Cairo University
Faculty of Engineering
Computer Engineering Department
CMP N103 **Fall 2018**

بسم الله الرحمن الرحيم

"نرفع درجاتٍ من نشاء وفوق كل ذي علم عليم"

Programming Techniques Project Requirements

Paint for Kids

Fall 2018

Introduction

A fancy colorful application is an effective way to teach kids some computer skills. Educational games are another enjoyable way for kids teaching. In this project (Paint for Kids) we are going to build a simple application that enables kids draw fancy shapes and also play some simple games with those shapes. Your application should help a kid draw a number of figures, fill them with different colors, save and load a graph, and so on. The application should provide a game playing mode to teach kids how to differentiate between figures types, colors …etc.

Your Task:

You are required to write a C++ code for Paint for Kids application. Delivering a working project is NOT enough. You must use ***object-oriented programming*** to implement this application and respect the **responsibilities** of each class as specified in the document. See the evaluation criteria section at the end of the document for more information.

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

Project Schedule

Project Phase	***Deliverables***
Phase 1 [25%]	**Input-Output Classes**
Phase 2 [75%]	**Final Project Delivery**

The project code must be totally yours. The penalty of cheating from any other source is not ONLY taking ZERO in the project grade but also taking some MINUS grades (the amount of the minus grades will be determined later), so it is way better to deliver an incomplete project other than cheating it.

Table of contents

Main Operations

The application supports 2 modes: **draw mode** and **play mode**. Each mode contains 2 bars: **tool bar** that contains the main operations of the current mode and **status bar** that contains any messages the application will print to the user.

The application should support the following operations (actions) in each mode. Each operation **must** have an **icon** in the tool bar. The user should click on the operation icon from the tool bar to choose it.

***Note:** Any **percentage** written below next to any operation is its percentage from **phase 2 grade**. See the evaluation criteria at the end of the document for more details.*

Percentages are added according to the **difficulty** of the task, so to divide the project load **equally** on team members, each member should take actions that their percentages add up to about **25%** of phase 2 (if 4 students in the team).

The main operations supported by the application are:

[I] Draw Mode: [75%]

Note: See the "General Operation Constraints" section to know the general constraints that must be applied in any operation.

1- [7.5%] Add Figure: adding a new figure to the list of figures. This includes:
- ❑ Adding a new **line**, a new **rectangle**, a new **triangle**, a new **rhombus**, or a new **ellipse** (the figures initially are unfilled)
- ❑ **Note:** Make all **rhombus** figures have the same size, so you need to take only the center point from the kid. The same note applies in **ellipse** too but not in line, rectangle or triangle.

2- [10%] Select Figure: selecting **one** of the figures.
- ❑ User first chooses the "**select**" icon from the tool bar
- ❑ Then clicks inside the figure or on its border (***same for filled and unfilled figures***)
- ❑ If the user re-clicks on the selected figure, this will un-select it.
- ❑ If the user clicks on an empty area (not inside any figure), this will make un-select too.
- ❑ If the user clicks on the intersected part of some overlapping figures, you should select/unselect the last drawn one from these figures.
- ❑ If the user selects a figure while another one is selected, the previous one will be unselected and the newly-selected one will be selected (NO multiple selection is allowed).
- ❑ When the user selects a figure,
  - ❑ This figure should be highlighted (Assume that the highlight color is "**magenta**"; a color in the library)
  - ❑ All information about the selected figure should be printed on the status bar.
    For example, the application can print (depending on figure type): the figure ID, start and end points, width, height, ...etc.

3- [5%] Change "Figure" Colors: changing the drawing or filling color for the selected figure and any subsequent drawings. You have to select figure first (using the "select" icon) before clicking the icon of changing its drawing or filling color. Existing figures will NOT be changed but any subsequent figures will be drawn using the changed drawing and filling colors (until they're changed again and so on). You may need to have two separate icons for this action. The available colors are: **black**, **white**, **red**, **green** and **blue**.

4- [5%] Delete Figure: deleting the selected figure.

5- **[7.5%] Copy Figure:** copying the selected figure to the application clipboard (**Hint:** clipboard could be a pointer that points to the copied/cut figure).

