An Experiment in Automatic Game Design

Julian Togelius and Jürgen Schmidhuber

***Abstract*—This paper presents a first attempt at evolving the rules for a game. In contrast to almost every other paper that applies computational intelligence techniques to games, we are not generating behaviours, strategies or environments for any particular game; we are starting without a game and generating the game itself. We explain the rationale for doing this and survey the theories of entertainment and curiosity that underly our fitness function, and present the details of a simple proof-of-concept experiment.**

I. Introduction

Can computational intelligence (CI) help designing games? One is tempted to answer "Yes, obviously, the whole field of Computational Intelligence in Games (CIG) is devoted to this, isn't it?"

However, the majority of CIG research is concerned with learning to play particular games as well as possible. There is nothing wrong with this type of research; indeed, it is very valuable for the science of artificial intelligence. Games provide the type of deep but accessible reinforcement learning problems that we badly need in order to develop better CI algorithms and controller representations. But these types of studies are less interesting to game development in general, and to commercial game developers in particular.

This is because commercial game developers are, in general, not interested in ways of making NPCs (non-playing characters, e.g. opponents) play better. This is because it is usually possible to make them play well enough using just a tiny bit of (virtually undetectable) cheating, such as giving the NPC more information than the human player, or slightly superior abilities. Further, having NPCs that play as well as possible is no end in itself; if the opponents are too hard to beat, the game quickly becomes uninteresting for human players.

Of course, there are exceptions. In particular, many strategy games (such as *Civilization* or *Starcraft*) are so complicated that it's hard to come up with NPC AI that plays competitively against good human players, without resorting to most blatant cheating. In such cases, CI techniques for beating the game better might contribute to the quality of the game. And let us not forget that many games are representations of real-life problems (e.g. battle tactics or vehicle control), and advances in learning how to play games might be transferable to related real-life problems. But for most game genres, such as racing games, platformers, puzzle games or first person shooters, there does not seem to be any interest from game developers in learning to play the game better *per se*.

Both authors are with IDSIA, Galleria 2, 6928 Manno-Lugano, Switzerland. Jürgen Schmidhuber is also with TU Munich, Boltzmannstr. 3, 85748 Garching, München, Germany. emails: {julian, juergen}@idsia.ch

Now, there is certainly other research being carried out in the CIG field that is more directly relevant to real game development (and often dependent on research done in learning to play games, which thus becomes indirectly relevant to game development). For example, we have CI techniques proposed to generate NPC controllers that play interestingly as opposed to just well [1], [2]; CI techniques for automatically finding exploits/bugs in games [3]; CI techniques for modelling the behaviour of human players[4], [5]; CI techniques for making NPCs trainable by human players [6]; and techniques for generating the content of a game, such as tracks, levels or mazes [4], [7].

While the above techniques all represent relevant research directions for game design, they all assume that there is a game there to begin with. Before we let CI loose on generating behaviours, strategies or environments, or testing the game, it needs to have something to play with. However, game design is concerned with all aspects of the game. In particular, it is designed with the very heart of every game, that which defines the game: its rules [8].

Game rules determine when the game begins and when it ends. They determine what the player can and can't do, and (together with the actions of other players) what happens as a result of the player's actions. The complexity of rules vary from something as simple as checkers, which can be expressed in a paragraph of text, to something as complex *Starcraft*, *Counter-strike* or other modern video games, where the rules cannot be expressed in its entirety without describing the whole game engine.

Of course, there are borderline cases regarding what is and what isn't a rule. While the set of legal capture moves in Checkers are clearly rules, would the throwing speed and blast radius of a grenade on *Counter-strike* count as rules, aspects of the environment or something else? In this paper, we will take a fairly liberal and inclusive view of what constitutes rules for a game in our argument, whereas in the experiment we will err on the side of exclusiveness and adopt a stricter view of what constitutes a rule.

In this paper, we describe an initial proof-of-concept experiment where we evolve the rules for a game. To our best knowledge, this represents the first time rules have been evoled for a single-player game; the first time rules have been evolved for a non-board game; and the first time a fitness function based on learning progress has been used for evolving game rules. An interesting recent study describes the evolution of board game rules using a fitness function based on measurements of historical play patterns [9]. It is worth noting that both the type of game and the fitness function differ drastically between these two independently conceived and executed studies.

