24-780 Engineering Computation Problem Set 04

You need to create a ZIP file (It may appear as a compressed folder in Windows) and submit the ZIP file via the 24-780 Canvas course. The file name of the ZIP file must be:

PS04-YourAndrewID.zip

For example, if your Andrew account is *hummingbird@andrew.cmu.edu*, the file name must be:

PS04-hummingbird.zip

If your ZIP file does not comply with this naming rule, you will automatically lose 5% credit from this assignment. If we are not able to identify who submitted the file, you will lose another 5% credit. If we finally are not able to connect you and the submitted ZIP file, you will receive 0 point for this assignment. Therefore, please make sure you strictly adhere to this naming rule before submitting a file.

The ZIP file needs to be submitted to the 24-780 Canvas course. If you find a mistake in the previous submission, you can re-submit the ZIP file with no penalty as long as it is before the submission deadline.

Notice that the grade will be given to the final submission only. If you submit multiple files, the earlier version will be discarded. Therefore, if you re-submit a ZIP file, the ZIP file MUST include all the required files. Also, if your final version is submitted after the submission deadline, late-submission policy will be applied no matter how early your earlier version was submitted.

Make sure you upload your Zip file to the correct location. If you did not upload your assignment to the correct location, you will lose 5%.

The ZIP file needs to include:

- C++ source file (ps4-1.cpp)
- C++ source file (ps4-2.cpp)

Submission Due: Please see Canvas.

START EARLY!

Unless you are already a good programmer, there is no way to finish the assignment overnight.

Make sure your program can be compiled with no error in one of the compiler servers. Don't wait until the last minute. Compiler servers may get very busy minutes before the submission deadline!

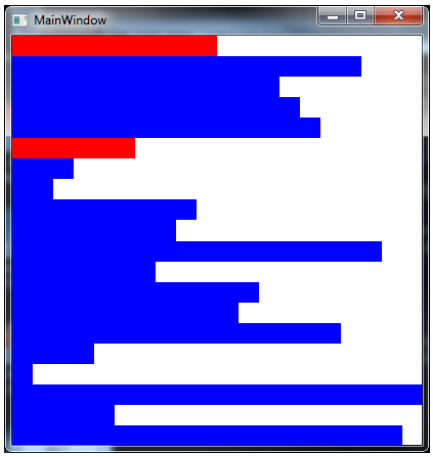
PS4-1 Visualization of Bubble Sort (ps4-1.cpp)[30pts]

In PS4-1, you write a program that visualizes how the bubble-sort algorithm sorts the values in an array. Your program should open 400x400 pixel window and visualize intermediate state the array contents during bubble-sorting by a stack of blue rectangles, except the lines that are about to be swapped which must be drawn by red rectangles. Each rectangle must be 20 pixels high, and the width must be 20 times the value of an item stored in the array in pixels.

Every time the lines are swapped, the program shows the current state (just before swap) and wait for a key stroke.

Complete your program by filling and correcting VisualizeArrayContents() and SwapInt() functions of the following program. Also modify BubbleSort() function so that the lines about to be swapped are drawn by red rectangles. In the final picture, all lines must be drawn by blue rectangles. The program must be saved as ps4-1.cpp and included in the Zip file.

```
#include <stdio.h>
#include "fssimplewindow.h"

void VisualizeArrayContents(int n,int x[],int movedIndex1,int movedIndex2)
{
    (You fill this function)
}

void SwapInt(int a,int b)
{
    (You fill this function.  Also correct errors.)
}

void Show(int n,int x[],int toBeMoved1,int toBeMoved2)
{
    FsPollDevice();
    while(FSKEY_NULL==FsInkey())
    {
        glClear(GL_DEPTH_BUFFER_BIT|GL_COLOR_BUFFER_BIT);
        VisualizeArrayContents(n,x,toBeMoved1,toBeMoved2);
        FsSwapBuffers();

        FsPollDevice();
        FsSleep(10);
    }
}

void BubbleSort(int n,int x[])
{
    // (Modify this function so that the lines about to be swapped are drawn red)
    int i,j;
    for(i=0; i<n; i++)
    {
        for(j=i+1; j<n; j++)
        {
            if(x[i]>x[j])
            {
                Show(n,x,0,0);
                SwapInt(x[i],x[j]);
            }
        }
    }
    Show(n,x,-1,-1); // You can leave this line as is. You'll see the first line red in the end.
}

int main(void)
{
    int x[20]={17,10,13,14,15,6,3,2,9,8,18,7,12,11,16,4,1,20,5,19};
    FsOpenWindow(16,16,400,400,1);
    BubbleSort(20,x);
    return 0;
}
```



