UNIVERSITY OF CALIFORNIA, MERCED

BERKELEY · DAVIS · IRVINE · LOS ANGELES · MERCED · RIVERSIDE · SAN DIEGO · SAN FRANCISCO

SANTA BARBARA · SANTA CRUZ

CSE 165: Introduction to Object Oriented Programming

Final Project Template Documentation

Angelo Kyrilov
Computer Science and Engineering
University of California, Merced

Contents

1 Introduction

This document provides an overview as well as detailed explanations of all aspects of the final project template. It is a 2D graphical application that allows easy incorporation of sprites in order to create simple games. It also illustrates important object oriented programming practices such as abstraction, encapsulation, inheritance, polymorphism, private inheritance, friendship, and object based hierarchies.

2 Project Structure

The main file of the project appears below:

```
#include "App.h"

int main(int argc, char** argv) {
    GlutApp* app = new App(argc, argv, 1280, 800, "Exploding Shroom");

    app->run();
}
```

2.1 The `GlutApp` class

In the main file, we simply create a new instance of `App` and upcast it as a `GlutApp` pointer. Here is the interface of the `GlutApp` class:

```
1  class GlutApp {
2  public:
3      void run();
4      void redraw();
5      void toggleFullScreen();
6
7      virtual void draw() const = 0;
8
9      virtual void keyDown(unsigned char, float, float){}
10     virtual void keyUp(unsigned char, float, float){}
11     virtual void specialKeyDown(int, float, float){}
12     virtual void specialKeyUp(int, float, float){}
13
14     virtual void leftMouseDown(float, float){}
15     virtual void rightMouseDown(float, float){}
16     virtual void leftMouseUp(float, float){}
17     virtual void rightMouseUp(float, float){}
18     virtual void drag(float, float){}
19
20     virtual void idle(){}
21
22     virtual ~GlutApp(){}
23 };
```

From the interface above, we can see that a `GlutApp` can run, it can redraw the scene and it can switch between full screen and windowed mode. We will skip the `draw` method in line 7 for now but will return to it soon.

The `GlutApp` can also handle all manner of user interactions, including `keyUp`, `keyDown`, `specialKeyUp`, `specialKeyDown`, `leftMouseDown`, `leftMouseUp`, `rightMouseDown`, `rightMouseUp`, and `drag`. The names of these functions are all self-explanatory.

There is also an `idle` function, which is called repeatedly all the time, and there is a virtual destructor, as it should be.

We now come back to line 7 and the pure virtual method `draw`. Since the method is a pure virtual one, then the whole `GlutApp` class is an abstract class. This means that it can not be instantiated. In order to use the `GlutApp` class, we need to create a child of it and implement the pure virtual methods, in this case `draw`.

2.2 The App class

Why would we want to do that? We are forcing the client programmer (whoever is using this template) to provide a `draw` method. We take care of all the other bits and pieces, such as defining the keyboard and mouse callbacks, but we do not specify a default `draw` behavior, we leave that task to the user. This is why the `GlutApp` pointer in the main function, actually points to an instance of `App`. Here is the interface of `App`:

```
1  class App: public GlutApp {
2
3      std::vector<AppComponent*> components;
4
5      void addComponent(AppComponent* component){
6          components.push_back(component);
7      }
8  public:
9
10     App(int argc, char** argv, int width, int height, const char* title);
11
12     void draw() const;
13
14     void keyDown(unsigned char key, float x, float y);
15
16     ~App();
17 };
```

The first thing we can see is that `App` is a child of `GlutApp`. Looking at just the public section for now, we have the appropriate constructor. We actually can not define a copy constructor or a default constructor because those have been defined as private in `GlutApp`.

In line 12, we are fulfilling the requirement of defining a function named `draw` so the class `App` is not abstract. There will be an implementation of `draw` in `App.cpp`.

In line 14, we can see that this application will only handle keyboard events with the `keyDown` method. Since we do not need to handle the other user interactions, se simply do not define the other callback

functions. The app will just use the default behavior of `GlutApp`.

In line 16, there is a destructor, so that we can free up all the memory taken up by this object.