978-1-4244-2974-5/08/$25.00 ©2008 IEEE

In the following, we first survey some theories of entertainment in games, and describe how, in general terms, they could be converted into fitness functions. We then describe the assumptions we make regarding the games we will create, and how these constitute a searchable rule space. As our ruleset fitness function is based on the learning of an agent, the neural network-based agent architecture and agent fitness function will be exemplified in the context of a sample game. Some examples of evolved games are then followed by a discussion about the many challenges remaining before this technique becomes practical and the vast impact it could have on game design once it does.

A. What is fun?

So we want to evolve rule sets that create entertaining games. But how do we do this? Can we measure entertainment in a quantitative way that does not involve using human subjects, so that it can be incorporated into a fitness function? To start with, we need some sort of theory about what makes a game entertaining.

Fortunately, there are a number of theories from psychology and game studies that might be of help here, many of them focusing on the challenge a human player experiences when playng the game. One of the oldest and best known of these is Csikszentmihalyi's concept of *flow*, the "optimal experience" [10]. When in the state of flow, a human is performing a task (not necessarily a game; the theory was developed with e.g. the creative process of artists in mind) such that he is fully concentrated on the task and loses his sense of self but has a sense of full control. One of the key prerequisites for reaching flow is that the amount of challenge of the task is just about right: not too hard, and not too easy.

Malone [11] distinguishes between three factors that make games engaging: challenge, fantasy and curiosity. While the right level of challenge is a key component of the flow concept as well, the other two factors are more game-specific; fantasy refers to how well the game evokes a sense of being somewhere else or doing something exotic, whereas curiosity means that environments should have an "optimal level of informational complexity": novel but not incomprehensible.

Sweetser and Wyeth proposed *GameFlow* as a an adaptation of the flow concept specifically to games [12]. GameFlow offers a way of evaluating how entertaining a game is based on a number of criteria: concentration, challenge, player skills, control, clear goals, feedback, immersion and social interaction.

Koster's informal "theory of fun for game design" also focuses on the challenge of the game, but in a different way. Koster sees the learning of how to play a game as its main source of entertainment; playing is learning, and learning is fun [13]. For Koster, well-designed games are games that are easy to start playing, but where the difficulty curve is such that the player continuously learns something new while playing. I.e., games that take a minute to learn, but a lifetime to master.

1) The theory of artificial curiosity: A theory which has not so far been explored in the context of game design is Schmidhuber's theory of artificial curiosity [14]. Unlike the theories above, it does not focus on the challenge of the task/game, but instead focuses on the predictability of environments. Agents prefer environments with some sort of algorithmic regularity that is not yet known but expected to be quickly learnable. A curious agent is rewarded whenever a separate adaptive predictor learns to predict the incoming data stream better than before. Thus the agent is motivated to explore those parts of the world that yield temporarily "interesting" input sequences, maximizing the derivative of the subjective predictability.

The theory of artificial curiosity has the advantage that it is quantitative, and a number of variants of it have already been successfully used in reinforcement learning. Early work (e.g. [15]) described a predictor based on a recurrent neural network, predicting inputs and reward signals from the entire history of previous inputs and actions. The curiosity rewards were proportional to the predictor errors, that is, it was implicitly and optimistically assumed that the predictor will indeed improve whenever its error is high.

Follow-up work [16] pointed out that this approach may be inappropriate, especially in probabilistic environments: one should not focus on the errors of the predictor, but on its improvements. Otherwise the system will concentrate its search on those parts of the environment where it can always get high prediction errors due to noise or randomness, or due to computational limitations of the predictor, which will prevent improvements of the subjective compressibility of the data.

Variants of this idea have subsequently found their way into the mainstream of both reinforcement learning and developmental robotics. A previous game-related application of the idea was presented in [17]. There, two agents design games for each others, realized as algorithms expressed in a universal programming language. One of the agents proposes an algorithm to execute, the other chooses whether to accept it, and modifications are proposed until both agents accept. As above, the goal is to maximize learning progress: find a set of rules such that the opponent accepts them because it thinks it will prevail, as he already knows the aspects of the world exploited by the current game rules, then surprise it by playing the game (running the algorithmic experiment) and winning. The other agent will adapt and next time prefer a different game because it has learnt something.

The experiment in this paper differs from the above experiment in generating single-player games rather than two-player games, in the learning algorithms used, and above all in the generated games being recognisably "gamey" rather than algorithms expressed in a universal programming language.

