VG101 — Introduction to Computer and Programming

Project 3

Manuel — UM-JI (Fall 2020)

- Include simple comments in the code
- Split the code over several functions
- Extensively test your code and impove it
- Start early and respect the milestones
- Update the `README` file for each milestone
- Update the `Changelog` file between two milestones
- Archive the files (`*.{zip|tar}`) and upload on Canvas

Project instructions

The goal of this project is to better understand Object Oriented Programming. In particular Classes, Inheritance, and Polymorphism are at the core of the project and must be applied in order to complete it. It is highly recommended to:

- Start thinking of the project as early as possible;
- Focus mainly on the organisation at the beginning;
- Define various objects and relate them to each others;

In this project many questions are left to the appreciation of the programmers. Based on your knowledge, research, and understanding argue on your choices in the README file.

The project splits into two parts: (i) the design of a generic parking lot, and (ii) the drawing of an interstellar parking lot using the OpenGL library. The two part should be written independently and be provided with their respective compiling commands.

A Canvas survey will be open for each student to briefly explain his contribution to the project. The result of the survey will be taken into account to determine the grade of each group member. A student not working much is unlikely to receive a good grade. Similarly a student who does not help weaker members of his group will be penalised.

Remark: do not exchange code among groups; Honor Code will be strictly applied.

General guidelines:

- Read, and understand the project specifications, then carefully follow them. If anything is unclear contact the teaching team as early as possible;
- Anything that is not precisely specified is left to your choice. This typically includes I/O format, number or random vehicles to generate etc. Take advantage of this freedom to surprise us, bonuses will be awarded.
- Document the most important aspects of your work in the `README` file. Non-exhaustive list of common information to include: How should we interpret the program output? What are the program main highlights? Why do you think your program is worth more points than others?

1 Project setup

To reward Haruka, Kana, and Chiaki for their good work in the C midterm their parents decide to drive them to the nearest shopping center. However things start to go bad as soon as then enter the

underground parking lot. Despite the sign showing more than half of the slots being empty their parents cannot find any place to park the car. After half an hour spent exploring the parking lot their parents decide to give up and go back home.

Although the three sisters are extremely disappointed, they start thinking and Kana comes to a conclusion: if they could not find any free parking slot it is because the parking lot is badly managed. Indeed when driving around the parking lot they saw that many cars were not properly recorded, that some motorbikes occupied slots reserved to cars, and that no information on where to find a fee spot was provided etc.

Chiaki therefore proposes to fix the problem by programming their own parking lot software in C. Kana and Chiaki feel very excited about it, but Haruka keeps thinking and suggests that C++ might be a better idea for such a project. In fact during the lectures it was mentioned that large projects are easier to write and organise in C++ than C.

After a brief discussion the sisters all agree on what to achieve and how to tackle the problem: they want to write a small program with a demonstration mode where random vehicles enter and exit a parking lot over a given period of time. They have also specified the minimum requirements they want to meet.

Minimum requirements

Parking area: more than one floor, each one being of different size

Vehicles: van, car, motorbike, bicycle

Price: depends on the type of vehicle and time spent

Arrival ticket: when a user arrives he receives a ticket containing:

- Time of arrival
- Type of the vehicle
- Some information (hint) on where to find an empty slot

Departure ticket: when a user leaves he receives a ticket containing:

- Time spent in the parking lot
- Type of vehicle
- Price

As the semester advanced the three sisters have discovered the OpenGL library and are eager to play with it. Upon studying and discussing the "Home sweet home" example from the lectures Haruka suggests to draw a parking lot in OpenGl and get the cars to drive and park in available slots. To add more fun to the original idea Kana encourages them to be more creative: "Why not implementing an interstellar parking lot?"

Inspired by this idea they decide to design an interstellar parking lot and drive a "machine" into a free parking slot. They agree on the following setup.

Parking setup

Initial composition:

- At least twelve slots;
- Amount and type of vehicles: randomly set;
- A barrier blocking the entrance;

Vehicle types:

- Teleported: have reserved slots marked by a rectangle randomly changing color;
- UFO: device that spins on itself;
- Spacecraft: vehicle which continuously zooms in and out;
- Car: regular car;

Vehicle's motion:

- When the barrier opens, move from the entrance to a free slot;
- Follow a smooth trajectory;
- Stop after an empty space and reverse into the slot following a smooth curve;

To clarify their idea Chiaki draws an example where the trajectories are shown in blue and red (Fig. 1). A car is ready to enter the parking lot as soon as the barrier opens. Then it drives to an empty slot following the blue line and reverses into the empty slot following the red line.

