Chapter 2 - VGDL and the GVGAI Framework

Diego Perez-Liebana

1 Introduction

The present chapter describes the core of the framework: the Video Game Description Language (VGDL) and the General Video Game AI (GVGAI) benchmark. All games in GVGAI are expressed via VGDL, a text based description language initially developed by Tom Schaul [6]. Having the description of games and levels independent from the engine that runs them brings some advantages. One of them is the fact that it is possible to design a language fit for purpose, which can be concise and expressive enough in which many games can be developed in. The vivid example of this is the 200 games created in this language since its inception. Another interesting advantage of this separation is that it is not necessary to compile the framework again for every new game is created, which helps the sustainability of the software. The engine can be kept in a compiled binary form (or a .jar file) that simply takes games and levels as input data.

But arguably one of the main benefits of having VGDL is the possibility of easily customize games and levels. Its simplicity allows researchers and game designers to create variants of present games within minutes. Put in comparison with other game research frameworks, it is much easier to tweak a VGDL game than modifying an Arcade Learning Environment (ALE) or Unity-ML one - in fact in some cases this may not even be possible. This feature is more than a simple convenience: it opens different research avenues. For instance, it facilitates the creation of a portfolio of games to avoid overfitting in learning agents [4], as well as enabling research on automatic game tuning and balancing (see Chapter 7) and game and level generation (Chapter 6).

Last but not least, the versatility of VGDL eases the organisation of a competition that provides new challenges on each one of its editions. Competitions have been part of the game AI community for many years, normally focusing on a single game at a time. As mentioned in the previous chapter, using more games at once allows research to focus more on the generality of the methods and less in the specifics of each game. However, this is not easy to achieve without the capacity of generating games of a medium complexity in a relatively fast and easy way.

Section 2 of this chapter describes the Video Game Description Language in detail, with special emphasis on its syntax and structure, also providing illustrative examples for single-player and two-player games, using both grid and continuous physics. Section 3 describes the framework that interprets VGDL and exposes an API for different types of agents to play the available games. This is followed, in Section 4, by a description of the competition and the different editions run over the years.

2 The Video Game Description Language

VGDL is a language built to design 2-dimensional arcade-type games. The language is designed for the creation of games around objects (*sprites*), one of which represents the player (*avatar*). These objects have certain properties and behaviours and are physically located in a 2D rectangular space. The sprites are able to interact among each other in pairs and the outcomes of these interactions form the dynamics and termination conditions of the game.

VGDL separates the logic into two different components: the game and the level description, both defined in plain text files. All VGDL game description files have four blocks of instructions.

- Sprite Set: This set defines the sprites that are available in the game. Each object or sprite is defined with a class and a set of properties. The are organised in a tree, so a child sprite inherits the properties of its ancestors. All types and properties are defined within an ontology, so each sprite corresponds to a class defined in a game engine (in this book, this engine is GVGAI).
- Interaction Set: These instructions define the events that happen when two of the defined sprites collide with each other. As in the previous case, the possible effects are defined in an ontology and can take parameters. Interactions are triggered in the order they appear in this definition.
- Termination Set: This set defines the conditions for the game to end. These conditions are checked in the order specified in this set.
- Level Mapping: This set establishes a relationship between the characters in the level file and the sprites defined in the Sprite Set.

The code in Listing 1 shows the complete description of one of the games in the framework: *Aliens*. This game is a version of the traditional arcade game *Space Invaders*, in which the player (situated in the bottom part of the screen) must destroy all aliens that come from above. In the following we analyse the code that implements this game.

```
1  BasicGame square_size=32
2   SpriteSet
3    background > Immovable img=oryx/space1 hidden=True
4    base     > Immovable img=oryx/planet
5    avatar   > FlakAvatar stype=sam img=oryx/spaceship1
6    missile > Missile
7      sam  > orientation=UP singleton=True img=oryx/bullet1
8      bomb > orientation=DOWN speed=0.5 img=oryx/bullet2
9    alien    > Bomber stype=bomb prob=0.01 cooldown=3 speed=0.8
10     alienGreen > img=oryx/alien3
11     alienBlue > img=oryx/alien1
12   portal  > invisible=True hidden=True
13     portalSlow  > SpawnPoint stype=alienBlue cooldown=16 total=20
14     portalFast  > SpawnPoint stype=alienGreen cooldown=12 total=20
15
16  InteractionSet
17   avatar  EOS  > stepBack
18   alien   EOS  > turnAround
19   missile EOS  > killSprite
20
21   base bomb > killBoth
22   base sam > killBoth scoreChange=1
23
24   base   alien > killSprite
25   avatar alien bomb > killSprite scoreChange=-1
26   alien  sam   > killSprite scoreChange=2
27
28  TerminationSet
29   SpriteCounter      stype=avatar               limit=0 win=False
30   MultiSpriteCounter stype1=portal stype2=alien limit=0 win=True
31
32  LevelMapping
33   . > background
34   0 > background base
35   1 > background portalSlow
36   2 > background portalFast
37   A > background avatar
```

Listing 1: VGDL Definition of the game *Aliens*, inspired by the classic game *Space Invaders*

In the first line, the keyword `BasicGame` marks the start of the definition of a standard VGDL game (chapter 7 will study other types of games). In this case, it specifies a parameter, `square_size`, that determines the size in pixels of each cell in the game board. This is purely an aesthetic aspect.