B. Measuring fun in practice

The psychologically oriented theories discussed above only talk about fun or interestingness in general, and not how to measure it for a particular game. In particular, they don't tell us how to measure it in the absence of a human player and a human observer. (Remember that we are talking

about *automatic* game design here.) The theory of artificial curiosity has so far been applied mostly to the behaviour of agents. The following is an example of how fun can be estimated and optimized for aspects of a game other than NPC behaviour:

In [4] we presented an approach to evolving interesting tracks for a driving game, for which we had previously evolved well-performing neural network-based players [18]. We let a human player drive a test track that contained different sorts of driving challenges, and measured aspects of the driving. We then further evolved a previously evolved controller to conform with these measures, creating an artificial player that drove similarly to the modelled human. The fitness function for the track was derived from Malone's factors. The fitness of a track was calculated in a rather indirect way: it depended on how the "human-like" controller performed on the track. The objectives were for the car to have the right (not too high, not too low) average speed, high maximum speed, and high variability in performance between trials, indicating driving "on the edge". Tracks were represented as b-splines. Evolution found tracks whose level of challenge matched the proficiency of the modelled players, and possibly more fun for the modelled players in other respects as well.

The key idea here, which can be transferred to evaluating the fitness of rulesets as well as other types of environments, is that fitness is measured *indirectly*. The genotype to be evaluated parameterizes some aspects of the game (the environment, the opponent(s), or the rules of the game itself), and its fitness is then determined by how the parameterized game makes an agent behave. The agent is controlled by a controller, which might or might not be made to imitate a human player, and might or might not be learning while playing the game.

For the purposes of constructing fitness functions out of theories of entertainment, we can divide the theories discussed above into *static* and *dynamic* theories of fun. Static theories are those that do not require the agent/player to learn while playing, and can be used to judge the entertainment of a game versus the capabilites and personality of a player at a given moment. Of the theories discussed above, Malone's theory is clearly a static theory, and for the most part this goes for GameFlow too. In contrast, dynamic theories puts learning in focus, and the entertainment of a game could only be judged by how the agent/player adapts to the game over time.

This distinction becomes relevant when trying to create fitness measures for game rules. When evaluating a new environment or a new opponent, it is perfectly possible to use a fixed controller for these evaluations, and thus base the fitness function on a static theory of entertainment. This is what we did when evolving racing tracks in the example above. However, when evaluating a complete game, it is impossible to provide a fixed controller to test the game with - such a controller does not exist. The controller has to learn to play the game through (possibly intelligently prejudiced) trial and error, like a human would do. Apparently, our only choice is to base the fitness function on a dynamic theory of entertainment, and a controller incorporating some form of reinforcement learning.

For the experiment in this paper, we will use the following implementation of Schmidhuber's theory of curiosity and interestingness, which is also a coarse approximation of Koster's theory of fun: a game is fun if it is learnable but not trivial. A game is no fun if it can be won by doing nothing at all or acting randomly, but it is also no fun if a learning mechanism cannot learn to beat it within a certain time. For learning mechanism, we will use evolutionary computation training a controller based on a neural network. But before we get to the details of that, we need to explain the rule space in which to search for games.

II. Game engine and rule space definition

It would not be possible to evolve a game without any constraints at all. A number of assumptions must be made, or in other words, a number of axioms must be laid down that define the rule space. One can see this as selecting the genre of the game, but not in the space of traditional genres. It would be perfectly possible to specify the axioms that (together with an appropriate simulation API) construct a rule space of games based on physical simulations, perhaps containing *Asteroids* in one end, *Breakout* in another and a car racing game somewhere else along the multidimensional spectrum. A rule space of board games could contain *Othello* and *Go*.

We have settled for a search space that at some points in space contain *Pac-man*-like games. The axioms are as follows:

- The game takes place on a discrete grid with dimensions 15×15.
- Each cell on the grid is either free space or a wall. In order to demonstrate that we are evolving rules and not environments, we have laid out the walls in advance; this layout can be seen in figure 1 and does not change regardless of rules or game state.
- A game will run for a finite number of time steps, starting at $t = 0$ and continuing until either $t = t_{max}$, $score >= score_{max}$ or the flag *agent death* has been set.
- If, at the end of a game, $score >= score_{max}$ the game is *won*; otherwise, the game is *lost*.
- At the beginning of a game, one of the cells (randomly selected among the 4×4 centralmost cells) contains the *agent*. At any time step, the agent can and must move one step either up, down, left or right. Any move that would result in the agent occupying the same cell as a wall is not executed.
- At the beginning of a game, zero or more cells are occupied by *things*. These cells are randomly chosen from the free space on the grid, except that no thing starts closer than two steps from the agent. Every thing can be either red, green or blue. Things can, but must

not, move one step every time step. Like agents, things can not pass through walls.
- Each colour has an associated *movement logic* that determines how things of that colour move; a *collision effects* table determines what happens to things when two things of the same or different colours collide, or when a thing and the agent collide; a *score effects* table determines how the score changes when a collision between two things or between a thing and the agent occurs.

In other words, the rule space consists of (1) eight simple parameters: t_{max}, $score_{max}$, the number of red, blue and green things, the movement logic for red, green and blue things, and (2) two tables: collision effects and score effects. The max and min values of the simple parameters are as follows:

- t_{max}: 0-100
- $score_{max}$: 1-10
- Number of items of each colour: 0-20 (three independent values)
- Movement logic for each colour: 1-5 (three independent values)

The following five movement logics are available:

1) Still: things of this colour do not move.
2) Random short: each thing chooses a new direction each time step, and tries to move in that direction.
3) Random long: each thing chooses a new movement direction and a number of steps n between 1 and 10, and for the next n steps it tries to move in that direction. After n steps, it chooses a new direction and a new n.
4) Clockwise: each thing chooses an initial movement direction. Each time step it tries to move in that direction. If it fails (i.e. bumps into a wall) it chooses a new direction which is to the right of the current (up becomes right, right becomes down, down becomes left, left becomes up).
5) Counterclockwise: like clockwise, except that the things of this colour turn left rather than right when they reach a wall.

The collision effects table has dimensions 4×4 and is used as follows: each time two things (or a thing and the agent) collide, the effects on both things (or the thing and the agent) are looked up separately at index (color of first thing, color of second thing). The axes are number (red, green, blue agent). So if two red things collide, the effects for both things are found at (0, 0), and if a green thing collides with the agent, the effect for the thing is found at (1, 3) and for the agent at (3, 1). Each of the 16 cells in the table has one of the following values:

1) None: nothing happens.
2) Death: the thing, or agent, dies. If a thing dies, it disappears from the grid and cannot interact with any other thing or with the agent for the rest of the game. If the agent dies, the game is over.
3) Teleport: the thing, or agent, is moved to a randomly chosen grid cell. The position is constrained so as not to be within two cells of the agent.

The score effects table has the same dimensions as the collision effects table, and is indexed in the same way. The difference is that the effects are constrained to incrementing or decrementing the score: the possible values of a cell is -1, 0 and $+1$.

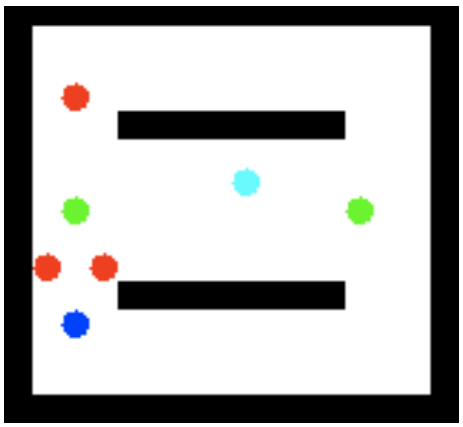

Fig. 1. The example game at $t = 0$. The cyan circle is the agent, the other circles are things of different colours. Note that all the thing positions are randomized, whereas the layout of the walls is hard-coded and subject to neither evolution nor changes during gameplay.

A. An example game

To illustrate the workings of the game engine and one sort of game that can be found in its rule space, an example game was hand crafted. Its parameters are:

- t_{max} : 50
- $score_{max}$: 50
- 3 red things; still
- 2 green things; counterclockwise movement
- 1 blue thing; clockwise movement
- collision: Red, Green → none, teleport, 0, 0[1]
- collision: Red, Blue → none, teleport, 0, 0
- collision: Red, Agent → none, death, 0, 0
- collision: Green, Agent → death, none, 0, -1
- collision: Blue, Agent → none, teleport, 0, 1

All other collisions are left inconsequential (none, none) and without score effects (0, 0).