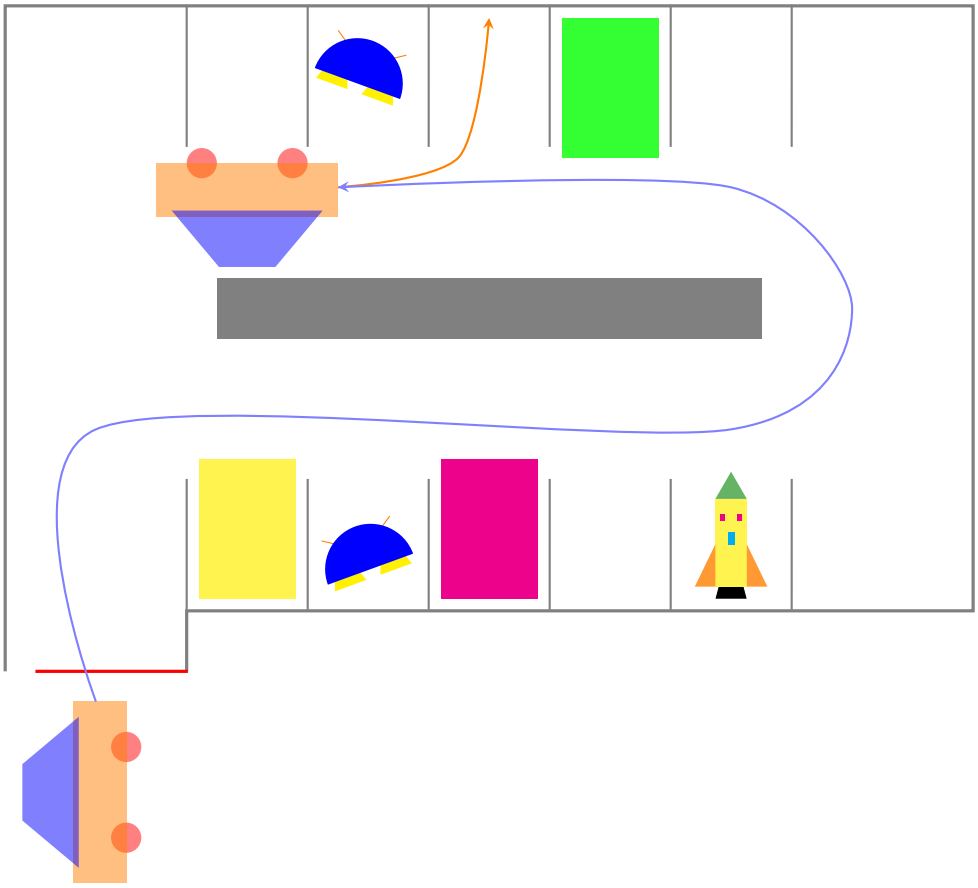

Figure 1: Interstellar car park, example of initial setup

While the three sisters are pretty happy with their setup they realise it will be a bit harder than the first part where they just implemented a parking lot management software. After much time spent thinking

of the best way to organise their project they come up with the following ideas.

Basic guidelines

Work strategy:

- Organise all the objects following a clear hierarchy (partial version shown on Fig. 2);
- Complete the partial classes interface:
  - The `Vec` class defines a mathematical vector; The class should be immutable, i.e. by definition no method is allowed to change any attribute at any time; It is intended for instance to define points without dealing with each coordinate;
  - The `figure` class defines a central point called `anchor` around which the figure can rotate, or zoom. Other methods are also listed;
  - The `Group` class inherits from `Figure` and as such is a figure. It is however composed of other "sub-figures", and as such can contain other `Group`;
- All the attributes should be either `private` or `protected`;
- STL vectors can be used to store objects;

