An Extensible Description Language for Video Games

Tom Schaul

***Abstract*—In this short paper, we propose a powerful new tool for conducting research on computational intelligence and games. "PyVGDL" is a simple, high-level, extensible description language for 2-D video games. It is based on defining locations and dynamics for simple building blocks (objects), together with local interaction effects. A rich ontology defines various controllers, object behaviors, passive effects (physics), and collision effects. It can be used to quickly design games, without having to deal with control structures. We show how the dynamics of many classical games can be generated from a few lines of PyVGDL. Furthermore, the accompanying software library permits parsing and instantly playing those games, visualized from a bird's-eye or first-person viewpoint, and using them as benchmarks for learning algorithms.**

***Index Terms*—Artificial general intelligence (AGI), benchmarking, description language, general game playing, video games.**

I. Motivation

The majority of research in computational intelligence and games is based on just one, or a handful of similar games. This has the advantage of producing a high diversity of methods and algorithms, tailored to some degree to the games of interest. The disadvantage is that because of a lack of generality, many of the proposed methods do not generalize easily to new problems, and even fewer are useful across different genres of games. One (ambitious) direction for computational intelligence in games (CIG) research is for game collections to be used as benchmarks for the development of general-purpose intelligence algorithms (AGI), as proposed in [1].

In a recent Dagstuhl workshop, 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—thousands of games in dozens of genres—which, in turn, would be suitable for evaluating architectures and algorithms that purport to be general purpose; the complete deliberations are published in a report [2]. Among the envisioned criteria are that 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. Its representation structure should be easy to parse and visualize, so that it maps directly onto a (human-)playable game, which in addition must run fast enough to support learning and evaluation. Finally, it should facilitate automatically generated games, in such a way that default settings and sanity checks enable most random game descriptions to be actually playable.

Existing description languages for games tend to focus on classes of classical games, such as board games [5] or text-based adventures [6]. The elegant Ludocore framework [4] permits the modeling of the logical structure of video games, but not creating playable games. On the other hand, high-level programming/scripting languages for video games exist (e.g., [7] and [8]), but none are as deliberately abstract, or viable for generative approaches.

The related approach most widely used by researchers is the general game playing framework (GGP; [3]), with its associated annual competition at the Association for the Advancement of Artificial Intelligence (AAAI). Its description language GDL, designed for logic-based games, is extraordinarily expressive, while still being clear, unambiguous and human readable (or at least, logician readable). On the other hand, GDL is less suited to the last two criteria; even when parsed, checking the validity in a GDL game is difficult, and move resolution can be slow. Also, game descriptions are rather lengthy (because there is no support for hierarchical concepts, all objects are defined from the ground up), and it is easy to incorrectly define games. Further, GDL is not designed primarily for human interactions, and is not able to automatically generate an intuitive interface for human play. A voluntary restriction to a video game-style interface (observations on a 2-D screen, a limited number of discrete actions), and favoring the reuse of common game elements at the cost of generality, may thus be an interesting tradeoff.

For the types of video games we are interested in, this work draws particular inspiration from the Arcade Learning Environment (ALE) [9], a unified framework for interacting with a broad collection of existing games from the classic Atari 2600 console—but with the desire to both extend it to new games that were never built for Atari, and to simplify existing ones. A related, but less ambitious precursor to our design was proposed in [10]. For a broader overview, limitations, and many additional references, see [2] and [11].

In a recent paper [12], we formally defined PyVGDL, a concrete instantiation of a VGDL that satisfies the desirability criteria, with a particular tradeoff between specificity (with 2-D object- and collision-based game dynamics) and generality (with an extensible ontology). At the same time, we introduced an accompanying implementation in Python, and demonstrated that the library interface allows a diverse set of learning approaches to learn in such games, including model-based and fully observable policy iteration, model-free and partially observable reinforcement learning, and direct evolution of bot controllers. This short paper restates the core elements of [12], complemented with follow-up work on the language and its ontology of components, and shows the extensibility aspect of the language.

II. The PyVGDL Language

The initial ideas for the description language were laid out in [2], and a fleshed-out prototype was presented in [12]. This section summarizes the main elements, and introduces some recent innovations.