Speaking of which, we can look at the private part of the class. We can see that the app just manages a collection of things called `AppComponent`. More on that later. For now, we can take a look at the implementation of `App`.

```
1  App::App(int argc, char** argv, int width, int height, const char*
       title): GlutApp(argc, argv, width, height, title){
2
3      addComponent(new Game());
4      addComponent(new Ad("See the best Android phones. Press 1.", "
           http://apple.com"));
5
6  }
7
8  void App::draw() const {
9      for (std::vector<AppComponent*>::const_iterator i = components.
           begin(); i != components.end(); ++i) {
10         (*i)->draw();
11     }
12 }
13
14 void App::keyDown(unsigned char key, float x, float y){
15     if (key == 27){
16         exit(0);
17     }
18     else if (key == 'f'){
19         toggleFullScreen();
20     }
21     else{
22         for (std::vector<AppComponent*>::iterator i = components.
               begin(); i != components.end(); ++i) {
23             (*i)->handleKeyDown(key, x, y);
24         }
25     }
26 }
27
28 App::~App(){
29     for (std::vector<AppComponent*>::iterator i = components.begin();
           i != components.end(); ++i) {
30         delete *i;
31     }
32     std::cout << "Exiting..." << std::endl;
33 }
```

In the constructor, lines 1-6, we can see that this app has two components. One is a `Game`, and the other is an `Ad`. They have both been allocated to the heap by the appropriate constructors, and added to the collection of components.

The function responsible for displaying stuff to the screen, lines 8-12, simply iterates over each component and calls its `draw` method. Now, someone might say: "But wait a minute, what if `Ad` does not have a `draw` method?". That is not a problem because `Ad` is guaranteed to have a `draw` method. As you will see later, `Ad` (and `Game` for that matter) is a child of `AppComponent` and it is forced to have a `draw` method. It kind of makes sense if we are creating a graphical application, we are requiring things to be able to draw themselves.

The function for handling keyboard events, lines 14-26, we handle the case for the escape key, which has ASCII value 27, in which case the app exits. We also handle the "F" key to switch between full screen and windowed mode. If any other key is pressed, we just tell all the components in our collection: "Hey, this key was pressed, can you handle it?". This is an illustration of the notion of separation of concerns. The concern of the function is to detect when a keyboard event occurs, and to tell us what key was pressed. There should be no game logic there for obvious reasons. If tomorrow we decided to stop using GLUT in favor of something like Qt, we will have to replace the files `GlutApp`, and `App`. When that happens, if there is game logic in the `App` file, then that logic will be lost. The way it is written now, there is no danger of that happening.

Finally, we have a destructor, lines 28-32, where each `AppComponent` is deleted, meaning that the heap memory taken by the component is released.

2.3 The `AppComponent` class

It is now time to look at the `AppComponent`. Here is a listing:

```
1  class AppComponent {
2  public:
3      virtual void draw() const = 0;
4
5      virtual void handleKeyUp(unsigned char, float, float){}
6      virtual void handleKeyDown(unsigned char, float, float){}
7
8      virtual void handleSpecialKeyDown(int, float, float){}
9      virtual void handleSpecialKeyUp(int, float, float){}
10
11     virtual void handleLeftMouseDown(float, float){}
12     virtual void handleRightMouseDown(float, float){}
13     virtual void handleLeftMouseUp(float, float){}
14     virtual void handleRightMouseUp(float, float){}
15
16     virtual void handleDrag(float, float){}
17
18     virtual ~AppComponent(){}
19 };
```

As you can see, an `AppComponent` is able to do all the things listed above. Specifically and implementation of `draw` is required, and all the handlers are optional. What I am basically saying with this file is: "If you want to put your component in my app, I will assume that it can do all of the above." I have provided default behaviors (do nothing) for all the handlers, but you have to implement the `draw` function, because we need to know how to draw your component and there is no obvious default way for me to do it. That's why I force you to provide it.

2.4 The `Ad` class

Now, let us look at one such component, namely `Ad`. As the name suggests `Ad` is simply meant to display some text and when the user presses a specific key (we could have also implemented it with a mouse click), the system opens up a web browser to a specific URL. (Disclaimer: this functionality does not work in Windows, but it does in UNIX-based operating systems.) Here is a listing of the `Ad` interface:

```
1  class Ad: public AppComponent {
2      const char* text;
3      const char* url;
4      TextBox* textbox;
5
6  public:
7      Ad(const char* text, const char* url);
8
9      void draw() const;
10     void handleKeyDown(unsigned char, float, float);
11
12     ~Ad();
13 };
```

We can immediately see that `Ad` is a child of `AppComponent`, which means it is an `AppComponent`. We have the necessary `draw` function, as well as the optional `handleKeyDown`.

