A Video Game Description Language for Model-based or Interactive Learning

Tom Schaul
Courant Institute of Mathematical Sciences
New York University, 715 Broadway,
10003, New York
schaul@cims.nyu.edu

***Abstract*—We propose a powerful new tool for conducting research on computational intelligence and games. 'PyVGDL' is a simple, high-level description language for 2D video games, and the accompanying software library permits parsing and instantly playing those games. The streamlined design of the language is based on defining locations and dynamics for simple building blocks, and the interaction effects when such objects collide, all of which are provided in a rich ontology. It can be used to quickly design games, without needing to deal with control structures, and the concise language is also accessible to generative approaches. We show how the dynamics of many classical games can be generated from a few lines of PyVGDL.**

The main objective of these generated games is to serve as diverse benchmark problems for learning and planning algorithms; so we provide a collection of interfaces for different types of learning agents, with visual or abstract observations, from a global or first-person viewpoint. To demonstrate the library's usefulness in a broad range of learning scenarios, we show how to learn competent behaviors when a model of the game dynamics is available or when it is not, when full state information is given to the agent or just subjective observations, when learning is interactive or in batch-mode, and for a number of different learning algorithms, including reinforcement learning and evolutionary search.

I. Motivation

During a session at a recent workshop in Schloß Dagstuhl, it was proposed to develop a video game description language (VGDL) in order to facilitate the generation (guided or automatic) of very large and diverse portfolios of games, which are in turn suitable for evaluating architectures and algorithms that purport to be *general-purpose*; the complete deliberations are published in a report [1].

Motivated by those discussions, this paper is following up and proposing a concrete instantiation of a VGDL, formally defined, and an accompanying implementation in Python, including many useful tools for interfacing with diverse learning paradigms, and which satisfies most of the envisioned criteria. For completeness, we reiterate the main motivations for a high-level VGDL from this report, and highlighting the desired features of such a language.

Working backward from the long-term goal of using games to foster artificial *general* intelligence [2], a recent proposal has argued for extending the framework of general game playing (GGP) to video games [3], and to establish an AI competition where agents must demonstrate their proficiency

```
wwwwwwwwwwwww
wA       w  w
w  w        w
w   w   w +ww
www w1  wwwww
w       w G w
w 1        ww
w     1    ww
wwwwwwwwwwwww
```

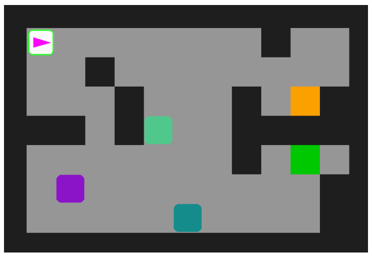

Fig. 1. Textual level description for a Legend of Zelda-like game (left), and its rendering (right). Here the avatar of Link 'A' starts in the top left, needs to find a key '+' and exit through to the goal 'G' while avoiding or killing monsters '1'. Impenetrable walls are found at the 'w' locations.

on a wide range of video games. To make this feasible, the authors proposed to limit the domain to arcade-style games in 2D, which form a sufficiently diverse space, given that they kept a generation of human gamers interested. We thus propose a video game description language designed for this purpose, which should eventually permit capturing the majority of mechanisms found in arcade games.

We reiterate here the desirable features for such a VGDL, as motivated in more depth in [1]: The language should be *clear, human-readable, and unambiguous*. Its vocabulary should be highly *expressive* from the beginning, yet still *extensible* to novel types of games. Finally, its representation structure should be easy to *parse* and facilitate automatically *generated* games, in such a way that default settings and sanity checks enable most random game description to be actually playable.

Previous work included description languages for logic-based games [4], [5], board games [6], or text-based adventures [7]. On the other hand, high-level programming/scripting languages for video games exist (e.g., [8], [9]), but none are as deliberately abstract. This work also draws inspiration from the Arcade Learning Environment (ALE) [10], a framework for games from the classic Atari 2600 console. A related, but less ambitious precursor to our design was proposed in [11]. For a broader overview, limitations and many additional references see [1], [3].