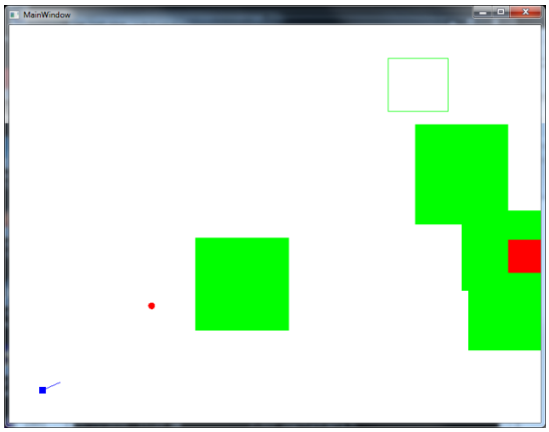
PS4-2 Cannonball Game (ps4-2.cpp) [70pts]

When you start the program, it opens a graphics window of the size 800x600 pixels. This window represents an 80x60-meters rectangular region on a vertical plane. The origin of the physical coordinate system is at the lower left corner of the window -- (0,600) --, and 1.0 meter in the physical coordinate system corresponds to 10 pixels in the window coordinate system.

In the window are the following objects drawn with different colors:

- a cannon (blue square and a piece of line),
- a cannonball (a red circle),
- a target (a red square), and
- five obstacles (five green rectangles).

The goal of the game is to hit the target using the smallest possible number of cannonballs. If there is no clear path to hit the target, you may hit an obstacle to destroy it, but you waste one cannonball. The game ends when you hit the target. A cannonball is subject to the gravitational force, but there is no other force applied to the cannonball – this makes the trajectory a perfect parabola.



Cannon is placed near the lower left corner, 5 meters to the right from the left edge and 5 meters up from the bottom edge, and initially points to the right 30 degrees up from the horizon. The direction of the cannon continuously rotates by 1 degree counter clockwise every time the screen is re-drawn. When the angle reaches 90 degrees (straight up), the angle needs to be brought back to 0 degree (horizontal).

You shoot a cannonball by pressing the space bar (FSKEY_SPACE). The initial speed of the cannonball is 40 m/sec. The trajectory of the cannonball should be animated. Your program needs to calculate the trajectory by a numerical integration scheme, such as Euler's method. Use time step of 25 milliseconds. Use FsSleep(25); every time you draw the picture so that the motion of the cannonball on the screen is roughly real time. The aim should keep moving for the next shot while the cannon ball is flying, however,

the user cannot shoot another cannonball until the ball disappears. The cannonball must disappear when it goes below the bottom edge or to the right of the right edge of the window.

When the game starts, the program places five rectangular obstacles at random locations. The width and height of each obstacle should range between 8m and 15m, randomly selected. Note that the effective size of the obstacle needs to comply with this requirement. If a part of an obstacle extends outside of the window, the effective size may become smaller.

Obstacles must be drawn by green filled rectangles initially. When the ball hits an obstacle, the obstacle is destroyed, and the destroyed obstacles must be drawn by green empty rectangles.

The target should be located initially at x=75m, y=60m, and its size is 5m x 5m. It moves downward at 10 meters per second. It can move through the obstacles, i.e., you don't have to consider collision between the target and an obstacle. Since the time step is 25 milliseconds, the target moves 0.25m each step. When it reaches the bottom of the window, it jumps back to y=60m. It can overlap with the obstacles, but the target should always be visible.

In PS4-2, you write a C++ program called ps4-2.cpp, which must be included in the Zip file.

Assuming that the time required for the numerical integration, managing key strokes, and drawing all the objects is negligible, let your program sleep 25 millisecond each iteration so that the program simulates the motion of the ball roughly real time.

You can cut & paste some functions from the sample code used in class to reduce the coding time.

Extra point: If you can draw a trajectory of the cannonball behind it, you get extra 5 point. Trajectory must be calculated by a numerical integration scheme. The trajectory must be drawn behind the ball, and stay visible until the user shoots the next cannonball.