A. Design

The language is designed around objects in a 2-D space. All objects are physically located in a rectangular space (the screen), with associated coordinates (and possibly additional physical properties). Objects can move actively (through internally specified behaviors, or player actions), move passively (subject to physics), and interact with other objects through collisions. All interactions are local, subject to rules that depend only on the two colliding objects, and can lead to object

Manuscript received September 10, 2013; revised May 13, 2014 and June 30, 2014; accepted July 16, 2014. Date of publication August 27, 2014; date of current version December 11, 2014. This work was supported in part by the National Research Fund Luxembourg under AFR Postdoctoral Grant 2915104.

The author was with the Courant Institute of Mathematical Sciences, New York University, New York, NY 10003 USA. He is now with Google DeepMind, London EC3M 5DJ, U.K. (e-mail: schaul@gmail.com).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCIAIG.2014.2352795

1943-068X © 2014 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

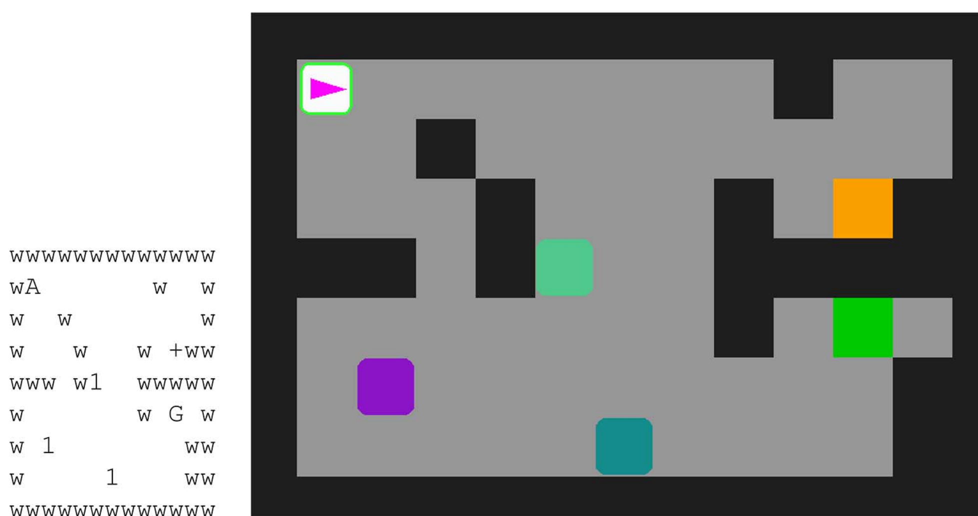

Fig. 1. Textual level description for a *Legend of Zelda*-like game (left), and its rendering (right). Here the avatar of Link "A" starts in the top left, needs to find a key "+," and exits through to the goal "G," while avoiding or killing monsters "1." Impenetrable walls are found at the "w" locations.

disappearance, spawning, transformation, or changes in object properties. To a certain degree, nonlocal interactions can still be achieved, e.g., through projectile particles or teleportation effects. The avatar(s) are special types of objects, in that they are also affected by control signals from the bot or human player.

Our emphasis on locality has the advantage that most decisions can be made with just local context, simplifying the task for bots that learn or plan in such games, especially if they only have subjective observations available, and as a side-effect speeding up the engine itself. Consistently with this, we deliberately do not allow for global variables or dynamics (with the exception of end-of-game timers)—while this may be common practice for programmers, it is unnecessary for describing most games (by localizing resources like health or mana within the avatar object). Enforced locality also makes it very easy to extend a game to a variant with two or more players: just initialize a level with more avatar objects.

A game instance is initialized from a starting configuration (defined in a textual level description; see below) and terminated after a finite number of timesteps with score value (which indicates win/loss). Longer games are formed by chaining such atomic instances together, usually by just providing more difficult levels subject to the same game dynamics, but possibly also by varying the game dynamics (e.g., increasing gravity in *Lunar Lander*).

B. Implementation

The canonical[1] video game description language is implemented in Python, and builds directly on the widely used `pygame` package for game development [8] (but is much higher level); henceforth, we refer to it as "PyVGDL." The 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 treelike structure which resembles more closely an XML schema. A formal description of the syntax's context-free grammar can be found in [12].

This is an open-source project, all code is licensed under the nonrestrictive Berkeley software distribution (BSD) license, and we welcome contributions—in the form of new games, new levels, new object or collision dynamics for the ontology, as well as improvements to the core library. The complete source code, including many example games, is available at: https://github.com/schaul/py-vgdl.

There is also a functionally equivalent Java implementation of the language,[2] which currently supports all the features of PyVGDL, except for the continuous physics aspects.

