Olympiads in Informatics, 2007, Vol. 1, 149–164

Increasing the Appeal of Programming Contests with Tasks Involving Graphical User Interfaces and Computer Graphics

Pedro RIBEIRO
Departamento de Ciência de Computadores, Faculdade de Ciências, Universidade do Porto
Rua Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
e-mail: pribeiro@dcc.fc.up.pt

Pedro GUERREIRO
Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa
Quinta da Torre, 2829-516 Caparica, Portugal
e-mail: pg@di.fct.unl.pt

Abstract. Programming contests should be capable of being appealing to both the contestants and the general public. We feel that the use of graphical user interfaces and computer graphics could help achieve this goal, providing new ways of viewing the task. We describe experiments we made with games (Tic-Tac-Toe, Snake and Ataxx, an Othello-like game), which were made available to students with graphical components, and discuss the results. We also present a simple graphic library where simple drawings can be made and show how it can be used in a programming contest environment. We then conclude by revisiting some past IOI problems, suggesting ways to enhance them with graphical components.

Key words: programming contests, graphical user interfaces, computer graphics.

1. Introduction

One of the main goals of programming contests is to draw the attention of the public to the contestants, to their achievements and to their extraordinary capabilities in the art and science of programming. Therefore, the more popular our programming contests become the closer we will be of reaching that goal. On the one hand, we need the public to understand or at least to have a reasonable idea of what the problems are about; on the other hand, we must design the tasks in a way that is appealing to the contestants, and especially to the newcomers, so that we can gather more and more participants, and so that our contests cause a greater impact on society. As observed in (Dagiene, 2006) the tasks are the keystones of contests. Their attractiveness to the contestants and public is vital. Therefore, anything we can make to improve this aspect will benefit our programming contests.

Traditional tasks in the International Olympiad in Informatics (IOI) usually involve simple text input and output. Even with improvements such as output only tasks or reactive tasks, the interaction between the program and the outside world is always done in a purely textual way. No other ways of observing the programs working exist, other than looking at the input and the output, which typically are just a bunch of numbers or strings. This approach lacks almost any attractiveness factor, and is meaningless to the general public, since it normally needs some pre-knowledge of the problem statement in order to minimally appreciate what the program is doing.

We have been experimenting with tasks that use graphical user interfaces and computer graphics in preliminary contests, in secondary schools in Portugal, and also in competitive assignments in our programming courses at the university. We believe that this is a domain that can be used to make the contests more attractive and more fun. The use of graphics could dramatically improve the manner in which people visualize the tasks, the test cases and the program outputs, creating a whole new level of ways to observe a programming contest. It also presents the non-informatics audience with an opportunity to perceive and more duly appreciate what the contestants have accomplished. Even the contestants could sometimes benefit of seeing how their program works in a different way. This graphic part should not take precedence over the educational and algorithmic component of the tasks, of course, but we claim that both these aspects can be merged, thus paving the way to interesting new problems.

In this paper we will present some of those experiments, discussing some of its advantages and disadvantages, and we will show how they could be adapted to programming contests like the IOI.

2. Graphical User Interfaces in Programming Tasks

Many IOI tasks are interactive, in the sense that there is an agent reading the output of the contestant's program, which, in turn, has to read the output by the agent. Typically the dialogue is carried out at the console, on line by line basis. Although this is sufficient in the strict setting of problem solving, we believe that in some cases the task could be made more appealing by having the agent respond graphically and by giving visual feedback to the moves by the program. In fact, the task not only becomes more appealing and more fun, but also the visual clues may indeed help the abstract reasoning that leads to the solution.

We have been experimenting along these lines in our own introductory programming courses, at secondary level and at university level. In these courses, some of the labs are competition-like: the students are given a task, they have to write a program for it, and submit it to an IOI-like automatic judge. The automatic judge we use is called Mooshak (Leal and Silva, 2003). It runs a sequence of test cases and informs "Accepted", "Wrong Answer", "Time Limit Exceeded", etc. In principle, "Accepted" means that all test cases passed, which is fine for class use. In competition mode, partial points are possible.

Enhancing competitions tasks with a graphical interface does not cause any penalty on the automatic judge, who still has to check text output only. The extra burden is a responsibility of the agent, and does not interfere greatly with the contestant's job.

We will now describe three of those experiments. The first is the well-known Tic-Tac-Toe problem, and uses an elementary form of ascii-graphics. The second is “Snake”: in this case, the snakes are represented by colored blocks on the screen. The third is an Othello-like board game, and input was done via the mouse, not via the keyboard. In all these cases, the games act as platforms for the programming tasks, providing immediate visual feedback to the students. This visual feedback offers new possibilities of reasoning about the program and checking its result.