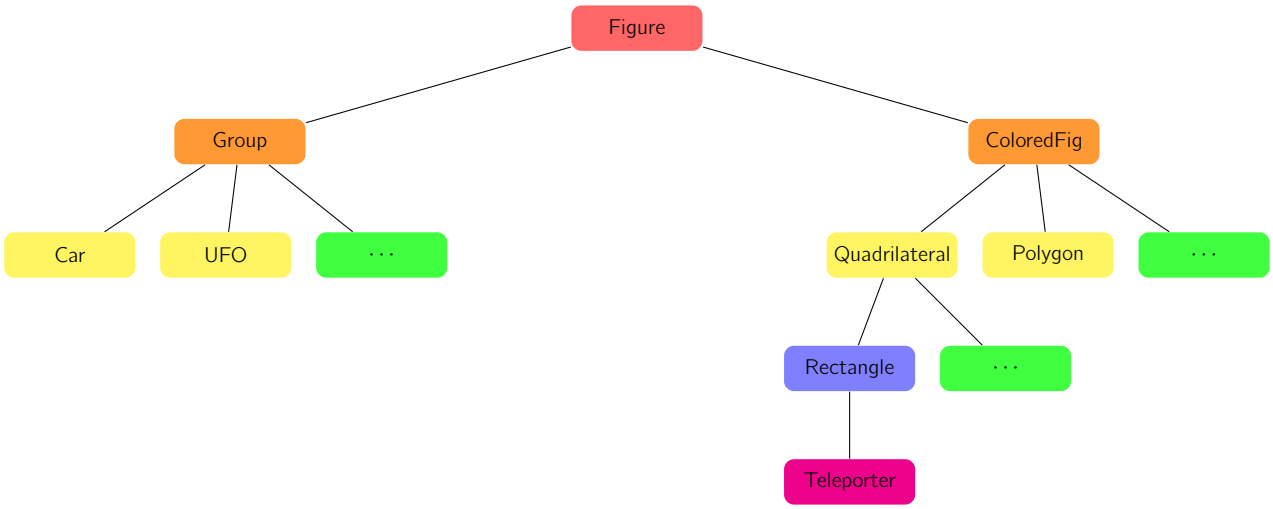

Figure 2: Partial interstellar parking slot hierarchy

Partial classes interface

```
1  class Vec {
2  private:
3      float x,y;
4  public:
5      Vec(float _x, float _y) {
6          x = _x; y = _y;
7      }
8      float getX() {return x;}
9      float getY() {return y;}
10
11     // Example Overloads + operator
12     // returns the sum of 2 Vec
13     Vec operator+ (Vec v) {
```

```
14          return Vec(x + v.getX(), y + v.getY());
15      }
16
17      // Overload - on 2 Vectors
18      // return thier difference Vector
19
20      // Overload * operator on a float k
21      // return current Vector scaled by k
22
23      // Overload * on 2 Vectors
24      // return thier inner product (scaler product)
25
26      // Overload << on an angle
27      // return current vector rotated counter clockwise
28      // by this angle
29
30      // Overload >> on an angle
31      // return current vector rotated clockwise
32      // by this angle
33  };
34
35  class Figure {
36  protected:
37      Vec anchor;
38  public:
39      Figure() : anchor(0, 0) {}
40
41      Vec getAnchor() {return anchor;}
42      void setAnchor(Vec a) {anchor = a;}
43      virtual void draw() = 0;
44      virtual void move(Vec dir) = 0;
45      virtual void rotate(float angle) = 0;
46      virtual void zoom(float k) = 0;
47
48      virtual ~Figure() {}
49  }
50
51  class Group : Figure {
52  private:
53      // A Group of figure "has" other figures.
54  public:
55
56      // We left out the constructor as well
57
58      void draw();       // Draw out all its figures
59      void move(Vec dir);     // Move all its figures
60      void rotate(float angle);     // Rotate the group as a whole.
```

```
61      void zoom(float k);      // Zoom the group as a whole.
62      ~Group() {}
63  }
```

More advanced strategy

While applying the following advice is not mandatory, it can greatly help in the design of a clean project.

As the use of global variables is forbidden it is tempting to "abuse" static variables. However a cleaner way is to implement a `Singleton` class. A singleton is a clean way to ensure an object is not instantiated more than once. This could be useful for instance in the `glDraw` function.

