Zewail City of Science and Technology
Communications and Information Eng. Prog
CIE202 Spring 2019

CIE202 Project

Game

Introduction

In this project we are going to build a simple game application. The general idea of the game is a player moving in a grid of cells having obstacles cells in addition to other types of cells that you should decide during game implementation. The player has health and lives and game is only one level.

Game Target:

The player is required to collect a number of items and reach a goal cell.

Game Result:

Win: The player could collect all the required items and reaches the goal cell.
Loss: The player lost all his lives.

Your Task:

You are required to write a **C++ code** for a Game. You must use ***object-oriented programming*** to implement this application and respect the **responsibilities** of each class as specified in the document.

You are given a framework code that you should extend to complete the project requirements.

NOTE: The application should be designed so that the types of objects and operations can be easily extended (*using inheritance*).

Project Phase	***Due Date***
Phase 1 [20%]	**Week 13 (23/4/2019)**
Phase 2 [80%]	**Week 17**

Main Operations

The application supports 2 modes: **Grid Mode** and **Game Mode**. In grid mode, the user interacts with the application through the menu icons that are in **tool bar** (top of the window). The application should also provide a **status bar** that to print messages to the user and to display game progress.

The player is moving in a grid that is divided into a number of cells; a cell is specified by its row and column.

[I] Grid Mode :

In this mode, the user interacts with the application through mouse clicks. Keyboard input is ignored.
The following operations should be supported by the Grid Mode

1- [5%] Save Grid:

- ❑ Saving the information of the drawn grid (all the cells) to a file (see "file format" section).
- ❑ The application must ask the user about the filename to create and save the grid in.
- ❑ Classes Player and Cell should provide a member function Save to be able to write their data in the file. Cell::Save should be virtual function and should be overridden in each derived class
- ❑ Grid should have SaveAll function that opens the files to save in ad then loops on every non-empty cell and call function Save for each cell. It should also call Player::Save function.

2- [5%] Load Grid:

- ❑ Loading a saved grid from a file and re-drawing it (see "file format").
- ❑ This operation re-creates the saved items and obstacles and re-draws them.
- ❑ The application must ask the user about the filename to load from.
- ❑ After loading, the user **can edit the loaded grid** and continue the application normally.
- ❑ If there is a grid already drawn and the load operation is chosen, the application should clear the area and make any needed cleanups then load the new one.

3- [20%] Edit Grid:

- ❑ When the application starts, an empty grid (i.e a grid whose cells are all of type EmptyCell) is loaded. The user can either load a new grid from a file (as described above) or can Edit the grid by replacing some of the empty cells with different types of cells to it.
- ❑ **[15%]** The following types of cells should be supported by your application
  a. ***PlayerCell*** (given): a cell where the player is located
  b. ***EmptyCell*** (given): a cell that has no objects
  c. ***ObstacleCell***: a cell the player cannot pass over.
  d. ***EnemyCell***: An enemy cell moves randomly trying to collide with the PlayerCell.
  e. ***GoalCell***: The player is trying to reach the goal cell.
  *f. **You are required to add FOUR more types of cells of your choice***
- ❑ Each type of cell (including EmptyCell) should have an icon in the menu so that the user can pick it a place it in any place in the grid as explained by the following steps.
- ❑ **[5%] Edit grid steps**:
  a. User picks the cell type from the menu icons.
  b. A message is printed to the user about the cell type and asking him to click on the grid on the required cell to replace.
  c. When the user selects a cell in the grid, the selected cell is replaced by the type of cell picked from the menu:
     ✓ You will need to add function getClickedCell in Grid class to know which cell is clicked by the user.

- ✓ Grid::getClickedCell function should call GUI::getClickedPoint to get the coordinates (x, y) of the mouse click and use them to identify the clicked cell and returns a pointer to it.
- ✓ Then call Grid::setCell to preform cell replacement.

d. If more information is required for the newly created cell, the application can ask the user to enter them.

4- [2.5%] Start Game: Any time the user clicks on "Start" icon, the application switches to the Game Mode. (see below)

5- [5%] Exit: end the application.

- ❑ Perform any necessary cleanup (freeing reserved memory) before exit. Ask the user if he wants to save current grid.

[II] Game Mode:

In this mode, the user interacts with the application though keyboard. Mouse clicks are ignored.