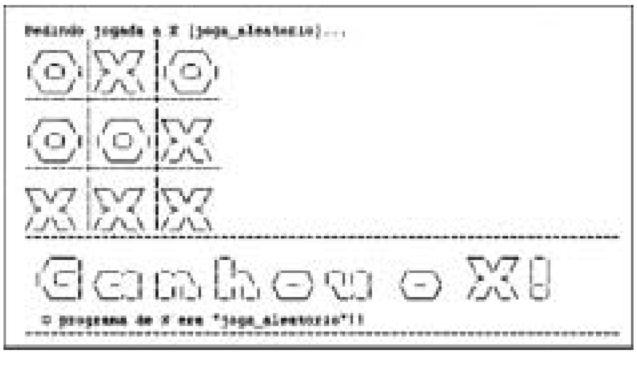
2.1. *Tic-Tac-Toe*

Tic-Tac-Toe is a well-know game and programming it is a typical exercise. We used it in an introductory C programming course for secondary students. The task was to create an agent with artificial intelligence for the game, in a way that agents could play against one another. We organized a tournament among all valid players and in order to validate the contestant submission we used black-box automatic evaluations with Mooshak.

Communication between agents is accomplished in a standard way, as in traditional IOI input/output programs. Each agent receives on the standard input the current state of the game, i.e., the configuration of the board, and writes the chosen move on the standard output. Each move is triggered by making a new call to the student program. This functions almost like an IOI test run, in the sense that the program is called from scratch for each move, without any type of memory between program calls. Each game is processed in the following way: the server initiates the game, with a circular list of the programs that are playing; then automatically calls the next program from the list, feeding it with correspondent input, reading its output and continuing in the same fashion until the game is over, where it communicates the final result.

In this example, the graphical component is built using character strings. It is a naïve design, but provided adequate visualization of the game. And, being very simple, it could be passed to the students, so that they could also learn from it. Fig. 1 shows a snapshot of this component in action.

This apparently simple scheme proved to be very addictive to the students, who had an immediate form of seeing the results of their programming effort. The students also used



```
Pedindo jogada a X [joga_aleatorio]...

Ganhou o X!
O programa de X era "joga_aleatorio"!!
```

Fig. 1. Graphical component of the Tic-Tac-Toe server (with Portuguese text).

the server to show their families and friends what they were doing in the programming course, thus helping explain to the "outside world" what programming is about. During the tournament, the games were projected in the classroom to a small audience, in an enticing way that could not be achieved if we had stuck to the elementary form of input and output.

This was precisely what we wanted to happen, and it corresponded perfectly to our aim of trying to show the "outside world" what programming can be.

2.2. *Snake*

"Snake" is another popular game. Each player controls a snake in a grid environment, striving to make it eat as much food as possible, without hitting the walls or other snakes (including itself). Each time a snake eats an "egg", its size increases by one. This game also provides a good platform for programming tasks.

We used this game in a Logic Programming course at the university, very much like we had used Tic-Tac-Toe for secondary school students. The students now had to program an artificial intelligence for a single snake and the final goal was to make a tournament with two available modes: all against all and solitary. In the first case, the winner is the last surviving snake; in solitary mode, the winner is the snake that eats the eggs fastest.

Communication between the server and programs driving the snakes was similar to the Tic-Tac-Toe program, and the server operated in the same way.

The graphical component was more sophisticated. It was designed for Linux, using XLib (Gettys and Scheifler, 2002). It displays a window in which all snakes, eggs and other food items are represented. Fig. 2 shows an example.

In this experiment, students only had the duration of a single lab (2 hours) for programming their snake agent. This matches contest time constraints.

Again, the experiment was very successful, and we could observe pride and joy on the face of some students when the final tournament was projected in the classroom. Some even brought in their friends (with no programming background), who came to just watch and have fun.

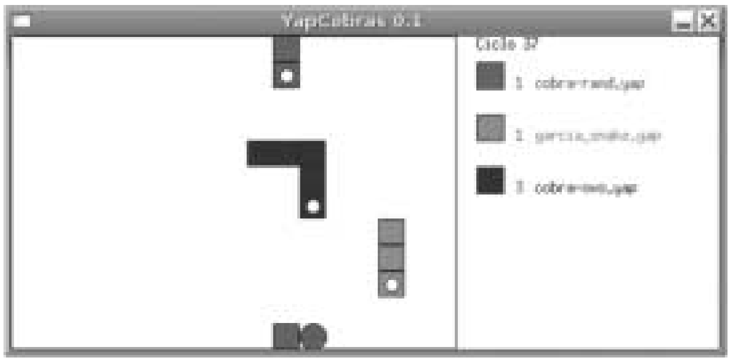


Fig. 2. Graphical component of the Snake server (with Portuguese text).