6- **[7.5%] Cut Figure:** cutting the selected figure and putting it in the application clipboard.
- ❑ The cut figure should **NOT** be removed from the drawing area before the paste action (mentioned below) is performed. Instead, when the selected figure is cut, its drawing and fill color will be drawn in **grey**.
- ❑ Note that the copy and cut operations use the same application clipboard, so each copy/cut should overwrite it.
- ❑ This means, for example, if a cut operation is followed by a copy/cut operation before pasting the first cut figure, the figure of the first cut will be lost (do NOT forget to deallocate it) and the paste operation will paste only the last copied/cut figure.
- ❑ Note that if there where a cut figure (**grey** figure), if the clipboard is overwritten by another cut or copy operation, the figure cut previously should return to its original draw and fill color.

7- **[10%] Paste Figure:** pasting the last copied/cut figure (the figure existing in the application clipboard).
- ❑ The application allows the user to paste the same copied/cut figure multiple times, so the user can make many paste operations.
- ❑ When pasting the figure, the application should ask the user to click at the new location to paste the figure to. You can assume this click to be the new center or upper left corner of the pasted figure.
- ❑ When pasting a copied figure, if the user copied a figure then updated it (e.g. its color) before pasting, the paste operation should paste the original copied figure without any recent changes.
- ❑ When pasting a cut figure (drawn in **grey**), paste it with its original draw and fill color.

8- **[5%] Save Graph**: saving the information of the drawn graph (all the figures) to a file (see "file format" section). The application must ask the user about the filename to create and save the graph in (overwrite if the file already exists).

9- **[5%] Save by Type**: the user first chooses a figure type (line, rectangle, …etc.) then this action will save all the figures that are of this type to a file using the same file format of the "Save Graph" action. When the user clicks on the "Save by type" icon, the application should ask the user to choose the figure type he wants to save (by choosing from 5 icons for the 5 types) then will ask the user about the filename to save the figures in (overwrite if the file already exists).

10- **[7.5%] Load Graph:** loading a saved graph from a file and re-drawing it (see "file format").
- ❑ This operation re-creates the saved figures and re-draws them.
- ❑ The application must ask the user about the filename to load from.
- ❑ After loading, the user can edit the loaded graph and continue the application normally.
- ❑ If there is a graph already drawn on the drawing area and the load operation is chosen, the application should clear the drawing area (make any needed cleanups of the current drawn graph) then load the new one.

11- **[2.5%] Switch to Play Mode:** by loading the tool bar and the status bar of the play mode. The user can switch to play mode any time even before saving.

12- **[2.5%] Exit:** exiting from the application.
- ❑ Perform any necessary cleanup (termination housekeeping) before exiting.

[II] Play Mode: [20%]

In this mode, the graph created in the draw mode (or loaded from a file) is used to play some simple kids' games. The operations supported by this mode are:

1- **[15%] Pick & Hide:** The user is prompted to pick the figures of a specific property (color or type). When he picks a figure, it should disappear and the user picks the next figure with the same property value. The application should print counters to count the number of correct and incorrect picks done by the user. Finally, when the user picks all the figures of the required property, a grade is displayed for him showing how many correct and incorrect picks he made.
The user should be able to pick figures by **one** of the following properties:
   - ❑ **Figure Type:** e.g. pick all rectangles, …etc.
   - ❑ **Figure Fill Color:** e.g. pick all red figures, …etc.

   You may need a toolbar icon for each one of the two picking properties (two icons). When the kid chooses a property (e.g. figure type or color) the application should randomly choose a property value (rectangle, line, …etc. if the property is figure type) and prompt the user to pick it. The application should be logical when it chooses a random property value, for example, do not ask the kid to pick ellipses and the drawn graph does not have ellipses.

   Note:
   At any time, the kid can restart the game or start another game by clicking its menu icon. When the user does so:
   - ❑ The original graph should be restored.
   - ❑ All changes done in the current game should be discarded.
   - ❑ Don't forget to make any needed cleanups.

2- **[5%] Switch to Draw Mode**: At any time, user can switch back to the drawing mode.
   - ❑ The original graph should be restored
   - ❑ All changes done in the play mode should be discarded.
   - ❑ Don't forget to make any needed cleanups.

The remaining 5% of phase 2 is left for code organization and styling. See "Phase 2 Evaluation Criteria" section for more info.