Lines 2 to 14 define the **Sprite Set**. There are six different types of sprites defined for *Aliens* (*background*, *base*, *avatar*, *missile*, *alien* and *portal*), some of them with sub-types. The class that corresponds to each sprite is the keyword that starts with an upper case letter on each type or sub-type. For instance, the *avatar* sprite is of

type `MovingAvatar` while the two types of portals (*portalSlow* and *portalFast*) will be instances of `SpawnPoint`. As can be seen, the definition of the tpye can be either in the parent of a hierarchy (as in *alien*, line 9) or in the children (as in *portal*, lines 13 and 14).

Some parameters that these sprites receive are common to all sprites. Examples in *Aliens* are `img`, which describes a file that will graphically represent the sprite in the game, or `Singleton`, which will indicate to the engine that only one instance of this sprite can be present in the game at all times. In this game, this sprite is `sam` (the bullet that the player shoots): in the original game Space Invaders, the player can only shoot one bullet at a time. Another interesting parameter is `hidden`, which indicates to the engine that the sprite must not be included in the observations the agent receives (see Section 3). This is used throughout the games in the GVGAI framework to model partial observability in games. A different parameter, `invisible`, only hides the sprite in the visual representation of the game, but it is still accessible in the game observations.

Many parameters, however, are dependent on the type of sprite. For instance, the avatar for *Aliens* is set up as a `FlakAvatar`, which the ontology defines as an avatar type (i.e. controlled by the player) that can move only LEFT and RIGHT and spawn an object, when playing the action USE, of the type defined in the parameter `stype`. In this case, `stype` has the sprite *sam* as value, which is defined below in the code.

sam has *missile* as a parent sprite. *missile* is of type `Missile`, which is a type of sprite that moves in a straight direction ad infinitum. In *Aliens*, there are two types of missiles: *sams* (shot by the player) and *bombs* (dropped by the aliens), which have different parametersations of the type `Missile`. One of these parameters is `orientation`, which determines the direction in which the sprite travels. Another one, `speed`, indicates how many cells per frame the sprite moves. As can be seen in lines 7 and 8, these missiles have different orientations and speeds (if the parameter speed is not defined, it takes the default value of 1.0).

The enemies the player must destroy are described in lines 9 to 11. *alien* is the parent sprite of a hierarchy that defines them as being of type `Bomber`. Bombers are sprites that move in a certain direction while spawning other sprites. As in the case of the missiles, their movement is defined by a `speed` and an `orientation` (which by default is set to RIGHT). Besides, the parameter `cooldown` specifies the rate at which the sprites execute an action (in the example, aliens move or shoot once every 3 frames). The parameter `stype` describes the sprite type that is spawned with a probability (per action) `prob` as defined in line 9.

Aliens are spawned by portals (lines 13 to 14), which can be seen as immovable bombers. They can define the maximum number of sprites to spawn, after which the

spawner is destroyed, using the parameter `total`. In this case, probability `prob` is set by default to 1.0, so the rate at which aliens are spawned is set only by the `cooldown` parameter.

The Sprite Set is arguably the most complex and richest set in VGDL, but defining complex sprites without the rules that describe the interactions between them does not constitute a game. The **Interaction Set**, lines 16 to 26, describes this. Each line in the interaction set describes an effect that takes place when two sprites collide with each other. In order to keep the description of the game short, it is possible to group pairs of interactions in instructions with two or more sprites (see line 25). An instruction of the type $A_1A_2A_3..A_N > E$ is equivalent to N effects $A_1A_2 > E$, $A_1A_3 > E..A_1A_N > E$, etc.

The first three interactions (lines 17 to 19) describe the effects to apply when sprites leave the game screen (keyword `EOS` - End of Screen). When an avatar reaches the end of the screen, the game will automatically execute the event `stepBack`, which sets the position of the sprite to its previous location (hence preventing it to leave the screen). When aliens reach the end, however, the event executed is `turnAround`, which moves the sprite one cell down and flips its orientation. This makes the enemies descend in the level while keeping their lateral movement. Finally, when missiles reach the `EOS`, they are destroyed. Note this interaction affects both subtypes of missiles defined in the Sprite Set (*sam* and *bomb*) and it is crucial for the avatar to be able to shoot again (as *sam* sprites are `Singleton`).

The rest of the interactions in this game regulate when a sprite should be destroyed. bases (structural defences) are destroyed when colliding with a *bomb*, a *sam* and an *alien*. In the first two cases, the other sprite is also destroyed (via the effect `KillBoth`). The *avatar* is destroyed when colliding with an *alien* or a *bomb* (the enemies' missile). Finally, aliens are killed when hit by a *sam* (player's missile).