The next section describes our proposed language, including the formal syntax. Section III provides an overview of its implementation, discussing the parser, interpreter and the multiple interfaces for human and non-human players.

```
BasicGame
  LevelMapping
    G > goal
    + > key
    A > nokey
    1 > monster
  SpriteSet
    goal   > Immovable color=GREEN
    key    > Immovable color=ORANGE
    sword > Flicker limit=5 singleton=True
    movable      >
      avatar     > ShootAvatar stype=sword
        nokey    >
        withkey > color=ORANGE
      monster    > RandomNPC cooldown=4
  InteractionSet
    movable wall   > stepBack
    nokey goal     > stepBack
    goal withkey   > killSprite
    monster sword > killSprite
    avatar monster> killSprite
    key  avatar    > killSprite
    nokey key   > transformTo stype=withkey
  TerminationSet
    SpriteCounter stype=goal    win=True
    SpriteCounter stype=avatar win=False
```

Fig. 2. Game description for a Legend of Zelda-like game. Words in violet are (arbitrary) user-defined identifiers for different sprite types (used elsewhere using the keywork parameter `stype`), the texts in blue is referring to elements from ontology. Note the hierarchy of class definitions in the 'SpriteSet' block, the on-the-fly specialization of ontology elements with the 'keyword=value' format.

Section IV demonstrates the simplicity and diversity through a few example games that can be defined very concisely. Finally, in section V we show how such games can be used to learn behaviors, in three different paradigms: model-based and fully observable policy iteration, model-free and partially-observable reinforcement learning, and direct evolution of bot controllers. Before concluding, we then outline some future directions of development.

II. The PyVGDL Language

The canonical video game description language is implemented in Python, and builds directly on the widely used `pygame` package for game development [9] (but is much higher level); henceforth we refer to it by "PyVGDL". The complete source code, including many example games, is available at:

`https://github.com/schaul/py-vgdl`.

Our design for the structure of the description language (as discussed in [1]) is to restrict ourselves to a 2-dimensional (rectangular) space, in which all game-relevant *objects* are located. Essentially, objects can move, interact with other objects, disappear, or spawn new objects, subject to global or object-specific rules and player actions.

A game is defined by two separate components, the *level description*, which essentially describes the positions of all objects and the layout of the game in 2D (i.e., the initial conditions), and the *game description* proper, which describes the dynamics and potential interactions of all the objects in the game.

The level description is simply a text string/file with a number of equal-length lines, where each character maps to (read: instantiates) one or more objects at the corresponding location of the rectangular grid. See Figure 1 for an example level description.

The game description is composed of four blocks of instructions. Figure 2 provides an example of a full game description, based on the game Legend of Zelda, and we will refer to it to illustrate the different concepts below.

- The `LevelMapping` describes how to translate the characters in the level description into (one or more) objects, to generate the initial game state. For example, each '1' spawns an object of the 'monster' class.
- The `SpriteSet` defines the classes of objects used, all of which are defined in the ontology, and derive from an abstract `VGDLSprite` class. Object classes are organized in a tree (using nested indentations), where a child class will inherit the properties of its ancestors. For example, there are two subclasses of avatars, one where Link possesses the key and one where he does not. Furthermore, all class definitions can be augmented by keyword options. For example, the 'key' and 'goal' classes differ only by their color and how they interact.
- The `InteractionSet` defines the potential events that happen when two objects collide. Each such interaction maps two object classes to an event method (defined in the ontology), possibly augmented by keyword options. For example, swords kill monsters, monsters kill the avatar (both subclasses), nobody is allowed to pass through walls, and when Link finds a 'key' object, the avatar class is transformed.
- The `TerminationSet` defines different ways by which the game can end, each line is a termination criterion, different criteria are available through the ontology and can be further specialized with keyword options. Here, it is sufficient to capture the goal (bring the sprite counter of the 'goal' class to zero) to win.

What permits the descriptions to be so concise is an underlying *ontology* which defines many high-level building blocks for games, including the types of physics used (continuous, or grid based, friction, gravity, etc.), movement dynamics of objects (straight or random motion, player-control, etc.) and interaction effects upon object collisions (bouncing, destruction, spawning, transformation, etc.).

Further, many components are easily *recombinable* through subclasses and keyword options. For example, an object class, when defined with the option `physicstype=GravityPhysics` dramatically alters its behavior (now suddenly being subjected to gravity). See Figure 8, which gives the game description of Lunar Lander. Also, as all projectiles are object classes themselves, their interactions with other objects or level structures can be altered in very simple ways. Recombinability goes as far as permitting level descriptions that were intended for one type of game to be used in a very different context, akin to what is done in the game "ROM Check Fail" [12].

Finally, it is relatively straightforward to *extend* the provided ontology of behaviors, effects and object classes. In fact, most of the many predefined classes and effects are coded in just a handful lines of Python code. In other words, this permits easy prototyping of novel game dynamics, which can immediately be recombined with existing ones.

