VoroBreeder: Semi-involuntary interactive evolution of tiny populations

Pietro Galliani

September 18, 2016

1 Introduction and background

It is occasionally said that while some people enjoy attention and company and find it difficult to cope with isolation, others cherish their time alone and find sustained interaction with others positively tiresome; and, as it is often the case for pronouncements about human nature, this is at best a gross overgeneralization and at worst complete nonsense. It would perhaps strike closer to home to say that many of us are not particularly clear or consistent about the degree up to which we desire other people's attention, but that we not rarely find ourselves wishing for more — or less – of it than we have.

This work could be thought of as an exploration of this concept, in terms of coevolution of two tiny (ten individuals each) populations of individuals which find themselves competing respectively for user *interaction* and for user *neglect*. After each brief (twenty-two seconds) run, the user is offered the possibility to discover which individuals were longing for the tender click of the cursor and which ones instead hoped against hope that the user would pass them by; then a new generation of each population is created, by means of a fairly standard genetic algorithm, and the game begins anew.

Another possible key of interpretation of this work, one which is also not without its own merits, is that this is the first part of what was supposed to be a somewhat more ambitious project. My initial aim was to investigate which kinds of visual patterns tend to capture or not capture someone's attention; and I planned to do so by having two families of such patterns coevolve while the user is engaged in some simple point-and-click computer game. The lineages of those individuals which are rewarded by being "captured" would then naturally evolve towards more and more garish and eye-catching patterns; and on the other hand, those of the individuals which are rewarded for being "missed"' would evolve towards more unobtrusive, easy-to-miss patterns.

As it is, this work is not successful in this respect: the tiny dimensions of the populations, the relative simplicity of the patterns, and the short number of iterations make it unlikely that its results might have some significance for the study of the psychology of visual perception. But, at least to some degree, this is also the result of a conscious design choice. It would certainly have been possible to have (or presume to have) the user (or, why not, different users, in order to better account for individual variability) slog their way through hundreds of different, rapidly-succeeding visual patterns and pick those that they can notice in time; but while this could constitute a passable preliminary outline for an experiment about visual perception, it also looks like something that not many would gladly subject themselves to.

This touches upon one of the principal problem in the study of interactive evolution of computer art: users *fatigue*, and fatigued individuals behave differently from attentive ones — and, eventually, they may stop paying any attention whatsoever to their specimen and choose semi-randomly. Of course, for my original intentions a certain lack of conscious deliberation is useful; but there exists a threshold of interest beneath which no useful data may be collected (especially if one is not *paying* people to undergo this sort of thing).

The idea of studying the effects of gamification on visual perception is clearly not without interest, nor is it of course in any sense novel — for instance, (Ong, 2013) examines the consequences of adding game points on the performance of forty individuals attempting a twenty minutes long perceptual search task (i.e., finding a faint human silhouette in a snowfield image). Furthermore, a main inspiration for this work was (Hastings et al., 2009), in which particle systems (representing the effects of futuristic weapons) are made to evolve on the basis of players' predilection or lack thereof for them, as inferred automatically by their usage of them. But for the purposes of this work, I eventually settled for the objective of creating something

- That illustrates a valid biological scenario (namely, the coevolution of non-competing small populations over short timescales);
- That, whilst making use of techniques and ideas from generative creativity and evolutionary computing, is not just a showcase for them and can also be appreciated (within its, admittedly very narrow, limits) on its own independent merits;
- That a user — at least, one in the right mood — might possibly find mildly engaging for perhaps a couple of minutes.