An important duty of the interaction set is to define the score system, which in turn can be used as reward for learning algorithms. Each interaction can define a parameter `scoreChange` which establishes how many points are awarded every time the interaction is triggered. Therefore, in *Aliens*, +1 point is given when the player destroys a *base* (line 22), +2 points when destroying an *alien* (line 26) and a point is lost when the avatar is destroyed (line 25).

The **Termination Set** (lines 28 to 30) describes the ways in which the game can end. All instructions must define the parameter `win`, which determines if the game is won or lost from the perspective of the player. The first condition (line 29) is of type `SpriteCounter`, which checks that the count of one particular type of sprites reaches the `limit`. In this case, this instruction determines that, when the player is killed, the game ends in a loss. The second condition (line 30), a `MultiSpriteCounter`,

requires that both types of sprites (`stype1` and `stype2`) have no instances in the game for the game to end. In this case, when there are no aliens in the game and the portal is gone (i.e. it has spawned all aliens already), the game is finished with a win for the player.

Finally, the **LevelMapping** set describes the sprites that correspond to the characters defined in the level file. All VGDL games can be played in many levels, which set the initial configuration of the sprites when the game starts. Figure 1 shows two examples of levels for *Aliens* available in the framework.

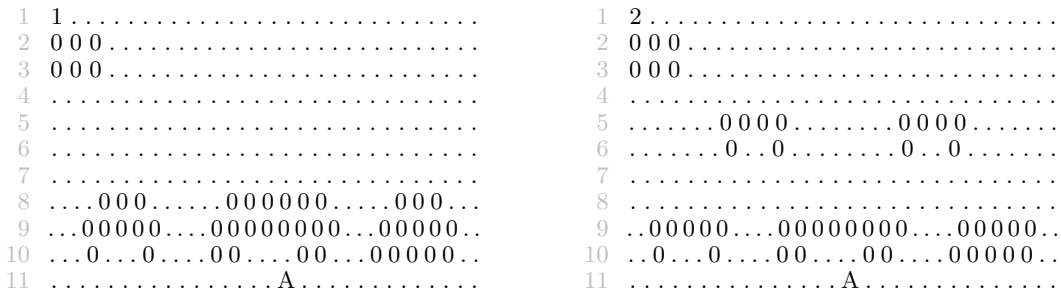

Fig. 1: Two VGDL levels for the game *Aliens*

Each character in the level definition and mapping can define one or more sprites that will be created at that position. In the case of *Aliens*, all characters define a common sprite: *background*. This sprite, defined in the Sprite Set, has the only purpose of being drawn behind all sprites (so when, for instance, a *base* is destroyed, something is drawn instead). Note also that having two different types of portals allows for the creation of different levels with different difficulties. *portalFast* (character "2") spawns aliens at a higher rate (making the game more difficult) than *portalSlow* (resp. "1"). The two levels in Figure 1 use different portals (easier case for the level on the left, harder for the one on the right). Figure 2 shows a screenshot of the *Aliens* game at an intermediate stage.

VGDL is a rich language that allows the definition of multiple 2-dimensional games in which sprites can move actively (through behaviors specified in the ontology or the actions of the player), passively (subject to physics), and collide with other objects that trigger effects. These interactions are local between two colliding objects, which can lead to some sprites disappearing, spawning, transforming into different

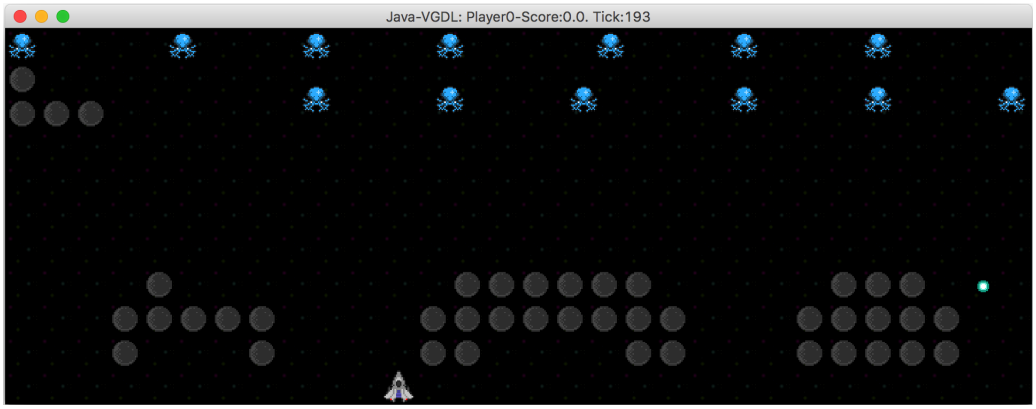

Fig. 2: Screenshot of the game *Aliens* in GVGAI.

types or changing their properties. Non-local interactions can also be achieved, as teleportation events or effects that affect multiple sprites (all, or by type) are possible.

