PA6: Zombie Apocalypse (Due 11/17 @ 11:59PM)

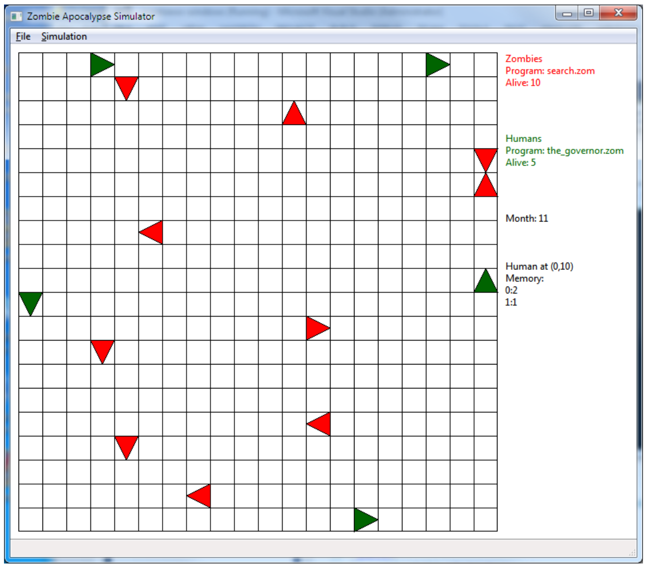
The GitHub classroom link for this assignment is: https://classroom.github.com/a/QSYyMQQS

Note that this assignment is due on a **FRIDAY**, not a Wednesday.

In this lab we are going to make a zombie simulation program. The inspiration is "Darwin's World" developed by Nick Parlante at Stanford.

The basic idea is that there are zombies and humans fighting for dominance in a post-apocalyptic world. Rather than hard-coding the behavior of the zombies and humans, their behavior is driven by ".zom" files. So for instance, "daryl_dixon.zom" is a pretty awesome ranged/melee zombie killer while "michonne.zom" only uses melee attacks.

The final version of the program looks something like this:



Zombies
Program: search.zom
Alive: 10

Humans
Program: the_governor.zom
Alive: 5

Month: 11

Human at (0,10)
Memory:
0:2
1:1

There are a few things you'll have to do in this lab. First, you'll have to parse the .zom file format and generating the correct set of instructions from it. After this, you also need the code that can execute those instructions.

You also need code to maintain the overall simulation. For this, you will randomly place zombie and humans in the world, and then running the virtual machine simulation for every zombie and human until either the zombies or humans win. Once the execution works properly for a single zombie or human, it's not a lot more to get this working. Drawing uses a few simple wxWidgets call, like the last lab.

Before you start coding, keep in mind that ***this lab uses exceptions***. This means you need to throw exceptions for things the user might do (such as try to pass in an invalid .zom file) **and** what a ZOM simulation might do. You also will need code to catch exceptions.

Part of your grade is how well you used exceptions. Make sure you create appropriate exception subclasses in the Exceptions.h header file as needed. Don't just use `std::exception` everywhere.

As in the prior lab, you want to run the "main" target for the most part.

ZOM Framework

Before coding, let's discuss the simulation. There are zombies and humans. Humans can perform actions more often than zombies (because they can use vehicles and other tools), and they also have a ranged attack.

If a zombie kills a human, the human becomes a zombie. However, if a human kills a zombie, the zombie just disappears from the world.

The world is a grid of 20x20 squares. Each square can either be empty, have a zombie, or have a human. A square cannot contain more than one human or zombie at any one time.

Zombies/humans face in one of the four cardinal directions. They can only move and attack in the direction they're facing. The rotate action allows the zombie/human to rotate either clockwise or counter-clockwise. The border around the grid is the "wall" so zombies/humans can't move outside the walls.

This is a turn-based simulation. Every turn, a zombie can only perform one "action" per turn (aka they have one action point). Actions are things like attacking, moving, rotating, etc. Humans can perform two actions per turn (they have two action points). Any programming logic such as conditionals, branches, etc., **do not count as actions**. So, if a ZOM file has a lot of conditions/branches, you can potentially perform a lot of ops in one turn, but only one (or two) actions.

Here is a small sample ZOM file, `stationary.zom`:

```
test_human,1; Is this a human?
je,5        ; If so, jump to line 5
rotate,0    ; Rotate clockwise
goto,1      ; Goto line 1
attack      ; Try to eat the human!
goto,1      ; Goto line 1
```

Anything after a semi-colon is a comment, so the parsing code should ignore it. Notice how some ops, like `test_wall` have no parameters, whereas others have a parameter, with a comma separating the op and the parameter. Each line can only have one op. And line numbers matter, because branches will always specify the line number.