More information on singleton can be found in the following resources:

- `https://en.wikipedia.org/wiki/Singleton_pattern`
- `http://www.yolinux.com/TUTORIALS/C%2B%2BSingleton.html`
- `http://stackoverflow.com/questions/1008019/c-singleton-design-pattern`

Minimal Singleton class implementation

```
1  class Singleton {
2    private:
3      Singleton() {};
4      ~Singleton() {};
5      // omit copy constructor
6      // omit overloading assignment operator
7
8    public:
9      static Singleton* getInstance() {
10       static Singleton *s = NULL;
11       if (s == NULL) s = new Singleton();
12       return s;
13     }
14
15     static void deleteInstance() {
16       Singleton *s = Singleton::getInstance();
17       if (s != NULL) delete s;
18       s = NULL;
19     }
20
21 };
```

2 Project tasks and milestones

The project features three milestones, the last one corresponding to the final submission. Each milestone should take about a week to complete. For each milestone students must submit their current code with

all the usual relevant files attached as well as with a short `Changelog.txt` file that describes the progress

Milestone 1

Tasks to be completed:

- Write a precise interface for all the classes from the regular parking lot;
- Implement a function which finds an empty slot;
- Implement a function which calculates the price to be paid on exiting the parking lot;
- Implements two function to print a ticket for entering and exiting vehicles;

Milestone 2

Tasks to be completed:

- Reorganise milestone 1 into proper classes using inheritance and polymorphism;
- Complete the demonstration mode for the regular parking lot;
- Draw a clear and precise hierarchy diagram for the interstellar parking lot; In particular define all the classes (with their attributes and methods) and use inheritance and polymorphism;

Final submission

Tasks to be completed:

- Following the hierarchy diagram from milestone 2, implement all the classes for the interstellar parking lot;
- Write a simple interface allowing the user to specify the number of slots in the parking and allowing a vehicle to enter and park in an empty slot;

Bonuses

In general, you need to identify parts of the code you believe to be "too complicated", such as

- Duplicated code: code that you are writing over and over again;
- Long functions: functions that are longer then 50 lines;
- Complex logic/calculation: simplify them using similar techniques as `Vec`;

In particular, some ways to get bonus for the interstellar parking lot are provided below.

- Add a small flag on the car, let it move up and down while the car is driving forward.

  The idea behind this is that all object are "rigid bodies" in our setup. In reality it is seldom true. A character will wave his hand when he is moving forward. Can you try and develop a good way of letting a figure has its "internal animation"?

- Avoid a very long drawing function that continuously creates (destroys) objects.

  In the display function (the function where you instantiate and draw figures), will implement all the animations. The code tends to be lengthy, and the logic quite complex. Besides for each frame

drawn, the objects are created then destroyed. This represents a waste of time and resources. It there a good workaround?

One obvious idea would be try to define object outside this function. But since this function does not take arguments, how can we pass our objects into the function? Remember here that global variables are not allowed.

- More complicated drawings.

  Try to make the teleporter more complicated. Can you try to implement a polygon teleporter, with the number of sides changing over time?

- Add an interactive user interface.

  The concept of "function pointer" is mentioned in chapter 12. The technique is used in `glutTimerFunc()`. This type of code is in fact rather common. It is called a `callback()` function. One very important use of callback functions are user interfaces. Things like buttons, text box, scrolls etc.. `Glut` allows you draw such things in your application very easily. Try to add some interactive elements to your code.

- Anything creative!

  We might give big bonus if you show something that we have never thought of and really impressive. Do not restrict your imagination as long as you strictly follow the basic specification!

3 Project submission

Before submitting the project on Canvas ensure the project compiles on JOJ.

- A project that has not been submitted to JOJ will not be graded;
- JOJ will compile the code using the flags `-Werror -pedantic -Wall -Wextra -std=c++11`;
- A project that is not compiling on JOJ will not be graded;
- No test case is offered for the project, the goal of JOJ is only to ensure the written code compiles and complies with the C++ standard;
- JOJ submissions will be open for the whole duration of the project such that compliance with the C++ standard can be tested at any time;
- Another instance of JOJ will be available only for the final submission. Submissions failing to pass this test will not be graded, i.e. receive a 0;

3.1 Basics on the grading policy

The project will be graded with respect to the following aspects.

- Running effects: what is the end result of the code? For instance, for the regular parking lot is the demonstration mode clear and easy to understand? Is the simulation really random, i.e. covers most of the common cases. For the interstellar parking lot, is the vehicles' trajectory smooth enough?
- Design and structure: is the program structured in a clean way? Are you abusing the `switch` statement? Do you have much duplicated, e.g. copy/paste drawing code for `Rectangle` and `Square`? Is the program organised into sub-functions and classes?

- As a general rule a function longer than 50 lines should be redesigned and split into smaller functions.
- Grammatical correctness: do all the `new` having a corresponding `delete`? Are the opened files closed? Have the keywords `public`, `protected` and `private` be used appropriately?
- Is the `Vec` class **immutable**? Remember that the member function is not allowed to change any attribute!
- README, highlights, and Bonus: is your README concise yet containing all necessary information? If you have written something you feel deserves a bonus, then convince us to give it to you, i.e. clearly explain everything in the `README` file.