All the ad logic is private, meaning that the client programmer does not need to be concerned with it. We are handling it at a `TextBox` with a specific message in it. There is also a variable to store the URL that we need to visit when the user performs the appropriate action. Here is also the implementation of everything in `Ad`:

```
1  Ad::Ad(const char* text, const char* url){
2      textbox = new TextBox(text, -0.3, 0.9);
3      this->url = url;
4  }
5
6  void Ad::draw() const {
7      textbox->draw();
8  }
9
10 void Ad::handleKeyDown(unsigned char key, float x, float y) {
11     std::string url_s(url);
12
13     url_s = "open " + url_s;
14
15     if (key == '1'){
16         system(url_s.c_str());
17     }
18 }
19
20 Ad::~Ad(){
21     delete textbox;
22 }
```

The constructor, lines 1-3 simply initializes the variables. The `draw` method, lines 6-8, displays the `TextBox`. The keyboard handler, lines 10-18, opens up the URL in the event that the "1" key was pressed by the user (does not work in Windows but whatever). Finally, the destructor, line 20-22, releases the memory occupied by the textbox.

2.5 The `Timer` class

Up to this point in the project description, we have not talked about any animation, as all the contents we have seen were stationary. Animation is achieved by manipulating the state of the scene objects (such as position or size), and redrawing the scene. This needs to happen often, and more importantly in a controlled manner, so that animations can be smooth and consistent. Have you ever played an old PC game on much newer hardware and found that the game runs impossibly fast, to the point that it is not playable. The game Commandos: Behind Enemy Lines is the most high-profile game I can think of, which suffers from this flaw.

In order to avoid this, we need to be able to measure time precisely. If we use the GLUT `idle` function, it is not precise and will cause the flaw described above. On a faster system, it will get called more often than on a slower system. To address this issue, use timers. There is a timer construct offered by GLUT but it is not working well, especially if you have multiple moving objects controlled by different timers.

In this project, we have a timer that is extremely precise and it is multi-threaded, meaning that if we have multiple timers in our application, they run on different threads, taking advantage of the fact that all our hardware is multi-core. We do not explicitly worry about that, we only hope that the OS will take advantage of it. All we do is we specify that we want more threads. Here is a listing of the `Timer` class:

```
1  class Timer {
2      int interval;
3      bool running;
4      int delta;
5      int initial;
6      std::thread timerThread;
7      static void repeat(Timer*);
8  public:
9      Timer();
10
11     void start();
12
13     void setRate(int);
14
15     void stop();
16
17     ~Timer();
18
19     virtual void action() = 0;
20 };
```

Looking at just the public section, we can see that the timer can start, stop, and set a rate in milliseconds. There is a pure virtual method called `action`, which is what the timer should do at every time interval. Once again, it doesn't make sense to guess a default behavior here, so we will force the client programmer to provide the action.

Here is a listing of the implementation of `Timer`:

```
1  void Timer::repeat(Timer* self){
2      while (self->running){
3          int tss = glutGet(GLUT_ELAPSED_TIME);
4
5          self->delta = tss - self->initial;
6
7          int delay = self->interval;
8          if (self->delta >= self->interval){
9
10             self->action();
11             glutPostRedisplay();
12
13             self->initial = glutGet(GLUT_ELAPSED_TIME);
14         }
15         std::this_thread::sleep_for (std::chrono::milliseconds(self->
               interval/2));
16     }
17
18 }
19
20 Timer::Timer() {
21     interval = 1000;
22     running = false;
23     initial = glutGet(GLUT_ELAPSED_TIME);
24     delta = initial;
25 }
26
27 Timer::~Timer(){
28     if (running) timerThread.detach();
29 }
30
31 void Timer::stop(){
32     if (running) timerThread.detach();
33     running = false;
34 }
35
36 void Timer::setRate(int mills){
37     interval = mills;
38 }
39
40 void Timer::start(){
41     running = true;
42     timerThread = std::thread(repeat, this);
43 }
```

Looking at line 44, we can see that when the timer starts, it spawns a new thread and runs the `repeat` function in there. The `repeat` function simply executes the `action` function at every interval, and it refreshes the display. It precisely measures the time elapsed between invocations of `action` and ensures that the interval is always obeyed, with at most 1 millisecond mismatch, which is not noticeable to the eye.

