

Writeup for Homework 3

200333470

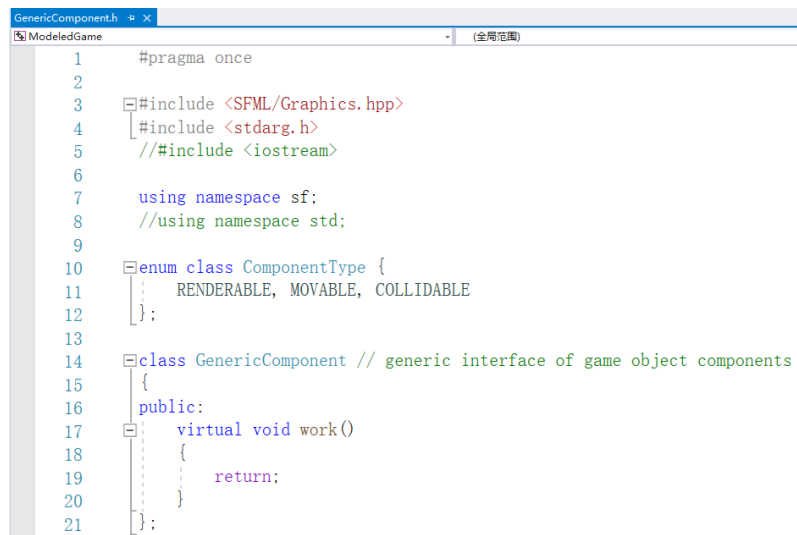
yhuang64

Yuanmin Huang

Section 1

In the first part of this homework, I decided to implement a model which is close to a generic component model. In my design, I have a bunch of components that provide different categories of functionalities and a bunch of game objects composite a map of generic components (component type and pointer) to utilize the functionalities of components. I said my model is close to a generic model because some game objects may need some extra self-defined attributes other than the pre-defined components, which makes the model not purely a generic component model.

For the components, I first designed a GenericComponent class to be inherited by all the other components. The header file contains nearly nothing except for an enumerate class of type of components and a virtual function work() to define different behaviors of each component. The class structure is as shown below.

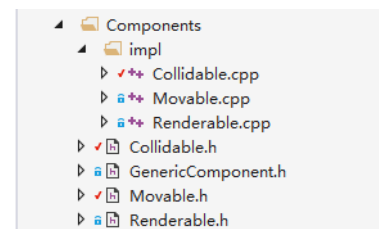


```
1  #pragma once
2
3  #include <SFML/Graphics.hpp>
4  #include <stdarg.h>
5  // #include <iostream>
6
7  using namespace sf;
8  // using namespace std;
9
10 enum class ComponentType {
11     RENDERABLE, MOVABLE, COLLIDABLE
12 };
13
14 class GenericComponent // generic interface of game object components
15 {
16 public:
17     virtual void work()
18     {
19         return;
20     }
21 };
```

At first, I wanted to design an extensible function work() using ... parameter of C++. Later, I found it too complicated and not practical to do so, so I turned to implement overload functions in subclasses instead of override ones so that I could have changeable parameters.

As is shown above, I have three kinds of components:

1. Renderable, which is responsible for defining an object's shape and color. I provided three shapes: rectangle, diamond and circle and three colors: red, green, and blue. They are pre-defined enumeration elements which can be chosen by users when



constructing objects. Within this component, the size of the shape can also be decided by users.

2. Movable, which is responsible for the movement of objects. Thus, the work() function within this class is used for moving. Obviously, to move an object, the class composites a pointer to a Renderable object to gain access to the object to be moved. To indicate which specific type of moving the object wants to implement, I defined HORIZONTAL, VERTICAL and KEYINPUT three ways. The work() uses a switch-case to decide which actual move() function would be called to move the object. As for the three types, KEYINPUT simply deals with character movements using keyboard input, while the other two deals with back and forth movement in either horizontal or vertical direction of platforms. To make it more extensible, the speed, start point and range of the movement can be defined by users easily by passing corresponding arguments to the constructor.
3. Collidable, which is responsible for dealing with the collision of objects. The work() function here works more like detectCollision() in previous homework. Again, a collidable object needs to be renderable and may need to be movable (for platforms and characters). Therefore, there are pointers of Renderable and Movable in this class. An enumeration class with CHARACTER, PLATFORM, BOUNDARY and DEATHZONE is defined to indicate the actual type of the game object compositing this component, which would be used to lead to different behaviors when collision is detected. I only implemented the function for PLATFORM here, and the other two would be implemented in section 3.