Out of curiosity, it should be noted that we also used Mooshak automatic evaluation for Logic Programming (Prolog), not only in this experience but also in other exercises. In fact, Mooshak is very configurable and we make extensive use of it in our courses, for different programming paradigms, both at introductory and advanced levels.

2.3. *Ataxx*

Our third experiment was more ambitious. The task was a more complete programming project that took a few weeks to complete. We chose Ataxx, a board game similar to Othello. For a better understating of the challenges involved, we will briefly explain the game rules, but a more detailed explanation can be seen on (Beyrand, 2007). Ataxx is a 2 player game played on a 7×7 grid board. The players start with two pieces each, on opposite corners of the board and take turns to play. Each time, we can make two types of moves: we either add a new piece, by making it appear on an adjacent square to an already existent piece, or we move a piece, by making it jump at most two squares. After each move, pieces that are adjacent to the piece that was added or that moved turn to our color. In the end, the player with more pieces wins.

Like before, the main goal was to make an agent capable of playing the game automatically. This was a Logic Programming task, to be programmed in Prolog, and the communication was handled directly, by calling a logic predicate with the game state as input. The game state was given as a list representing the board and an integer representing the available time. This second parameter was necessary because the agents had a time limit for their moves.

We implemented Ataxx using a general architecture where a text-only server was responsible for implementing the game, its rules, and calling the agents. This server could be called to automatically run a game in the fastest possible way, in batch, with no human interaction. On the other hand, a Graphical User Interface (GUI) could be attached to the text-server, making use of standard input/output interaction. This module is independent from the game itself. Fig. 3 shows a basic view of the architecture we used.

The text server functions basically as the previous ones. It initiates the game, calls the programs for the players in the right order, and announces the final result when the game is over.

The GUI itself is more refined that the previous ones. We used Java Swing (Project Swing) for it and included a clickable interface to be used by humans playing the game. This way, it was possible to have computer vs. computer, human vs. computer and human vs. human matches. Actually, this possibility may be used by the students to become more

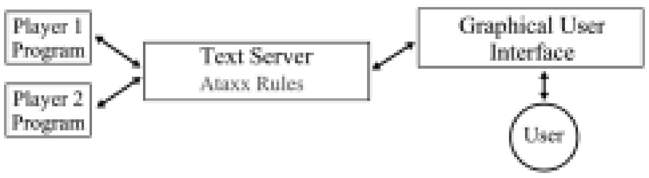


Fig. 3. Ataxx experience architecture.

familiar with the game and devise the strategies that they will later implement. The GUI included animations for the piece captures and provided ways to automatically run a full game, advance turn by turn, undo moves and to load/save game logs. Fig. 4 displays a snapshot of the GUI in action.

For guiding the students in developing their solutions, we designed a sequence of tasks, each of which was handled as a competition problem: students should complete each task and submit it to the automatic judge Mooshak for validation. For example, we had a task for computing the list of adjacent squares, for checking whether a move was valid, and for generating the list of all possible moves, given a board description. We held several "mini-tournaments" where students could participate in order to see how their agents were doing, and also just for the fun of it.

In the end, we had 47 functional programs to grade. Some of them were quite good players, constantly beating known Ataxx programs, by a large margin. For grading, we first created a set of agents, with which we made an initial estimate of the agent's "merit". With this we could identify, for example, the programs that were not even able to win against a random opponent. After the initial analysis by the grading agents, we grouped the programs in three different leagues, according to their strength, and then, in each league we carried out a large-scale tournament, in which all played against all. The final score was a function of the ranking in the tournament. In the end, the best programs were announced in the department, and their authors publicly recognized.