[III] [Bonus] Operations [10%]:

The following operations are bonus and you can get the full mark without supporting them and you will not receive more bonus if you implemented any additional feature other than the features mentioned here in the bonus part. This bonus will be delivered in **Phase 2.**

1- **[5%] Add Voice to your application:**
   - ❑ When the kid makes any action like adding a new figure (e.g. rectangle), the application should say a word indicating what happened (e.g. say: "Rectangle" in a funny child voice).
   - ❑ The icon of this action enables or disables the voice in your application.
   - ❑ To take the full mark of this action, you need to add voice to at least 5 actions of the actions mentioned above.

2- **[2.5%] Resize Figure:** resizing the selected figure by 1/4, 1/2, 2 or 4 times their current size**.**

3- **[2.5%] Send to Back / Bring to Front**: sending the selected figure to the back or the front of other figures overlapping with it which will affect how they are drawn because the front figures (only if filled) hides the overlapped parts of the back figures.

General Operation Constraints

The following rules must be satisfied even if not mentioned in the operation's description:

1. Any needed **cleanups** (freeing any allocated memory) must be done.

2. Many operations need some figures to be selected first. If no figures are selected before choosing such operations, an **error message** is displayed in the status bar and the chosen operation will make no change.

3. All kid inputs (except filename) must be obtained through **mouse clicks** on icons not keyboard typing. This is better to avoid typing errors and checks and also because the application targets kids so typing should be minimized.

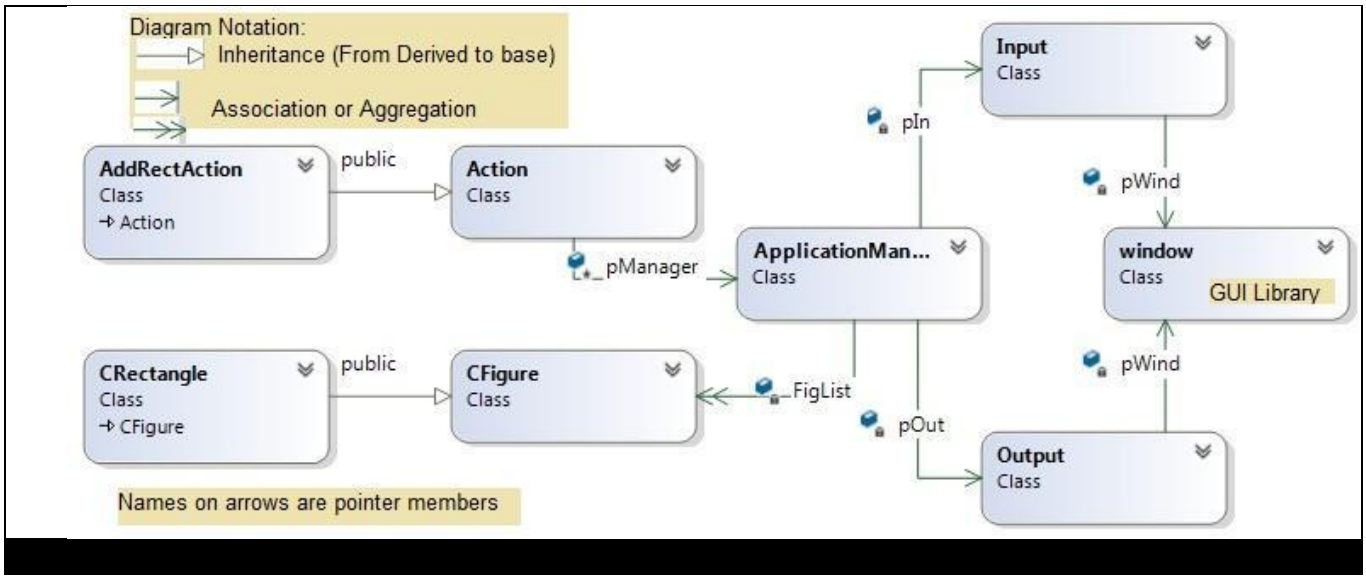
Main Classes

Because this is your first object-oriented application, you are given a ***code framework*** where we have ***partially*** written code of some of the project classes. For the graphical user interface (GUI), we have integrated an open-source ***graphics library*** that you will use to easily handle GUI (e.g. drawing figures on the screen and reading the coordinates of mouse clicks …etc.).