The result of these rules is an unremarkable little game where the agent (which can be controlled by a human player using the arrow keys of the keyboard) is supposed to catch the solitary blue thing twice within 50 time steps. This is made more difficult by the blue thing moving counterclockwise (if the player chooses to chase it around the outer walls of the grid he quickly runs out of time, as the agent moves with the same speed as the thing) and the agent teleporting

[1]This line should be read as follows: when a red and a green thing end a time step in the same grid cell, nothing happens to the red thing, the green thing is teleported to a randomly chosen grid cell, and the score does not change $(0 + 0 = 0)$

to a random place whenever he reaches the blue thing. Also complicating the matter are the red things which act rather like land mines in this game, remaining in place and being deadly to the touch, and the green things that move around counterclockwise and causes the player to lose score if he touches them.

Using the current user interface, where time does not pass independently of the player but waits for his input before advancing to the next time step, the game is not hard to win for a player that understands the rules. But it is not altogether trivial either, and with a real-time interface it would probably be moderately hard. However, it is not very fun to play. See figure 1 for a screenshot of the game engine interface when playing this game.

III. Evolvable controllers

Our proposed technique for generating entertaining games builds on a measure of their learnability. A necessary preliminary step is thus devising a controller representation and learning mechanism that can learn to play games represented in the rule space. For this first experiment, we will base the controllers on neural networks and train them with evolutionary algorithms.

The neural network is a standard multi-layer perceptron, with 60 inputs, 10 hidden neurons and 4 outputs. The first 4×12 inputs are fed by four different visual fields around the agent. Each visual field consists of the 12 cells that are within a (Manhattan) distance of two cells from the agent, but sees only one colour: red, green, blue or black (walls). Each input to the neural network from a visual field is set to one if there is a thing of the correct colour (or a wall, if the colour of that visual field is black) in the corresponding cell, and zero other wise.

The next 3×4 inputs signify directions to the nearest thing of each colour. For each colour, there are 4 inputs representing the four directions; one of them is set to one and the others to zero. This is calculated through finding the thing of the specified colour with lowest Manhattan distance to the agent, and looking at the sign of the axis with largest difference.

At the other end of the network, the four outputs are interpreted as the desired movement direction of the agent; the direction associated with the output with highest activation is greedily chosen.

Learning is done with a standard evolution strategy (ES) without self-adaptation. Mutation consists in adding Gaussian noise with mean 0 and standard deviation 0.1 to all weights in the neural network.

The fitness function is defined thus:

$$fitness = \begin{cases} -1 & \text{if the agent dies} \\ score/score_{max} & \text{otherwise,} \end{cases}$$

A. Evolving controllers for the example game

A $50+50$ ES was used to evolve controllers (as described above) for the example game. To counteract the very noisy fitness evaluations, fitness for each controller was calculated as the mean of 20 iterations. Using this setup, each generation takes about two seconds. At the beginning of an evolutionary run, the best fitness found in the population is typically around 0.2. At the end of 100 generations, the best controller found always had significantly higher fitness than that, typically around 0.6.

This means that while evolution makes steady progress, it fails to find controllers that reliably win the game. A typical evolved controller sometimes wins the game, and other times fails through some mistake of varying severity. Ocular inspection of the behaviour of evolved controllers show that the agents often adopt the simple strategy but rather effective strategy of moving to one of the side walls of the grid and wait for the blue thing to come around. However, the movement is seldom in a straight line and typically quite erratic; sometimes the agent moves back and forth a few times before continuing, even though nothing in its immediate neighbourhood has changed. Given that the controller is stateless, it seems that the direction-to-nearest-thing inputs sometimes confuse the agent. (Removing those inputs did not significantly improve performance, nor did using recurrent neural networks, at least in initial experiments.) Moreover, the avoidance of red and green things is never perfect; it is quite typical to see the same agent elegantly sidestepping a moving green thing that comes from above, but just stupidly standing there when a green thing approaches from below.

While it might be expected that the proposed mechanism should learn to play such a simple game perfectly, this is a proof-of-concept study, and so we relegate improving the performance of controller learning to future work, and move on the evolving the game rules themselves.

IV. Evolving rules

For rule evolution, the fitness function is defined as follows: first, the game is tested with two different types of random controllers (controllers taking moving in random directions, changing direction either every time step or every few time steps). If any of these controllers score above 0.3, averaged over ten games, the game is deemed to easy, and is assigned a fitness of -1. Otherwise, the fitness is the average of the best controller fitness fund after 100 generations of evolution with a $5 + 5$ ES, and only 5 trials per fitness controller evaluation. In other words, the fitness function gives very low fitness to games that do not require any skill to play, low fitness to hard and impossible games, and high fitness to games that can be learnt quickly.

