PH30056 Comp Phys B – Monday 1 February 2021 – **Single-particle code with openGL**

This handout *should* be complete in the sense that if you follow the instructions then everything will work. You should end up with a moderately complicated program that makes a particle move around the screen at random. You *do not* need to understand every detail of this program – the idea is to get a sense of how to build up a program from scratch.

0. Setup

We will start from the last openGL program at the end of the session 1 handout. If you completed that handout then you should have a program which consists of a *Window* class (that involves two files `Window.h` and `Window.cpp`). Also a main program file that includes the `main` function, a global pointer to a *Window*, etc. The contents of these three files are on moodle (*glMulti.cpp* etc).

Assuming that most people didn't finish that handout then here is a summary of how to get to this point (mostly copy-pasted from the session 1 handout)

- Once Visual Studio 2017 (VS) is open, create a *New Project*. It will ask for the project type – select *Empty Project Visual C++*. You will need to select a sensible name for the project (perhaps *myProject01*) and a location in your user area where you want to store it. This information goes in boxes near the bottom of the window. Now, go to the *Project* menu and click *Add New Item*, and select *C++ File (.cpp) Visual C++*.
- Import openGL libraries, as follows
  - go to: *Project* $\rightarrow$ *Manage nuget packages.* a window will open.
  - where it says "search" type "nupengl" (this is the graphics package, do not include the quotes)
  - click on nupengl.core in the left panel of the window
  - check that the description in the right panel says "this library adds the various libraries to your project..."
  - click the Install button in the right panel
- Copy-paste *glSimple.cpp* from Moodle into the main window of visual studio. Compile using 'Build Solution' in the Build menu. Run using 'Start Debugging' from the Debug menu (or using the Play button in the toolbar). You should see a window with a string in it. Hit q or e within that window to quit.

At some stage you should go back and work carefully through the first handout. There are quite a lot of explanations in there about this program. But I suggest for now you carry on with this session.

1. A one-particle system (see Moodle file part1.txt)

We are going to overwrite that C++ project in order to do something more interesting. If you need to reconstruct it then you can do so following the previous instructions.

The first job is to add a new function to the *Window* class, so that the user can draw a square. You need to add the declaration to the header file and the definition to the cpp file (see moodle). Once you have done that, go to the project menu and select Build Solution, to check that it still compiles.

Next, add a new class to your project called *systemClass*. This is going to represent the physical system that we are modelling. (Use the Add Class option from the Project menu.) A header file and a cpp file will appear in the Solution Explorer. Within the header file, edit the class definition by adding a declaration for a Window pointer (`Window *win;`). Also add some coordinates for a particle that will live in the system (`double posX; double posY;`). Add a variable for the particle size (`double pSize`). You also need to add a line `#include "Window.h"` at the beginning of *systemClass.h* so that the systemClass "knows about" the Window class. Also, add in your main file `#include "systemClass.h"`. See moodle for hints on this part.

Add a declaration for a `drawSystem` function within `systemClass.h` (the output type should be void and it takes no inputs). Also add a declaration for a constructor that takes a Window pointer as input. Write a definition for the constructor in `systemClass.cpp`, it should set the Window pointer within the class to the pointer that is input to the constructor. So now the system "knows what window it is in". It is sensible to set `posX` and `posY` to zero within the constructor, and to set `pSize` to 0.05 (see later). Again, see moodle.

The definition of the `drawSystem` function is provided on moodle, copy this into `systemClass.cpp`. This will draw a square of size `pSize` at position `posX,posY`. You also need to move the piece of code

```
namespace colours {
...
}
```

from the main file to the systemClass.cpp file (put it near the beginning, before any function definitions). You also need to add a line

```
#include <gl/freeglut.h>
```

near the beginning of systemClass.h, to be sure that the systemClass knows about openGL.

At this point the program will not compile: you need comment out the line `win->displayString(...);` because `colours::red` is now defined only in sytemClass.cpp so the main program "does not know about it". If you turn that line into a comment then the program should compile (check this). Note however that no systemClass objects are created in the code and so nothing appears in the graphics window when you run it.

To make progress, we need to connect up the `systemClass` to our main program. Within the main program file, remove the global window pointer and replace it with a global system pointer `systemClass *sys;` We will use this pointer to access the systemClass.