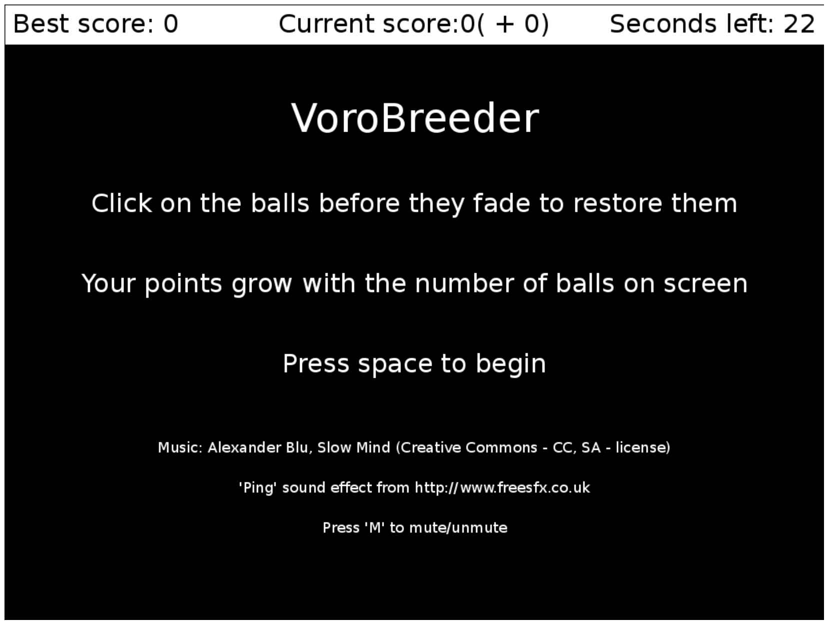
Some words should be spent here discussing some of the ways in which this third point affected my design choices. First of all, the short duration of each playthrough has the express purpose of preventing them from being too boring. Each "level" was initially supposed to last one minute; but I found that halving the duration of the game, and thus giving the user more opportunities to compete against their own previous scores, made for a far less tiresome experience. Trimming it down a bit more, to twenty-two seconds, improved matters even further, at least in my opinion.

Similarly, the relatively small number of specimen encountered through each play, and the somewhat sedate (one every second) rate of their appearance is also meant to stave off fatigue: at least in my personal experience, my initial idea of having several patterns zip around at every moment (thus forcing the user to make near-continuous split-second choices between them) tended to elicit rather quickly a sentiment of confused disinterest for the whole silly thing.

Finally, I found that some background music and a simple sound effect — a 'ping' every time a target is hit — greatly increased user immersion. Neither the sound effect nor the music are of my own making: as declared in the initial screen of my program, I downloaded the sound effect from an online repository, while the music was created by Alexander Blu and published on `https://www.jamendo.com` with a Creative Commons (Non-Commercial, Share-Alike) license.

2 Illustration

The user is shown a brief introduction screen and is told to press space to start, or 'm' to start/stop the sound effects.



Best score: 0 Current score:0(+ 0) Seconds left: 22

VoroBreeder

Click on the balls before they fade to restore them

Your points grow with the number of balls on screen

Press space to begin

Music: Alexander Blu, Slow Mind (Creative Commons - CC, SA - license)

'Ping' sound effect from http://www.freesfx.co.uk

Press 'M' to mute/unmute

Figure 1: The start screen.

When the user presses space, the program loads the two populations from two text files (seekers.JSON and avoiders.JSON), stored together with the applet in my university filesystem; then it creates two new generations, by keeping the three "best" seekers and the three "worst" avoiders and generating the rest through crossover and mutations, and starts the game.

Each specimen is visualized as a colored ball: as I will discuss more in detail in the **System Overview** section, its pattern is the Voronoi diagram generated by a set of points (which are specified in the specimen's genome, together with