4 FAQ

This section lists Frequently Asked Questions (FAQ).

1. For the regular parking lot where do the vehicles in the simulation come from, files or user input?

   Neither, you randomly generate them inside the program. You should generate randomly arriving vehicles, with random type, random parking duration, etc.. The program should inform the user of all necessary attributes of the vehicle whenever a it arrives or leaves.

2. What are the `Vec` class and the + operator?

   The `Vec` class models a mathematical vector, or equivalently the concept of "Point". As it makes sense to add two vectors together, we define their sum with the + operator. This is called operator overloading. Search online how to write operator overloads. We provide a simple example showing how to add two vectors using the + operator.

   ```
   Vec v1(1,0);
   Vec v2(1,1);
   Vec v3 = v1 + v2; // v3 is now Vec(2,1);
   ```

   Advice: avoid directly computing coordinates, and instead use the `vec` class as much as possible;

3. Do we need to draw the trajectory in the window?

   No, you are not require to, but some good implementation may lead to bonus. But you do need to make sure the trajectory of the car is smooth enough (avoid sharp turns, spinning on a spot, etc.).

4. Do we need to submit the class inheritance relationship? In what form?

   Yes. This can done either by providing a picture of the relationship (a photo of a clear and neat handwritten drawing is acceptable, but a computer generated version is preferred, e.g. LaTeX). A text file is also acceptable as long as it looks like a drawing, e.g.

   ```
   figure------group ------- UFO
           |           |---- ...
           |
           |---coloredFig------- triangle
           |               |---- ...
           ...
   ```

5. Can we change the given class interface/inheritance relation?

   Yes. In such a case do not forget to document and argue on why the chosen strategy is better than the suggested one.

   For the new inheritance relation, argue why the old inheritance relation is invalid in your design, and why the new one is more logical and efficient.

6. Can we decide to use public attributes instead of private? Can we decide that a `vec` is not immutable?

   No, those requirements are part of good coding quality. Breaking them will bring large penalties on the project.

   Even if the class interface and inheritance relations are altered the following rules always apply: (i) all attributes must be `private`, and (ii) the `Vec` must remain immutable;

7. What is the use of the "anchor"?

   Think of this concept as the "Frame of reference" in physics. Each figure has its own frame of reference, and the anchor vector (points) specifies where the origin of this frame of reference is. Since the object always rotate/zoom around its anchor point, think how you can take advantage of this together with << in `Vec` to simplify things.

8. Is there an easy way to perform rotation?

   Yes, instead of using many `arcsin` and `arctan` etc., an easy way to perform rotation is to use a "rotation matrix". Refer to `https://en.wikipedia.org/wiki/Rotation_matrix` for more details om how to achieve it. Then a vector can easily be rotated by using simple multiplication/addition.

9. Is there a "criteria" for a "good" design?

   Yes, but part of it is related to personal taste. In general a good design should be able to be regarded as good by most people looking at it.

   A minimum requirement is *consistency*. For example when implementing `Group`, it is possible to say that a `group` contains ("has") another `group`. This is consistent as a `group` "is" a `figure` it can be part of a larger `group`.

   Another example is related to the operations performed on a `group`; If one applies the following methods

   ```
   Move(Vec(1,0)); Zoom(3.0); Rotate(PI / 6); Draw();
   Move(Vec(-1,0)); Rotate(-PI / 6); Zoom(1.0/3.0); Draw();
   ```

   then logically, the figure should be identical to the one before these operations. As a result two identical figures should be drawn at the same place. In particular, neither the `Draw()`, nor the order of the `Move`, `Zoom`, and `Rotate` operations should affect the result.

   Ensure your design and implementation fit these minimum criteria.

10. Can we use Windows or MacOS specific functions?

    Simple answer is "No". But if for any reason you decide to do so, explain your reasons clearly in the README. This however does not mean your reason will be accepted, and major deductions might be applied. **Before using any OS specific function refer to the instructor.**

11. Can we change the setup?

    As this may surprise you, yes you can. You are allowed to change the setup, as long as you are "generalizing" the problem, or equivalently, not making the problem "easier". This is especially true for exercise 2. We encourage you to try more directions, and develop new methods.