A. Features

To whet the reader's appetite, we list some of the elements available in the current ontology:

- Spawning, cloning and elimination of objects, as well as transformation from one type into another.
- Self-propelled movements of objects, taking consistent or random actions, or erratically changing direction.
- Non-deterministic chasing and fleeing behaviors.
- Projectile objects, spawned at the location of arbitrary objects, on fixed or random schedules, based on user actions, or triggered by collision effects.
- Stickiness, i.e., one object pulling another one.
- Bouncing and wrap-around behavior, from other objects or the edge of the screen.
- Teleportation of objects, to fixed or random end-locations.
- Continuous physics effects like inertia, friction and gravity.
- Stochastic effects like slipping in the current direction, or wind gusts.

B. Syntax

The PyVGDL *syntax* is based on a simplified version of the syntax of the Python language itself, retaining white-space based indentation, comments and keyword arguments. However, game descriptions must conform to a strict tree-like structure which resembles more closely an XML schema. Figure 4 gives the formal description of the syntax' context-free grammar in the notation of Extended Backus-Naur Form.

The *parser* implemented for the syntax handles both level descriptions and game descriptions, and given one of each (provided as text strings), it generates the full code for the game (in Python). The generated game object includes the dynamics, on-screen visualization, and interactions with the (human or artificial) player. Parsing and instantiation takes less than a second – making all generated games instantly playable.

III. Interpreter and Interfaces

As detailed above, the PyVGDL library defines a game description language, and an initial ontology of behaviors, but it also encompasses a wide range of additional tools that are designed to make it directly useful to the computational intelligence researcher.

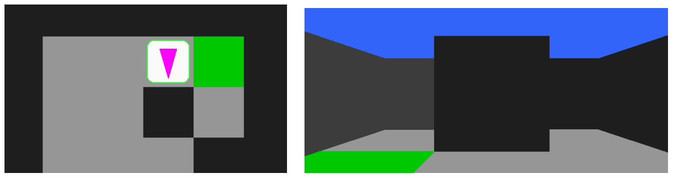
Fig. 3. For the same game state, we show the rendered game from the objective (birds-eye) perspective, left, and the subjective (first-person) perspective, right. Note how in the subjective view, impenetrable objects like walls are shown as blocks, while other objects (the green one in the corner) are drawn on the level floor.

There are a number of distinct dimensions in which learning frameworks may vary. Below, we list which of these options are currently available, and how they are implemented:

- *Player type*: we currently support direct interactive play with a human player (through the keyboard), and an interface to artificial players (bots), which may take one action per cycle. The interface for the bots is conforming to the "Agent/Environment/Task" model of the PyBrain machine learning library [13], allowing an arbitrary controller to interact with the game chiefly through calls to the `getSensors` and `performAction` methods. This could easily be extended to richer interfaces as well.
- *Number of players*: The library currently supports a single player, which is the dominant means of measuring absolute performance. Nevertheless, we plan to extend the interface to multiple agents (mixed human/non-human control), and connect it to a tournament environment for bot play.
- *Perspective*: By default, a game is played from the birds-eye perspective (objective), with the full rectangular 2D space visible at once. As an alternative, we also provide an option to play from a first-person viewpoint (subjective), where the game becomes effectively *partially observable* (not implemented for all types of physical dynamics yet). For an illustration, see Figure 3.
- *Observation*: Orthogonally to the perspective of observation, we also provide to different types of encoding for observations provided to the player: they are available rendered visually as a medium-resolution image, or in 'clean' form, representing only the functionally different components. The latter is making learning easier, as the representation is capturing all the essential information without redundancy, but eventually, strong general-purpose agents should be capable of dealing with the visual observation stream directly, like humans do.
- *Model*: Some learning and planning approaches rely on the availability of a complete forward-model of the game dynamics, in order to simulate (roll-out) action sequences before taking a decision. This is always available, as the game state can be read, stored and reset after the roll-out. To further accommodate model-based approaches, we provide a conversion tool, which transforms the game dynamics into the full transition matrices of a Markov Decision Process