You should **stick to** the **given design** (i.e. hierarchy of classes and the specified job of each class) and complete the given framework by either: extending some classes or inheriting from some classes (or even creating new base classes but after instructor approval).

We want you to **stick** to the given design because it is your **first OOP project** and we want to give you an example of a fairly-good designed code to work in it. In most of the projects of next years, you will be free to design your OOP classes the way you want.

Below is the class diagram then a description for the basic classes.



Input Class:
ALL user inputs must come through this class. If any other class needs to read any input, it must call a member function of the input class. You should add suitable member functions for different types of inputs.

Output Class:
This class is responsible for **ALL** GUI outputs. It is responsible for toolbar and status bar creation, figures drawing, and for messages printing to the user. If any other class needs to make any output, it must call a member function of the output class. You should add suitable member functions for different types of outputs.

Notes: - No input or output is done through the console. All must be done through the GUI window.
- Input and Output classes are the **ONLY** classes that have direct access to **GUI library**.

ApplicationManager Class:
This is the ***maestro*** class that controls everything in the application. Its job is to manage or instruct other classes to do their jobs (**NOT** to do other classes' jobs). It has pointers to objects of all other classes in the application. This is the **ONLY** class that can operate directly on the figures list (**FigList**).

CFigure Class:
This is the base class for all types of figures. To create a new figure type (Ellipse class for example), you must **inherit** it from this class. Then you should override virtual functions of class **CFigure (**e.g. Draw, save, etc.). You can also add more details for the class CFigure itself if needed.

Action Class:
Each operation from the above operations must have a ***corresponding action class***. This is the base class for all types of actions (operations) to be supported by the application. To add a new action, you must **inherit** it from this class. Then you should override virtual functions of class **Action**. Each action may have action parameters. **Action parameters** are the parameters needed to be read from the user, after choosing the action icon, to be able to execute the action. You can also add more details for the class Action itself if needed.

Example Scenarios

The application window in draw mode may look like the window in the following figure. The window is divided to **tool bar**, **drawing area** and **status bar**. The tool bar of any mode should contain icons for all the actions in this mode (***Note***: the tool bar in the figure below is not complete and you should extend it to include all actions of the current mode).

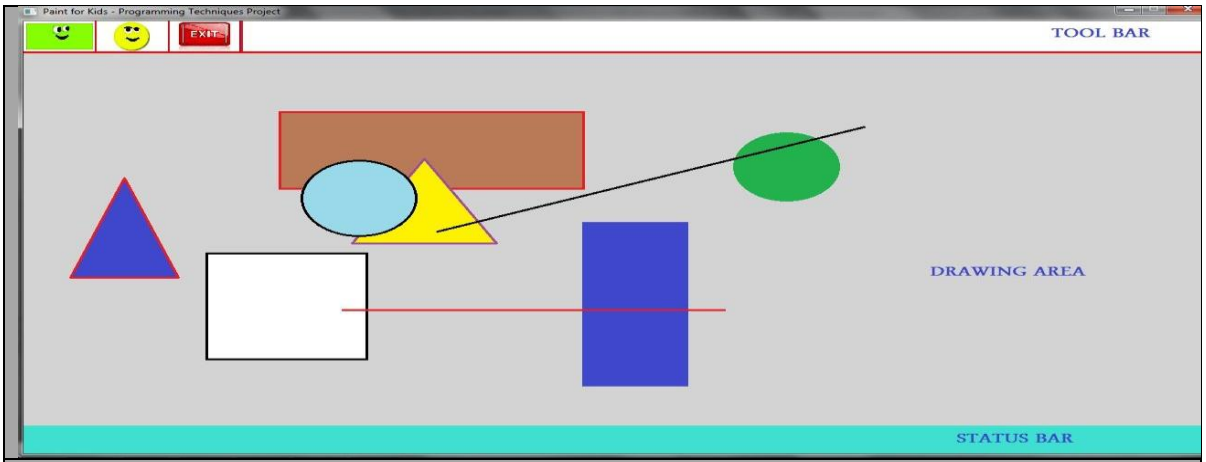


Figure 1 – Application screenshot

Example Scenario 1: AddRectAction

Here is an example scenario for **drawing a rectangle** on the output window. It is performed through the four steps mentioned in 'Appendix A - implementation guidelines' section. These four steps are in the "main" function of phase 2 code. You **must NOT** change the "main" function of **phase 2**. The 4 steps are as follows:

Step 1: Get user input

1- The ***ApplicationManager*** calls the ***Input*** class and waits for user action.
2- The user clicks on the "Add Rectangle" icon in the tool bar to draw a rectangle.
3- The ***Input*** class checks the area of the click and recognizes that it is a "draw rectangle" operation. It returns **`DRAW_RECT`** (an **"enum"** value representing the action: **ActionType**) to the manager.