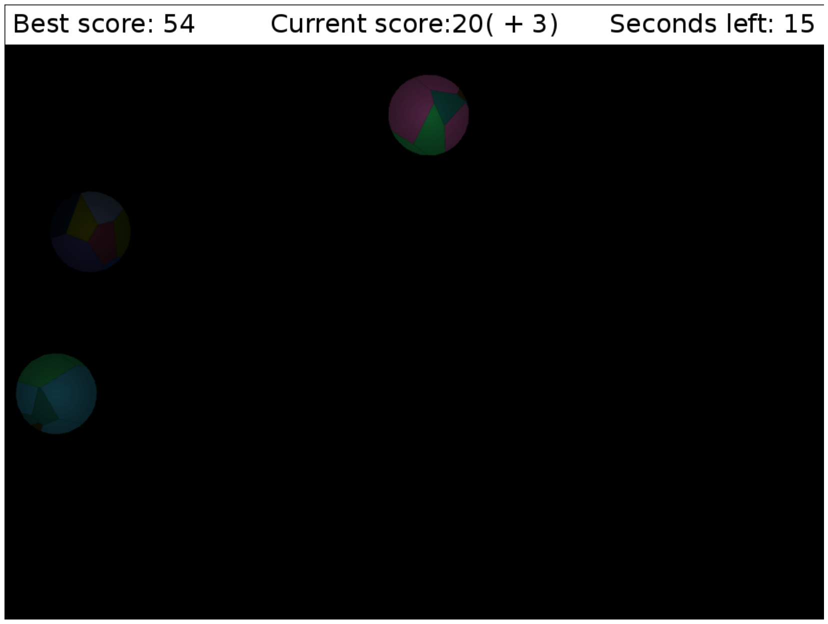
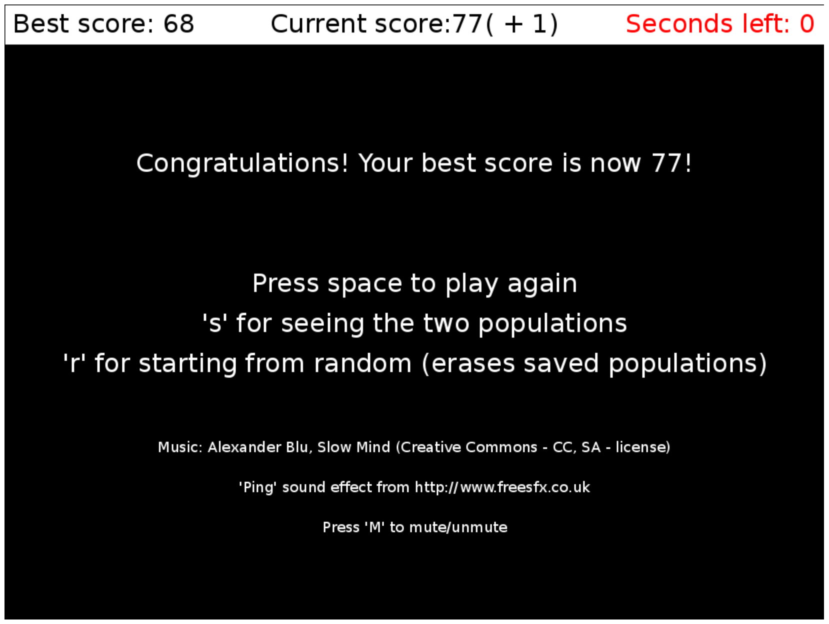


Figure 2: A scene from the game. Three balls (one almost entirely faded), so +3 points for this second.

the colors of the regions of the diagram). These balls appear at a rate of one per second and bounce around in the game area, rotating (at a rate of half a turn per second) and fading over time. Their initial lifespan is of about one second before they fade away entirely and "die"; but whenever the user clicks on them, they are regenerated to full health once more. Every second, the user gains as many points as the number of balls currently on the screen; thus, they are encouraged to juggle their attention between balls, and keep as many of them "alive" as they can.

After the game is ended, the user is told whether their score has improved, and they are given the opportunity to play again, to erase the saved data from the server and start from new random populations, or to see the two populations separately.

As we can see in Figure 4, the "attention-seekers" look overall considerably less bright and contrasting than the "attention-avoiders". I was very surprised by this, as it runs exactly contrary to what I was expecting, and I suspected that a programming error had me swapping the two populations somehow; but after comparing the average "lifespans" of the attention-seekers and the attention-avoiders and controlling the code, I have come to the conclusion that the program is working as intended. My best guess is that, as a user trying to keep several balls "alive" at the same time, I am more likely to click on a specimen if I think that it is about to fade away; and thus, the pre-faded looks of


Best score: 68 Current score:77(+ 1) Seconds left: 0

Congratulations! Your best score is now 77!