⟨*game*⟩	::= **game_class** ⟨*eol*⟩ INDENT ⟨*level-block*⟩ ⟨*sprite-block*⟩ ⟨*interaction-block*⟩ ⟨*termination-block*⟩
⟨*level-block*⟩	::= `LevelMapping` ⟨*eol*⟩ INDENT { ⟨*char-map*⟩ NEWLINE } DEDENT
⟨*sprite-block*⟩	::= `SpriteSet` ⟨*eol*⟩ INDENT { ⟨*sprite-def*⟩ NEWLINE } DEDENT
⟨*interaction-block*⟩	::= `InteractionSet` ⟨*eol*⟩ INDENT { ⟨*interaction-def*⟩ ⟨*eol*⟩ } DEDENT
⟨*termination-block*⟩	::= `TerminationSet` ⟨*eol*⟩ INDENT { ⟨*termination-def*⟩ ⟨*eol*⟩ } DEDENT
⟨*char-map*⟩	::= CHAR ‘ > ’ ⟨*sprite-type*⟩ { ‘ ’ ⟨*sprite-type*⟩ }
⟨*sprite-def*⟩	::= ⟨*sprite-simple*⟩ [⟨*eol*⟩ INDENT { ⟨*sprite-def*⟩ ⟨*eol*⟩ } DEDENT]
⟨*sprite-simple*⟩	::= ⟨*sprite-type*⟩ ‘ > ’ [**sprite_class**] { ‘ ’ ⟨*option*⟩ }
⟨*interaction-def*⟩	::= ⟨*sprite-type*⟩ ⟨*sprite-type*⟩ ‘ > ’ **interaction_method** { ‘ ’ ⟨*option*⟩ }
⟨*termination-def*⟩	::= **termination_class** { ‘ ’ ⟨*option*⟩ }
⟨*eol*⟩	::= { ‘ ’ } [‘#’ { CHAR | ‘ ’ }] NEWLINE
⟨*option*⟩	::= IDENTIFIER ‘=’ (⟨*sprite-type*⟩ | **evaluable**)
⟨*sprite-type*⟩	::= IDENTIFIER | ‘`avatar`’ | ‘`wall`’ | ‘`EOS`’

Fig. 4. Grammar of the video game description language PyVGDL, in Extended Backus-Naur Form (square brackets indicate optional terms, curly brackets indicate repetition). The terms in **bold** denote terms that are defined in the ontology that accompanies the grammar. Furthermore, "**evaluable**" could be any valid Python expression that can be evaluated within the scope of the ontology. Note that implicitly, only sprite types that are defined in the level-block (or are defined by default) can be used in the other blocks.

(MDP). Given that those matrices scale quadratically with the number of distinct states, this is only feasible for games that do not suffer from a combinatorial explosion of reachable states.

Other useful tools include the possibility to enable/disable the visualization (resulting in dramatic speed gains), to record the actions taken during a game and replay them, and to create animated GIF videos from such replayed action sequences.

Our objectives are for PyVGDL to remain lightweight, fast and agile, so very little emphasis has been on sophisticated rendering of the objects, creatures and environment, nor are there any flashy animated effects (most objects are just solid colored squares). This aspect is extensible however, because care was taken to keep the graphical aspects and the game dynamics separate, so adding a new (or interfacing to an existing), more advanced rendering engine should be easy.

IV. Example Games and Benchmarks

To demonstrate the wide spectrum of games that can be encoded in PyVGDL, we implemented simplified versions of a number of classical, well-known games, and some classical reinforcement learning (RL) benchmarks. They include:

- *Space invaders*, with shooting, complex movement sequences, timed spawning points.
- *Frogger*, with sticky/attachable objects and wrap-around movement.
- *Lunar lander*, with delicate continuous gravity and inertial effects.
- *Physical Traveling Salesman Problem* (PTSP [14]), with continuous control of salesman spaceship that can bounce off walls.
- *Pac-Man*, showing off simple ghost chasing behaviors and transformative power pills.
- *Sokoban*, where the agent can push blocks around in a maze, but not pull them.
- *Dig-Dug*, with diggable ground, and sacks of gold that fall through the tunnel system when touched.
- *Portal*, exploring different types of teleportation mechanisms.
- *Legends of Zelda*, with a unique directional attack, keys, and locked doors; see Figures 1 and 2.
- *Super Mario*, including moving elevator platforms, Goombas and Koopa Paratroopas.
- *Windy gridworld* [15], demonstrating stochastic environmental features.
- *89-state maze* [16], with stochastic actions and observations; see Figure 5.
- *T-maze*, a classical RL task for generalizing state memorization across long time-scales [17].

The first three of these were the motivating examples in [1], and a few of them are depicted in Figure 9. All of them ship with the library, and serve the double purpose of providing a potential game developer with a tool to learn the language by example. The game descriptions are all concise and simple, the descriptions in Figure 2 and 8 are typical in that respect, and indeed none of the games mentioned require game descriptions of more than 40 lines.