This experiment was very successful and the students were strongly motivated by it. The graphic component, including the animations, had a great impact, and we know that some students used it to show their work to their non-informatics friends. The Ataxx project also had a positive influence on the results in the course. In the survey we carried at

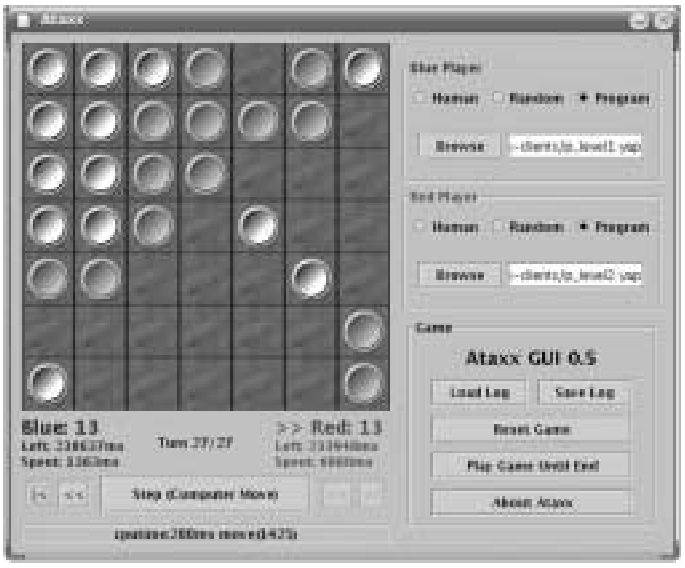


Fig. 4. Snapshot of the graphical user interface for Ataxx.

the end of the semester, the course was rated highly, due in large part to this programming task.

We now use this program as an example of student projects that we present when prospective secondary school students visit us. Needless to say, without the graphical component, that would not be possible. The Ataxx server and related resources are publicly available at (Ribeiro, 2007).

3. Simple Graphics for Simple Drawings

Many problems in IOI competitions are geometric, and would become more dramatic with a graphical output. Take, for example, problem Joining Points from IOI 2006 in Merida, Mexico (IOI Secretariat). The goal is to join pairs of *green* points and pairs of *red* points according to certain rules. The output is merely the sequence of pairs of points, given by their position in the input files. Of course this is enough, as far as algorithmics go, but the problem would be more appealing and less abstract if the task was to actually draw the figure that comes out as the points are joined. Indeed, it might happen that by actually watching the drawing appear on their screens, albeit in the wrong form in their early experiments with the problem, more contestants would be able to figure out the correct strategy.

This example suggests that two additional aspects need be considered when designing contest problems with a graphical output: we need to agree on a common graphical library and we must be able to automatically judge the graphical output.

The graphical library is necessary because students use different languages and operating systems, and we must ensure that no environment gives an unfair advantage to contestants using it. On the other hand, professional graphical libraries are too vast for the needs of common problems and it would be a waste of time trying to completely master them, in preparation for IOI. By providing a simple graphical library, we could achieve an informal competition "standard" that would be followed worldwide. As a side effect, it could eventually spread out and be used in general for teaching programming in general.

In programming competitions such as the IOI and ACM-ICPC (ICPC site) most tasks have text output. Automatic judging is performed either by directly comparing the output of the contestants program to the official solution or by running a validating program, specific to each problem, that checks whether the output is correct. This second solution is necessary when there are many solutions and it is not adequate to pick one of them as a representative. We anticipate this will be the common case with graphical programs: the programs must draw a picture but it does not matter which parts of the figure are drawn first. Therefore, except for very simple cases, graphical programs will be judged via ad-hoc tester programs.

We will now present the sketch of simple graphical library along the lines we have discussed. It has been tested within our own programming courses, where most of the programming assignments are contest-like.

3.1. *Classes for Figures*

The first class in our library is class Point, representing points in the two dimensional plane. It is useful directly in many problems and it is used in many other classes: Circle, Polygon, Polyline, Segment, etc. For example, in the Joining Points problems that we mentioned, we are given two arrays of Points. Actually, there are green points and red points, which means perhaps we should handle them via subclasses for colored points. The result is an array of segments. If from a pure algorithmic perspective, segments are just pairs of points, and this point of view would be enough for solving the problem as presented, the graphical variant that we suggest would be better handled by using a class for colored segments as well.

All these classes – Point, Circle, Polygon, Polyline, Segment – and also, the classes for colored points and colored segments, represent figures, and this leads us to an abstract class Figure, from which the others derive. Class Figure is another fine example, because it declares as pure virtual functions those functions that all figures must implement and also declares and implements abstractly a number of template functions in terms of the pure virtual ones. These will be available without further effort in all derived classes. Here is a sketch of the class declaration, in C++:

```
class Figure {
public:
    // basic functions
    virtual void Scale (double fx, double fy) = 0;
                                       pre IsDeformable () || fx == fy;
    virtual void Translate (double dx, double dy) = 0;
    virtual void Rotate (double angle) = 0; // pre IsRotatable ();
    // preconditions
    virtual bool IsRotatable () const;
    virtual bool IsDeformable () const;
    // template functions
    virtual void RotateAround (double angle, const Point& p);
    ...
};
```

The three basic functions, Scale, Translate and Rotate, are pure virtual. The template functions have default implementations or are programmed in terms of those three. Each class derived from Figure will provide an implementation for the three basic functions and inherit the remaining functions. Boolean functions IsRotatable and IsDeformable are used as abstract preconditions for commands Rotate and Scale (Meyer, 1997). They are defined here returning true: descending classes whose objects cannot be rotated (for example, rectangles with horizontal and vertical sides) must redefine IsRotatable; descending classes whose objects cannot be scaled differently in along the two axes (for example, circles) must redefine IsDeformable.