On the status bar, you should show: the player's health, remaining lives, score, and the number of items the player should collect.

The operations supported by this mode are:

1- [5%] Player Move and Pick: By pressing on the arrow keys (check the given code).

- ❑ Player can't pass through an obstacle
- ❑ While moving, the player should collect certain items and avoid non-friendly cells until he reaches the goal cell.

2- [15%] Collision Resolution:

- ❑ Collision means that the player is trying to move over a given cell.
- ❑ Collision is resolved according to the type of the cell colliding with the PlayerCell. For example collision with an obstacle cell prevent the player from moving over the cell
- ❑ Collision should be resolved in function Cell::ActOn(Player *). This function should be overridden in each cell type.

3- [5%] Moving enemies: (Randomly)

- ❑ EnemyCells should move randomly trying to collide with the PlayerCell.
- ❑ If they collide the player's health decreases. If health becomes zero, the player loses a life.

4- [2.5%] Pause the game: By pressing on ESC key

- ❑ The game is paused and the application switches to the Grid Mode
- ❑ While the game is paused, the user can perform any of the "Grid Mode" operations
- ❑ To resume the game gain, the user should click on the "Start" icon

[30%] Individual participation

Each member must be responsible for some part of the code and must answer some questions showing that he/she understands both the program logic and the implementation details.
Note: we will reduce the individual grade in the following cases:

- ❑ Not working enough
- ❑ Neglecting or preventing other team members from working enough

You should **inform the instructors** before the deadline **with a sufficient time (some weeks before it)** if any of your team members does not contribute in the project work and does not make his/her tasks. The TAs should warn him/her first before taking the appropriate grading action.

[5%] Code Organization & Style

- ❑ Every class in .h and .cpp files
- ❑ Variable naming
- ❑ Indentation & Comments

For teams of 4 members (bonus for others)

Note: This part is mandatory for teams of 4 members and bonus for other teams

WeaponCell: (derive this class from Cell class)

- ❑ add a new type of cells that contains a weapon

GrabWeapon: (A member function in class Player)

- ❑ A player can pick a weapon if he collides with a WeaponCell
- ❑ The player can hold only one weapon at a time.
- ❑ Class Player should have a pointer to class Weapon as a data member.

The WeaponCell may contain one of the following two types of weapons:

BombWeapon: (Create a class ***Weapon*** and derive this class from it)

- ❑ Kills enemies in the region ±3 cells in the vertical and horizontal direction from the player's location only if no obstacle cell is between them.
- ❑ Takes effect at any time the user presses **B** while carrying this weapon.
- ❑ Once used, the player loses this weapon.

SlowDownWeapon: (derive this class from ***Weapon*** class)

- ❑ Decreases the speed of an Enemy by half for next 40 steps it moves.
- ❑ Takes effect when the player collides with an enemy while carrying this weapon.
- ❑ Once used, the player loses this weapon.

File Format

Your application should be able to **save/load a grid** to/from a simple text file. In this section, the file format is described

- **File Format**

Player Information
Total_Number_of_nonEmpty_Cells (say N)
Cell_1_Type Cell_1_row Cell_1_col Cell_1_other_info(if needed)
Cell_2_Type Cell_2_row Cell_2_col Cell_2_other_info(if needed)
……..
……..
Cell_N_Type Cell_N_row Cell_N_col Cell_N_other_info(if needed)

- ❑ **The Load Operation**: revisited
  For each line in the above file, The **Load function** first **reads** the cell type then **creates** (allocates) an object of the required cell type. Then, it **calls Cell::Load** virtual function that can be overridden in the subclasses of class cell to make the cell load its data from the opened file by itself (its job). Then, the created cell is added to the grid in its correct coordinates.

Project Phases

A code framework is given to you. You should extend to complete the project phases.

Phase 1 [20%]

In this phase, you are required to implement "Edit Grid" operation described in the "Grid Mode" above. No move logic or collision logic is required at this phase. Also Save and Load functions are not required at this phase

What is required at this phase:

1- Create a class for each type of cell you will use in the game
- ❑ Derive from class Cell
- ❑ Add an image for each cell type.
- ❑ Function ActOn is **NOT** needed at this phase.

2- Add more icons to the game menu so that each type of cell has a corresponding icon

3- The "Edit Grid Steps" described above should be implemented at this phase.