V. Demonstrations

Producing agents that exhibit competent behavior in a game environment is a very general problem, and not surprisingly,

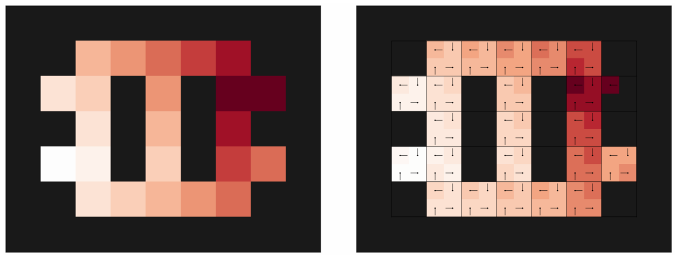

Fig. 5. Value function of the optimal policy, as computed by PI on a maze task. Darker red corresponds to higher values, black denotes inaccessible locations. The unique goal is in the upper left dead end. **Left**: the agent can move into any of the four cardinal directions, leading to 23 unique states. **Right**: the agent can move forward or rotate by an angle of 90, 180 or 270 degrees, leading to 89 distinct states, consisting of cell location plus agent orientation (marked by the black pointers).

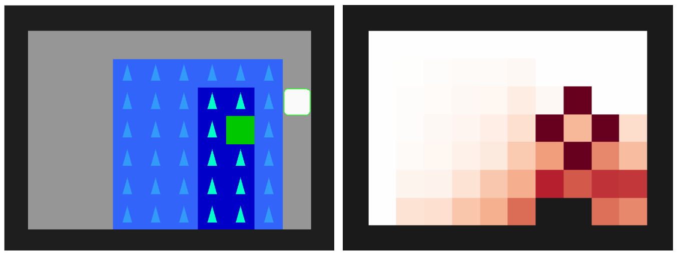

Fig. 6. Windy gridworld. **Left**: the game rendered for a human player, with the agent (in white with a green boundary) close to the goal (in green) already. Partial observability means that the agent will only see the colors in the four positions immediately adjacent to it, no further. **Right**: the value function of the agent policy learned by LSPI: darker red corresponds to better values.

drastically different methodologies exist, including model-based planning, Monte-Carlo tree search [18], various flavors of RL [15], [19], neuro-evolution [20], etc. PyVGDL aims to be agnostic with respect to how its games are used in that context.

In this section we will give examples of how some of these types of learning can use PyVGDL to define and interface to game benchmarks. In our case, all the algorithms that do the actual learning[1] are available in the PyBrain open source machine learning library [13].

A. Optimal policies with model and full state information

As an initial demonstration, we place a minimal burden on an agent's learning capabilities, namely providing it with perfect state information and a full model of the game dynamics: for each state and chosen action, it knows the true probabilities P of reaching the subsequent state, and the accompanying reward. The PyVGDL library can convert any sufficiently simple game into a $P \in \mathbb{R}^{|A|} \cdot \mathbb{R}^{|S|} \cdot \mathbb{R}^{|S|}$ tensor, where A and S denote the discrete action and state spaces.

For a given policy, it is therefore possible to compute the expected (discounted) future reward for each state in closed form, and we use *policy iteration* (PI [15]) to iterate between this evaluation step and the policy defined by greedily value-maximizing actions, until no further improvement is possible. Figure 5 shows the value function of the optimal agent, after convergence, for two maze variants using the same level map.

B. Least-squares approximation with partial observability

Often, state information is not available directly to the agent, instead it has access only to a set of *observations* or features, in each state. Among the algorithms that learn from sequences of actions and observations, without requiring a model or perfect information, we chose Least-Squares Policy Iteration (LSPI [21]), a batch method that minimizes the errors of the value function in a least-squares sense. LSPI accumulates observation-based transition data from which the approximate value function can then be found in closed form.

[1]All the scripts used to produce the results and figures in this paper are distributed with the PyVGDL code, as use-case examples, and to guarantee reproducibility.

The outer loop then iterates over increasingly better greedy policies, just like PI.

To add an additional difficulty, we test LSPI on a game benchmark with stochastic transitions, the classic windy gridworld [15]. Each state provides as observations the colors on the current and the four adjacent grid positions, where colors can take five different values (one each if the position has a wall, goal, weak wind, strong wind, or is empty), which makes many states ambiguous. For our experiment, we gather sufficient data that every action is attempted at least 20 times in each state. The resulting (approximate) value function is shown in Figure 6.

C. Training a neural network controller

An alternative to value-function based methods, particularly useful if the state-space is large, is to learn the policy of an agent directly, by encoding a mapping from observations to actions. A general-purpose tool for such an encoding are neural networks (NN), which can in principle approximate any continuous function, but may have many trainable parameters ('weights'). In this case, we use the interactive interface of PyVGDL, connecting the agent/policy directly to the game environment in which it can take actions for a while, after which it is evaluated based on performance achieved ('fitness').