3.2. *Clone and Draw*

Sooner or later, we will need containers of polymorphic figures. In C++ these can be implemented as vectors of pointers to Figure. When adding a figure to a container, we prefer not to manipulate the pointers explicitly. Therefore, each class derived from Figure must provide a virtual function Clone that creates a dynamic copy of its object. The situation would be simpler in Java, where the objects have reference semantics. Still, when adding a figure to a composite figure, we do not want to add a reference to the argument: we want to automatically create a copy of the argument and store the copy, so the argument and the composite figure can evolve independently. Therefore we also need to clone in Java, and this should be done avoiding the complications of the built-in clone function (Bloch, 2001).

For graphics, we want each Figure to be able to draw itself, and thus class Figure must declare a function Draw. The case for function Draw introduces another issue: where will the drawing be made? In other words, when we write, we write on a stream; when we draw, where do we draw? In C++, when we write a value of type double on a stream, it is the stream that decides how the number will be written: fixed point or scientific notation, how many decimal places, etc. Likewise, when we draw, there must be some structure (i.e., some class) that "decides" how to draw: which color, which line thickness, etc. Well, that structure will be called canvas and we posit an abstract class Canvas to represent it.

Function Draw should also be pure virtual in class Figure:

```
class Figure {
public:
    // ...
    virtual Figure* Clone () const = 0;
    virtual void Draw (CanvasPtr) const = 0;
};
```

Please note that the argument of Draw is of type CanvasPtr. This is a typedef that represents a pointer to Canvas.

3.3. *Classes for Canvases*

A canvas is an object where figures are drawn. Canvases will be implemented in terms of an underlying graphics library, and will effectively shield the users from having to deal with that particular library. Instead, they will only have to learn a few functions: those provided by class Canvas, the abstract class for canvases:

```
class Canvas {
public:
    virtual void DrawPolygon (const std::vector<Point>& p) = 0;
    virtual void DrawRectangle(const Point& p1, const Point& p2) = 0;
    virtual void DrawEllipse (const Point& p1, const Point& p2) = 0;

    virtual void DrawPoint (const Point& p) = 0;
    virtual void SetColor (const Color& c) = 0;
    virtual void SetPenWidth (double t) = 0;
    virtual void SetFilled (bool b) = 0;
    // and the corresponding getters ...
};
```

There are several Draw functions, all with arguments of type Point or std::vector<Point>. These functions handle the common cases, but a few more could be added. Then we have functions for setting the color, the pen width and the font, and the corresponding getters. Function SetFilled determines that the figure will be filled with the current color if the argument is true, or not if it is false.

Class Canvas is abstract. Each concrete class that derives from class Figure will implement its function Draw in terms of the re-sources provided by the abstract class Canvas. The library provides four concrete canvases: CanvasText, CanvasBasic, CanvasExtensible and CanvasEuclidean. When drawing, the user must select one of these canvas classes.

CanvasText is actually not a graphical canvas: it is linked to a stream, and figures are drawn by writing their description on the stream. This textual output can be processed by a viewer program, but we may use it for unit testing (SUnit site) and in the realm of programming contests, we need it for automatic evaluation.

CanvasBasic makes a one to one mapping between window coordinates and the coordinates of the figure, but leaves the origin on the lower left corner of the window, and has the y-axis growing upwards. It is useful in simple cases and it is the base class for the other CanvasExtensible and CanvasEuclidean.

CanvasExtensible is associated to a form and is extensible in the sense that the constructor sets the extension of the larger side of the form. The x-axis runs through the bottom of the window, from left to right, and the y-axis runs through the left side, from bottom to top. If, for example, the extension is 300, and the window is wider than tall, then the visible y-range is zero to 300, and the visible x-range is from zero a number greater than 300, computed so that x units and y units measure the same on the screen.

CanvasEuclidean is similar to CanvasExtensible, but allows for negative coordinates in both axes. More precisely, users can specify the x-range and the y-range arbitrarily.