The VGDL underlying ontology permits descriptions to be concise. This ontology defines high-level building blocks for games, such as physics, movement dynamics and interactions. Given these tools, VGDL is able to describe a wide variety of arcade games, including versions of classic arcades such as *Space Invaders* (described here), *Sokoban*, *Seaquest*, *Pong*, *Pac-Man*, *Lunar Lander* or *Arkanoid*.

Some of these games, however, could not be defined in the original VGDL for GVGAI. The following two sections describe the introduction of real-world physics (Section 2.1), to include effects like gravity and inertia, and the ability to define 2-player games (Section 2.2).

2.1 Continuous Physics

The type of games that the original VGDL could describe for GVGAI is limited to arcade games with discrete physics. All games before 2017 were based on a grid structure that discretises the state of the game and the available actions. For instance, the orientation of the sprites described above could be either `Nil`, `Up`, `Down`, `Left` or `Right`, without providing a finer precision. While this is enough to create games like *Aliens*, it falls short for games such as Lunar Lander, which requires finer precision for orientation and movement.

Perez-Liebana et al. [5] introduced in 2017 an update in VGDL that enhanced the ontology to widen the range of games that can be built. The main addition to the framework was the ability of establishing a *continuous* physics setting that would allow sprites to move in a more fine-tuned manner. Sprites can now be asked to

adhere to the new physics type by setting a new parameter `physicstype` to `CONT`. Sprites that align to these physics acquire three extra parameters that can be defined: `mass`, `friction` and `gravity`. Although it is normal for gravity to be set equal to all sprites, it can also be set separately for them.

Movement of sprites is still active, like in the case of the traditional grid physics, but continuous physics do not limit movement to four directions. Instead, orientation can be set to 360°. Furthermore, speed is not defined proportional to cell but to screen pixels, allowing the sprite to move in any possible direction and with and speed.

Sprites are also subject to a passive movement, which occurs at every game frame. This passive movement alters the direction of the sprite, either by friction or by gravity. If the sprite is affected by friction, its speed will be multiplied by $1 - f(s)$ (where $f(s)$ is the sprite's friction). In case it responds to gravity, a vector $\overrightarrow{g}$ that points downwards and has a magnitude equal to the sprite's mass multiplied by the sprite's gravity is added to the sprite's current velocity vector.

Figure 3 shows an example of a game built with continuous physics, *Arkanoid*. In this game, the player controls the bat in the bottom of the screen. It must bounce a ball to destroy all the bricks in the level in order to win. The player loses a life when the ball falls through the bottom end, losing the game when all lives have been lost.

Listing 2 shows a simplified version of the Sprite Set for this game. The avatar sprite, of type `FlakAvatar` (line 5, defines the types of physics to be continuous. `friction` and `inertia` are set to 0.2, which establish the dynamics of the bat movement. Other interesting parameters are the ones that define the number of lives the player has (`healthPoints` and `limitHealthPoints`) and the width of the bat (`wMult`).

The *ball* (of type `Missile`) also adheres to this type of physics. For this game, no gravity is needed (the ball in *Arkanoid* does not really obeys the law of physics), but `speed` is defined at 20 speeds per second.

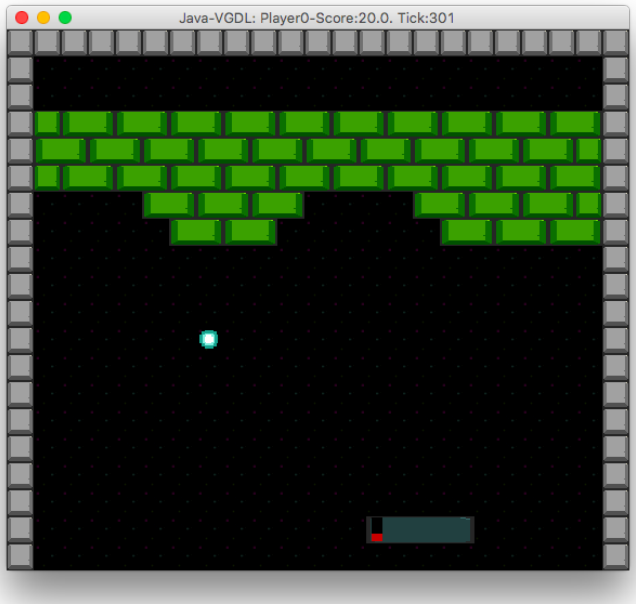

Fig. 3: Screenshot of the game *Arkanoid* in GVGAI.

```
1     SpriteSet
2         background > Immovable img=oryx/space1 hidden=True
3
4         avatar > FlakAvatar stype=ball physicstype=CONT wMult=4 friction=0.2
5             mass=0.2 img=oryx/floor3 healthPoints=3 limitHealthPoints=3
6
7         ball > Missile orientation=UP speed=20 physicstype=CONT
8         ballLost > Passive invisible=True hidden=True
9
10        brick > Passive img=newset/blockG wMult=2
11        block > Passive img=newset/block2 wMult=2
```

Listing 2: VGDL Sprite Set block of the game *Arkanoid*.