We employ *Separable Natural Evolution Strategies* (SNES [22]), an evolutionary direct search algorithm, which is based on updating search distributions according to the estimated natural gradient. SNES is a state-of-the-art algorithm for non-Markov continuous control problems like the classic double-pole balancing task.[2] In addition, it's computational complexity scales linearly with the number of parameters, permitting the training of large networks. For the implementation, we use again the tools provided in the PyBrain library, which includes implementations of SNES, and of numerous neural network variants.

For this experiment, SNES was run for 12 generations (with its default population size of 17), tuning the 94 parameters of a 3-layer feed-forward network with softmax output and tanh transitions functions on its 6 hidden units. The fitness of individuals was averaged over three independent roll-outs. The results are depicted on Figure 7, which shows that a reactive observation-based navigation policy is easily learned.

[2]Also, its acronym suits the theme of the paper.

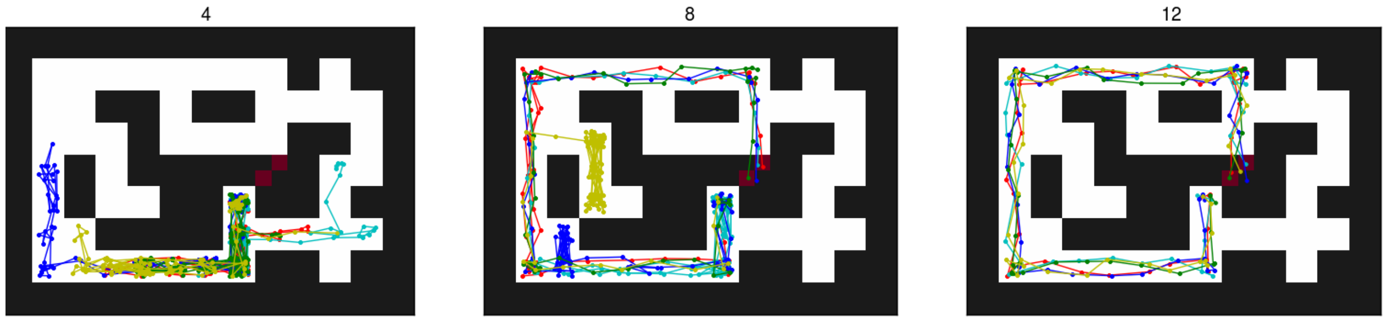

Fig. 7. Policies of evolved neural controllers using SNES, after a certain number of generations. Each colored line represents the trace of one roll-out, given the same initial position and following the currently best-found policy. The goal, in the middle of the maze, is marked in dark red. **Left**: after 4 generations, the best policy is a very exploratory one. **Center**: after 8 generations, the agent does sometimes reach its goal, and after 12 generations (**right**), it does so consistently and without detours. Note that the optimal policy here is reactive, that is does not require any form of memory of past observations or actions to succeed.

```
BasicGame
   LevelMapping
      G > pad
   SpriteSet
      pad    > Passive          color=BLUE
      avatar > InertialAvatar   physicstype=GravityPhysics
   InteractionSet
      avatar wall > killSprite
      avatar EOS  > killSprite
      pad avatar  > killIfSlow
   TerminationSet
      SpriteCounter  stype=pad     limit=4  win=True
      SpriteCounter  stype=avatar           win=False
```

Fig. 8. Game description for Lunar Lander, a continuous physics-based game. The game is over if the rocket can land one of five landing pads with low velocity (the pad is destroyed by such an interaction through the 'killIfSlow' method), after which the count of remaining pads goes to four, which triggers the winning condition. On the other hand, any collision with a wall or the end-of-screen ('EOS') ends the game in a loss.

VI. Future Work

The development work of PyVGDL is far from completed, and we propose extending it in a number of directions. In terms of game dynamics, a key lacking component are resource-type objects (health, heat, coins) which could be integrated in numerous ways, possibly borrowing from the Machinations framework [23]. Another important extension for the near future is a more complete, fine-grained score system with intermediate rewards to help RL approaches. Other desirable features are momentum-preserving object splits, area-of-effect events, or line-of-sight conditions. Some of these should arise naturally in the process of implementing more example games, such as Asteroids or Pong. More generally, we foresee permitting multi-player games (with mixed human/non-human controllers), additional visualization and characterization tools for learned agents, and parallelized execution of forward models to speed up learning.

VII. Conclusions

We proposed a powerful new tool for conducting research on computational intelligence and games. The simple, high-level language allows the design of games by non-programmers, and its concise descriptions are accessible to potential generative approaches (not just of game levels, but of full game dynamics). These games are immediately human-playable, from a global or first-person viewpoint, and can be directly interfaced to (learning) agents.