Even though pictures provide rich visual feedback for the workings of the program, situations occur, typically when debugging, where we want to observe precisely the sequence of elementary strokes that make up the drawing. In principle, this can be achieved directly by simply changing the canvas where the figure is drawn from one of the graphical ones to CanvasText. We have observed that this technique is very effective.

We already mentioned that CanvasText is for automatic evaluation. We have tested this in the programming assignments. The idea was to validate some new transformations, before they were applied in the figure that was being composed. This validation was performed by the automatic judge that we use, Mooshak, on a program drawing on an object of type CanvasText, wrapping the output console. Once the drawing was accepted, students could switch to one of the graphical canvases.

3.4. *Example*

To illustrate the use of our library, let us consider a program to draw simple fractals that was used in a programming assignment with automatic judging. A fractal is a polygon; therefore, class Fractal inherits from class Polygon:

```
class Fractal: public Polygon {
private:
   // some data members for fractals
public:
    virtual void Read (std::istream& input);
    virtual Fractal Next() const;
    virtual void Transform();
};
```

Function Read reads the fractal data from an input stream; function Next yields the next fractal in the generation process; function Transform assigns the result of Next to the target object; function Draw does not appear because it is inherited from Polygon. Here is a simple test function that reads the description of a fractal from the standard input, transforms it twice and "draws" it on the standard output using a CanvasText object:

```
void TestFractal () {
    Fractal f;
    f.Read (std::cin);
    f.Transform(); f.Transform();
    Canvas* t = new CanvasText (std::cout);
    f.Draw (t);
    delete t;
}
```

This test function is part of a console application. In order to draw graphically, we need a form application, but the main difference from the console application is that now we draw on a graphical canvas that wraps the form, not the console. Typically, this happens in a member function of the form class.

3.5. *Composite Figures*

The fractal example that we mentioned before was very simple: there is a single polygon that must be drawn. In general, we want more complicated figures. Anyway, we can still use exactly the same technique provided we can handle composite figures, and we can do that using the Composite design pattern (Gamma *et al.*, 1994): a composite figure is a figure that is made up of a sequence of figures:

```
class FigureComposite: public Figure {
private:
    std::vector <Figure *> components_;
public:
    // Constructors
    // Inherited functions to be redefined

    virtual void Put (const Figure& f);
};
```

Function Put is the only novelty: it adds another figure to the composite figure. Scaling, translating, rotating, or drawing a composite figure means scaling, translating, rotating or drawing each of the components.

We mentioned that the solution for the joining points problem is an array of pair of points. Actually, we could use a composite figure, made up of colored segments.

3.6. *Colored Figures*

From what we have described, we may infer that each figure is drawn using the current value for the color, the pen width, and filled attributes. This implies that the components of a composite figure are all drawn with the same color, same pen width, etc. We do not want that, in general. On the contrary, we need colored figures, which are drawn according to their own properties, not the properties of the canvas.

This means that for each figure in our class library we need a colored version. We can accomplish that using inheritance, of course: for example, class ColoredPolygon inherits from Polygon and adds data members and functions to deal with the color and other graphical attributes.

This solution, however, is not very attractive: it would duplicate the number of classes we have to maintain. A better approach that we can use with C++ relies on generic programming. We define a template class Colored<T> and the instantiate generically with subclasses of Figure, as needed. Here is Colored<T>:

```
template <class T>
class Colored: public T {
private:
    Color c_;
    int w_; // pen width;
    bool f_; // filled
public:
    Colored (const T& t, Color c = Color::black, int w = 1,
                      bool f = false):
       T (t), c_(c), w_(w), f_ (f) {
    }

    virtual Colored<T>* Clone() const {
       return new Colored<T>(*this);
    }


    virtual void Draw (CanvasPtr x) const {
       Color c1 = x->GetColor ();
       double w1 = x->GetPenWidth ();
       bool f1 = x->GetFilled ();
       x->SetColor (c_); x->SetPenWidth (w_); x->SetFilled (f_);
       T::Draw(x);
       x->SetColor (c1); x->SetPenWidth (w1); x->SetFilled (f1);
    }
};
```

Note that template class Colored<T> inherits from its formal argument. Hence, Colored<Polygon> inherits from Polygon, and exhibits normal polygon behavior, to which it adds color and pen width, as required.

This property of C++, of allowing generic classes to inherit from its template argument, is uncommon, but offers an elegant solution to our problem. Note that this approach could not be applied to Java or Eiffel, for example, even though these languages cater for generic programming as well. In Eiffel or Java, the solution would be to design a class Colored, inheriting from Figure, with a component of type Figure, representing the object whose graphical properties we want to control. This is also an interesting situation (also possible in C++, of course) that corresponds more closely to the design pattern Decorator (Gamma *et al.*, 1994).