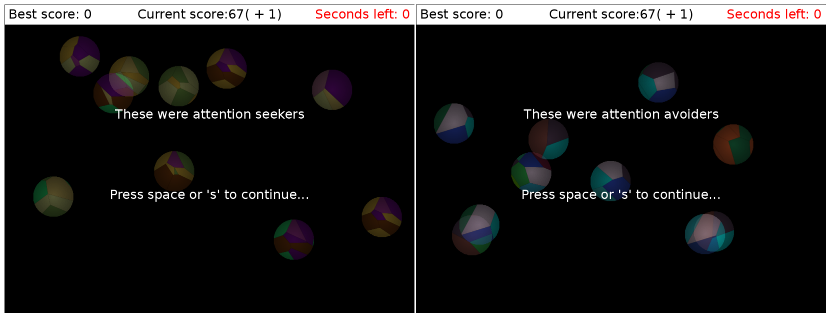
Press space to play again
's' for seeing the two populations
'r' for starting from random (erases saved populations)

Music: Alexander Blu, Slow Mind (Creative Commons - CC, SA - license)

'Ping' sound effect from http://www.freesfx.co.uk

Press 'M' to mute/unmute

Figure 3: After the end of the play.


Best score: 0 Current score:67(+ 1) Seconds left: 0

These were attention seekers

Press space or 's' to continue...

Best score: 0 Current score:67(+ 1) Seconds left: 0

These were attention avoiders

Press space or 's' to continue...

Figure 4: Attention seekers and attention avoiders

the attention-seekers and the brighter looks of the attention-avoiders constitute genuine (and frankly unexpected) evolutionary adaptations to this scenario.

To return to the social metaphor of the introduction, it is as if some "introverts" found it useful to present themselves as more involved already than they actually are (thus avoiding further interaction with well-meaning people on top of that) and some "extroverts" found instead useful to present themselves as less busy (and, thus, more needful of interaction) than they are.

3 System Overview

This work has a both a server and a client side. The server side consists of the two files `seekers.JSON` and `avoiders.JSON`, storing the two populations as JSON strings, and of the PHP program `updateSpecimen.php`, which merely receives two strings in input and overwrites the two files with them (unless a different process is already writing on them, in which case it just fails).

The content of the strings is not checked in any way, which poses an obvious vulnerability; but on the upside, only the two files can be written, and since the string is passed as a POST parameter there is a strict limit (usually 20MB, I think, although it depends on the server configuration) on the possible amount of information that may be sent to the files.

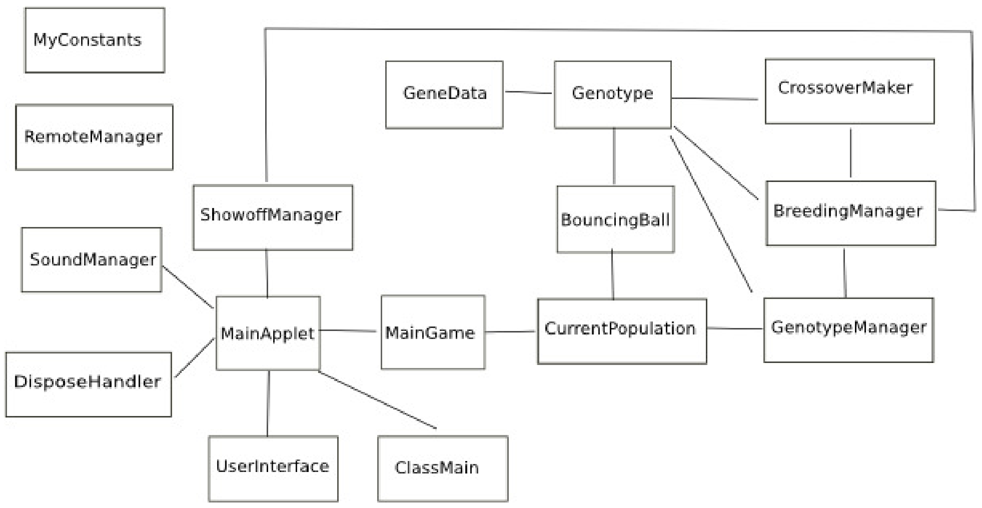


Figure 5: VoroBreeder client: classes and relationships

The client side is considerably more complex. Figure 5 contains a graphical representation of the classes which are part of it and their relationships; and in the rest of this section I will go through each one of them, in alphabetic order, and say a few words about their roles and my design decisions (if nontrivial).