C. Game Descriptions

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 2-D (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 with a number of equal-length lines, where each character maps to one or more objects at the corresponding location of the rectangular grid. See Fig. 1 for an example level description.

The game description is composed of four blocks of instructions. Fig. 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 `VGDL-Sprite` 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.

[1]Some subtle aspects like the exact execution order of contradicting instructions upon simultaneous collisions are difficult to specify explicitly, which is why a reference implementation is relevant.

[2]https://github.com/EssexUniversityMCTS/gvgai

```
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 scoreChange=1
    avatar monster> killSprite
    key  avatar    > killSprite scoreChange=5
    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 text in blue is referring to components from within the ontology. Note the hierarchy of class definitions in the "SpriteSet" block, the on-the-fly specialization of ontology elements with the "keyword = value" format. For example, effects that set the `scoreChange` attribute affect the global game score.

- The `TerminationSet` defines different ways by which the game can end, with termination criteria available through the ontology that can be further specialized with keyword options. Here, it is sufficient for winning to capture the goal, i.e., bring the sprite counter of the "goal" class to zero.

D. Ontology

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, movement dynamics of objects, and interaction effects upon object collisions. The hybrid approach of a simple description language with an extensible ontology is a tradeoff between simplicity and extensibility: it keeps game descriptions simple and allows nonprogrammers to quickly build new games, while at the same time giving the freedom to more advanced users to use new types of dynamics by adding a few lines of Python code to extend the ontology. Some examples from the current ontology are:

- 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;
- player-controlled movements, including grid-navigation, artillery-angle control, jumping, shooting (= spawning a projectile), continuous flight control;
- nondeterministic chasing and fleeing behaviors;
- projectile objects, spawned at the location of other objects, triggered by schedules, user actions, or collisions;
- stickiness, i.e., one object pulling another one;
- bouncing and wraparound behavior, from other objects or the edge of the screen;
- teleportation of objects, to fixed or random end locations;
- continuous physics like inertia, friction, and gravity;
- stochastic effects like slipping or wind gusts.

Some of these effects can also be executed conditionally on local state, depending on relative velocity or resources in possession (see below).

Currently under development are smarter pathfinding behaviors, and in the near future the ontology should also include features like momentum-preserving object splits, area-of-effect events, and line-of-sight conditions.

E. Resources

Objects can have a number of properties that determine how they are visualized (color, shape, orientation), and how they are affected by physics (mass, momentum). One additional kind of property is resources, like health, mana, or ammunition, which increase or decrease by interactions with other objects (finding health packs, shooting, taking fall damage). Some interaction effects are conditional on the amount of resources available to the objects (e.g., if health is too low, an object might be killed by a collision, but not otherwise).

Here, resources are always local to an object; they are automatically added to the object that collected resources, and visualized with a progress bar (in the same color as the resource); see Fig. 3.

F. Mix, Match, Extend

Many ontology 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, for example, the game description of *Lunar Lander* in [12]. 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* [13].

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 of lines of Python code. This permits easy prototyping of novel game dynamics, which can immediately be recombined with existing ones. For example, for the game *Super Mario*, it was necessary to distinguish the direction of movement of a collision—only when hitting a Goomba from above is it killed by Mario. So we added the following code to the ontology:

```
def killIfFromAbove(s, p, game):
    """Kills the sprite, only if the other
       one is higher and moving down."""
    if (s.lastrect.top > p.lastrect.top
        and p.rect.top > p.lastrect.top):
        killSprite(s, p, game)
```

where `rect` and `lastrect` denote the current and previous position of an object, and the more general method `killSprite` was defined previously.

G. Limitations

As with all description languages, the design of PyVGDL involves tradeoffs which favor some classes of games over others. The two obvious limitations are the restriction to 2-D games, and the reliance on local interactions as discussed above. Some typical video game features, such as an inventory filled with various types of items, would be unnatural to implement in the current framework because it would

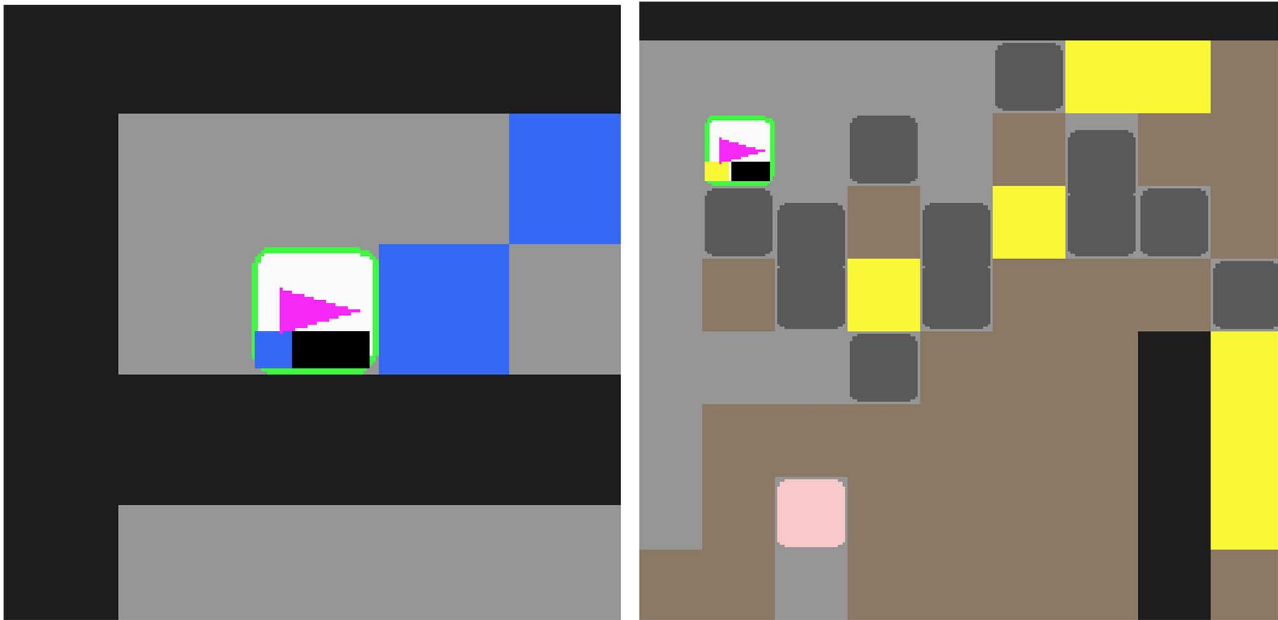

Fig. 3. Resource bars are visible on the avatar. (Left) The ammunition/mana of a first-person shooter, with additional mana blocks (blue) available for pickup. (Right) The Boulder Dash agent has accumulated some gold, but not yet enough to exit the level, as the yellow progress bar indicates.

involve a combinatorial number of avatar types. It is also lacking the logical rules and composability available in classical languages such as GDL, which impoverishes the expressibility to more superficial types of interactions. For example, it is not currently possible to distinguish legal and illegal moves and reason about them, except by making illegal moves have no effect (such as when Link tries to move into a wall object). Also, as a procedural language, it lacks the advantages of more declarative approaches, making it difficult to reason about PyVGDL games.

III. Example Games

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, for example:

- *Space Invaders*, with shooting, complex movement sequences, timed spawning points;
- *Frogger*, with sticky objects, a simplistic log resource that protects from drowning, and wraparound movement;
- *Lunar Lander*, with gravity and inertial effects;
- *Physical Traveling Salesman Problem* (PTSP [14]), with continuous control of wall-bouncing spaceship;
- *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 teleportation mechanisms;
- *Legends of Zelda*, with a unique directional attack, keys, and locked doors; see Figs. 1 and 2;
- *Super Mario*, including elevator platforms, Goombas, and Koopa Paratroopas, and direction-sensitive collisions;
- *Pong*, showcasing a simple two-player game;
- *Tank Wars*, a two-player game with aimed artillery pieces;
- *Zombie Apocalypse*, with infinitely many opponents and a replenishable health resource;
- *First-Person Shooter*, with ammunition-limited firing of missiles, and ammo packs to be found in the level;
- *Boulder Dash*, where the exit can be used only under the resource condition that sufficient gold has been collected.

The first three of these were the motivating examples in [2], and a few of the others are depicted in Fig. 4. 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 Fig. 2 are typical in that respect, and indeed none of the games mentioned require game descriptions of more than 40 lines.