Action Ops (one action point each)

`attack`	Try to kill whatever is right in front of you. If a zombie attacks a human, the human becomes a zombie. If a human attacks a zombie, the zombie dies. If a zombie attacks a zombie, nothing happens. If a human attacks a human, the human being attacked dies. If you try to attack a wall or there's nothing in front, it just wastes the action.
`ranged_attack`	Attack two tiles in front of you with a ranged attack. Only humans can use a ranged attack. Whatever a human attacks dies. If there's nothing to attack there, it wastes the action. If you try to load a zombie file with a ranged attack, it should throw an exception on loading file.
`rotate,n`	If n=0, rotate 90° clockwise, otherwise counter-clockwise.
`forward`	Move forward, if possible. If the movement is invalid (wall or otherwise occupied), it consumes an action but nothing else happens.
`endturn`	Automatically ends the turn, even if there are more actions left.

Other Ops

`test_human,n`	Set test flag to true if a human is n tiles in front, where n can be either 1 or 2.
`test_wall`	Set test flag to true if facing a wall.
`test_zombie,n`	Set test flag to true if a zombie is n tiles in front, where n can be either 1 or 2.
`test_random`	Randomly set test flag to true or false.
`test_passable`	Set test flag to true if facing an open tile.
`je,n`	If test flag is true, jump to line number n. Otherwise continue to next line of execution. If line n is invalid, throws an exception on loading file.
`jne,n`	Exactly like je, but jumps if the flag is false.
`goto,n`	Automatically go to line number n. If line n is invalid, throws exception on loading file.

Right now, the only op implementations given are `goto` and `rotate`. You'll have to implement the rest.

Hopefully at this point you understand roughly how the simulation works. Now let's look at the source code that's provided for you.

Part 1: ZOM Implementation

There are four main files related to the ZOM virtual machine framework: Op.h/cpp, Machine.h, and Traits.h. I'm going to briefly describe what's in all the files. Read the description below first, ***don't try to implement anything until you get to*** "What to implement first."

Op.h/cpp defines and implements all the ops. There is an abstract base class called `Op` from which all the different operations derive from. All operations can have up to one parameter.

More than one parameter is not supported. There are currently implementations for `OpGoto` and `OpRotate`. You will need to implement every other op. Notice how each `Op` has an `Execute` function, which takes a `MachineState` reference as a parameter. The `Execute` function is what performs the logic of the `Op`.

`MachineState`, in Machine.h, represents the state of the current zombie/human. Right now, it has several variables which are useful, but **it does not have all the variables you will need**. You will have to add them as needed.

Here are the variables `MachineState` currently has:

`mProgramCounter` – Tracks current line number of execution. This will change after every op.

`mActionsTaken` – Tracks the number of actions taken this turn.

`mFacing` – Stores the current facing of the zombie/human.

`mTest` – Boolean used for all the test_* ops.

Each zombie/human in the world has a corresponding `MachineState`. However, the `MachineState` does not store a vector of ops. This is intentional, and is what the `Machine` class is for. The `Machine` class will parse in the .zom file, load in all the appropriate ops, and keep track of all that. All zombies in the world use one Machine, and all humans in the world use another Machine. Think of the Machine as a CPU and the MachineState as the threads that run on the CPU.

`Machine` has temp code in it right now. For now, hard code a few instructions for testing. Later, you will need to implement `LoadMachine` to parse in ZOM files properly.

Zombies and humans have slightly different characteristics. Rather than sub-classing `Machine`, will use the Policy/Traits design pattern. There are two different trait classes, one for zombies and one for humans, and they're defined in Traits.h. It basically is just a couple of constant values. The `Machine` class can access these values as needed.

The `Machine::BindState` function is important. Whenever a `MachineState` is created, it should be bound to a specific `Machine`. That's because it sets the important traits related to that `Machine`. There is no other way to get the traits from a `Machine` inside `MachineState`, so you must remember to bind when creating a human/zombie. Don't forget to change the binding when a human gets infected!

What's Working So Far

The only menu option that's hooked up currently is Simulation>Start/Stop. What this does is sets a timer in ZomFrame.cpp, so that every second the turn timer ticks, and one turn runs (via the `ZomFrame::OnTurnTimer` function).

The instructions the program currently runs are the hard-coded ones in `LoadMachine`. So, you'll see it `rotate` and `goto`, and debug text output that shows what commands are running (on Windows, this debug output will be in the Output window in Visual Studio, and on Mac it'll just show up in the normal console).

What to Implement First
This section outlines one way you can approach the implementation. This is not the only way to do it, but it's along the lines of how I implemented it when creating the assignment.

What I'd recommend doing is first implement a class to represent the entire world (one way is to use the Singleton class that's provided for this purpose). You then need some sort of data structures to represent all the zombies/humans in the world.