BouncingBall:

Instances of this class correspond to specific individuals in our population. This class keeps track of their lifespan, checks if they have been clicked (and, if so, restores their original lifespan), bounces them around the screen (without worrying about collisions with other balls — I considered adding that functionality, but I feel that it would be an unnecessary complication that would add little to this work), and displays them.

As already mentioned, the appearance of our individuals is based on the notion of Voronoi Diagrams (Voronoï, 1908). In brief, the **Genotype** associated to a **BouncingBall** (or better, the corresponding **GeneData**) consists of a sequence of points, described in polar coordinates, and a sequence of colors (not necessarily of the same length). After converting these points to Cartesian coordinates, we identify — through Fortune's $O(n \log n)$ time sweepline algorithm (Fortune, 1987), as implemented by Lee Byron's **Mesh** library (`http://leebyron.com/mesh/` — the regions of the sphere which are closer to each one of them than to the others, and we paint them of their given color. If we have finished the list of all colors and we still have regions, we simply loop back to the first color: this allows for the possibility that a single point mutation might affect the look of multiple regions of the individual, a phenomenon not without biological parallels, and it also permits for instance that a genome might code for "substantial color uniformity" independently on the exact color (and preserve this uniformity through multiple mutations).

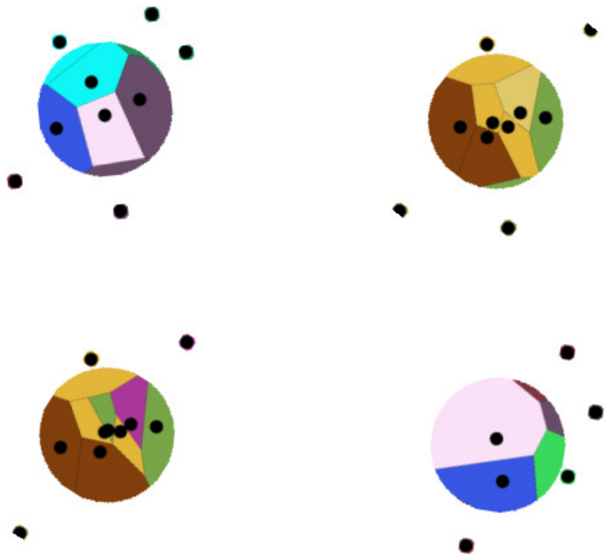


Figure 6: Four of our individuals and the points for the corresponding Voronoi diagrams. Two of these like to be clicked and two dislike it. Which are which? The answer could be unexpected...

BreedingManager:

This class keeps track of the lists of **Genotypes** corresponding to the seekers and the avoiders, loading them from the server (or generating them *ex novo*, if necessary), using **CrossoverMaker** to build the new generations, and returning the list of genotypes (seekers and avoiders together, in random order) to

GenotypeManager.

ClassMain:

This class exists for the sole purpose of being able to run the program as a stand-alone application rather than an applet.[1] Beyond this technical issue, it can be safely ignored.

CrossoverMaker:

This class contains the bulk of the genetic algorithm. Given the current generations of genotypes for attention seekers and attention avoiders, we keep the three "best" ones of each generation (that is, the seekers which have lasted the most on the screen in the last play and the avoiders which have lasted the least).

For both seekers and avoiders, we fill each of the other seven slots as follows: we extract two parents (using a roulette wheel function, using weights which are directly proportional to past lifespans in the case of seekers and inversely proportional to them in the case of avoiders). Then we create a new genotype through crossover, we add random mutations, and we store it in our list.