2.6 The `Game` class

The `Game` class is the most complex construct in this project as it is responsible for visualizing a flying mushroom as well as a cannon at the bottom of the screen. When the user pushes the spacebar, the cannon is fired directly upwards. If the projectile collides with the mushroom, then the mushroom explodes and falls to the ground as it is burning.

First, let's take a look at the interface of `Game`:

```
1  class Game: public AppComponent, private Timer{
2      TexRect* mushroom;
3      Rect* projectile;
4      AnimatedRect* explosion;
5      bool projectileVisible;
6      bool mushroomVisible;
7      bool up;
8      bool left;
9      bool hit;
10     float theta;
11     float deg;
12 public:
13     Game();
14
15     void draw() const ;
16     void handleKeyDown(unsigned char, float, float);
17
18     void action();
19
20     ~Game();
21
22 };
```

The first thing we see here, on line 1, is that `Game` is a child of two different classes. That's right, C++ supports multiple inheritance. More interestingly, the inheritance from `AppComponent` is public, while the inheritance from `Timer` is private. This means that `Game` *is-a* `AppComponent`, so any instance of `Game` can be upcast to an `AppComponent`. This is not true of `Timer`. The `Game` is not a `Timer`. The relationship between the two is that `Game` *is-implemented-in-terms-of* `Timer`, which means that `Game` has a private instance of `Timer`, that can be used for implementing whatever `Game` needs.

At this point, one might ask: "Why don't we just use composition?". Remember that `Timer` is an abstract class because its `action` method is pure virtual. This means that we can not create an instance of `Timer`. We would first have to create a child of `Timer`, say for example `GameTimer` and we would have to create an instance of it inside `Game`. The problem with that, other than the fact it's more work, is that `GameTimer` does not have access to the private variables in `Game`, so we will have to give it access to them, which is even more work. Private inheritance solves all this in a very elegant way.

From looking at the public section, we can see that `Game` knows how to draw itself, it knows how to handle a key pressed by the user, and it has an action, which specifies what happens every time interval (the animation stuff). As usual, there is also a destructor, whose job it will be to release all the memory held by the game objects.

Looking at the private section, we can see that there are going to be 3 objects in the scene, namely `mushroom`, `projectile`, and `explosion`. They are not all visible at the same time but they are all in memory. The data types have been chosen specifically to illustrate the different objects in the `Shape` hierarchy (more on that later). In addition to the shapes, there is some game logic stored by variables from lines 5-11.

2.7 The `Shape` object-based hierarchy

Shapes in a graphical application have a lot in common. They all have to be positioned somewhere on the scene, which means they have coordinates. They also have a size, and they all should have the ability to draw themselves.

This is an object-based hierarchy for shapes with rectangular regions. Their regions are specified by the xy-coordinates of the top left corner, the width, and the height.

2.7.1 The `Rect` class

The most basic rectangular shape is `Rect`. In addition to the position and size variables described above, it also stores its own color in three variables for red, green, and blue respectively. As convention dictates, there are getters and setters for all the instance variables. There is also a `contains` method, which will be used for collision detection, and as it is required by the `Shape` abstract class, `Rect` has a `draw` method. Here is a listing of the class, notice the instance variables are protected so that any children (descendents) of `Rect` would be able to use these "private" variables, while no one else would.

```
1  class Rect: public Shape{
2  protected:
3      float x;
4      float y;
5      float w;
6      float h;
7
8      float r;
9      float g;
10     float b;
11
12 public:
13     Rect(float=0.0f, float=0.0f, float=0.4f, float=0.2f, float=1.0f,
           float=1.0f, float=1.0f);
14
15     float getX() const;
16     float getY() const;
17     float getW() const;
18     float getH() const;
19
20     float getR() const;
21     float getG() const;
22     float getB() const;
23
24     void setX(float);
25     void setY(float);
```

```
26      void setW(float);
27      void setH(float);
28
29      void setR(float);
30      void setG(float);
31      void setB(float);
32
33      bool contains(float, float) const;
34
35      void redrawScene();
36
37      virtual void draw() const;
38
39      virtual ~Rect();
40  };
```

