Project 1 – Simple Artillery Game
CSIS3700 – Dr. Shaffer
Due: 3/31/2015

---

1 Introduction

In this project you will create a simple artillery game. In this game play will alternate between two players. Each player has a tank with a muzzle. On a player's turn they select the angle of their muzzle (up and down arrow keys) and the velocity of their shell (left and right arrow keys). Once the player is happy with the angle they press space to fire the shell which may or may not destroy the opponent's tank.

Below I will describe some of the considerations you will need to make as **you** design your solution.

2 Images

I recommend starting with hand-drawn images of the following:

- Tank base (without muzzle) around 50x50 pixels
- Tank muzzle oriented horizontally...this should be a short and wide image around 50x5 pixels
- A background image which includes a relatively flat "ground". I recommend increasing the size of your window from 800x600 to 1024x768 (or choose something more appropriate for the aspect ratio and resolution of your monitor). The background image should have dimensions that correspond to your selected size.
- A "missed" explosion image...the shell exploding when it hits the ground, missing the opponent's tank
- A "hit" explosion image...the shell exploding when it hits the opponents tank

3 The world

Sprites will be coming and going from the world. As a simple example, when an explosion is added to the world it will want to remove itself after a certain time has passed (I used 2 seconds). The world needs to have methods to support this. In this particular example I added:

- `void begin_explosion(float x, float y, bool hit)` – creates the appropriate kind of explosion sprite at the specified location. Updates the world state to reflect that an explosion is occurring. (See below for discussion of the world state.)
- `void end_explosion(explosion_sprite *)` – remove the supplied explosion sprite. Update the world state so that it is the next player's turn or the game is over.

You may have several messages like these to control sprites coming and going from the world. **Do not allow sprites to add and remove arbitrary sprites from the world. Your game will become unmanageable.** Finally, messages like those above complicate your loops in the world. Consider the world's `advance_by_time` method. As it loops over the sprites:

```
void world::advance_by_time(double dt) {
  for(vector<sprite*>::iterator it = sprites.begin(); it != sprites.end(); ++it)
    (*it)->advance_by_time(dt);
}
```

the list of sprites may change. This will cause significant problems with the iterator. The solution is to modify this method so that it loops over a **copy** of the sprite list:

```
void world::advance_by_time(double dt) {
  vector<sprite*> copy;
  copy = sprites;
  for(vector<sprite*>::iterator it = copy.begin(); it != copy.end(); ++it)
    (*it)->advance_by_time(dt);
}
```

In this game the world moves through a sequence of states in a predictable fashion. I recommend enumerating these states and explicitly checking the state during event handling. The states I used were:

- aiming
- shell in air
- explosion animation
- Game over (one of the tanks destroyed)

I recommend either defining an `enum` to represent the state or create a state-related classes (more advanced approach). The world will need to have an instance varibale, `currentState`, which it will use to make decisions during event handling. I also have an instance variable to track which player is taking their turn.2

Finally don't be surprised if the world needs to treat some of the sprites in a special way. That is, it will still need to keep a list of sprites so that they animate properly etc. but it may also what to keep separate pointer variables to specific important sprites (the tanks, for example).

4 Sprites

You will need specialized sprite subclasses:

- `tank_sprite`
- `shell_sprite`
- `missed_explosion_sprite`
- `hit_explosion_sprite`

Unlike your first animation, these sprites will need to interact with the world. For example, a `shell_sprite` will need to know if it hit the other tank (just use the width and height of the shell and the tank and their positions to determine if a hit occurred). This means that these sprites will need to have a pointer to the world in an instance variable.

5 Tank Muzzle

As the player aims, the angle of the muzzle should change. Allegro supports drawing a rotated bitmap via `al_draw_rotated_bitmap`. See the documentation of that function for details.

6 Motion of the shell

At each time step the shell's position after that time step $(x(t + \Delta t), y(t + \Delta t))$ depends on its position and velocity before that time step $(x(t), y(t))$ etc. Similarly for the velocities. They should be updated with the following:

$$\begin{aligned} x(t + \Delta t) &= x(t) + v_x(t)\Delta t \\ y(t + \Delta t) &= y(t) + v_y(t)\Delta t \\ v_x(t + \Delta t) &= v_x(t) + a_x\Delta t \\ v_y(t + \Delta t) &= v_y(t) + a_y\Delta t \end{aligned}$$

The accelerations should be constant:

$$\begin{aligned} a_x &= 0 \\ a_y &= 300 \end{aligned}$$

The **initial** velocities are determined by the angle θ and velocity v selected by the user:

$$\begin{aligned} v_x(0) &= v\cos\theta \\ v_y(0) &= v\sin\theta \end{aligned}$$

At each time step the shell sprite should check to see if it hit the ground or the enemy tank. In either case the sprite should ask the world to delete it and replace it by the appropriate explosion.

7 Text

During the aiming phase, the player needs some feedback about they direction and velocity of their shell. Display text showing them this information. In addition it is helpful to display text indicating the current active player (whose "turn" it is) and when the game ends to display "GAME OVER" in a large font.

First select and download font files for the muzzle information and the GAME OVER display. To draw text you must make sure that you initialize the font add-on via `al_init_font_addon`. If your font is a TrueType font, you will also need to call `al_init_ttf_font`. Add these calls to `main`. Next, you'll need to load the fonts from the files using `al_load_font`. This can be done in `world` who also should keep pointers to the return fonts. Finally, to display text on the screen use `al_draw_text` or `al_draw_justified_text`. See the documentation for details.

8 To hand in

Submit a `zip` file of your complete solution (all source code) on the submission system.