In this initial experiment, we will use a simple hill-climber to search rule space. The search starts with a randomly initialized rule set, where all the variables (simple parameters and parameters in tables) defined in section II are set to uniformly randomly selected values within their legal ranges. At each generation, a copy is then made of the rule set, and the copy is mutated. The mutation operator resets at least one randomly selected parameter to a legal random value, and as long as a freshly drawn random number is below 0.8, more parameters are reset. After the copy is mutated, the fitness of

both the original and the copy is evaluated, and if the copy does not have lower fitness it replaces the original.

It is obvious that in the current rule space, the vast majority of games are unplayable; they are either impossible, or the player wins on walk-over. As hill-climbers are particularly sensitive to starting positions, this means that many runs of the hill-climber find only games with zero or negative fitness. However, other runs do find playable games.

A. Examples of evolved games

The games below are all playable, but surely no *Portal*. They do not represent good game design and are not particularly fun. But they are sampled from a limited number of runs of the algorithm described above, and are guaranteed free from human design involvement.

1) Chase the blue:

- $t_{max} = 37$
- $score_{max} = 2$
- 0 red things; random long
- 18 green things, counterclockwise
- 1 blue things, counterclockwise
- collision: Red, Green →none, teleport, -1, 0
- collision: Red, Blue → death, teleport, 0, -1
- collision: Red, Agent → teleport, death, 0, 0
- collision: Green, Blue → teleport, none, -1, 1
- collision: Green, Agent → teleport, none, -1, 1
- collision: Blue, Agent → none, teleport, 0, 1

The whole point of this game is to get to catch the blue thing, twice and fast. It works if you play well and the blue thing doesn't start way off. The green things move around prettily and teleport when you touch them, but are inconsequential.

2) Red/green suicide:

- $t_{max} = 94$
- $score_{max} = 1$
- 4 red things; random long
- 10 green things, clockwise
- 11 blue things, still
- collision: Red, Green → death, death, -1, 1
- collision: Red, Blue → teleport, none, -1, 0
- collision: Red, Agent → none, teleport, 0, -1
- collision: Green, Blue → teleport, none, -1, 0
- collision: Green, Agent → death, none, 0, -1
- collision: Blue, Agent → none, none, 1, 0

This game initially seems impossible. Whatever one does, the score rapidly drops by several points per time step, sometimes reaching levels close to -100. However, after a while, this stops happening, as the red things kill themselves and bring an equal number of green things with them. The player can start incrementing the score, with the goal of bringing it over zero, and usually he is successful in this through just going back and forth on a blue thing or between two blue things if they happen to be placed next to each other. The fact that the agent can kill green things, though losing a point while doing it, suggests going for the green things as soon as possible to stop the score dropping sooner. However, we are not sure this is a better strategy than just going back and forth on a blue thing.

3) Race against green:

- $t_{max} = 28$
- $score_{max} = 6$
- 0 red things, random short
- 4 green things, clockwise
- 9 blue things, still
- collision: Red, Green → none, death, -1, -1
- collision: Red, Blue → death, death, 1, 1
- collision: Red, Agent → death, death, -1, 0
- collision: Green, Blue → none, death, -1, -1
- collision: Green, Agent → teleport, none, -1, 1
- collision: Blue, Agent → death, none, 1, 1

You have only 28 time steps to reach score 6. Fortunately, the blue things are just lying around, waiting to be picked up and reward you with two points each. Unfortunately, the green things are also out to get (though they are just moving clockwise) the blue things, and when they do, you lose two points! But if you see a green thing in the vicinity, you can easily bump him away to another part of the board without score penalty...

B. Observations on evolved games

Apart from the games described above, many evolutionary runs get stuck on games on which a good player can score better than a random player, but which are not winnable. In some cases there is not enough time to get the required score, and sometimes things of a particular colour add to the score only as they are consumed, and there are not enough of these. In these cases, we have a working game mechanic with maladjusted parameters. It is currently not clear why the evolutionary process does not correct these seemingly simple cases through adding more things of the specified colour, increasing the time limit or decreasing the score limit.