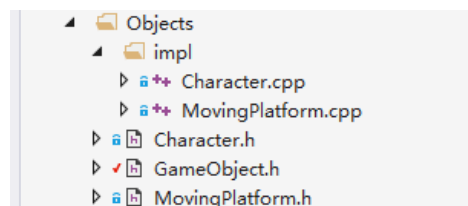
```

91     for (Collidable* object : objects)
92     {
93         // get bound of object
94         FloatRect bound = object->render->getShape()->getGlobalBounds();
95
96         if (cbound.intersects(bound)) { // collision happens
97             switch (object->collision)
98             {
99                 case Collision::PLATFORM:
100                     platformWork(object, bound, boundary_lines);
101                     break;
102                 case Collision::DEATHZONE: // implement in section 3
103                     break;
104                 case Collision::BOUNDARY: // implement in section 3
105                     break;
106                 default:
107                     break;
108             }
109         }
110     }

```

Then, for the game objects, similarly, I designed a GameObject class to composite the common map from ComponentType to GenericComponent* and functions to add and get GenericComponents.

MovingPlatform and Character are the only two classes of game objects till now (, static platforms can be implemented using moving platform of velocity zero). They all composite the three components designed above to implement the functionality they need. Specifically, Character has a vector of four Collidable pointers to indicate if the character is having collision with any collidable objects in all four ways. The pointer to the vector would



be passed to the character's Collidable component to implement the functionality of work().

```

GameObject.h
1  #pragma once
2  #include <map>
3  #include "../Components/GenericComponent.h"
4
5  using namespace std;
6
7  class GameObject
8  {
9  private:
10     map<ComponentType, GenericComponent*> gcs;
11 public:
12     GameObject() {} ;
13
14     GameObject(map<ComponentType, GenericComponent*> gcs) { this->gcs = gcs; }
15
16     GenericComponent* getGC(ComponentType type)
17     {
18         auto pair = gcs.find(type);
19         return pair != gcs.end() ? pair->second : nullptr;
20     }
21
22     void addGC(ComponentType type, GenericComponent* component)
23     {
24         gcs.insert({ type, component });
25     }
26 };

```

Originally, I made the constructor of game objects relatively simple, which only receives pointers of pre-constructed components and stores them by calling addGC() of GameObject. However, this makes the construction of game objects looks quite heavy in main.cpp.

```

63 // init character
64 Renderable cr(::Shape::DIAMOND, ::Color::BLUE, Vector2f(60.f, 120.f), Vector2f(200.f, 100.f));
65 Movable cm(&cr, Vector2f(250.0f, 0.0f), gameTime, Move::KEYINPUT);
66 Collidable cc(Collision::CHARACTER, &cr, &cm);
67 Character character(&cr, &cm, &cc);
68 dynamic_cast<Collidable*>(character.getGC(ComponentType::COLLIDABLE))->setBoundaryPtrs(character.getBoundaryPtrs());
69 objects.emplace_back(dynamic_cast<Renderable*>(character.getGC(ComponentType::RENDERABLE))->getShape());
70 movingObjects.emplace_back(dynamic_cast<Movable*>(character.getGC(ComponentType::MOVABLE)));

```

I felt that this part should be encapsulated into game objects as it is part of the construction of the object. So, I refactored the constructors and got the code as follow.

```

62 // init character
63 Character character(
64     ::Shape::DIAMOND, ::Color::BLUE, Vector2f(60.f, 120.f), Vector2f(200.f, 100.f),
65     Vector2f(250.0f, 0.0f), gameTime
66 );
67 objects.emplace_back(dynamic_cast<Renderable*>(character.getGC(ComponentType::RENDERABLE))->getShape());
68 movingObjects.emplace_back(dynamic_cast<Movable*>(character.getGC(ComponentType::MOVABLE)));