2.2 2-Player Games

The original VGDL only allowed for single-player games to be created. Gaina at al. [2] and [1] introduced the possibility of defining 2-player simultaneous games in this language.

One of the first modifications to the language was the addition of a new parameter (`no_players`) to specify the number of players that the game is for. This is added to the first line of the game description, as Listing 3 shows.

```
BasicGame square_size=50 no_players=2
```

Listing 3: Specifying the number of players in VGDL.

Additionally, the score interactions must specify the number of points each player receives when an event is triggered and the termination instructions have to determine he end state (win or lose) for each player. In both cases, this is separated by a comma, as Listing 4 shows.

```
avatarA npc > killSprite scoreChange=-1,0
avatarB npc > killSprite scoreChange=0,-1

SpriteCounter stype=avatarA limit=0 win=False,True
SpriteCounter stype=avatarB limit=0 win=True,False
```

Listing 4: Score and termination instructions for a 2-player VGDL game.

An important aspect to note is that games can be cooperative or competitive, which is inferred from the interaction and termination sets that describe the game. Figure 4 shows two examples of 2-player games in GVGAI. The one on the left, *Minesweeper*, in which two players compete for finding their own bombs in boxes scattered across the level. In the one of the right, *Sokoban*, two players cooperate to push all crates into target holes.

2.3 The future of VGDL

By July 2019, 140 single-player and 60 two-player VGDL games have been created, with 5 different levels per game. As can be seen in the rest of this book, VGDL has been a crucial part of GVGAI, as a rapid language to prototype and design new challenges. The integration of continuous physics and two-player capabilities have widened the variety of possible games that can be created.

However, VGDL is still limited. Despite its expressiveness, it is hard to make VGDL games that are truly fun for humans to play. Furthermore, creating complex games is also limited in the language, which in turn puts a limit in the type of challenges that can be posed for intelligent agents. Apart from interfacing GVGAI with non-VGDL games (see Chapter 8), further work in VGDL is envisioned.

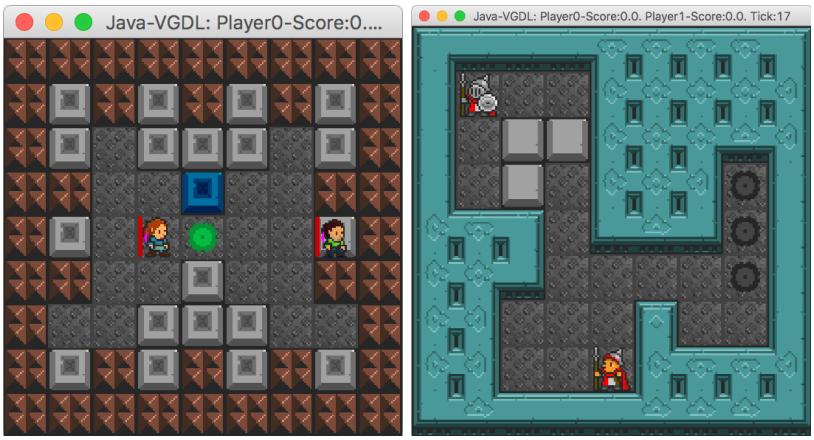

Fig. 4: Examples of two 2-player VGDL games: *Minesweeper* (left) and *Sokoban* (right)

One of the potential changes to VGDL is to adapt it for games in which the player doesn't necessarily need to be in control of an avatar (which is a requirement in the current version). For instance, games like Tetris or Candy Crush can't be created in the current version of VGDL without an important number of instructions and hacking. Another logical next step is to move towards the possibility of creating 3D games, as well as moving from 2-player to N-player challenges.

As will be seen later in chapters 6 and 7, VGDL is used for content generation and game design. VGDL was not originally designed for these kind of tasks, and as such it is not trivial to adapt it for them. Advances towards a more modular language can ease the automatic design of rules, levels and games and facilitate the creation of new interesting games and challenges.

3 The General Video Game AI Framework

The original implementation of VGDL was parsed in `py-vgdl`, a Python interpreter written by Schaul [7] that permitted agents to play the original VGDL games. In order to provide a fast Forward Model for planning agents, py-vgdl was ported to Java for the first GVGAI framework and competition. This framework is able to load games and levels defined in VGDL and expose an API for bots with access to a forward model. Figure 5 shows a scheme of the GVGAI framework at work.