The evolved games which are winnable are typically very easy to win. This is no doubt because of the very limited capability of the current controller learning process; only games which are very easy to win gets good fitness.

V. Discussion

This paper lays no claim to provide a fully working technique for creating new game. It merely aims to point out an new, unexplored research direction, argue for the relevance of such research, and provide a proof-of-concept experiment that shows that the main idea is tenable at all. Incidentally, this experiments highlights some of the challenges we are facing.

One clear challenge is that an evolutionary process where the fitness function includes another evolutionary process takes very long time. The time taken for experiments is the reason (together with there being a deadline for submission of the paper) that not more results are included in this paper. One remedy for this (that does not include using more computer power) could be to replace evolution in the controller learning "inner loop" with some other reinforcement learning

algorithm. We have previously seen that temporal difference learning in many cases learn good behaviour much faster than evolutionary algorithms, but often reaches a lower final fitness [19].

At the same time, we would like to improve the quality of the learning in the inner loop. The results in section III-A are not satisfactory. If our controller learning method cannot learn to play such a simple game, how could it be a good guide to which games can be learnable by humans (who are, after all, quite clever)? The controller architecture and learning method could probably be improved in many ways. For example, the state representation seems to be far from optimal; one could start by introducing symmetry, and then take cues as to how state is represented to the controller in published CIG studies on learning to play games of similar genres, such as *Pac-man* [20]. Given that the state space is discrete, a neural network might not be the best way of representing the controller; quite possibly, expression tree-based genetic programming or a rule-based system would do a better job. And a better learning algorithm and controller representation would probably have resulted in better games in section IV-A.

A deeper question about the controller learning is to what extent it is similar to human game learning. It is plausible that we could develop controller representations and learning algorithms that learned to play a given game better or worse, faster or slower than a typical human, or any human in particular. But would this algorithm have problems with the same things? Maybe the algorithm and the human learns to play *Asteroids* in the same amount of time and to the same level, but the same algorithm can also learn some obscure game that just seems random to the human, and the human can learn games where the algorithm gets nowhere. This issue requires further study, involving trials with human subjects. Ideally, we would like to be able to model the learning profile of individual humans, and base our rule set fitness function on a "learning model".

It could be argued that the games described in the section above are really only minor variations of the same game, and that the rules of this game are what we choose to call the axioms for the rule space. This is a valid point to the extent that the border is a bit arbitrary. But the games above are arguably as different from each other as *Tetris* from *Columns* or *Dr. Mario*, or *Pong* from *Breakout*: they share an underlying mechanic, but what objects appear in the game, its goal and what you need to do to win all differ.

The representation of the rule space in the current experiment is deliberately basic. At least one other attempt to create a representation for arbitrary game rules exists, namely the the more complex Lisp-like *Game Description Language* used in the *AAAI General Game Playing Competition*[2]. It would be worth studying the advantages and disadvantages for expressivity and evolvability of that language versus simpler representations.

One could also choose to take a more inclusive view of what constitutes a rule in order to achieve a richer search space. For the game engine used in this paper, rules could coevolve (either as a part of the same genotype or cooperatively) with the layout of the walls. In a car racing game, the track could coevolve with car parameters (motor effect, tyre friction, damage tolerance etc.) and aspects of the game rules (will you get disqualified for pushing your opponents off the track?).

In a bigger perspective, the question is for what automatic game design would be useful, given that effective enough methods were developed.

When a new video game gets a mediocre review in a newspaper or on a game site, one of the most common complaints of the reviewer is that the game lacks imagination. It's just another game of this or that subgenre (e.g. WWII shooter); not necessarily bad, but certainly not innovative. One would be forgiven for thinking that there is simply a lack of fresh ideas among game developers[3]. Enter evolutionary computation: what evolutionary algorithms do best is producing unexpected solutions to problems. Numerous experiments in evolutionary art testify to this.

One use of automatic game design would be to develop prototypes of completely new games. The game designer specifies a game engine (something as simple as in this paper, or something as sophisticated as the *Unreal* engine) and the axioms that define a rule space, and sits back to watch evolution produce new game ideas. These would naturally need to be refined and elaborated on by human artists and programmers.

But it is equally plausible to use automatic game design at the other end of the design process. After humans have designed and implemented a game, it could be useful to automatically be able to fine-tune it; the techniques described here could be used to ensure that all levels or areas of the game were playable and of a certain approximate difficulty level, and implicitly to rid them of bugs.