IV. 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.

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.

A. Visualization

Games are visualized on screen either in bird's-eye or first-person view, but it is possible to disable the visualization, which results in dramatic speed gains (e.g., for planning or learning). Furthermore, tools are included for recording the actions taken during a game and replaying them, for creating animated GIF videos from such replayed action sequences, and even to automatically uploading game videos to YouTube.

Our objective was for PyVGDL to remain lightweight, fast, and agile, so very little emphasis has been placed 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 interfacing to a more advanced rendering engine should be easy.

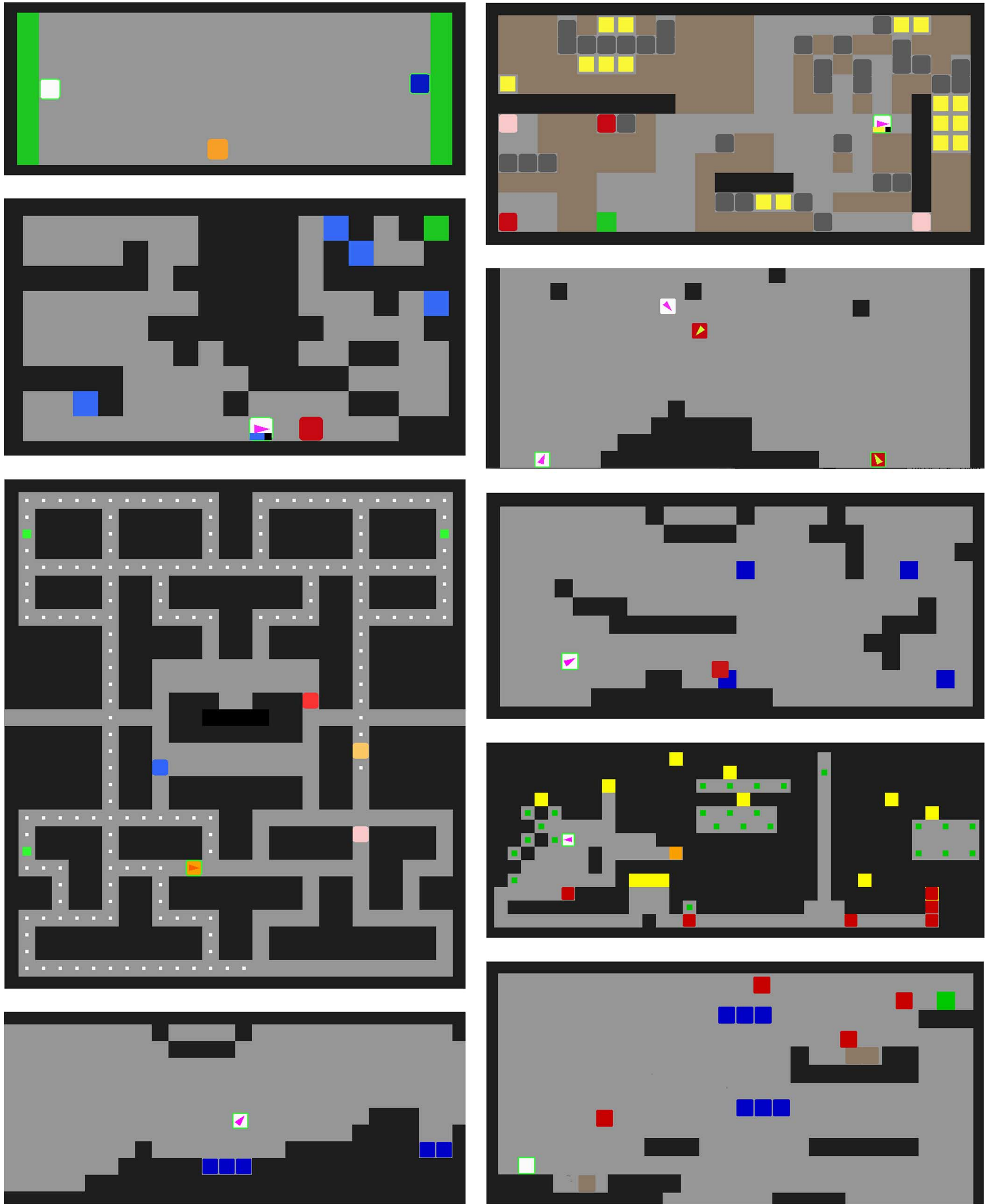

Fig. 4. Renderings (bird's-eye view) of a few example games. Clockwise from the top left: *Pong*, *Boulder-Dash*, *Tank Wars*, *Physical TSP*, *Dig-Dug*, *Super Mario*, *Lunar Lander*, *Pac-Man*, and a shooter game.

B. Interfaces

Games can be interfaced in a number of different ways.

- Player type: We currently support direct interactive play with human players (via 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 [15].

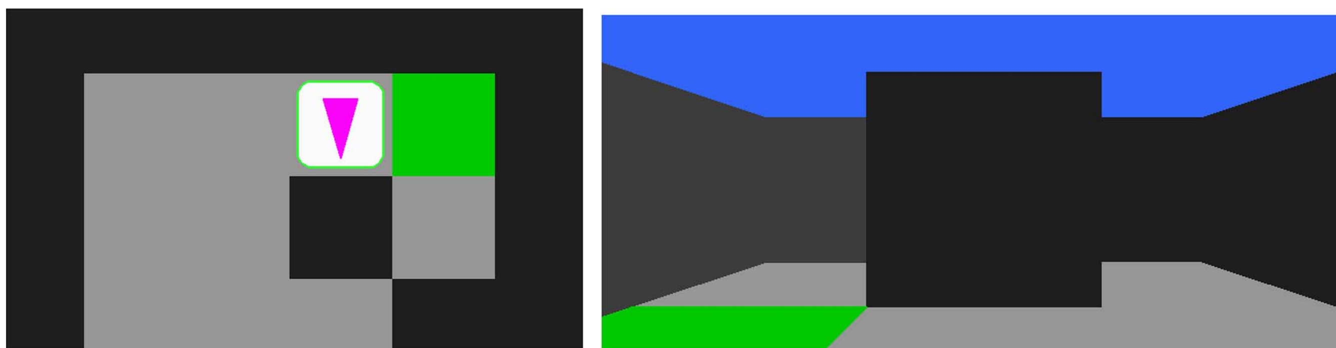

Fig. 5. For the same game state, we show the rendered game from (left) the bird's-eye perspective, and (right) the first-person perspective. Note how in the subjective view, impenetrable objects like walls are shown as blocks, while other objects (the green one in the corner) are drawn on the level floor.

- Number of players: The library currently supports one or two players.
- Perspective: By default, a game is played from the bird's-eye perspective (objective), with the full rectangular 2-D 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 Fig. 5.
- Observation: Orthogonally to the perspective of observation, we also provide to different types of encoding for observations provided to bots: they are available rendered visually as a medium-resolution image, or in "clean" form, representing only the functionally different components.
- 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 rollout. 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 (MDP).

C. Learning

Producing agents that exhibit competent behavior in a game environment is a very general problem, and not surprisingly, drastically different methodologies exist, but PyVGDL aims to be agnostic with respect to how its games are used in that context. The companion paper [12] showed how the PyVGDL framework can be used to learn competent behaviors in a broad range of scenarios: 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 (including dynamic programming, value-based reinforcement learning, and evolution strategies). All the scripts used to produce the results of that paper are distributed with the PyVGDL code as use-case examples. Since then, the framework has been extended to include a more complete, fine-grained score system, with intermediate rewards that should help to speed up learning, at least for reinforcement learning (RL) approaches.

