MIPS and Floating Point

COMP273 Assignment 3 - Fall 2015, Prof. Kry

School of Computer Science, McGill University
Available: 29 October 2015
Due date: 11:30 PM, 12 November 2015
Submit electronically via MyCourses

In this assignment, you will use the memory mapped display tool in MARS to animate 3D geometry. The assignment is broken into three parts: basic utilities, line drawing, and then matrix multiplication.

Bitmap display

We will use the default settings of the bitmap display. That is, when you select *Bitmap Display* from the Tools menu, it will have a base address set to the start of static data (0x10010000), and will be 512 pixels wide and 256 pixels high. Do not forget to press the *Connect to MIPS* button so that the display to works with your program.

The top left corder of the memory mapped bitmap display corresponds to the first memory address (0x10010000) with one word (four bytes) encoding the pixel colour. The top byte of the word is unused, while the lower three bytes provide the red, greed, and blue components of the pixel's colour (e.g., red is 0x00ff0000, green is 0x0000ff00, blue is 0x000000ff, black is 0x00000000, and white is 0x00ffffff).

The memory in the display is in row-major order. That is, the pixel just to the right of the top left pixel will be at 0x10010004, while the pixel just below it in the row below will be at 0x10010800, which is the memory address that comes after all the pixels in the first row (Note that the display is 512 pixels wide, or 0x200, and given 4 bytes per pixel). Suppose we would like to set a pixel to a given colour. Let x be an integer specifying the column (valid values going from 0 to 511 inclusive), and let y be an integer specifying the row (valid values going from 0 to 255 inclusive). To set pixel (x, y) to white, we would store 0x00ffffff at memory location $b + 4(x + wy)$, where b is the base address of the memory mapped display, and $w = 512$ is the width.

With a total of 256 rows, or 0x100, the amount of memory needed for the display is 0x80000 bytes (i.e., 512 time 256 times 4). In this assignment, we will want to reserve the memory in the static data segment for not only the display, but an equal amount of memory to allow us to draw first into an *off-screen* buffer, and then later copy that memory into the memory mapped buffer. To reserve the space, place the following labels and directives at the top of your assembly file.

```
.data  # start data segment with bitmapDisplay so that it is at 0x10010000
.globl bitmapDisplay # force it to show at the top of the symbol table
bitmapDisplay:   .space 0x80000 # Reserve space for memory mapped bitmap display
bitmapBuffer:    .space 0x80000 # Reserve space for an "offscreen" buffer
width:           .word 512 # Screen Width in Pixels
height:          .word 256 # Screen Height in Pixels
```

Note that later in the assignment, you will use data provided in a separate file, and you will need to select *Assemble all files in directory* under the Settings menu. As this other file will also have declarations to put data into the data segment, we need to worry that the bitmapDisplay label ends up in the right place. To ensure that MARS correctly concatenates the data segments, you need only ensure that you press the *assemble* button when you have your main assignment asm file open. If you were to hit the assemble button with the auxiliary data file open, the data in that file would appear first in the data segment. If you have doubts, please check the list in the *Labels* window in the *Execute* tab that opens after you assemble. The *.globl* directive is included above not to make the bitmapDisplay label to be visible in other asm files (you will only be writing one), but instead to make it show near the top of the list in the *Labels* window.

1 Utility functions (5 marks)

Several simple utility functions will be needed for drawing with an off-screen buffer and a bitmap display. Implement the following functions. Note that these functions, being small and simple and not needing to call any other functions, will not need to use the stack.

`clearBuffer( int colour )`

> This function takes the clear colour, and sets every pixel in the **off-screen** *bitmapBuffer* to be this colour. You may find that partial loop unrolling (i.e., setting more than just one pixel inside the loop) will make your function faster by reducing the total number of instructions necessary to get the job done.
>
> For testing, consider changing the *Display height in pixels* to 512 instead of 256 so that the bitmap display shows both your normal display buffer, as well as the off-screen display buffer in the lower half of the window.