The Java-vgdl engine parses the game and level description to creates two components: a game object and a controller, which can be a interfaced by a bot or a

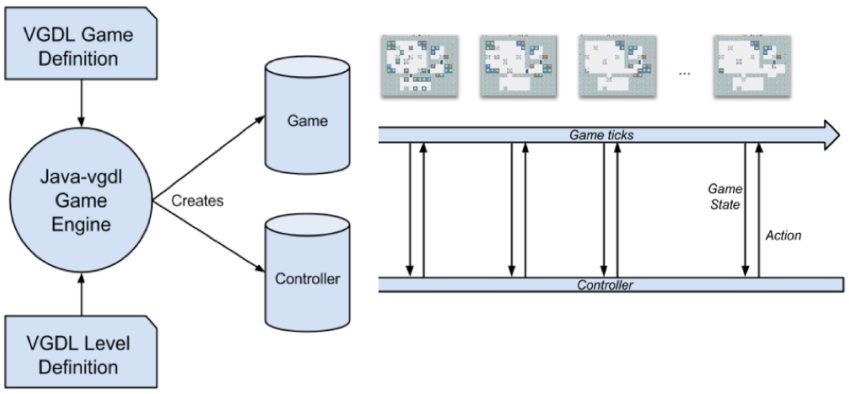

Fig. 5: The GVGAI framework

human player. Then, the game starts and request an action from the controller at each game frame. The controller must return an action to execute in the real game in no more than $40ms$, or penalties apply. If the action is returned in between 40 and 50 ms, a null action (NIL) will be applied. If the action is obtained after more than $50ms$, the controller is disqualified. This rule is set up in order to preserve the real-time nature of the games played.

GVGAI bots must implement a class according to a given API. In single-player games, this class must inherit from the abstract class `AbstractPlayer` (`AbstractMultiPlayer` for two-player games). The API for single-player games controllers is included in Listing 5.

```
1    public <ClassName>(StateObservation, ElapsedCpuTimer);
2
3    Types.ACTIONS act(StateObservation, ElapsedCpuTimer);
4
5    void result(StateObservation, ElapsedCpuTimer);
```

Listing 5: API for single-player games.

The function `act` is called from the game at every game frame and must return a `Types.ACTIONS` value at every $40ms$. Although each game defines a different set of available actions, all games choose a subset of all possible moves: RIGHT, LEFT, UP, DOWN, USE and NIL. The first four are movement actions, USE is employed as a wild card for non-movement actions that the avatar type may define (for instance, shooting) and NIL is used to return a void action. The second function, `result`, is optional and is called only when the game is over.

This function `act`, however, *must* be implemented in the bot. Both functions receive two parameters: a `StateObservation` and a timer. The timer can be queried to control the time left until the action needs to be returned. The state observation provides information about the state:

- Current time step, score, victory state (*win*, *loss* or *ongoing*), and health points of the player.
- The list of available actions for the player.
- A list of observations. Each one of the sprites of the game is assign a unique integer type and a category attending to its behaviour (static, non-static, non-player characters, collectibles and portals). The observation of a sprite, unless it's a hidden one, contains information about its category, type id and position. These observations are provided in a list, grouped by category (first) and sprite type (second).
- An observation grid with all visible observations in a grid format. Here, each observation is located in a two-dimensional array that corresponds to the positions of the sprites in the level.
- History of avatar events. An *avatar event* is an interaction between the avatar (or a sprite produced by the avatar) and any other sprite in the game. The history of avatar events is provided in a sorted list, ordered by game step, and each instance provides information about the place where the interaction happened, the type and the category of each event.

One of the most important functionalities that the `StateObservation` provides is the Forward Model (FM). The function `advance` from `StateObservation` receives one action and rolls the state of the game forward one step applying such action. The next state will be one of the possible next states that the agent will find if that action were applied in the real game. The games are stochastic in nature, so unless the game is fully deterministic, the next state will be sampled from the possible next states that could exist. Additionally, `StateObservation` provides the function `copy`, which allows to make an exact copy of the current state. These two functions are essential to be able to run statistical forward planning (SFP) algorithms like Monte Carlo Tree Search (MCTS) or Rolling Horizon Evolutionary Algorithms (RHEA).

Apart from the function `act`, the controller must also implement a constructor that receives the same parameters. This constructor builds the agent and is called just before the game starts. This function must finish in $1s$ and can be used to analyse the game and initialise the agent. It's important to notice that the agent never has access to the VGDL description of the game - it must guess the rules, dynamics and ways to win using the FM.

3.1 API for 2-player games

Bots created for the 2-player games need to adhere to a different API. These are the main differences.

- The class that implements the agent must inherit from the base class `AbstractMultiPlayer`.
- The same functions as in the single-player case are available, although in this case the state observation is received as a `StateObservationMulti` object.
- Each player receives a player ID in the form of an integer, which is supplied as a third parameter in the constructor. This player ID can be 0 or 1 and it is assigned by the game engine. Listing 6 shows the API for a 2-player games agent.

```
1    public <ClassName>(StateObservationMulti, ElapsedCpuTimer, int);
2
3    Types.ACTIONS act(StateObservationMulti, ElapsedCpuTimer);
4
5    void result(StateObservationMulti, ElapsedCpuTimer);
```

Listing 6: API for single-player games.

- Score, victory status, health points and list of available actions are still available through the `StateObservationMulti` object. However, the respective methods now receive a parameter: the player ID from the player whose information is requested. Note that the agents may have a different set of actions available.
- Access to a Forward Model is available as in the single-player case, but the function advance now receives an array of actions for all players in the game. The index in this array corresponds to the player ID and the actions are executed simultaneously to roll the state forward.