Within the `main` function, we need to make some changes to create a new instance of the `systemClass`, full details are on moodle. We also need to change the `drawFuncs::display` function so that it uses the global systemClass to draw the particle. Find that function and replace the line `win->displayString(...)` (which you commented out earlier) by `sys->drawSystem();`

With a bit of luck, it should now compile. If so, run it. You should see a square window with a small blue square (particle) in the middle. Hit q or e within that window and the program

should exit. Congratulations!

If it does not compile then there is a mistake. Check the Error List in visual studio, also check to see if it has underlined some parts of your code in red (this is a bad sign). You may want to ask a demonstrator. . .

2. Particle class (see moodle file part2.txt)

Since a particle within the system is a physical object, it makes sense for it to be an object in the code. Create an *inline* class called `Particle`. (You need to tick the "inline" box when you add the class.) The class should include an array of 2 doubles to store the position (call this `pos`). Write a constructor that initialises the position according to an input array. See moodle for a hint.

You need to add `#include "Particle.h"` at the top of *systemClass.h* to be sure that the system knows about the new Particle class.

Now remove the position variables `posX` and `posY` from the systemClass (in *systemClass.h*) and replace them with a pointer to a Particle, using `Particle *currentParticle;` Edit the systemClass constructor to remove `posX` and `posY` and instead initialise `currentParticle=0;` (this is a null pointer, ie there is no particle in the system). Edit the drawSystem function so that it only draws the square if `currentParticle!=0`. Also you need to replace the inputs `posX` and `posY` to the drawSquare function with the position vector of the particle (use `currentParticle->pos`). See moodle for hints.

Compile and run. If all is working then you should see a blank window. (Remember that the constructor set `currentParticle` to be a null pointer so there is no particle in the system any more.)

To get beyond this blank window we need to do some more work. We will add a `createParticle` function to the `systemClass`. This function is given on moodle. It checks if a particle already exists; if there is no particle then it creates a new one. Check that the program still compiles ("Build Solution").

Now, we are going set up interaction with the `systemClass` via the keyboard. Go back to the main program and find the `handleKeypress` function. Add a new case to the switch so that if the user hits 'c' then the program prints a message (eg "create") to `cout` and then calls the `createParticle` function for our system (using `sys->createParticle();`). Remember `sys` is a global pointer to our system. At the very end of the `handleKeypress` function add a line `glutPostRedisplay();` This tells the program that it needs to redraw the window (something may have changed). See moodle for a model answer.

Compile and run. Hit 'c' in the window. A particle should appear(!).

At this point we can try a few things to check that we understand. (For now you may want to keep going and come back to these later.)

- Add a `removeParticle` function to the systemClass. This should check if a particle exists (`currentParticle!=0`). If so then `delete currentParticle;` and then initialize `currentParticle = 0;` if you wish to be able to create a new particle after deleting the old one. Modify the code so that pressing 'd' will remove the particle.
- Add a `moveRight` function to the systemClass. This should check if a particle exists (`currentParticle!=0`). If so then move the particle a distance `pSize` to the right (by modifying the array `currentParticle->pos`). Modify the code so that pressing 'r' will move the particle to the right.
- Try adding a `moveLeft` function associated to the 'l' key. What about a `moveToOrigin`

function that puts the particle back at (0,0).

You are encouraged to experiment and try any other things that you fancy.

3. Random walk (see moodle file part3.txt)

Suppose that we want the particle to hop around at random, in a *random walk* (this will be useful for diffusion-limited aggregation). For this we need a random number generator. To do this, we create an inline class called `rnd`. Copy the contents of *rnd.h* from moodle into the relevant header file (see the instructions for using the class which are at the top). Add `#include "rnd.h"` near the beginning of *systemClass.h*.

Add an instance of the random number generator to the systemClass as `rnd rgen;` You may want to add (for example) `rgen.setSeed(0);` to the systemClass constructor, so that the random seed is initialised to a specific value.

Check that everything still compiles.

In the random walk, the particle will hop at random to a neighbouring position. We are going to set this up in a particular way that will be useful later. On moodle there is a declaration and definition for a function `getNborPosition` which should go in the systemClass. This function takes 4 inputs. The idea is that the function takes a particular position (second input) and calculates the a neighbouring position (either right, left, up or down). The choice of which neighbour is given by the 3rd input (this should be an integer 0,1,2,3) and the distance between neighbouring sites is the 4th input. The neighbouring position that the function calculates is stored in the array that is the 1st input.