```

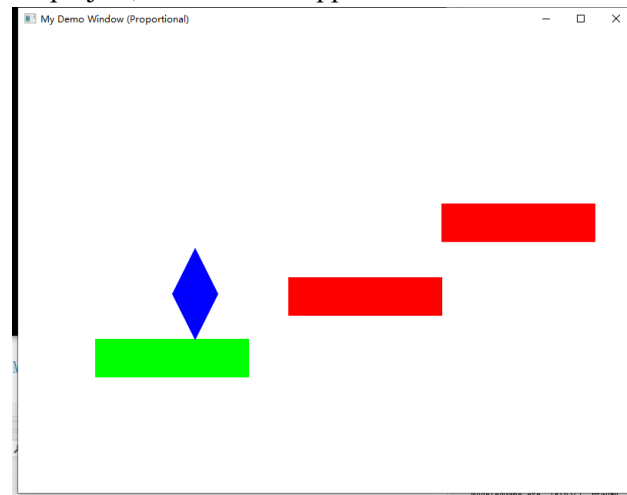
The new constructor looks like this. The similar refactor was done to MovingPlatform as well.

```

11 Character::Character(::Shape shape, ::Color color, Vector2f size, Vector2f pos, Vector2f velocity, Timeline& timeline)
12     : GameObject(), boundary_ptrs({ nullptr, nullptr, nullptr, nullptr })
13 {
14     this->addGC(
15         ComponentType::RENDERABLE,
16         new Renderable(shape, color, size, pos)
17     );
18     this->addGC(
19         ComponentType::MOVABLE,
20         new Movable(
21             dynamic_cast<Renderable*>(this->getGC(ComponentType::RENDERABLE)),
22             velocity, timeline, Move::KEYINPUT
23         )
24     );
25     this->addGC(
26         ComponentType::COLLIDABLE,
27         new Collidable(
28             Collision::CHARACTER,
29             dynamic_cast<Renderable*>(this->getGC(ComponentType::RENDERABLE)),
30             dynamic_cast<Movable*>(this->getGC(ComponentType::MOVABLE))
31         )
32     );
33     // set boundary ptrs to collidable
34     dynamic_cast<Collidable*>(this->getGC(ComponentType::COLLIDABLE))->setBoundaryPtrs(&boundary_ptrs);
35 }

```

After you run the project, the effect is supposed to be like the following picture.



Section 2

This section is relatively easy because the most important design for networking has already been completed in last homework. What I need to do in this section is to put the modeled design of game objects and the networking communication design together.

Thus, firstly, I substituted the original game object package of the server/client project with the new object package of the modeled game project and then modified the original implementation of networking package to fulfill the requirements of communication.

Afterwards, what's new this time is to implement a graceful disconnection operation. My design is to let a disconnecting client send a different message ("CLIENT-NAME D") to server to indicate that it's disconnecting before it closes the window. Then, the server reset the stored character pointer of that client to NULL so that when the publisher of server broadcast message to all the other clients next time, it can notify everyone that one client has disconnected (, using "C CLIENT_NAME D"). In this way, the figure of the disconnecting client on other clients' windows can be removed properly. After completing all these things, the server erases the pair of information of that client in the map of client name to character pointer to get ready for the next connection.

In conclusion, the most important design of my client/server model lies in:

1. The disconnection functions in both client end and server end, which directly deal with the disconnection behavior.

```

91 void Client::disconnect()
92 {
93     // prevent from sending messages out
94     connected = false;
95
96     // generate message accordingly
97     string message = (string)CLIENT_NAME + " D";
98
99     // send message and receive response
100     s_send(sender, message);
101     string response = s_rcv(sender);
102 }

106 void Server::disconnectHandler(const string& name)
107 {
108     // find and release the character pointer,
109     // reset the second value to nullptr
110     auto pair = characters.find(name);
111     delete pair->second;
112     pair->second = NULL;
113
114     // send response back to client
115     s_send(receiver, "Disconnected");
116
117     cout << "Client " + name + " disconnected" << endl;
118 }

```

2. The modified part of the publication and subscription function dealing with disconnections in server and client.

The server publication function would send disconnecting messages when

encountering pairs with character pointer is NULL. After sending the message, the

```

81     string deleted = "";
82     // generate clients message
83     for (auto pair : characters)
84     {
85         message += ClientMessage(pair.first, pair.second);
86     }
87     if (!pair.second) // disconnected client, delete it
88     {
89         //characters.erase(pair.first);
90         deleted = pair.first;
91     }
92 }
93 // delete pair of disconnected client after notifying everyone
94 if (characters.find(deleted) != characters.end())
95 {
96     characters.erase(deleted);
97 }

```

pair would be deleted from the map.

The client subscription function would then delete the stored pair of client name and character position when receiving the disconnection message. Accordingly, the disconnected character would no longer be drawn on the window.

Section 3

What I have done for this section is mainly some add-ons to the object model I have designed for section 2.

I implemented SideBoundary, SpawnPoint and DeathZone for this section.