V. Future: Video Game Olympiad

Working backward from the long-term goal of using games to foster artificial general intelligence [1], a recent proposal has argued for extending the framework of general game playing to video games [11], and to establish an AI competition where agents must demonstrate their proficiency on a wide range of video games. To make this feasible, the authors proposed to limit the domain to arcade-style games in 2-D, which form a sufficiently diverse space, given that they kept a generation of human gamers interested.

Directly in this vein, we propose to use PyVGDL as the basis for setting up a Video Game Olympiad, because it is explicitly designed to facilitate using a very broad range of games with a unified interface, which eventually capture the majority of mechanisms found in arcade games. The games on which agents are evaluated can further be split into some that are seen during training/agent design, and another set of variants, or completely unseen ones that are used for the final evaluation, akin to the Polyathlon in the RL competition [20], [21]. It stands to reason that building algorithms that are competitive under such circumstances will improve our understanding of general-purpose and transfer learning. A first edition of such a competition based on PyVGDL was held at the 2014 IEEE Conference on Computational Intelligence and Games (CIG 2014, Germany, Dortmund).[3]

VI. Conclusion

We proposed "PyVGDL," a powerful new tool for conducting research on computational intelligence and games. To conclude, we revisit the six original objectives set out in [2].

- Our language is human readable and high level, enough for non-programmers to design new games.
- It is unambiguous and instantly parsable into playable games.
- The concise structure lends itself to game-evolutionionary approaches, in particular to crossover operators.
- It is expressive enough to describe a very broad range of arcade-style video games, and concisely so.
- As this paper aims to demonstrate, it is extensible through its flexible ontology of components.
- While a generative approach (not just of game levels, but of full game dynamics) remains to be tried, it is plausible that many random descriptions (with a few weak consistency constraints) lead to viable games.

The library's availability as an open-source package is an invitation for others to use and build upon it. As we have demonstrated, the ontology can easily be extended, but even without further extensions, many interesting new games can be written in PyVGDL and are instantly playable.

Acknowledgment

The author would like to thank the anonymous reviewers for their constructive feedback, the other contributors to the PyVGDL library, namely J. Togelius, S. Samothrakis, D. Perez, and C.-U. Lim; and also the participants of the 2012 Dagstuhl Seminar on Artificial and Computational Intelligence in Games, in particular, M. Ebner, J. Levine,

