COMP 2659 Course Project – Stage 5: Animation and Input

Given: Tuesday, February 23, 2015
Target Completion Date: Sunday, March 8, 2015

Overview

Before starting on this stage, it is assumed that you have a working renderer module which can render an *entire*, single frame of animation based on the state of the model's data structures.

Ideally, if you have completed stage 4b, the renderer can also render an *updated* frame of animation (i.e. without redrawing the entire frame), based on information about which elements of the model have changed since a previous invocation of the renderer.

In this stage you will create a main game module. The module will contain a main game loop which polls for and detects clock ticks and keyboard input. These will allow the model to evolve in response to synchronous and asynchronous events, respectively. The main game loop will also periodically invoke the renderer, so that a sequence of frames of animation are produced over time. In this stage your game world ceases to be static, and becomes dynamic. In other words, your game will start to become playable!

If you haven't yet completed stage 4b, it is still possible to make progress on stage 5. However, the animation will be unnecessarily slow and choppy. Therefore, you will be forced to refine your renderer so that plotting is minimized. At the very latest, it will be necessary to address this issue prior to stage 6.

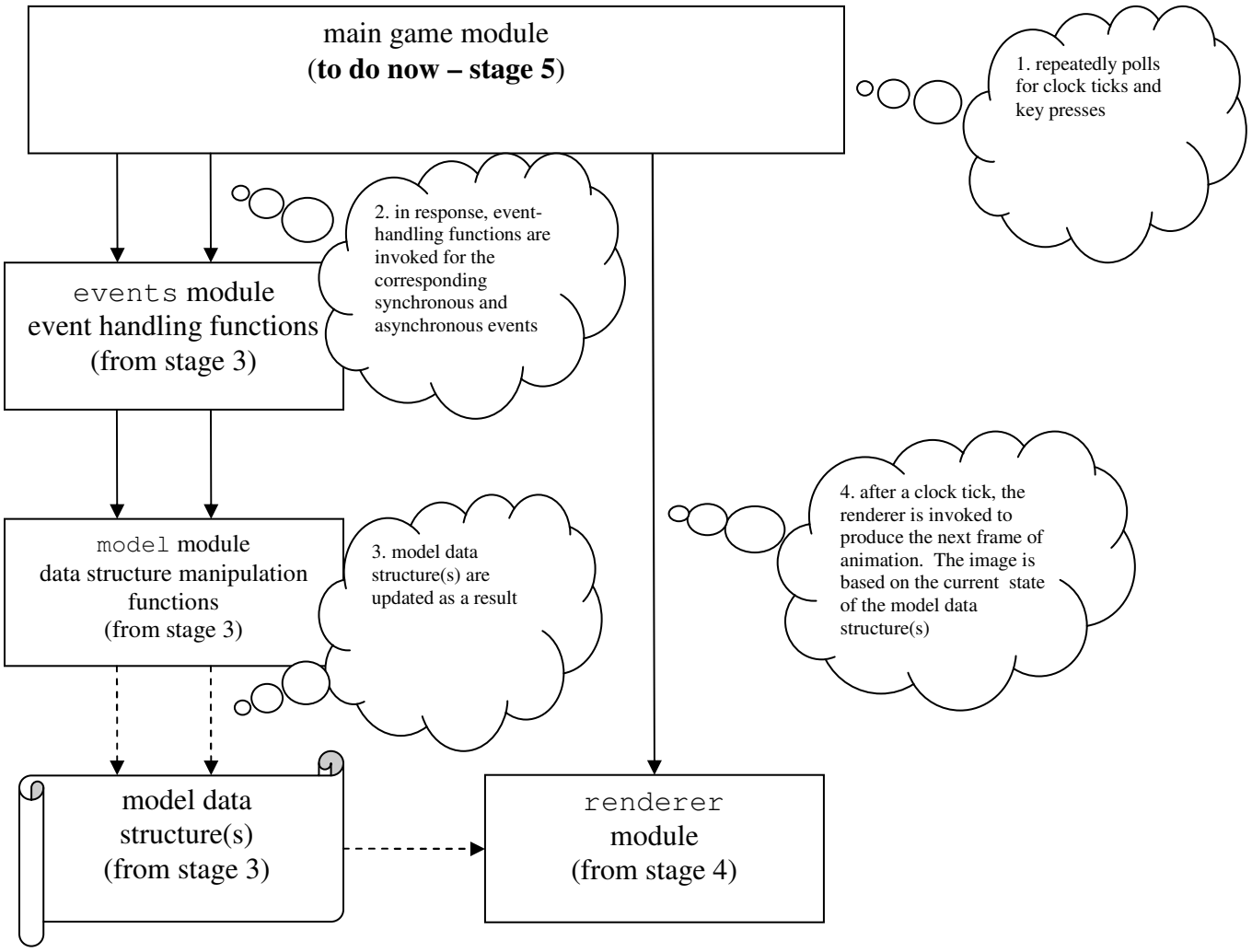
Regardless of how well you complete stages 4, 4b and 5, the animation will remain imperfect. You will probably notice graphical "flickering" and "tearing" effects. In short, this will be because plotting will not be synchronized with the screen refresh rate of the video hardware. Later, in stage 6, you will fix this problem using a technique called "double buffering".