Step 2: Create a suitable action

1- ***ApplicationManager::ExecuteAction(ActionType)*** is called to create an action object of type ***AddRectAction*** *class*.

Step 3: Execute the action

1- ***ApplicationManager::ExecuteAction(…)*** calls ***AddRectAction::Execute( )***
2- ***AddRectAction****::Execute( )*
   a. calls **AddeRectAction::ReadActionParameters( )** which calls the ***Input*** class to get rectangle parameters (i.e. the 2 corner points of the rectangle) from the user. **Notice** that when ***AddRectAction*** wants to print messages to the user on the status bar, it calls some functions from the **Output** class.
   b. Creates (allocates) a figure object of type ***CRectangle*** class and asks the ***ApplicationManager*** to add it to the current list of figures by calling ***ApplicationManager*** **::AddFigure(…)** function.

   At this step the action is complete but it is not reflected yet to the user interface.

Step 4: Reflect the action to the Interface.

1- The ***ApplicationManager::UpdateInterface( )*** is called to draw the updated list of figures.
2- ***ApplicationManager::UpdateInterface( )*** calls the virtual function ***CFigure::Draw( )*** for each figure in **FigList**. (in this example, function **CRectangle::Draw( )** is called)
3- ***CRectangle::Draw( )*** calls ***Output::DrawRect*****(…)** to draw a rectangle on the output window.

This means there is a draw function in **Output** class for each figure which takes the figure parameters (e.g. the 2 corners of the rectangle and drawing color …etc.) and draw it on the window. The draw in each **Figure** class calls the function draw that draws that figure from Output class. Then the update interface function of **ApplicationManager** loops on **FigList** and only calls function **Draw** of each figure.

Example Scenario 2: SaveAction

- **Note: Save/Load** has NO relation to the **Input** or **Output** classes. They save/load graphs to/from files not the graphical window.

- Here we explain the calling sequence in the execute of 'save' action as an example.
  Note the responsibility of each class and how each class does only its job or responsibility.

- There is a save function in **ApplicationManager** and in each figure class but they perform different jobs:

1. **CFigure::Save(…)**
   It is a pure **virtual** function in **CFigure**. Each figure class should **override** it with its own implementation to save itself because each figure has different information and hence a different way or logic to save itself.

2. **ApplicationManager::SaveAll(…)**
   It is the responsible for **calling** Save/Load function for each figure because **ApplicationManager** is the only class that has **FigList**. Note that it only calls function save of each figure; ONLY calling without making the save logic itself (not the responsibility of **ApplicationManager** but the responsibility of each figure class). This note is important and has a huge grade percentage.

3. **SaveAction::Execute()**
   It does the following:
   - first reads action parameters (i.e. the filename)
   - then opens the file
   - and calls **ApplicationManager::SaveAll(…)**
   - then closes the file

File Format

Your application should be able to save/load a graph to/from a simple text file. At any time during the draw mode, the user can save or load a graph. In this section, the file format is described together with an example and an explanation for that example.

- **File Format**

```
Current_Draw_Color    Current_Fill_Color
Number_of_Figures
Figure_1_Type   Figure_ID    Figure Parameters (coordinates, draw color, fill color ...etc.)
Figure_2_Type   Figure_ID    Figure Parameters (coordinates, draw color, fill color ...etc.)
Figure_3_Type   Figure_ID    Figure Parameters (coordinates, draw color, fill color ...etc.)
……..
……..
Figure_n_Type   Figure_ID    Figure Parameters (coordinates, draw color, fill color ...etc.)
```

- **Example:** The graph file will look like that

```
BLUE    GREEN
5
LINE    1    100    200    17    30    BLUE
RECT    2    20     30     154   200   RED    NO_FILL
TRIANG  3    10     20     70    30    220    190    BLACK    RED
LINE    4    10     200    45    90    BLACK
TRIANG  5    20     30     80    90    220    190    BLUE  RED
```

