunday, February 15, 2015

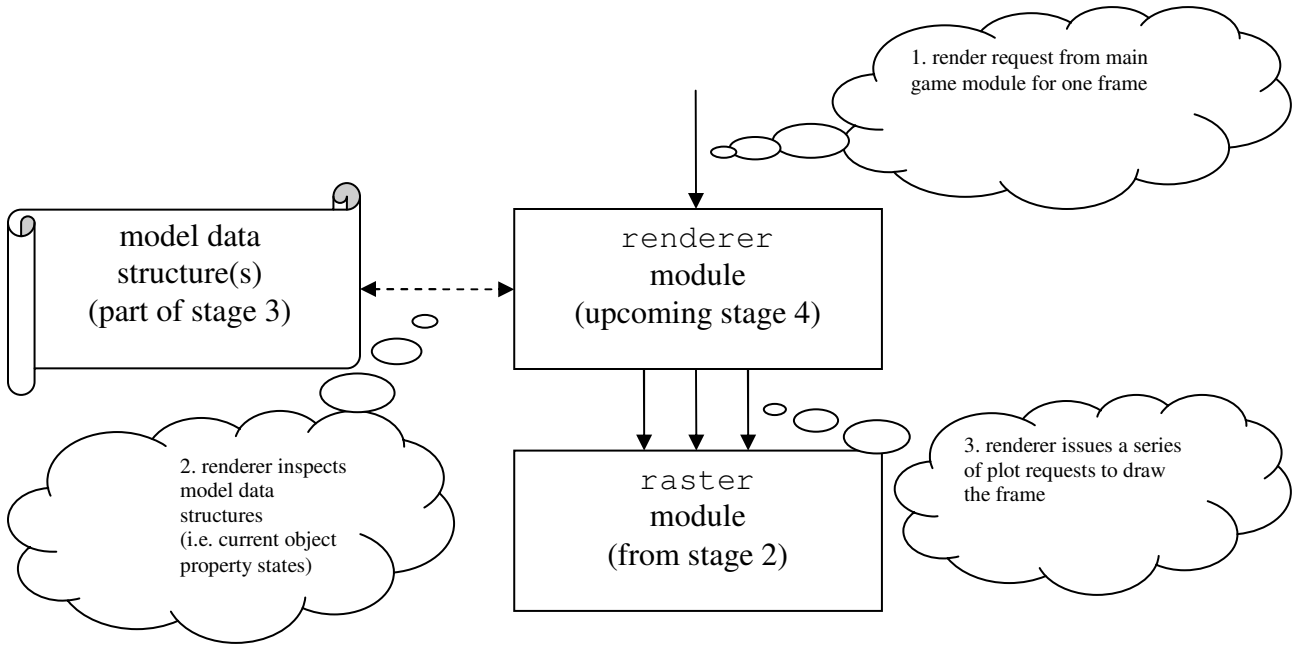
Overview

In this stage you will design and implement the data structures for your game world's object types. At run time, objects will be represented by structures whose fields correspond to the objects' properties. Also, you will implement functions for manipulating objects according to their specified behaviours. Finally, you will implement event-handling functions for each of the specified asynchronous, synchronous and condition-based events.

Background: The Design of Video Games, Part 1

A video game is based on a "model" which describes how the game world is represented internally by the computer. The model captures the essential properties of the world's objects, their behaviours and interactions, the game's logic and the world's physics. In other words, it describes how the world's state is represented at any point in time, as well as how the world's state evolves over time. *The model is independent of how a view of the world is presented to the user* (e.g. as an image on the screen). The model has very little or nothing to do with bitmaps, plotting algorithms or other aspects of how the objects are ultimately displayed.

"Rendering" is the process of generating an image from a model[1]. Later, stage 4 will deal with rendering an individual frame of animation based on a static model state.



Model data structures are updated in response to events – perhaps asynchronous events triggered by key presses or synchronous events triggered by a clock tick. Later, stage 5 will add event triggers and periodic render requests, so that the game world becomes animated.

[1] In 3D computer graphics, the term "rendering" has a more specific (but non-contradictory) meaning.

Requirement: Model Data Structures

Begin by designing the data structure(s) for representing the game world. You must identify each game world "object"[2] and object type, and decide on a representation for each of its properties. Ideally, the important details have already been described in your stage 1 specification.

For example, a ball in Pong has position[3] as well as velocity[4] state. In C, this can be modelled as:

```
struct Ball                  /* type definition for Pong's ball object     */
{
   unsigned int x, y;        /* position coordinates                        */
   int delta_x, delta_y;     /* horiz. & vert. displacement per clock tick */
};
```

Place these type definitions in a file named `model.h`.

Try to stick to integer values whenever possible. When representing fractions, it is often preferable to use a pair of integers instead of a floating-point value.

If your game world has a large number of objects with heterogeneous types, consider collecting them within a single `Model` type. For example:

```
struct Model
{
   struct Ball ball;
   struct Paddle paddles[2];     /* 0=left, 1=right */
   /* ... etc ... */
};
```

Make intelligent use of arrays for multiple objects of homogenous type.

Later, it will be possible to declare instance(s) of the model type(s). Initialize them according to the desired initial state of the game world. For example:

```
struct Model testPongSnapshot =
{
   { 320, 200, -2, 2 },      /* the single Ball instance */
   {
         { ... },            /* left paddle instance  */
         { ... }             /* right paddle instance */
   },
   /* ... etc ... */
};
```

[2] Although we are not using an object-oriented programming language, we will still think of the model as being comprised of various "objects".
[3] Note: the Atari ST's monochrome monitor resolution is 640 × 400. The ball position is therefore within this area.
[4] Time-relative properties such as velocity should be measured against a 1/70th of a second clock rate, as discussed elsewhere.

Requirement: Object Behaviour Functions

Each object type has a set of behaviours. Add corresponding functions to `model.c` for each of these. For example:

```
#include "model.h"

void move_ball(struct Ball *ball)
{
    ball->x += ball->delta_x;
    ball->y += ball->delta_y;

    /* ... what about collision detection? */
}
```

Also, declare ("prototype") these functions in `model.h`, because these will need to be callable by other modules.

Requirement: Event Handler Functions

Finally, develop an `events` module (with corresponding `events.h` and `events.c` files). Within, develop "event handler" functions which update the model's state for each kind of synchronous, asynchronous and condition-based game event. The actual events won't be occurring yet, but their handlers can still be implemented and tested. These functions will delegate some or all of their tasks to `model` functions.

Requirement: Test Driver

Write one or more test driver programs which help verify your model implementation, including event handling. Text output is acceptable for this stage.

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. This should also include a normal ID header, but should also include team member names and the author's name.

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.