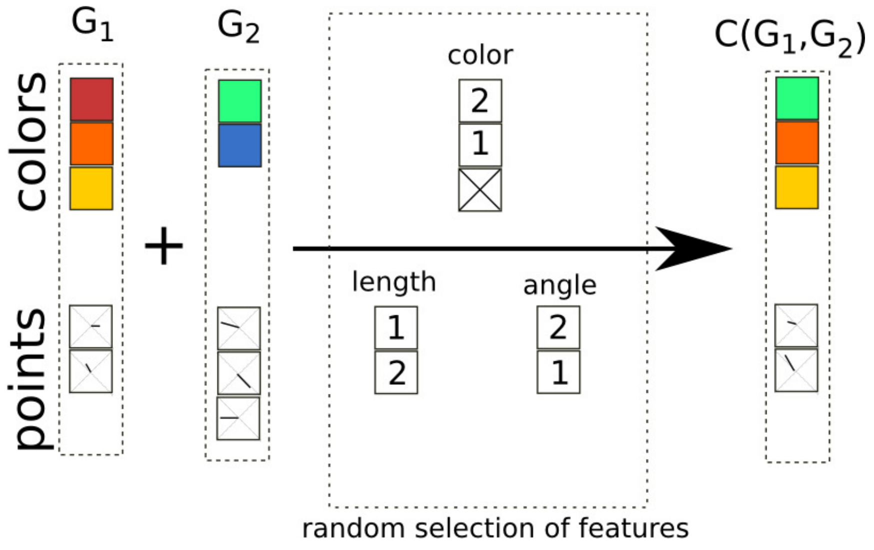


Figure 7: Graphical representation of the crossover procedure.

The crossover procedure operates as depicted in Figure 7: first we choose the number of colors and of points of the child, picking a random number between the values of the father and those of the mother.[2] Then we pick each color

[1]More precisely, it provides a `static void main()` function which, when executed, tells **MainApplet** to start.

[2]Not that gender is modeled in any way in this work, of course.

randomly from either the father or the mother (if one has less colors than the other, we chose the rest from the remaining one), and we do likewise for the radii and the angles of the points. It is worth pointing out here that radii and angles are extracted *independently*, so that the child genotype might have a point whose radius corresponds to that of the father's point and whose angle corresponds to that of the mother's. Individual colors, however, are not mixed in any way, and come always entirely from the father or entirely from the mother — I considered doing otherwise, but I wanted to avoid the risk that all colors converged spontaneously towards the same uniform tone as evolution goes on.

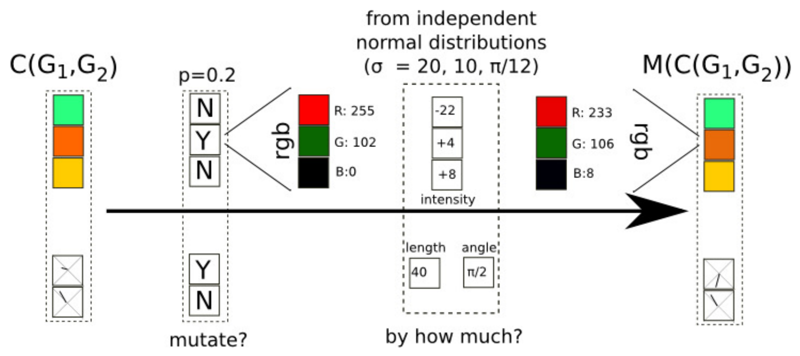


Figure 8: Graphical representation of the mutation procedure. For illustration purposes, the magnitudes of the length and angle mutations are unusually large.

As for the mutation procedure, illustrated in Figure 8, each color and each point of the child genotype has a fixed 2% probability of being mutated. If a color mutates then its red, green and blue parts are all modified by an amount extracted from a normal distribution with mean 0 and standard deviation 20; and if a point mutates then, similarly, the values of its angle and its radius are modified by amounts extracted from normal distributions (with standard deviations $\pi/12$ and 10 respectively).

CurrentPopulation:

This class keeps track of the current population of individuals, updating their positions, displaying them, removing the dead ones, and asking **GenotypeManager** for new ones when necessary.

DisposeHandler:

This is a small utility class used for closing automatically the sound files when the application exits.

GeneData:

This simple class is just a container for the information stored in a **Genotype** — that is, the number of colors and points, the polar coordinates of the points, and the color values.

Genotype:

Instance of this class represent genotypes, consisting of a **GeneData** instance and a counter of the number of steps through which the genotype has survived. This class also contains functions from loading a genotype from a JSON object (such as the client might receive from the server), or storing it in one.