Deliverables:

(1) **Cells Description document**: a **printed document** containing decryption for each cell type in your game.
(2) One submitted zipped file containing:
   a. ID.txt file. (Information about the team: names, IDs, team email)
   b. The cell description document.
   c. The project code and resources files (images, saved files, …etc.).

Phase 2 [80%]

The full working project

Deliverables:

(1) **Workload division**: a **printed page** containing team information and a table that contains members' names and the actions each member has implemented.
(2) One submitted zipped file containing:
   a. ID.txt file. (Information about the team: names, IDs, team email)
   b. The workload division document.
   c. The project code and resources files (images, saved files, …etc.).
   d. Sample grid files: at least three different grids. For each grid, provide:
      - Grid text file (created by save operation)
      - Grid screenshot for the grid generated by your program
   e. The sample files should represent different game cases ranging from easy to hard game files.

Deliverables and Evaluation Criteria

The weight (percentage) of each required operation is shown above in "Main Operations" section

General Evaluation Criteria for any Operation:

1. **Compilation Errors → MINUS 50%** of Operation Grade
   - ❑ The remaining 50% will be on logic and object-oriented concepts (see **point no. 3**)

2. **Not Running** (runtime error in its **basic functionality**) → **MINUS 40%** of Operation Grade
   - ❑ The remaining 60% will be on logic and object-oriented concepts (see **point no. 3**)
   - ❑ If we found runtime errors but in corner (not basic) cases, that's will be part of the grade but not the whole 40%.

3. **Missing Object-Oriented Concepts → MINUS 30%** of Operation Grade
   - ❑ **Separate class** for each item and action
   - ❑ Each class does its **job**. No class is performing the job of another class.
   - ❑ **Polymorphism:** use of pointers and virtual functions
   - ❑ **See the "Implementation Guidelines" in the Appendix which contains all the common mistakes that violates object-oriented concepts.**

4. **For each corner case** that is not working → **MINUS 10% to 20%** of the Operation Grade according to instruction evaluation.

Notes:

- ❑ The code of any operation does NOT compensate for the absence of any other operation; for example, if you make all the above operations except the delete operation, you will lose the grade of the delete operation no matter how good the other operations are.

APPENDIX A

[I] Implementation Guidelines

- **Any user operation is performed in 4 steps:**
  - Get user action type.
  - Create suitable action object for that action type.
  - Execute the action
  - Reflect the action to the Interface (i.e. Draw affected cells).

- **Use of Pointers/References**: Nearly all the parameters passed/returned to/from the functions should be pointers/references to be able to exploit **polymorphism** and **virtual** functions.

- **Classes' responsibilities:** Each class must perform tasks that are related to its responsibilities only. No class performs the tasks of another class. For example, when a cell needs to draw itself on the GUI, it calls function ***GUI::DrawCell*** because dealing with the GUI window is the responsibly of class **GUI**.

- **Abusing Getters:** Don't use getters to get data members of a class to make its job inside another class. This breaks the classes' responsibilities rule.

- **Virtual Functions:** In general, when you find some functionality (e.g. saving) that has different implementation based on each type, you should make it virtual function in the base class and override it in each derived class with its own implementation.
  - A common mistake here is the **abuse** of **dynamic_cast** (or similar implementations like **class type** data member) to check the object type outside the class and perform the class job there (not inside the class in a virtual member function).
  - This does not mean you should never use **dynamic_cast** but do NOT use it in a way that breaks the constraint of class responsibilities.

- You are not allowed to use **global variables** in your implemented part of the project, use passing variables as function parameters instead. That is better from the software engineering point of view. Search for the reasons of that or ask your TA for more information.

- You need to get instructor approval before using **friendships.**

- For fast navigation in the given code in Visual Studio, you may need the following:
  - **F12 (go to definition):** to go to definition of functions (code body), variables, …etc.
  - **"Ctrl" then "Minus":** to return to the previous location of the cursor.

Notes on The Project Graphics Library*:*

- The origin of the library's window **(0, 0)** is at the **upper left** corner of the window.
- The direction of **increasing** the **x coordinate** is to the **right**.
- The direction of **increasing** the **y coordinate** is **down**.
- The images extension that the library accepts is "**jpg**".