Agents in 2-player games may have different objectives and ways to win the game. The API does not provide information about the cooperative or competitive nature of the game, leaving the discovery of this to the agents that play the game.

3.2 GVGAI as a multi-track challenge

The original use of the GVGAI framework was for playing single-player **planning**, later expanded to also cover 2-player games. We refer to this setting (or track) as *planning* because the presence of the Forward Model (FM) permits the implementation of statistical forward planning methods such as Monte Carlo Tree Search (MCTS) and Rolling Horizon Evolutionary Algorithms (RHEA). The agents do not require previous training in the games, but the FM allows the bots to analyse search trees

and sequences of actions with the correct model of the game. Chapter 3 shows simple and advanced methods that can be used for the planning challenge in GVGAI.

A different task is to learn to play games episodically without access to an FM. The **learning** track was later added to the framework to tackle this type of learning problem. For this task, agents can be also written in Python apart from Java. This was included to accommodate for popular machine learning (ML) libraries developed in Python. With the exception of the FM, the same information about the game state is supplied to the agent, but in this case it is provided in a JavaScript Object Notation (JSON) format. Additionally, the game screen can also be observed, for learning tasks via pixels only. Torrado et al. [8] interfaced the GVGAI framework to the OpenAI Gym environment for a further reach within the ML community. Chapter 5 dives deeper in this setting of GVGAI.

The GVGAI framework has primarily be used for agents that play games, but games as an end in themselves can also be approached. In particular, the framework was enhanced with interfaces for automatic **level** and **rule** generation. These are Procedural Content Generation (PCG) tracks where the objective is to build generators that can generate levels for any game received and (respectively) create rules to form a playable game from any level and sprite set given. In both cases, access to the FM is granted so the generators can use planning agents to evaluate the generated content. Chapter 6 describes these interfaces in more detail.

GVGAI thus offers a challenge for multiple areas of Game AI, each one of them covered by one of the settings described above. The planning tracks offer the possibility of using model-based methods such as MCTS and RHEA and, in the particular case of 2-player games, proposes research in problems like general opponent modelling and interaction with another agent. The learning track promotes the investigation in general model-free techniques and similar approaches such neuro-evolution. Last but not least, the content generation settings put the focus in algorithms typically used for PCG tasks, such as solvers (Satisfiability Problems and Answer Set Programming), search-based methods (Evolutionary Algorithms and Planning), grammar-based approaches, cellular automata, noise and fractals.

4 The GVGAI Competition

Each one of the settings described in the previous sections has been proposed as a competition track for the game AI community. Table 1 summarises all editions of the competition until 2018.

Single-Player Planning				
Contest	**Winner**	**Approach**	**% Victories**	**Entries**
CIG, 2014	adrienctx	Open Loop Expectimax Tree Search (OLETS)	51.2%	14
GECCO 2015	YOLOBOT	MCTS, BFS, Sprite Targeting Heuristic	63.8%	47
CIG 2015	Return42	Genetic Algorithms, Random Walks, A*	35.8%	48
CEEC 2015	YBCriber	Iterative Widening, Danger Avoidance	39.2%	50
GECCO 2016	MaastCTS2	Enhanced MCTS	43.6%	19
CIG 2016	YOLOBOT	(see above)	41.6%	24
GECCO 2017	YOLOBOT	(see above)	42.4%	25
WCCI 2018	asd592	Genetic Algorithms, BFS	39.6%	10
Two-Player Planning				
Contest	**Winner**	**Approach**	**Points**	**Entries**
WCCI 2016	ToVo2	SARSA-UCT with TD-backups	178	8
CIG 2016	adrienctx	OLETS	142	8
CEC 2017	ToVo2	(see above)	161	12
FDG 2018	adrienctx	(see above)	169	14
Level Generation				
Contest	**Winner**	**Approach**	**% Votes**	**Entries**
IJCAI 2016	easablade	Cellular Automata	36%	4
CIG 2017	(suspended)	-	-	1
GECCO 2018	architect	Constructive and Evolutionary Generator	64.74%	6
Rule Generation				
Contest	**Winner**	**Approach**	**% Votes**	**Entries**
CIG 2017	(suspended)	-	-	0
GECCO 2018	(suspended)	-	-	0

Table 1: All editions of the GVGAI competition, including winners, approach, achievements and number of entries submitted (sample agents excluded). All editions ran in a different set of games and some competition winners did not participate in all editions.

Unless it's specified differently, all tracks follow a similar structure: games are organised into sets of 10 (plus 5 levels per game). N sets[1] are made public and available within the framework code[2]. Participants can therefore use those games to train their agents.

One extra set is used for *validation*. Its games are unknown and not provided to the participants, although they can submit their agents to the competition server[3] to be evaluated in them (as well as in the training sets). The server provides rankings

[1] N depends on the competition. The collection of GVGAI games grows by 10 (1 set) for each successive edition.