GenotypeManager:

This class keeps track of the genotypes in the corresponding population. It does *not* keep track of which is a seeker and which is an avoider — this is a job for **BreedingManager**, from which **GenotypeManager** receives the list of genotypes — but it creates new individuals from them when requested from **CurrentPopulation**. Furthermore, it passes commands about generating genotypes, loading them, and storing them between **CurrentPopulation** and **BreedingManager**.

MainApplet:

This class is the central hub of the application. Mostly it keeps track of user inputs — keypresses and mouse clicks — and tells the game (**MainGame**), the graphical interface (**UserInterface**) and the "show all seekers/all avoiders" utility (**ShowoffManager**) what to do.

MainGame:

This class does nothing but pass messages from **MainApplet** towards **CurrentPopulation** about creating/loading and reproducing a new population, drawing it, checking if some ball has been clicked, and sending the population back to the server. Not an overly necessary class, but it provides some encapsulation.

MyConstants:

This file merely contains a list of numerical constants used by the other classes, for ease of reference and modification.

RemoteManager:

This class contains the functions for low-level communication between the client and the server (and also for reading and writing to files, when in local mode).

ShowoffManager:

Instances of this class can be in three states (or "phases"), transitioning from each to the other when the function `next()` is invoked by **MainApplet**:

- **Phase 0**: Do nothing.
- **Phase 1**: Load the genotypes of the current seekers and avoiders (through **BreedingManager**), and display all the seekers on the screen.
- **Phase 2**: Display all the avoiders on the screen.

SoundManager:

This class takes care of playing the background music and the sound effect when a ball is clicked, by means of Damien di Fede's `Minim` library.

I had a surprisingly difficult time getting sound to work and the applet to load the sound files without getting permission errors: now sounds seems to work fine in the applet, but not in the stand-alone application (although I suspect this might be due to my computer's configuration, as using different libraries did not seem to improve matters).

As said in the introduction, I find that music and sound effects add much to the quality of the experience; so even despite these difficulties, I find myself glad that this apparently works now.

UserInterface:

The name of this class is a bit of a misnomer. All it does is display the header with the information about the best score, the current score, the number of variables, and the time left, and display messages on the screen when the applet first starts and between plays.

4 Limitations and Possible Improvements

The greatest limitation of this work, as it is now, lies in the tiny dimensions of the populations of the seekers and the receivers and in the fairly small number of trials used to evolve them. These factors, by the way, are also one of the reasons why I chose to focus on simple patterns generated by Voronoi diagrams rather than on more complex forms of representations such that Compositional Pattern-Producing Networks (Stanley, 2007). However, I find myself quite satisfied with the overall visual effect — the individuals look like mutant beach balls, and their aspect is similar enough to grant to the work as a whole a consistent visual style — and I now think that going for the generality and expressivity of CPPNs for this kind of project would have been a mistake anyway. The same can be said about approaches such as the *Fractal Flame* algorithm (Draves and Reckase, 2003) employed by Draves' Electric Sheep project (Draves, 2005) or Lindenmayer Systems (Lindenmayer, 1968) — if the game is to present an overall coherent, unified aesthetics, excessive expressivity in the visual language describing its entities is a drawback and not an improvement.

Another obvious problem is that, while different players should be able to play the applet at the same time, there is no mixing of the resulting populations: the populations of the one that finishes last simply overwrite those of the former, nothing more.

Both these problems could be dealt with together by associating each player to *distinct* populations of seekers and receivers and having the server implement a simple form of communication between populations, periodically moving random individuals between the populations corresponding to different individuals.

An interesting extension along these lines could be to have the rate of communication between the populations of seekers and of avoiders depend on (rough estimates of) the *physical* distances between the corresponding users. It might then be possible (at least, for a sufficiently big number of users and plays) to see the development of different "species" of seekers and avoiders, on the basis of the individual preferences of the users (for instance, a player might happen to be more likely to click on red balls than on blue ones, while the other might do the opposite) and of their spatial relationships; and the ebb and the flow of these species could then be given a suitable graphical representation and published online.