`copyBuffer()`

> This function performs a memory copy. It should copy all pixels from the off-screen buffer to the on-screen buffer. Again, partial loop unrolling may improve performance.

`drawPoint( int x, int y )`

> This function takes x and y coordinates of a pixel as signed integers, and sets the given pixel in the **off-screen** buffer to a colour. The colour choice is arbitrary, as will be the choice of the colour you use to clear the buffer. Be sure to use contrasting colours. Setting pixels white after clearing black would be one good example among many.
>
> The drawPoint function must do bounds checking on the input parameters. Use `sltu` to simultaneously check lower and upper bounds of the x coordinate, and similarly for the y coordinate. If either is out of bounds, your funciton should do nothing so as not to overwrite memory that is not part of the display.

2 Line Drawing (5 marks)

With the basic utility functions of screen clearing, off-screen to on-screen buffer copying, and drawing points, we now want to be able to draw lines. You will use an algorithm similar to Bresenham's line drawing algorithm, which is an efficient integer based solution using only addition and subtraction for determining which pixels need to be set to draw a line between two points. The algorithm assumes that the line has a slope less than one, and that the starting x position (column) $x0$ is less than or equal to the end column $x1$. With these assumptions, the problem is reduced to a simple loop where the x position is stepped one pixel at a time from $x0$ to $x1$, while the y position is periodically increased depending on how far the current pixel is from the line. The trick is to keep track of a measure of the distance, or the error. After drawing pixel (x, y), the question is if pixel $(x + 1, y)$ is closest to the line, or if it is farther from the line than $(x + 1, y + 1)$.

To have a useful line drawing case that works for all lines, of any slope, and not assuming that $x0 < x1$, we can write a simple function that gracefully deals with all cases. See the end of this assignment for the code for `drawLine( int x0, int y0, int x1, int y1)` which will step x and y in either positive or negative pixel increments depending on the input. Implement this function and have it call your drawPoint call. You should not use any multiplication or division in your implementation. Test your code by drawing some different lines, and note that you can even draw lines that have an endpoint that is off-screen thanks to the bounds checking you do in drawPoint.

3 Matrix multiplication (5 marks)

To lines that represent 3D geometry on our 2D bitmap display, we will need a matrix vector multiplication. Specifically, the matrix will be a 4-by-4 matrix of floating point numbers, and the 4 component vector will be the *homogeneous* representation of a 3D point (x, y, z), which is simply the vector $(x, y, z, 1)$.

`mulMatrixVec( float* matrix, float* vec, float* out )`

> This function takes 3 pointers as parameters, the first being a 4-by-4 matrix is row-major order, and the second and third being 4 component vectors. Given that the size of the matrix and vectors is fixed, you might consider implementing this function without using loops! As you will want to test your code, consider making sample matrices and vectors to multiply, such as the following.
>
> ```
> testMatrix: .float
>      1  2  3  4
>      5  6  7  8
>      9 10 11 12
>     13 14 15 16
> testVec1: .float 1 0 0 0
> testVec2: .float 0 1 0 0
> testVec3: .float 0 0 1 0
> testVec4: .float 0 0 0 1
> testResult: .space 16
> ```
>
> Multiplying `textMatrix` by `testVec1` will result in a vector equal to the first column of the matrix, that is, (1,5,9,13). Note the use of a .space directive to reserve memory for storing the answer in `testResult`, specifically 4 floats, each being 4 bytes, or a total of 16 bytes of space reserved. Remember that you can use the code presented in class for printing a 4 component float vector to check your results.

4 Geometry animation (5 marks)

Given that you have completed the other parts of the assignment, you are now ready to drawn and animate a teapot. The 3D line data for the teapot is in the provided in the `lineData.asm` file. This file contains the labels `LineCount` and `LineData`. The line count is the number of lines. For each line there are 8 floating point numbers, specifically, the two endpoints of the line, where each point is represented as a four component vector (i.e., in homogeneous coordinates).