2.7.2 The `TexRect` class

This is a shape that can load a texture from a file and display it in its rectangular region. All we need to do is inherit from `Rect`, and redefine the `draw` function, as it now should specify texture coordinates in addition to vertices. Here is a listing of the code.

```
1  class TexRect: public Rect {
2  protected:
3      GLuint texture_id;
4
5  public:
6      TexRect(const char*, float, float, float, float);
7
8      void draw(float z) const;
9  };
```

The only thing different is in addition to all the instance variables of `Rect`, we now also have, on line 3, a pointer to the texture. The implementation of `TexRect` takes in a filename, and uses the SOIL library to load the image as an OpenGL texture. The `draw` function maps the texture coordinates to the rectangle coordinates.

2.7.3 The `AnimatedRect` class

Another thing that would make our graphical applications better is to have objects in the scene that are not static. This is achieved by having the sprite switch between different pre-defined frames. This is how the explosion is implemented in this project. Typically a sprite sheet will be divided into a number of frames and the scene object should switch between them at given time intervals so as to simulate motion. Here is the code for the `AnimatedRect` class:

```
1  class AnimatedRect: public TexRect, protected Timer {
2      int rows;
3      int cols;
```

```
4       int curr_row;
5       int curr_col;
6       bool complete;
7       bool loop;
8       bool animating;
9       bool visible;
10      void advance();
11      bool done();
12      bool flipped;
13      void action();
14      const char* filename;
15  public:
16      AnimatedRect (const char*, int, int, int, bool, bool, float,
            float, float, float);
17      void setMap(const char*, int, int);
18      void draw(float z);
19      void playLoop();
20      void playOnce();
21      void reset();
22      void pause();
23      void resume();
24      void play();
25      void flip();
26      ~AnimatedRect();
27  };
```

When we create an `AnimatedRect`, we need to specify the filename where the sprite sheet is, then with two `int` variables the number of rows and columns in the sprite sheet. This is followed by another int to specify the time interval for the animation, the next constructor argument is a `bool` to specify if the `AnimatedRect` should be visible when upon creation, and the next `bool` specifies whether the animation should start playing when the object is created. Finally, we have 4 floats, that specify the position and the size of the `AnimatedRect`, the same way as usual.

3 Guidlines

In order to create a meaningful graphical application, all you need to do is to create an `AppComponent` and add it to the `App` object, which will display it for you and will let you know about all the user interactions that have occurred. How you handle them is up to you.

We have already seen the interface of the `Game` class, so the rest of this document is devoted to its implementation. As mentioned earlier, it is a very simple game that features one moving target, a mushroom, and a stationary cannon at the bottom, displayed as as simple rectangle. When the user presses the spacebar, the projectile from the cannon is shot upwards, and if it collides with the mushroom, then the mushroom blows up and falls to the ground as it is still burning.

To get started, we look at the constructor of the game.

```
1  Game::Game(){
2
3      // Some cross-platform compatibility stuff
4
5      const char* shroomFileName;
6      const char* fireballFileName;
7
8      // In Windows (Visual Studio only) the image files are found in
           the enclosing folder in relation to the project
9      // In other environments the image files are in the same folder
           as the project
10
11     #if defined WIN32
12     shroomFileName = "../mushroom.png";
13     fireballFileName = "../fireball.bmp";
14     #else
15     shroomFileName = "mushroom.png";
16     fireballFileName = "fireball.bmp";
17     #endif
18
19     mushroom = new TexRect(shroomFileName, -0.25, 0.5, 0.5, 0.5);
20     projectile = new Rect(-0.05, -0.8, 0.1, 0.1);
21     explosion = new AnimatedRect(fireballFileName, 6, 6, 64, false,
           false, -0.25, 0.8, 0.5, 0.5);
22
23     up = false;
24     left = true;
25     projectileVisible = true;
26     mushroomVisible = true;
27     theta = 0;
28     deg = 0;
29     hit = false;
30
31     setRate(1);
32     start();
33 }
```

Everything above line 18 is just so that the code will compile unmodified on Windows Visual Studio, as well as other platforms.