To demonstrate the package's usefulness in a broad range of scenarios, we provided a number of simple but diverse examples of gameplay, and showed how to learn competent behaviors when a model of the game dynamics is available or when it is not, when full state information is given to the agent, or just subjective observations, and for a number of different learning algorithms.

The library's availability as an open-source package, under the non-restrictive BSD license, is an invitation for others to use and build upon it, for example to diversify RL competitions [24], [25], more generally to benchmark learning algorithms in the games domain, or even to use it in teaching.

Acknowledgments

The author wants to thank the participants of the 2012 Dagstuhl Seminar on Artificial and Computational Intelligence in Games, in particular, Marc Ebner, John Levine, Simon Lucas, Tommy Thompson and Julian Togelius for the discussions that triggered and inspired the development of this paper; additional helpful input came from Mark Ring, and all four reviewers. This work was funded in part through AFR postdoc grant number 2915104, of the National Research Fund Luxembourg.

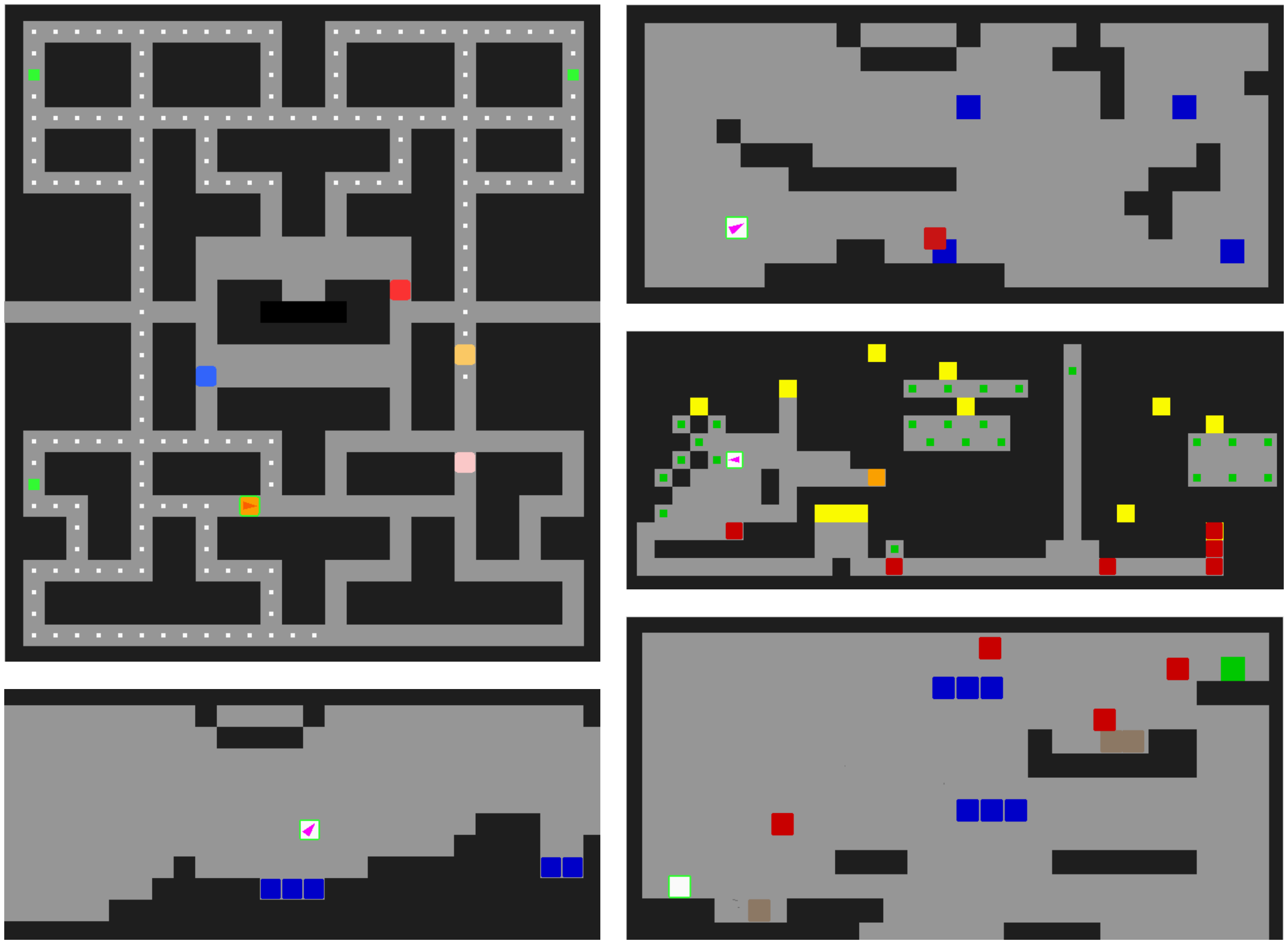

Fig. 9. Renderings (bird-eye view) of a few example games; clockwise from the top left: Pac-Man, Physical TSP, Dig-Dug, Super Mario and Lunar Lander.