To draw the lines on the bitmap display, you will transform the 4D end point vectors into (x,y) display coordinates, and then send these 2D endpoints to your line drawing function. The following matrix provides a perspective projective suitable for drawing the teapot.

```
M: .float
331.3682, 156.83034, -163.18181, 1700.7253
-39.86386, -48.649902, -328.51334, 1119.5535
0.13962941, 1.028447, -0.64546686, 0.48553467
0.11424224, 0.84145665, -0.52810925, 6.3950152
```

To compute the 2D display point from 4D line end-point p, compute the product Mp, and take the first two components divided by the last. That is, if $(x, y, z, w) = Mp$, then our display coordinates are $(x/w, y/w)$. The reason for this division is that we would like points which are far to appear smaller on the screen, and it is this w component that will contain the distance of the point after multiplication by the matrix.

While this matrix was prepared specially for this assignment, it is actually the product of a number of very simple matrices (the details are covered in COMP 557, Fundamentals of Computer Graphics).

Note that you will naturally use `div.s` for floating point division, but you will need to convert the floating point values to integers to use with your line drawing code!

`drawTeapot()`

> Write a function that will loop over all the line data, transform the points, and draw the lines. As this function will do looping and call your matrix multiplication and line drawing functions, you will need to use the stack! Feel free to write other helper functions (for instance, to compute screen space (x, y) integer coordinates from the result of a matrix vector multiply).
>
> Test your drawTeapot function by either viewing the off-screen buffer by increasing the display height, or by using the clear buffer and copy buffer utility functions from the first part of this assignment.

Animation

To animate the teapot, you can repeatedly draw the teapot, with a transformation applied to all the points between each drawTeapot call. The following 4-by-4 homogeneous matrix represents a small rotation about the z axis.

```
R: .float
 0.9994 0.0349 0 0
-0.0349 0.9994 0 0
 0      0      1 0
 0      0      0 1
```

`rotateTeapot()`

> This function loops through all the line data transforming each point by the matrix `R`. Note that multiple calls to this function will have a cumulative effect, that is, we can call rotateTeapot to increase the total rotation of the teapot by a small amount on each drawing pass.

You should now create a main function to produce an animation. In a loop, repeatedly call clear, copy, draw, and rotate. You may make an endless loop and let the program run forever, or instead run the loop a fixed number of times (e.g., 30) before doing a syscall to terminate the program.

Submission Instructions

All work must be your own, and must be submitted by MyCourses. **Include your name and student number in your source files**. If you are submitting multiple files, use a **zip archive** to bundle your submitted files (do not use other types of archives). You should not include any directory structure in your submitted zip file. You need not submit the provided lineData.asm file. Be sure to **check your submission** by downloading your submission from the server and checking that it was correctly submitted. You will not receive marks for work that is incorrectly submitted.

```
void drawLine( int x0, int y0, int x1, int y1 ) {
   int offsetX = 1;
   int offsetY = 1;
   int x = x0;
   int y = y0;
   int dX = x1 - x0;
   int dY = y1 - y0;
   if ( dX < 0 ) {
        dX = -dX;
        offsetX = -1;
   }
   if ( dY < 0 ) {
       dY = -dY;
       offsetY = -1;
   }
   drawPoint( x, y );
   if (dX > dY) {
       int error = dX;
       while (x != x1) {
           error = error - 2*dY;
           if (error < 0) {
               y = y + offsetY;
               error = error + 2*dX;
           }
           x = x + offsetX;
           drawPoint(x,y);
       }
   } else {
       int error = dY;
       while (y != y1) {
           error = error - 2*dX;
           if (error < 0) {
               x = x + offsetX;
               error = error + 2*dY;
           }
           y = y + offsetY;
           drawPoint(x,y);
       }
   }
}
```