Writeup for Homework 1

200333470

yhuang64

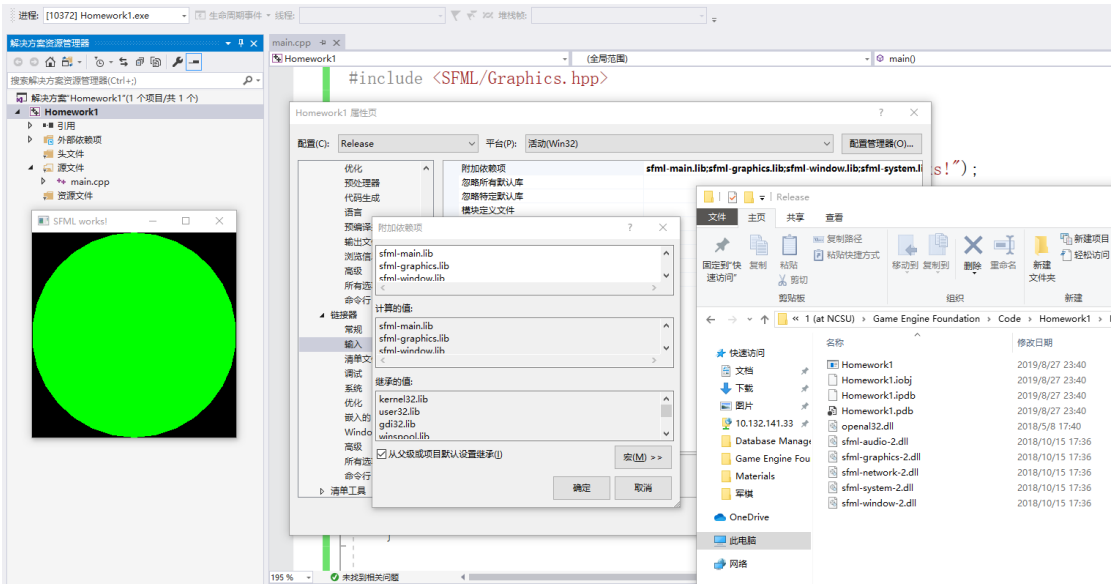
Yuanmin Huang

*To write something before the document begins, I'd like to say that this is my first time to write a "writeup" like this, and it's my first time to write a document like this in English (as I'm an exchange student). So, I'm not sure whether this kind of style is ok for the coming work in the future or needs to be improved. I'd appreciate it a lot if I could receive some guidance for that. *

Section 1

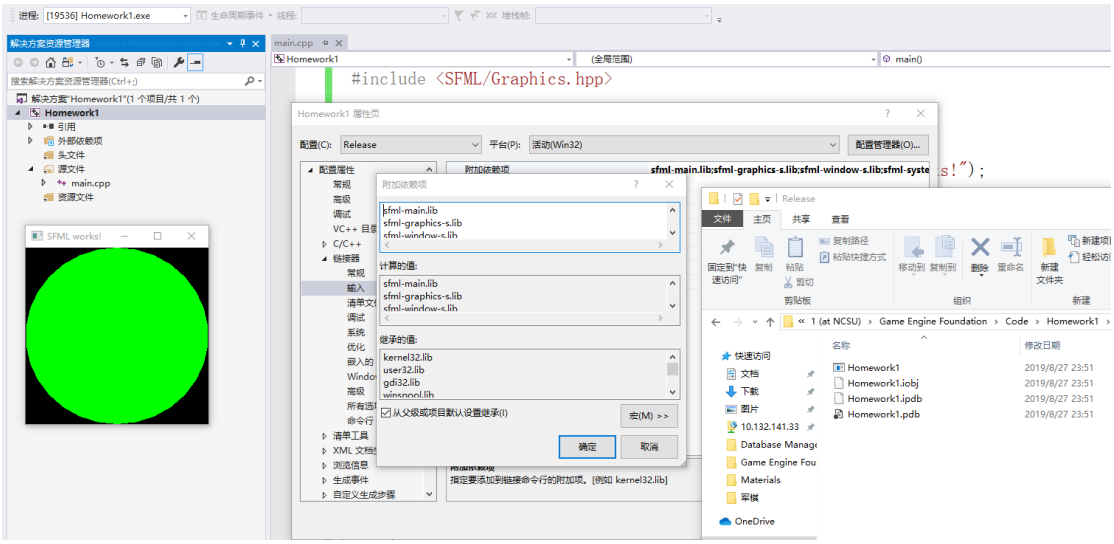
To be honest, I haven't written codes using Visual Studio before. Previously, I used a bunch of IDEs of JetBrains, like Intellij IDEA for Java, CLion for C/C++, PyCharm for Python… After all, it took me some time to create a new project in C++ using VS. Then, the work just became much easier.