Check that you understand what this function does. Example: if `pos` is a position (x, y) then `getNborPosition(newPos,pos,0,0.1)` will set the array `newPos` to contain $(x + 0.1, y)$. (The third input being zero says that newPos should be to the right of pos.)

The next part is a bit more difficult: you need to create a function `moveRandom` that moves the particle to a random neighbour. You can get a random number 0,1,2,3 using `rgen.randomInt(4);` You need to pass this number as the third input to `getNborPosition` in order to calculate the (random) new position of the particle and then you need to update the particle position to this new position. (The first input to `getNborPosition` will be an array `newPos`, the second will be the particle's current position, the third will be a random number and the fourth will be `pSize`)

Try writing this function yourself (there is a model answer on moodle).

If that is working then then you should be able to make the particle make a random hop whenever you hit the 'u' key.

Some more things to try. (For now you may want to keep going to the next part and come back to these later.)

- If you hit 'u' very many times, the particle may hop off the screen. Try adding a check so that if the particle tries to hop to a point that is further than some distance R from the origin, it gets deleted. (You can pick a suitable value for R.) I suggest you include this check inside the `moveRandom` function.
- Try adding another check so that if you hit 'u' and there is no particle in the system then a particle gets created at the origin. Again, include this check inside the `moveRandom` function.
  If you now hit 'u' many times you have a process where particles appear at the origin, move around at random, and eventually disappear.

4. Automated update (see moodle file part4.txt)

Within openGL there is a function called
`glutTimerFunc`
which takes 3 inputs which are an integer (`wait`), a function (`func`) and another integer (`val`). The function `func` should take a single integer as argument. The function `glutTimerFunc` causes openGL to do nothing for `wait` milliseconds, then to call the function `func` with input `val`. This is very useful because we can use it to automate our program.

To do this: Note that a function `void update(int val);` is declared within the `drawFuncs` namespace, although this function is never defined. Put a definition of this function near the bottom of the main program file in a similar way to `drawFuncs::handleKeypress`. The definition is provided on moodle.

Before the `glutMainLoop()` command in the `main` function, add a function call
`glutTimerFunc(10,drawFuncs::update,0);`
This means that once the main loop starts, the program will wait 10ms and then call the update function.

Note also: at the end of the `update` function, we tell openGL to update the screen using `glutPostRedisplay();` and then there is another call to `glutTimerFunc`. So you should imagine that the computer will keep calling this function many times, with some short waiting times in between.

If you run the code you should see that the particle hops around the window. If you did the extra tasks in part 3 then this particle may get deleted if it moves too far from the origin, and a new particle may also be created. This is an automated “simulation” of a random walk. Maybe you can imagine that this general method can be used to simulate more complicated processes as well. . .

5. More exercises... (see moodle file part5.txt)

On moodle there is a function that opens a file for writing, and a function that prints some data to that file, and a function for closing the file. Can you integrate these functions within the `systemClass` so that the position of the particle is printed to the file after every hop? Perhaps change the program so that you press a specific key to close the file and exit? (If you exit without closing the file then some data may be lost.)

Maybe you want to include a counter so that after a certain number of hops then the file gets closed and no more positions are printed (otherwise the file may get very large).

Can you edit the code so that instead of hopping to the four nearest neighbours, it can also hop diagonally? (I suggest you do this by adding extra cases to the switch in the `getNborPosition` function.)

Can you see how to edit the program so that the particle moves with periodic boundaries? (That is, if it leaves the window from the right then it comes back on the left.)

Note also: there are ways to keep track of multiple code versions so that you can go back to a working version if you break something but we are not going to deal with that here. You can always make backup copies of your .cpp and .h files, either by copy-pasting the code or by finding the files themselves, which are in a directory called something like
Visual Studio 2017/Projects/*projectName*/*projectName*/
The repetition is deliberate here, but you need to replace *projectName* with the actual name of your project.

Note however, just copying files into this directory will not add them into your project. That needs some further action within visual studio.

Version history
AS, 1 Feb 2021