- **Explanation of the above example**

```
   BLUE     GREEN      //current draw and fill colors that will be used to draw any new figures.
   5  //Total number of figures is 5

   LINE     1     100   200   17    30    BLUE
//Figure1:Line, ID=1, start point (100,200), end point(17,30), color = blue
//note that lines cannot be filled so we skipped this parameter

   RECT     2     20    30    154   200   RED   NO_FILL
//Figure2:Rectangle,ID=2, corner1(20,30), corner2(154,200),color = red, not filled

   TRIANG   3     10    20    70    30    220   190   BLACK      RED
//Figure3:  Triangle,  ID=3,  corner1(10,20),  corner2(70,30),  corner3(220,190),
//color=black, fill=red

   LINE     4     10    200   45    90    BLACK
//Figure4: Line, ID=4, start point (10,200), end point (45,90), color = black

   TRIANG   5     20    30    80    90    220   190   BLUE  RED
//Figure3:  Triangle,  ID=5,  corner1(20,30),  corner2(80,90),  corner3(220,190),
//color=blue, fill=red
```

Notes:

- You can give any IDs for the figures. Just make sure ID is **unique** for each figure.
- You are allowed to modify this file format if necessary **but after instructor approval**.
- You can use numbers instead of text to simplify the "load" operation. For example, you can give each figure type and each color a number. This can be done using ***enum*** statement in C.
- **The LoadAction**:
  For lines in the above file, the **LoadAction** first **reads** the figure type then **creates** (allocates) an object of that figure. Then, it **calls** **CFigure::Load** virtual function that is overridden in the class of each figure type to make the figure load its data from the opened file by itself (its job). Then, it **calls** **ApplicationManager::AddFigure** to add the created figure to **FigList**.

Project Phases

A partially implemented code frameworks for Phase 1 and for Phase 2 are given to you to complete them. We want you to gain the skill of how to continue on a partially implemented code.

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.

1- Phase 1 (Input / Output Classes) [25% of total project grade]

In this phase, you will implement the **Input** and the **Output** classes as they do not depend on any other classes. The **Input** and **Output** classes should be **finalized** and ready to run and test. Any expected user interaction (input/output) that will be needed by phase 2, should be implemented in phase 1.

Input and Output Classes Code and Test Code

You are given a code for phase 1 (separate from the code of the whole project) that contains both the input and output classes partially implemented. Each team should complete such classes as follows:

1- ***Output Class***:

- ❑ **Draw the full tool bars**.
  Output class should create 2 FULL tool bars: one for the **draw mode** and one for the **play mode**. Each contains icons for every action in this mode.
  [Notes]:
  - ❑ Some operations need **more than one icon**, for example, "Pick and Hide" should contain two icons for the two types of picking properties (by type or by color).
  - ❑ The toolbar of the draw mode must contain icons for the **colors** that may be used as drawing or fill color (the required colors are specified above in the "Change Current Colors "operation description).
  - ❑ All icons must be added in phase 1.

- ❑ **A draw function for each figure type** (Line, Rectangle, Triangle, Rhombus and Ellipse). These functions must be able to draw the figures: either filled or not filled, with all supported colors for borders or filling, and draw them either normal or highlighted.
- ❑ Add any other needed member data or functions

2- Input Class:

- ❑ Complete the function **Input::GetUserAction(…)** where the input class should detect all possible actions of any of the 2 modes according to the coordinates clicked by the user.
- ❑ The input class should also recognize the color selected by the user from the color **palette.**
- ❑ Add any other needed member data or functions

3- ***Test Code:*** this is <u>not</u> part of the input or the output classes; it is just a test code (the **main**).

- ❑ Complete the code given in **TestCode.cpp** file to test both Input and Output classes.
- ❑ Do NOT re-write the main function from scratch, just complete the required parts.

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**".
- ❑ You can use an alternative graphics library if you want, but under the condition of keeping the same interface (function prototypes) of the **Input** and **Output** classes and any other classes of phase 2 (same interface but the function body will be using your new library). However, we as TAs do not guarantee the support if you have any problem with your new library. It is your responsibility to deal with it.