As an example of this technique at work, observe a class representing the flag of Sweden: a cross made of two yellow rectangles on top of a blue rectangle:

```
class Sweden: public FigureComposite {
public:
    Sweden ():
       FigureComposite () {
       Rectangle r(Point(0, 0), Point(10, 8));
       Colored<Rectangle> rc (r, Color::blue);
       Rectangle v(Point(3, 0), Point(5, 8));
       Colored<Rectangle> vc (v, Color::yellow);
       Rectangle h(Point(0, 3), Point(10, 5));
       Colored<Rectangle> hc (h, Color::yellow);
       Put(rc); Put(vc); Put(hc); Scale(100, 100);
    }
};
```

This concludes the presentation of our simple graphical library.

4. Graphical Tasks

Having discussed our experiments in the area, we will now show how some past IOI tasks could have been graphically enhanced using the approach we advocate. With these kinds of problems, the IOI tasks could more easily be brought to the general media, like the press and the television, which are inherently graphical biased.

We will now give list of some of those problems. Note that we select at least one problem per year since 1994, as an indication that this approach applies widely. All task statements can be seen on (IOI Secretariat):

- *IOI'94, task "The Clocks"*: compute the minimum number of moves to turn 9 clock dials to 12 o'clock. A GUI could have been provided, and it could graphically show the moves happening, with animations. Since the output of the tasks was the set of moves that should be made, nothing had to be changed. Generally speaking, almost all search problems could be made visual if the states can be represented visually in a simple suggestive manner (for example, *MagicSquares from IOI'96* is similar).
- *IOI'95, task "Packing Rectangles")*: Find the smallest enclosing rectangle into which other four rectangles may be fitted without overlapping. Instead of numbers, the output could be really graphical, with drawing commands, giving a new insight into how are the rectangles being packed, and which kind of algorithm is being used.
- *IOI'96, task "A Game"*: A 2-player game where players take turns to collect numbers from the ends of a sequence of integers, winning the best sum of numbers. A fully functional GUI could have been made, which would also allow the player to make games between two versions of his program. Generally speaking, almost every 2-player game that appeared in IOI could have a GUI (for example, *The Game of Hex from IOI'97* could also have a GUI).
- *IOI'97, task "Stacking Containers"*: A crane operates several containers, for storage and removal, in a 3-dimensional world. A GUI could have been used to show the effects of the algorithms used, giving a visual depiction of the simulation. Generally speaking, almost every problem that is related to a 3D world, can have a

visually improved interface (another example is *The Toxic iShongololo, also from IOI'97*).

- *IOI'98, task "Starry Night"*: Detect similar clusters of stars in the sky. This is an example with a great potential public appeal if it used a graphical GUI to show the stars.
- *IOI'99, task "A Strip of Land"*: Find a rectangular region for an airport with the largest area subject to width and height difference constraints. A GUI could show in 3D the area to calculate, giving an attractive view of the task.
- *IOI'00, task "Building With Blocks"*: Find the way to decompose a 3D solid in a minimum number of different small blocks. Once more, the 3D nature of the task could be really used, and a GUI would even allow the contestants to have a better understanding of the problem.
- *IOI'01, task "Ioiwari game"*: A Mancala like 2-player game. Do we need to say more? In this year, there was also another 2-player game task (*Score*).
- *IOI'02, task "Xor"*: An output only-problem where one has a *xor* operation to transform a blank screen into a figure. A GUI would have permit a visual depiction of the solution, but would also give the contestants a whole new way of trying to make "manual solutions" in their heads (that could have algorithmic nature).
- *IOI'03, task "Seeing the Boundary"*: Count how many fence posts one can see from a determined position with some rocks obscuring posts. This is a very graphical problem, and a GUI could have helped the contestants to see exactly why a determined input gives the correspondent output.
- *IOI'04, task "Polygon"*: An output-only problem where we must do a reverse Minkowski sum on a polygon. Even the name of the task is telling us it could use graphics. A GUI could have shown much more clearly the structure of the problem and allow for different approaches to the task. The graphical library could have been used to draw the polygon, instead of simple text output.
- *IOI'05, task "Rectangle Game"*: A two-player game where players take turn to divide the rectangle. All that has been said before about games can be applied.
- *IOI'06, task "Joining Points"*: This problem was already approached before (on chapter 3) and it constitutes a paradigmatic example of a task with almost everything in favour for using a graphical approach.