REFERENCES

[1] M. Ebner, J. Levine, S. Lucas, T. Schaul, T. Thompson, and J. Togelius, "Towards a video game description language." *Dagstuhl Follow-up. To appear.*, Preprint available at http://www.idsia.ch/∼tom/publications/dagstuhl-vgdl.pdf.

[2] T. Schaul, J. Togelius, and J. Schmidhuber, "Measuring intelligence through games," *Arxiv preprint arXiv:1109.1314*, 2011.

[3] C. B. Congdon, M. Bida, M. Ebner, G. Kendall, J. Levine, S. Lucas, R. Miikkulainen, T. Schaul, and T. Thompson, "General video game playing," *Dagstuhl Follow-up. To appear.*, 2013.

[4] M. Genesereth and N. Love, "General game playing: Overview of the aaai competition," *AI Magazine*, vol. 26, pp. 62–72, 2005.

[5] A. M. Smith, M. J. Nelson, and M. Mateas, "Ludocore: A logical game engine for modeling videogames," in *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*. IEEE, 2010, pp. 91–98.

[6] C. Browne, "Evolutionary game design," *IEEE Transactions on Computational Intelligence and AI in Games*, pp. 11–21, 2011.

[7] G. Nelson, *The Inform Designer's Manual*. Placet Solutions, 2001.

[8] G. Maggiore, A. Spanò, R. Orsini, M. Bugliesi, M. Abbadi, and E. Steffinlongo, "A formal specification for casanova, a language for computer games," in *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems*. ACM, 2012, pp. 287–292.

[9] W. McGugan, *Beginning Game Development with Python and Pygame: From Novice to Professional*. Apress, 2007.

[10] M. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *Arxiv preprint arXiv:1207.4708*, 2012.

[11] J. Togelius and J. Schmidhuber, "An experiment in automatic game design," in *Proceedings of the IEEE Symposium on Computational Intelligence and Games (CIG)*, 2008.

[12] Farbs, "Rom check fail (game)," http://www.farbs.org/games.html, 2008.

[13] T. Schaul, J. Bayer, D. Wierstra, Y. Sun, M. Felder, F. Sehnke, T. Rückstieß, and J. Schmidhuber, "PyBrain," *Journal of Machine Learning Research*, vol. 11, p. 743746, 2010.

[14] D. Perez, P. Rohlfshagen, and S. M. Lucas, "The physical travelling salesman problem: WCCI 2012 competition," in *IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2012, pp. 1–8.

[15] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, Cambridge, MA, 1998.

[16] M. L. Littman, A. R. Cassandra, and L. P. Kaelbling, "Learning policies for partially observable environments: Scaling up," in *International Conference on Machine Learning*, 1995, pp. 362–370.

[17] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, pp. 1735–1780, 1997.

[18] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A

survey of monte carlo tree search methods," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 4, no. 1, pp. 1–43, 2012.

[19] J. Togelius, T. Schaul, D. Wierstra, C. Igel, F. Gomez, and J. Schmidhuber, "Ontogenetic and phylogenetic reinforcement learning," *Zeitschrift Künstliche Intelligenz - Special Issue on Reinforcement Learning*, pp. 30–33, 2009.

[20] L. Cardamone, D. Loiacono, and P. L. Lanzi, "Evolving competitive car controllers for racing games with neuroevolution," in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*. ACM, 2009, pp. 1179–1186.

[21] M. Lagoudakis and R. Parr, "Least-squares policy iteration," *Journal of Machine Learning Research*, 2003.

[22] T. Schaul, T. Glasmachers, and J. Schmidhuber, "High Dimensions and Heavy Tails for Natural Evolution Strategies," in *Genetic and Evolutionary Computation Conference (GECCO)*, Dublin, Ireland, 2011.

[23] J. Dormans, "Machinations: Elemental feedback structures for game design," in *Proceedings of the GAMEON-NA Conference*, 2009, pp. 33–40.

[24] S. Whiteson, B. Tanner, and A. White, "The reinforcement learning competitions," *AI Magazine*, vol. 31, no. 2, pp. 81–94, 2010.

[25] B. Tanner and A. White, "Rl-glue: Language-independent software for reinforcement-learning experiments," *The Journal of Machine Learning Research*, vol. 10, pp. 2133–2136, 2009.