[2] Competition Framework: `https://github.com/GAIGResearch/GVGAI`

[3] Competition website: `http://www.gvgai.net/`

of the agents based on training and validation sets, keeping the validation games anonymised. Finally, a hidden and secret *test* set is used to evaluate the agents and form the final rankings of the competition. Participants can't access these games nor evaluate their agents on them until submissions are closed. Once the competition is over, the games and levels of the validation set are released and added to the framework, the test set will be used as validation for the next edition of the competition and 10 new games will be created to form the next test set.

Competition rankings, in a set of games, are computed as follows: First, all the entries play each game in all the available levels (once for training and validation sets, 5 in the test set). All games end after 2000 frames with a loss for the player if the natural game end conditions are not triggered before. A game ranking is established by sorting all the entries first by average of victories, using average score and game duration as tie breakers, in this order. All indicators but the latter are meant to be maximized. For the time-steps count, t are counted for games that end in a victory and $2000 - t$ for those ended in a loss; this rewards games being won quickly and losses reached late.

For each game ranking, points are awarded to the agents following an F1 system: from first to tenth position, $25, 18, 15, 12, 10, 8, 6, 4, 2$ and 1 points are given, respectively. From the tenth position down, no points are awarded. The final rankings are computed by adding up all the points achieved in each game of the set. Final ranking ties are broken by counting the number of first positions in the game rankings. If the tie persists, the number of second positions achieved is considered, and so on until the tie is broken.

In the 2-player planning case, the final rankings are computed by playing a round-robin tournament among all submitted agents in the final set, with as many iterations as time allows. In this case, all levels are played twice, swapping positions of the agents to account for possible unbalance in the games. Training and validation sets are run differently, though, as it wouldn't be possible to provide rapid feedback if a whole round-robin needs to be run every time a new submission is received. In this case, a Glicko-2 score system [3] is established for selecting the next two agents that must play next. In these sets, Glicko-2 scores become the first indicator to form the rankings. The test set also provides Glicko-2 scores but only as an extra indicator: final rankings are computed as in the planing case (victories, score and game duration).

The learning track has been run in two different editions (2017 and 2018). For the 2017 case, controllers were run in two phases: learning and validation. In the learning stage, each entry has a limited time available (5 mins) for training in the

first three levels of each game. In the validation step, controllers play 10 times the other two levels of each game.

The framework used in the 2018 competition[4] interfaced with OpenAI Gym and the competition was run with some relaxed constraints. The set used for the competition only had three games and all were made public. Two of their levels are provided for training to the participants, while the other three are secret and used to obtain the final results. Also, each agent can count on $100ms$ for decision time (instead of the traditional $40ms$). For full details on the learning track settings, the reader is referred to Chapter 5.

Finally, the winner of the PCG tracks is decided by human subjects who evaluate the content generated by the entries. In both settings, the judges are presented with pairs of generated content (levels or games) and asked which one (or both, or neither) is liked the most. The winner is selected based on the generator with more votes. For more details on the content generation tracks, please read Chapter 6.

References

1. R. D. Gaina, A. Couëtoux, D. J. Soemers, M. H. Winands, T. Vodopivec, F. Kirchgessner, J. Liu, S. M. Lucas, and D. Perez-Liebana, "The 2016 Two-Player GVGAI Competition," *IEEE Transactions on Computational Intelligence and AI in Games*, 2017.
2. R. D. Gaina, D. Perez-Liebana, and S. M. Lucas, "General Video Game for 2 Players: Framework and Competition," in *IEEE Computer Science and Electronic Engineering Conference*, 2016.
3. M. E. Glickman, "Example of the Glicko-2 system," *Boston University*, 2012.
4. N. Justesen, R. R. Torrado, P. Bontrager, A. Khalifa, J. Togelius, and S. Risi, "Illuminating Generalization in Deep Reinforcement Learning through Procedural Level Generation," *arXiv:1806.10729*, 2018.
5. D. Pérez-Liébana, M. Stephenson, R. D. Gaina, J. Renz, and S. M. Lucas, "Introducing Real World Physics and Macro-Actions to General Video Game AI," in *Conference on Computational Intelligence and Games (CIG)*. IEEE, 2017.
6. T. Schaul, "A Video Game Description Language for Model-based or Interactive Learning," in *IEEE Conference on Computational Intelligence in Games (CIG)*, 2013, pp. 1–8.
7. ——, "A Video Game Description Language for Model-based or Interactive Learning," in *Proceedings of the IEEE Conference on Computational Intelligence in Games*. Niagara Falls: IEEE Press, 2013, pp. 193–200.
8. R. R. Torrado, P. Bontrager, J. Togelius, J. Liu, and D. Perez-Liebana, "Deep Reinforcement Learning in the General Video Game AI framework," in *IEEE Conference on Computational Intelligence and Games (CIG)*, 2018.

[4] GVGAI-Gym Learning track framework: `https://github.com/rubenrtorrado/GVGAI_GYM`