On line 19 we instantiate the mushroom object with an appropriate constructor call. This is followed by a call to the constructor for the projectile object in line 20, and finally, we instantiate the explosion object in line 21. Notice that upon creation, the explosion object is not visible and it is not animating, since both boolean parameters are given as `false`. We then set the state variables to meaningful initial values, lines 23-29. We then set the interval for the timer to repeat every 1 millisecond, which is incredibly brief but it results in smooth animation. Feel free to experiment with this setting to see if you can call it less often while still having smooth animations. The last line of the constructor, line 32, is a call to start the timer, meaning that whatever we specified in the `action` method, will be repeated every millisecond.

We then look at the `action` method, which as the name suggests, is where all the action is.

```
1  void Game::action(){
2      float mx;
3      float my;
4
5      if (theta >= 2* M_PI) theta = 0;
6
7      mx = 0.5 * cos(theta);
8      my = 0.5 * sin(theta);
9
10     mushroom->setX(mx - mushroom->getW()/2);
11     mushroom->setY(my + mushroom->getH()/2);
12
13     theta += 0.001;
14
15     if (!hit && up){
16         float ypos = projectile->getY();
17         ypos +=0.005;
18         projectile->setY(ypos);
19
20         if (mushroom->contains(0, ypos-0.005)){
21             up = false;
22             hit = true;
23             projectileVisible = false;
24             mushroomVisible = false;
25             explosion->setX(mushroom->getX());
26             explosion->setY(mushroom->getY());
27             explosion->playOnce();
28         }
29     }
30
31     if (hit){
32         explosion->setY(explosion->getY()-0.0001);
33     }
34 }
```

This timer action is responsible for several things. The first one is to move the mushroom in a circular motion around the origin. The second one is, after the occurence of a keyboard event, to start moving the projectile upwards. If and when the projectile collides with the mushroom, the mushroom object disappears from view, and the explosion becomes visible, and plays once. As the explosion is playing, it is also reducing its y-coordinate, to simulate the fact that it has been shot down and it's falling due to gravity.

The first sub-action, moving the mushroom in a circular fashion is implemented in lines 2-13. Remember `theta` is a state variable initialized at 0. Every time `action` is executed (which is once every millisecond), the value of `theta` is increased by 0.001. This happens in line 13. If theta gets to 2π, that's the same as 0, so we reset it to 0 at that point, which happens in line 5. Now we want the mushroom to trace a circle around the origin, and the radius of that circle is 0.5. This is hard-coded in lines 7 and 8, which are the lines responsible for calculating the xy-coordinates of where the mushroom should go in the next frame. These coordinates, are then sent to the mushroom, in lines 10 and 11. Notice that for the x-coordinate we subtract half the width of the mushroom, and for the y-coordinate we add half the height. This is done so that the center of the mushroom is at the calculated coordinate, not its top-left corner.

Lines 15-29 contain an `if` statement, which will evaluate to `true` when the user presses the spacebar. That basically sets the `up` variable to true. Inside the `if` statement, lines 16-18 are responsible for moving the projectile upwards.

Line 20 is an if statement that detects the collision between the projectile an the mushroom, and lines 21-27 specify what needs to happen on that collision. One of the things to happen is that the variable `hit` becomes `true`. This means that from now on, the `if` statement on line 31 will evaluate to true, and the explosion will decrease its y-coordinate as it is going through its frames.

The next function we have is the `draw` function of `Game`. It is responsible for displaying the objects on the screen, and it only does so it the right conditions.

```
1  void Game::draw() const {
2      if (projectileVisible){
3          projectile->draw();
4      }
5      if (mushroomVisible){
6          mushroom->draw(0.0);
7      }
8      explosion->draw(0.1);
9  }
```

For completeness, we also provide the code for the keyboard handler:

```
1  void Game::handleKeyDown(unsigned char key, float x, float y){
2      if (key == ' '){
3          up = true;
4      }
5      else if (key == 'p'){
6          stop();
7      }
8      else if (key == 'r'){
9          start();
10     }
11 }
```

As you can see from the code, in addition to responding to a spacebar, the game also responds to "P", which stops the timer and the animation freezes, and "R" which starts up the timer again, so the animation continues.

Finally, the destructor of `Game`:

```
1  Game::~Game(){
2      stop();
3      delete mushroom;
4      delete explosion;
5      delete projectile;
6  }
```