Background: The Design of Video Games, Part 2

In an animated video game, individual frames of animation are rendered in real time based on the current state of the model. The "frame rate" is the rate at which successive frames are displayed. The faster the frame rate, the smoother the animation will appear to be. For this game, frame rates up to 70 fps (frames per second) are achievable[1]. This should appear smooth to most human observers. By contrast, frame rates down to about 12 fps may be perceived by most humans as animated, but jumpy[2].

A clock is used to drive the frame rate. It can also be used to produce synchronous (internally generated, clock synced) events within the game world. Of course, asynchronous (externally generated, non-clock synced) events such as user input must also be dealt with. Each event will typically entail updating some aspect of the model. Consequently, the next time a frame is rendered, the view of the world may have changed.

[1] The Atari ST's monochrome monitor has a 70 Hz refresh rate.
[2] Some commercial video games have frame rates in the 25-35 fps range. This is considered only minimally acceptable by some players. Other games attempt to match the refresh rate of the monitor, which is around 85 Hz on an average modern monitor. Still other games have an even higher frame rate, even though the monitor can't match it. In this case, the extra frames are discarded (or the frames aren't synchronized with the refresh rate, which leads to visual disturbances such as "tearing").



An essential part of a game's code is its *main game loop*, found within the main game module. The algorithmic structure of this module is illustrated below:

```
initialize model
render model (first frame)
set quit = false

repeat until quit
     if input is pending
          process async event              ← update model
     if clock has ticked
          process sync events              ← update model
          render model (next frame)
```

Requirement: Main Game Module

You must create a main game module. This must be coded as a source code file with an appropriate game-specific name (e.g. "`pong.c`"). This module must contain your game's `main` function. It may also contain various helper functions.

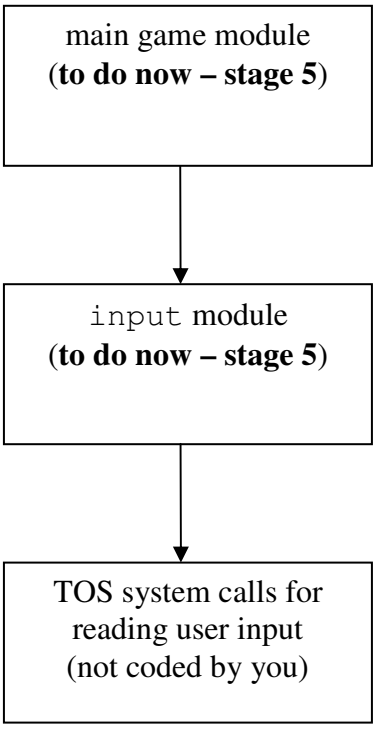
The main game module must contain game initialization code, as well as the main game loop. The game is not to display welcome menus or splash screens at this stage.

TOS maintains a 70 Hz clock at memory location 0x462. Every 1/70th of a second, the longword value at this address is auto-incremented. The main game loop must repeatedly poll this value. Each time it changes synchronous events can be triggered. The rendering of the next frame of animation must also be triggered by clock ticks. You must not use user input events as rendering triggers!

To the main game loop, you must also add checks for user input so that asynchronous events can be triggered.