To check if I am correct in each step of setting up, I firstly used the dynamic version of linking, which meant that I put all the .dll files into the debug directory and the release one to get the demo run correctly. Here is a screenshot of the running.



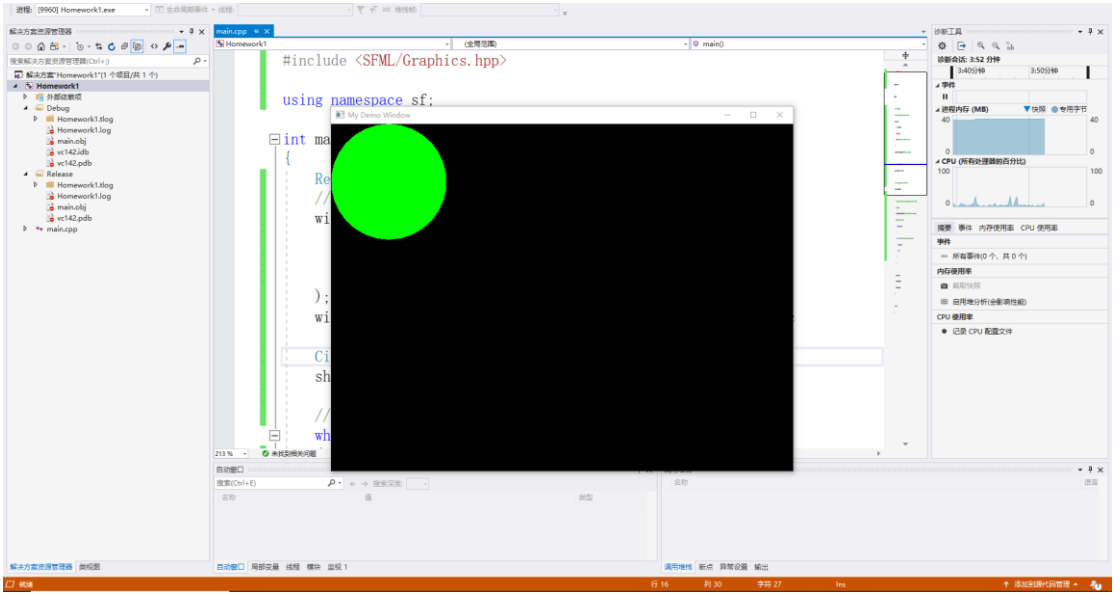
Then, I found it annoying that I need to copy all the .dll files to the working directory every time I start a new project, so I turned to the static version, which asked me to add "-s" to sfml libs and another bunch of libs. After some work, I got the version without .dlls as shown below.

Next, I gone through the tutorial of the window part to optimize the code for a decent window.



In this part, I first split the declaration and creation of the window by using the create function. In this way, I believe that it would be more convenient to insert some code between these two steps in the future. Secondly, I added a line of “using namespace sf;” to avoid typing “sf::” repeatedly every time. Thirdly, I turned on the vertical synchronization in case there would be a need later. Moreover, I changed the code of handling events from an if-statement to a switch-case statement for better expansibility, even though we wouldn’t be using event handling more than close event in this homework.

According to the tutorial, I got a Default (, which is Title bar | Resize | Close,) window with size of 800*600. The result of section 1 is shown below.



Section 2

In this part, I designed and implemented three classes for Platform, Moving Platform and Character respectively. Below is a screenshot of the result I get. Then, I will go through the detail of them one by one.

For platform, which is the green rectangle on the left in the picture. It is an instance of class Platform, which inherits sf::RectangleShape and has a constructor with a parameter of type sf::Vector2f indicating the size of the rectangle.

For moving platform, the right one. Its class inherits Platform. Additionally, it has a function named *around* to keep it moving from left to right and then moving back on the screen.

For character, the one with a diamond shape. Its class inherits sf::ConvexShape. In its constructor, the four points of its shape is set.

For all the three shapes above, they are all textured by images. The textures are loaded by a function defined in main.cpp to be reused. Moreover, they are all Drawable objects as well. Thus, I used a list of Drawable objects to manage the drawing of all shapes in the main loop. In addition, to manage the movable objects similarly, I designed an abstract class Movable to provide attributes and functions concerning movement to be inherited by moving object classes. In this way, I can also use a list to manage all the movable objects.

Besides, I also made some attempts to modulize the main function because it's growing bigger and bigger. I extracted the window initialization part and window event handling part out to form two separate functions.

However, there are still some points can be optimized. For instance, the way of moving of Moving Platforms is hard-coded now, so it's not reusable.