Finally, it would also be possible to include more than one learning algorithm in the fitness function, and select for games that maximize the difference between the different learning algorithms' performance. Such games could be very interesting as reinforcement learning benchmark functions.

VI. Conclusion

We have outlined a technique for automatically generating complete games using evolutionary computation, a described a proof-of-concept experiment. The main originality in this paper probably lies in the fitness function for game rules, which is based on learning progress, and inspired by Koster's theory of fun and Schmidhuber's theory of artificial curiosity. Further, we have argued that this is an important research direction that might make CIG research more relevant to the game industry. Our main aim with this paper is to show what sort of things can be done, and how much there is left to do,

[2]http://games.stanford.edu/

[3]Another hypothesis is that making another copycat game pays off better on average than trying to be innovative, because consumers don't value innovations highly enough.

in this exciting new research field. And of course to persuade you too to do research in automatic game design.

ACKNOWLEDGEMENTS

This research was supported in part by the Swiss National Science Foundation (SNF) grant number 200021-113364/1. Thanks to Tom Schaul for insightful discussions.

REFERENCES

[1] G. N. Yannakakis, "Ai in computer games: Generating interesting interactive opponents by the use of evolutionary computation," Ph.D. dissertation, University of Edinburgh, 2005.

[2] B. D. Bryant, "Evolving visibly intelligent behavior for embedded game agents," Ph.D. dissertation, Department of Computer Sciences, University of Texas, Austin, TX, 2006.

[3] J. Denzinger, K. Loose, D. Gates, and J. Buchanan, "Dealing with parameterized actions in behavior testing of commercial computer games," in *Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games (CIG)*, 2005, pp. 37–43.

[4] J. Togelius, R. De Nardi, and S. M. Lucas, "Towards automatic personalised content creation in racing games," in *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2007.

[5] S. Priesterjahn, "Online imitation and adaptation in modern computer games," Ph.D. dissertation, University of Paderborn, Paderborn, Germany, 2008.

[6] K. O. Stanley, B. D. Bryant, and R. Miikkulainen, "Real-time neuroevolution in the nero video game." *IEEE Transactions on Evolutionary Computation*, vol. 9, no. 6, pp. 653–668, 2005.

[7] D. Ashlock, T. Manikas, and K. Ashenayi, "Evolving a diverse collection of robot path planning problems," in *Proceedings of the Congress On Evolutionary Computation*, 2006, pp. 6728–6735.

[8] K. Salen and E. Zimmerman, *Rules of Play: Game Design Fundamentals*. MIT Press, 2004.

[9] C. Browne, "Automatic generation and evaluation of recombination games," Ph.D. dissertation, Queensland University of Technology, Brisbane, Australia, 2008.

[10] M. Csikszentmihalyi, *Flow: the Psychology of Optimal Experience*. Harper Collins, 1990.

[11] T. W. Malone, "What makes things fun to learn? heuristics for designing instructional computer games," in *Proceedings of the 3rd ACM SIGSMALL symposium and the first SIGPC symposium on Small systems*, 1980, pp. 162–169.

[12] P. Sweetser and P. Wyeth, "Gameflow: A model for evaluating player enjoyment in games," *ACM Computers in Entertainment*, vol. 3, 2005.

[13] R. Koster, *A theory of fun for game design*. Paraglyph press, 2005.

[14] J. Schmidhuber, "Developmental robotics, optimal artificial curiosity, creativity, music, and the fine arts," *Connection Science*, vol. 18, pp. 173–187, 2006.

[15] ——, "A possibility for implementing curiosity and boredom in model-building neural controllers," in *Proceedings of the International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, 1991, pp. 222–227.

[16] ——, "Curious model-building control systems," in *Proceedings of the International Joint Conference on Neural Networks*, 1991, p. 14581463.

[17] ——, "Exploring the predictable," in *Advances in Evolutionary Computing*. Springer, 2002, pp. 579–612.

[18] J. Togelius and S. M. Lucas, "Evolving robust and specialized car racing skills," in *Proceedings of the IEEE Congress on Evolutionary Computation*, 2006.

[19] S. M. Lucas and J. Togelius, "Point-to-point car racing: an initial study of evolution versus temporal difference learning," in *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2007.

[20] S. Lucas, "Evolving a neural network location evaluator to play ms. pac-man," in *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2005, pp. 203–210.