Phase 1 Deliverables*:*

On the discussion day, each team should deliver a CD that contains IDs.txt (team number, member names, IDs, email) and phase 1 code that has Input and Output classes and test program completed.

2- Phase 2 (Project Delivery) [75% of total project grade]

In this phase, the completed I/O classes (without phase 1 test code) should be added to the project **framework code** (given for phase 2) and the remaining classes should be implemented. Start by implementing the base classes then move to derived classes.

Phase 2 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) A CD that contains the following:
   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 graph files: at least three different graphs. For each graph, provide:
      i. Graph text file (created by save operation)
      ii. Graph screenshot for the graph generated by your program
      iii. Screenshot of one action of play mode

On CD cover, each team should write: **semester or credit**, **team number** and **phase number**.

Note that Each project phase should be sent first **by mail** (same time for all groups). **No modifications are allowed after the mail delivery.** After that the face-to-face **discussion** of the project will be held. The day of the week of sending each project phase by mail and the discussion schedule of each phase will be announced later. The content of the CD is the same content that should be sent by mail in the mail delivery due date.

Phase 2 Evaluation Criteria

Draw Mode [75%]

- ❑ Each operation percentage is mentioned beside its description.

Play Mode [20%]

- ❑ Each operation percentage is mentioned beside its description.

Code Organization & Style [5%]

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

Bonus [10%]

- ❑ Each operation percentage is mentioned beside its description.

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 figure 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.
- ❑ **Each of the above requirements will have its own weight. The summation of them constitutes the group grade (GG).**

Individuals Evaluation:

Each member must be responsible for some actions and must answer some questions showing that he/she understands both the program logic and the implementation details. Each member will get a percentage grade (**IG**) from the group grade (**GG**) according to this evaluation.
The overall grade for each student will be the product of GG and IG.

You should **inform the TAs** 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.

Note: we will reduce the IG in the following cases:

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

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 (i.e. function **Action::Execute()** which first calls **ReadActionParameters()** then executes the action).
  - Reflect the action to the Interface (i.e. function **ApplicationManager::UpdateInterface()**).

- **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. **FigList** is an array of **CFigure pointers** to be able to point to figures of any type. Many class members should be pointers for the same reason

- **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 class **CRectangle** needs to draw itself on the GUI, it calls function ***Output::DrawRect*** because dealing with the GUI window is the responsibly of class **Output**. Similarly, class **ApplicationManager** must not contain any logic. It only should call functions (the **maestro**). Read the "main classes" section to know the responsibility of each class.

- **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. For example, do NOT add in **ApplicationManager** function **GetFigList()** that gets the array of figures to other classes to loop on it there. **FigList** and looping on it are the responsibility of ApplicationManager. See "Example Scenario 2".

- **Virtual Functions:** In general, when you find some functionality (e.g. saving) that has different implementation based on each figure type, you should make it virtual function in class **CFigure** and override it in each figure type 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.

- **Not all the actions** need to add a corresponding function inside **ApplicationManager**. This will make **ApplicationManager** perform the responsibility of these actions. However, some actions need to loop on **FigList** (e.g. ***SaveAction*** …etc.). In this case only (looping on FigList), you can add functions for them in **ApplicationManager** that loop on the lists and just call functions without making any further logic.

- 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.**

[II] Workload Division Guidelines

Workload must be distributed among team members. A first question to the team at the project discussion and evaluation is "who is responsible for what?" An answer like" we all worked together" is a **failure** and will be penalized.

Here is a recommended way to divide the work based on **Actions**

- Divide workload by assigning some actions to each team member. Each member takes an action, should make any needed changes in any class involved in that action then run and try this action and see if it performs its operation correctly then move to another action.

- For example, the member who takes action **'save'** should create **'SaveAction'** and write the code related to **SaveAction** inside **'ApplicationManager'** and **'CFigure'** hierarchy classes. Then run and check if the figures are successfully saved. Don't wait for the whole project to finish to run and test your implemented action.

- It is recommended to give similar actions to the same member because they have similar implementation. For example: copy, cut, paste together and save and load together … etc.

- After finishing and trying few related actions, it's recommended to integrate them with the last integrated version and any subsequent divided action should increment on this project version and so on. We call this ***'Incremental Implementation'***.

- It's recommended to first divide the actions that other actions depend on (e.g. adding and selecting figures) then integrate before dividing the rest of the actions.