Section 3

This section is relatively simpler than the previous two. I'm leaving the jump instruction to next part because it needs collision detection functionality and a gravity simulation.

Therefore, I just make the class Character inherit class Movable and implement the move left and move right instruction. Then in the main.cpp, the character is also put into the moving objects list so that the function that make it move will be called in the main loop.

Section 4

I have gone through some quite tricky bugs in this section, and it took me about two days to figure out what's going wrong. I was forced to make some changes to my codes so that I could fulfill the requirement of the functionality.

My general idea for the collision detection is that it should tell me if the left, right, up or bottom side of the character has hit something (which is a platform in this scenario). Then, the character should behave according to its current situation, that is to move with the moving platform or to fall if it's not standing on a platform, etc.

However, the first problem I've met with is that the getGlobalBound function only provides a rectangle around the object, instead of telling me which part of the rectangle is intersecting with the other. Hence, I tried to generate four small rectangles simulating four points around the character (as my character has a diamond shape). Then, I use these points' global bounds to check the character's intersection with platforms. The corresponding codes are as follows.

```
// calculate four bounding points
FloatRect box = getGlobalBounds();
Vector2f
    leftBound(box.left, box.top + 0.5 * box.height),
    rightBound(box.left + box.width, box.top + 0.5 * box.height),
    upBound(box.left + 0.5 * box.width, box.top),
    bottomBound(box.left + 0.5 * box.width, box.top + box.height);
// create rectangles for four points
Vector2f point(1.f, 1.f);
RectangleShape l(point), r(point), u(point), b(point);
l.setPosition(leftBound);
r.setPosition(rightBound);
u.setPosition(upBound);
b.setPosition(bottomBound);
```

After that, when it comes to the collision detection and velocity setting part (to make the character move physically properly). I'm having a loop to go through every platform to check whether it is intersecting with the character's one part. If so, the velocity of character would be set correspondingly.

I used to put these two steps together, which caused a weird bug to occur that the character would only stand on the last platform in the platform list and fall through all the other ones slowly. Finally, I realized that I should split the task into two parts: check the collision and set the velocity get from outside.

Also, to get the velocity of the platform, the list of platforms passed to the character must be a list of moving platforms, or the attributes and functions of Movable would be sliced because of the grammar of C++. Thus, I must change the normal platform object into a moving platform object with a velocity of zero. Semantics is a little bit given up implementing the work.

When the physics are all done, it's time to implement a jump for character. My setting is that only when the character is standing on something, it can jump, and when it jumps, it gets an initial velocity of -10 on axis y. So, it would jump up, reach to the highest point and fall faster and faster because of the gravity added to it in each iteration of the loop (which is 0.5 on

```
if ((Keyboard::isKeyPressed(Keyboard::Up) || Keyboard::isKeyPressed(Keyboard::W))
    && bottom != nullptr) // character should be on a platform to jump
{// jump
    outVelocity.y = -10.f;
}
```

axis y in my implementation). My code on this part is shown above.

At the meantime, in the main.cpp, I stepped further to make it a map of textures and their names to manage the texture resources better. Also, I made a separated function that loads textures and puts them into the map, trying to make the main function look clearer.

I've tried to split the initialization of platforms from the main function as well but failed because the objects would become temporary variables if they were initialized in other functions, which made it hard to manage them by global lists.

Section 5 (Optional)

For this part, I referred to the View Tutorial on Github.com. I understood how the Window and View work together to show the specific part of the whole scene.

In addition, I learned from the official tutorial that I could capture the Resized event to customize the behavior of resizing the window.

Then things become easy. It is obvious that in the proportional mode (default), objects are zoomed with the window zoomed because the size of the view of the window isn't changed when the size of the window is changed; while in the constant mode, objects remain unchanged when the window zooms, which is because the size of the view also changes when window zooms.

According to the official tutorial, we can simply make a new view of event.size under constant mode. However, I have found out that there is a tricky point. When I first drag to make the window bigger under P (proportional) mode and then switch to C (constant) mode and make the window smaller, the view will first become bigger (under P), then shrink to normal when it reacts to the resizing event under C, which I think is unnatural.

In my opinion, the adjustments under P mode should have influence on all the coming resizing tasks in either mode. To implement this, I found a equation: previous window size / previous view size = new window size / new view size. In my code implementation, the previous window size is maintained by a Vector2u variable; the new window size would be event.size; the previous window size can be obtained by window.getView().getSize(). Thus, the new view size can be calculated from these variables.

Finally, concerning the real-time input, I made ctrl + C as C mode, ctrl + P as P mode. Moreover, I made ctrl + R as returning to the normal window (with size of 800 * 600).