On a simpler note, it might be worthwhile to make the effect of clicking on an individual *decrease* gradually with the number of clicks. As of now, one individual could be made to survive indefinitely by clicking repeatedly on it, at the expense of paying attention to others; and it comes to my mind that strategies based on this fact — ones which, for instance, focus mainly on the first seen individual, ignoring the others unless they are easy to reach from the current mouse position — could make the fitnesses of the genomes after each play relatively independent from the colors of the corresponding individuals.

Finally, an easy modification that could improve the flexibility of the genetic algorithm consists in adding a new mutation operation, of comparatively high probability, which simply swaps around the orders of colors and points in the genome: such a mutation would not have any *immediate* effect, of course, but in the reproduction stages of future generations it would allow it to interact with genes which it could never have interacted with otherwise. I am halfway tempted to add this as a last-minute change, but it's getting more and more absurdly late and if I keep making changes I'll end up missing my plane tomorrow.

5 Conclusions

From a practical perspective, this project has been, for me, an object lesson in feature creep. The main elements of the work — a genetic algorithm, a visual representation of the genotypes in terms of Voronoi diagrams, and a simple point-and-click game — were pretty easy and quick to implement; and on the other hand, secondary features such as background music or the ability to run on a browser as an applet proved themselves surprisingly time-consuming to set up, and fine-tuning parameters such as play duration or rate of appearance of individuals in order to make the overall experience not overly unenjoyable also required an unexpected amount of effort.

Still, I find myself quite happy with the final result, not only as a proof of concept in the "artistic" use of evolutionary programming but as a coherent and self-contained piece of work – one that, within its own limits, can also be appreciated for its own sake. And, as a welcome side bonus, I learned a few technical tricks I was not aware of during the course of the implementation.

The most surprising result of this work lies without doubt in the fact that apparently seekers can evolve drab, "faded" appearances in order to get the user to click on them more often. I was expecting precisely the opposite, and I found myself digging deep in the code for bugs before even entertaining the idea that the algorithm was working as intended — it was *I* who was not, in the sense that, against my own expectations, I was more likely to click on faded-looking balls than on bright-looking ones!

I am honestly not sure how replicable this result is, or how much it depends on the idiosyncrasies of the strategies which I tended to personally adopt when playing this game; but it comes as a not unwelcome reminder that with the adaptability of evolutionary computing comes along the "danger" of one's algorithm developing solutions entirely different from the kind that one was expecting; and that when the function that the evolutionary algorithm is optimizing against consists in one's own activity, the result may well reveal unforeseen aspects of oneself. This, after a fashion and on a much smaller scale, may be thought of as an exemplification of Annunziato and Pierucci's programme of

> *...exploring artificial beings and artificial societies as a digital mirror in which human culture is reflected.*
> (Annunziato and Pierucci, 2002).

References

Mauro Annunziato and Piero Pierucci. Relazioni emergenti: experiments with the art of emergence. *Leonardo*, 35(2):147–152, 2002.

Scott Draves. The electric sheep screen-saver: A case study in aesthetic evolution. In *Applications of Evolutionary Computing*, pages 458–467. Springer, 2005.

Scott Draves and Erik Reckase. The fractal flame algorithm, 2003. Forthcoming.

Steven Fortune. A sweepline algorithm for voronoi diagrams. *Algorithmica*, 2 (1-4):153–174, 1987.

Erin J Hastings, Ratan K Guha, and Kenneth O Stanley. Automatic content generation in the galactic arms race video game. *Computational Intelligence and AI in Games, IEEE Transactions on*, 1(4):245–263, 2009.

Aristid Lindenmayer. Mathematical models for cellular interactions in development i. filaments with one-sided inputs. *Journal of theoretical biology*, 18(3): 280–299, 1968.

Michael Ong. Gamification and its effect on employee engagement and performance in a perceptual diagnosis task. 2013.

Kenneth O Stanley. Compositional pattern producing networks: A novel abstraction of development. *Genetic programming and evolvable machines*, 8 (2):131–162, 2007.

Georges Voronoï. Nouvelles applications des paramètres continus à la théorie des formes quadratiques. deuxième mémoire. recherches sur les parallélloèdres primitifs. 1908.