To allow the user to select a file to load, you need to add two options to the Simulation menu: "Load Zombie..." and "Load Survivor...". To add menu options, you need to:

1. In ZomFrame.h, declare a new member function that takes in a `wxCommandEvent` by reference.
2. In ZomFrame.cpp, add an implementation for this function
3. In the enum towards the top of ZomFrame.cpp, add a new ID corresponding to your new menu option.
4. In the event table near the top of ZomFrame.cpp, add a new `EVT_MENU` entry that associates the new ID to the new member function
5. In the `ZomFrame` constructor, find the corresponding menu (in this case mSimMenu) and call the `Append` function a new element to the menu

Once you have the load options, use the `wxFileDialog` class (as in the last assignment) to select a file. I recommend passing in a path of `"./zom"` and setting the filter to `"ZOM Files|*.zom"`.

First, try to get the basic_movement.zom file working. Implement the parsing of ZOM files in `Machine::LoadMachine`, and all the ops in basic_movement.zom. You will need some way to query what is in front of you for the `test_wall` function. You also should probably add x/y coordinate information to `MachineState`. Have your single test guy start out at (0,0). Then in the debug output, you can output the current position of the zombie/human as well, because it will really help debugging the execution.

After you think you've implemented the parsing and these first few ops properly, you should track the position of the sample zombie as he goes all the way to the right, before hitting the wall and going down (or in another direction, depending on the randomness). And more or less hugging the wall all the way. This should give you an idea if the ops are implemented properly.

Once you have the test zombie working in the debug console, you should now implement the visualization. Implement the drawing code in `ZomDrawPanel::DrawGrid`. The grid is setup so the overall square is 600x600 pixels. So for a 20x20 grid, you want each square to be 30x30 pixels. The grid outline can be drawn with the `DrawLine` member function of `wxDC`, and the triangles for the zombies/humans can be drawn using `DrawPolygon`, which takes in an array of `wxPoints` representing the polygon.

After you get the single guy moving around in the window, add a “Randomize” option to the simulation menu. This should populate the world randomly with basic_movement.zom zombies on screen. Make sure you don’t have any issue with trying to move to an invalid square.

Once you get the visualization of basic_movement.zom working properly, you can then move on to implementing everything else. Once you think you’re done with the instructions, try placing a couple of humans and zombies and simulate them. See if it works. You could try using search.zom for the zombies and daryl_dixon.zom for the humans. Daryl is awesome so he will usually win this battle.

Note: When a new turn starts, all zombies should take their turns first, then all humans should take their turns after that. Don’t let them go in just any order.

Part 2: Overall Simulation Additions

Now we need to add a few more features. First, add a “Reset” menu option to the Simulation menu, which should remove any zombies/humans currently in the world and set the month back to zero. It should also stop the timer from firing (via `mTurnTimer->Stop()`).

Then implement the “New” option in the file menu. This should be like “Reset” except it also unloads the current zombie/human files. In essence, the state of the program after “New” should be the same as it is when you first open the program.

Next, update Simulation>Randomize so that it randomly place 20 zombies and 10 humans in the world. This may seem like an unfair ratio, but it is true to the zombie apocalypse! Plus, the humans generally are stronger than the zombies. This ratio should make it a bit fairer. They should have not only a random position but a random orientation.

You also need to make sure that only Load Zombie/Load Survivor are enabled initially. Once both a zombie and survivor has been loaded, then the other three options should be enabled. You can disable a menu by using the following function call:

```
mSimMenu->Enable(ID_SIMSTART, false);
```

To recap, the simulation menu should have the following options once you are done:

- Load Zombie...
- Load Survivor...
- Reset
- Randomize
- Start/Stop

You also should draw status text on the right part of the screen which displays the following:

1. Number of zombies alive
2. Current zombie file loaded
3. Number of humans alive
4. Current human file loaded
5. Current “month” (how many turns have elapsed)

To draw text, you can use the DrawText function on the wxDC. Note that this function expects to take in a wxString, not a std::string as a parameter. If you want to convert a std::string to a wxString, you can use wxString::FromUTF8(stringName.c_str()).

Finally, you need to check for a winner in the `OnTurnTimer` function in ZomFrame.cpp. If there are zero zombies or humans left, you should stop running the timer and declare the winner with a message box.

Grading Rubric

Part	**Points**
Part 1 – ZOM Framework	
***Each op implemented to spec (3% each)	52
***Implementation of “World” class	8
Part 2 – Menu/Glue	
***Each “Simulation” menu command works	10
***Contents of window draw correctly including the grid and text status messages	10
Your Test Cases	10
Code Quality (incl. following style guide)	10
Total	**100**