[3]http://www.gvgai.net

S. Lucas, T. Thompson, and J. Togelius for the discussions that triggered and inspired the development of this work.

REFERENCES

[1] T. Schaul, J. Togelius, and J. Schmidhuber, "Measuring intelligence through games," 2011 [Online]. Available: http://arxiv.org/abs/1109.1314

[2] M. Ebner, J. Levine, S. Lucas, T. Schaul, T. Thompson, and J. Togelius, "Towards a video game description language," Dagstuhl Follow-up, 2013 [Online]. Available: http://www.idsia.ch/~tom/publications/dagstuhl-vgdl.pdf

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

[4] A. M. Smith, M. J. Nelson, and M. Mateas, "Ludocore: A logical game engine for modeling videogames," in *Proc. IEEE Symp. Comput. Intell. Games*, 2010, pp. 91–98.

[5] C. Browne, "Evolutionary game design," *IEEE Trans. Comput. Intell. AI Games*, vol. 2, no. 1, pp. 11–21, Mar. 2011.

[6] G. Nelson, *The Inform Designer's Manual*. Oxford, U.K.: Placet Solutions, 2001.

[7] G. Maggiore *et al.*, "A formal specification for Casanova, a language for computer games," in *Proc. 4th ACM SIGCHI Symp. Engineering Interactive Comput. Syst.*, 2012, pp. 287–292.

[8] W. McGugan, *Beginning Game Development With Python and Pygame: From Novice to Professional*. New York, NY, USA: Apress, 2007.

[9] M. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," 2012 [Online]. Available: http://arxiv.org/abs/1207.4708

[10] J. Togelius and J. Schmidhuber, "An experiment in automatic game design," in *Proc. IEEE Symp. Comput. Intell. Games*, 2008, pp. 111–118.

[11] C. B. Congdon *et al.*, "General video game playing," Dagstuhl Follow-up, 2013.

[12] T. Schaul, "A video game description language for model-based or interactive learning," in *Proc. IEEE Conf. Comput. Intell. Games*, 2013, DOI: 10.1109/CIG.2013.6633610.

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

[14] D. Perez, P. Rohlfshagen, and S. M. Lucas, "The physical travelling salesman problem: WCCI 2012 competition," in *Proc. IEEE Congr. Evol. Comput.*, 2012, DOI: 10.1109/CEC.2012.6256440.

[15] T. Schaul *et al.*, "PyBrain," *J. Mach. Learn. Res.*, vol. 11, pp. 743–746, 2010.

[16] C. B. Browne *et al.*, "A survey of Monte Carlo tree search methods," *IEEE Trans. Comput. Intell. AI Games*, vol. 4, no. 1, pp. 1–43, Mar. 2012.

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

[18] J. Togelius *et al.*, "Ontogenetic and phylogenetic reinforcement learning," *Zeitschrift Künstliche Intelligenz*, vol. 3, Special Issue on Reinforcement Learning, pp. 30–33, 2009.

[19] L. Cardamone, D. Loiacono, and P. L. Lanzi, "Evolving competitive car controllers for racing games with neuroevolution," in *Proc. 11th Annu. Conf. Genetic Evol. Comput.*, 2009, pp. 1179–1186.

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

[21] B. Tanner and A. White, "Rl-glue: Language-independent software for reinforcement-learning experiments," *J. Mach. Learn. Res.*, vol. 10, pp. 2133–2136, 2009.