Requirement: `input` Module and Three-Level Design

When you code the user input portion of your game, you must follow a three-level design:



The middle layer contains the "high-level" input routines needed by the game. These in turn call the "low-level" input routines provided by the operating system. The former are hardware-independent, whereas the latter deal directly with the input hardware.

In summary, identify the input routines needed by your game and then implement a corresponding high-level input library named `input`. Note that these library routines must actually be relatively game independent (i.e. it should be possible to reuse the library in other applications). These routines will wrap underlying, O/S-specific code for performing keyboard and mouse input. This will ensure that game code is as O/S-independent as possible.

The advantage of this approach will become apparent in later stages, when the O/S support will be replaced by custom low-level code. If the main game module code has not been decoupled from the O/S, it will make removal of the O/S support difficult. Instead, it should be possible to leave the game code alone and to only modify the middle layer wrappers.

Like for the previous stages, you must provide one or more test driver programs which invoke the middle layer routines.

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: How to Detect Clock Ticks

Recall that TOS maintains a 70 Hz timer: a longword at address 0x462 which is auto-incremented 70 times per second. Your game can monitor the passage to time by periodically reading the contents of this memory location.

The timer memory location is protected: the ST hardware monitors memory accesses at this address, and prevents them unless the CPU is in a "privileged" mode. Normally, only TOS code runs in this privileged mode, while ordinary user programs that attempt to access the timer crash with a bus error:

```
long *timer = (long *)0x462;     /* address of longword auto-inc'ed 70 x per s */
long timeNow;

...

timeNow = *timer;                /* crashes with bus error (2 bombs) here! */

...
```

However, TOS allows[3] ordinary user programs to run in privileged mode! All that is required is a call to the `Super` system call:

```
#include <osbind.h>

...

long old_ssp;

old_ssp = Super(0);     /* enter privileged mode */
timeNow = *timer;
Super(old_ssp);         /* exit privileged mode as soon as possible */

...
```

The above code should be incorporated into a helper function called `get_time` which takes no parameters and returns the current timer value as a `ULONG32`.

Aside: While it may seem like a good idea to run your entire program in privileged mode it is actually a terrible idea. When in privileged mode you have unfettered access to everything. Thus, if you inadvertently alter a system variable or peripheral the operation of the system can change, possibly causing the system to hang or crash. For example disabling the keyboard will result in the system appearing to hang. In non-privileged mode this action would have resulted in 8 bombs indicating a Privileged Violation. In privileged mode there will not be any bombs and you will have no idea what is the problem and where it occurred or why. Thus, it is common to change into privileged mode, perform only those actions that require privileged access and then leave privileged mode.

Once the above is working, it should be possible to incorporate code like the following into the main game loop:

```
ULONG32 timeThen, timeNow, timeElapsed;

...

timeNow = get_time();
timeElapsed = timeNow - timeThen;

if (timeElapsed > 0)
{
   ...                      /* trigger synchronous events based on timeElapsed */
   ...                      /* render model */
   timeThen = timeNow;
}
```

Background: How to Detect Keyboard Input

TOS provides a `Cconis` system call for checking whether or not keyboard input is available for reading. And, it provides a `Cnecin` system call for actually reading it (without echoing the key press to the screen). Code like the following would read and process a key press only if one was pending:

```
#include <osbind.h>

...
```

[3] Foolishly for TOS, but luckily for us.

```
if (Cconis())                    /* is there user input read to be read? */
{
   char ch = (char)Cnecin();     /* if so, read it right away (otherwise don't */
   ...                           /* waste time waiting for any)                */
}
```

If `Cnecin` is called without first checking if a key has been pressed, it might "block" until a key is pressed.

More information on these TOS system calls is available in the reference subfolder of the course resources folder. The Devpac manual is a particular useful source of system call information.

Note: some games require that key press and key release events be detected separately. Although this will be possible later, the routines above do not allow for this functionality, so these only key press events can be detected for now. However, TOS will cause a key that has been pressed and held down to auto-repeat. So, until a later stage, detection of auto-repeated key presses must temporarily be used in place of detection of a key release.

A Note on Detecting Mouse Input

At this stage, the core game is expected to be keyboard-driven. A welcome splash screen with a mouse-driven main menu will be added later. Therefore, mouse input detection is not normally required at this stage. Students who would like to incorporate mouse input now must consult with the instructor for more information.