5. Conclusion

IOI is an algorithmic competition. The emphasis has been, from the beginning, on problem solving by designing appropriate algorithms that solve the proposed tasks. Typically, these tasks accept their data from the console and display the result of the computation also at the console, using numbers and strings. This is understandable, since we really want to focus on algorithm development, and input and output are there merely for being able to validate the submissions. Besides, when IOI started, almost 20 years ago, the available programming environments were very crude, compared to what is common nowadays. Therefore, other possibilities were not even contemplated.

At present, when we review the problem sets of past editions of IOI, we cannot help noticing that some have a strong graphical inclination, but it remains implicit and solutions are not meant to exploit it. This is a pity, since proper use of graphics in programming competitions might make them more attractive and more fun for the contestants, and also more understandable to the general public. This second aspect is very important, since one of the goals of programming competitions is to "shine the spotlight" on the students and on the great programming feats they are able to accomplish.

Graphics can be brought into the competition in two ways, at least. In some tasks the problem may be specified in term of controlling an agent whose behavior is represented graphically; we offered examples of this case in chapter 2. Other tasks may actually have graphical output, instead of plain text output; we explained how this can be approached in chapter 3. In either case, the graphics need not superfluously burden the task. On the contrary, they may actually help the contestant devise the correct strategy for solving the problem.

Complementarily, graphics open the way to a new kind of task, the "tournament" task, as illustrated by the Ataxx example in sub-chapter 2.3. Indeed the possibility of a tournament in IOI was already mentioned in (Opmanis, 2006), as way of improving the contest. In tournament tasks, points would be awarded as a result of a series of matches to be held live, before an audience. The players are the programs for that task, written by the contestants during the competition. A preliminary validation may carried out, in the conventional way, which could also be used to form "leagues" grouping programs whose performance falls within given ranges. This kind of tasks and the associated staging would bring an extra level of excitement to the competition, and could easily be followed by the public in general, thus boosting the IOI spirit.

References

Beyrand, A. (2007). *Ataxx !!*. `http://www.pressibus.org/ataxx/` (accessed May 2007).

Bloch, J. (2001). *Effective Java*. Addison Wesley.

Dagiene, V. (2006). Information technology contests – introduction to computer science in an attractive way. *Informatics in Education*, **5**(1), 37-46.

Gamma, E., R. Helm, R. Johnson and J. Vlissides (1994). *Design Patterns*. Addison-Wesley, Reading, Mass.

Gettys, J., and R.W. Scheifler (2002). Xlib – C Language X Interface. X Consortium Standard. X Version 11, Release 6.7 Draft.

ICPC site, The ACM-ICPC International Collegiate Programming Contest. `http://icpc.baylor.edu/icpc/` (accessed May 2007).

IOI, International Olympiads in Informatics. `http://www.ioinformatics.org/` (accessed May 2007).

IOI Secretariat. `http://olympiads.win.tue.nl/ioi/` (accessed May 2007).

Leal, J.P., and F. Silva (2003). Mooshak: a Web-based multi-site programming contest system. *Software Practice & Experience*, **33**(6), 567–581.

Meyer, B. (1997). *Object-Oriented Software Construction*, 2nd Ed. Prentice Hall.

Opmanis, M. (2006). Some ways to improve olympiads in informatics. *Informatics in Education*, **5**(1), 133–124.

Project Swing. `http://java.sun.com/j2se/1.5.0/docs/guide/swing/` (accessed May 2007).

Ribeiro, P. (2007). *Ataxx Server and GUI.* `http://www.dcc.fc.up.pt/~pribeiro/ataxx/` (accessed May 2007).

P. Ribeiro is currently a PhD student at Universidade do Porto, where he completed his Computer Science degree with top marks. From 1995 to 1998 he represented Portugal at IOI-level and from 1999 to 2003 he represented his university at ACM-IPC national and international contests. During those years he also helped to create new programming contests in Portugal. He now belongs to the Scientific Committee of several contests, actively contributing new problems. He is also co-responsible for the training campus of the Portuguese IOI contestants and since 2005 he has been deputy leader for the Portuguese team. His research interests, besides contests, are data structures and algorithms, artificial intelligence and distributed computing.

P. Guerreiro is an associate professor of Informatics at Universidade Nova de Lisboa. He has been teaching programming to successive generations of students, using various languages and paradigms for the over 30 years. He has been involved with IOI since 1993. He is also the current director of the South-Western Europe Regional Contest, within ACM-ICPC, International Collegiate Programming Contest. He is the author of three popular books on programming, in Portuguese. His research interests are programming, programming languages, software engineering and e-learning.