SideBoundary is a class designed to define a side boundary in the game. My goal is to set two boundaries at the left and right side of the window. They are both 100 units far away from the side (this value can be set to whatever reasonable value by game designers). When a character collides with a side boundary, it will then add a value of offset to the variable render offset in main.cpp. The value added is computed based on the given window size, how far the side boundary is from the side of the window and which side is the collided boundary on. The render offset is later used to add to the position of objects other than the character of the client before being drawn. In this way, I made my client move to “next” view relatively. Of course, when a collision happens, the side boundaries should also “move” with the character, which means I would add the negative offset to all the side boundaries. Actually, what I have described above is mainly implemented in the work() function of Collidable component.

```

35     void Collidable::sideBoundaryWork(Collidable* sideBoundary,
36     Vector2f& renderOffset, vector<SideBoundary*>* sideBoundaries)
37     {
38         //cout << "hit side boundary" << endl;
39         SideBoundary* boundary = dynamic_cast<SideBoundary*>(sideBoundary->getGameObject());
40
41         // add offset this time to overall offset
42         Vector2f offset = boundary->getOffset();
43         renderOffset += offset;
44
45         //cout << "offset " << offset.x << " " << offset.y << endl;
46         //cout << "render offset " << renderOffset.x << " " << renderOffset.y << endl << endl;
47
48         // update the position of all the sideBoundaries
49         for (SideBoundary* boundary : *sideBoundaries)
50         {
51             boundary->updatePos(offset);
52         }
53     }

```

The design of SpawnPoint is quite easy because it needs nothing, but a position defined in Renderable component. (I still make it have a Renderable component even though it needn't to be rendered.)

Finally, the DeathZone is an object with Renderable, Collidable. When a character collides with it, the character should be transferred back to a random spawn point in the list in my design, which is quite plain.

However, when I was implementing my design on the original generic component model, I encountered big problems which pushed me to refactor the model. The problem happens when I want to cast a pointer to a SideBoundary type in Collidable.cpp when implementing the work() function. I found that I needed to include the header file of SideBoundary if I wanted to do so. Whereas, SideBoundary had included Collidable.h, which made it become a recursive include. It made me impossible to link the cpp files together when compiling.

This made me to reconsider my design. I realized that I should composite a pointer of the game object in my game component so that I could gain access to the data I need when implementing some of the behaviors. My original design asked me to keep several attributes of all the game objects may be possible to composite the component in the corresponding component, which I found is inextensible and redundant.

Although the new approach couldn't prevent me from recursive including, I could at least remove all the redundant attributes in my components and no longer need to worry about the inconsistency of attributes values in components and corresponding game objects.

As for the recursive including, I googled the problem and found out that it can be solved by declare a incomplete definition of game object class in component's header file and then include the header file of objects in cpp files of components.

```
3  #include <SFML/Graphics.hpp>
4  #include <stdarg.h>
5  #include <iostream>
6
7  using namespace sf;
8  using namespace std;
9
10 class GameObject;
11
12 enum class ComponentType {
13     RENDERABLE, MOVABLE, COLLIDABLE
14 };
15
16 class GenericComponent // generic interface of game object components
17 {
18 protected:
19     GameObject* gameObject;
20 public:
21     GenericComponent(GameObject* gameObject)
22     {
23         this->gameObject = gameObject;
24     }
25 }
```

As a result, I designed a game with size of 1600 * 600. Four death zones stand around the area, with which character collides would cause a transfer back above the static platform in the first view. In the second and third view, there is a horizontally moving platform and a vertically

moving platform respectively. The window size is $800 * 600$, and the padding of side boundaries from the side of window is 100. Thus, in my design, every time a character collides with a boundary will cause a 600 change in render offset.

Moreover, after I started to run the whole program, I found that there were some instants that when the character was in the second view, the static platform in the first view emerged. I tried to print out the value of the render offset because I thought that it would be caused by render offset being modified unexpectedly, but I found nothing wrong. Until I realized that this was caused by read/write conflicts of multithreading. There were surely be moments when the main thread was rendering the window, the position of the platforms were modified by `subscribeHandler()` of the client. Therefore, I added a mutex to prevent these conflicts and protect the platforms from being modified when the main thread was drawing them. After that, everything moves as expected.

Section 4 (Optional)

Though I may not have the time to implement this section, I thought something about it.

For the networking model, one way to implement it would be only transferring the position of objects and generate the actual object accordingly on the other end. The other way would be use json or other libraries to transfer a whole serialized object and only needs decoding on the other end.

Then what I should do would be creating objects and run the program to measure the time used under different number of objects and under different transfer model.