Chapter 8 - GVGAI without VGDL

Simon M. Lucas

1 Introduction

In this chapter, we outline the motivation and main principles behind writing GVGAI compatible games in a high-level language instead of using VGDL. The specific examples described below are all written in Java, but the principles apply equally well to other efficient high-level languages. Java, including its graphics APIs, runs on Windows, macOS and Linux without requiring any additional libraries or plugins. We are currently evaluating the use of the new Kotlin[1] programming language as an even better alternative to Java. Kotlin has many language features that lead to clearer and more concise code while being fully inter-operable with the Java platform (Kotlin code compiles down to Java Virtual Machine - JVM - code). Kotlin also has other advantages, such as strong support for writing Domain Specific Languages [3] (which would support flexible languages for describing particular game variants, or the development of new versions of VGDL). Furthermore, Kotlin can also be transpiled to JavaScript and then run in a web browser, if all the code is pure Kotlin and does not call any Java or native libraries. While it used to be possible to run Java Applets in a web browser, in recent years this has become increasingly difficult with most web browser security managers preventing it.

First the motivation. VGDL is a good choice for expressing relatively simple games where the interactions of the game objects and the effects they trigger can be expressed via condition action rules involving rudimentary calculations. When these conditions hold, games can be authored quickly with concise code that is easy to understand and describes the core elements of the game while avoiding any distractions.

However, the restrictive nature of VGDL makes it relatively hard to implement games with complex and strategically deep game-play. Furthermore, existing VGDL game engines run much more slowly than a carefully implemented game in an efficient high-level language such as Java, C++ or C#. To give an idea of the scale of the difference, using a high-level language instead of VGDL may make the game 100 times faster. The increase in speed is important not only to run the experiments more quickly, but also to enable Statistical Forward Planning (SFP) methods to be

[1] `https://kotlinlang.org/`

run in real-time with larger simulation budgets. This enables the agents to behave in more intelligent and more interesting ways and provide a better estimation of the skill-depth of the game.

Another factor is programmer preference: despite the elegance of VGDL, many programmers would still prefer to author games in a more conventional high-level language.

GVGAI games are designed primarily to test the ability of AI agents to play them, in the case of most of the GVGAI tracks, and to design content for them in the level generation track. In judging the quality of the AI play and the generated content however, it is also important to make them human-playable. This enables us to compare the strength of the AI with human play and also allows players to evaluate the generated games and content. Although recent efforts have extended VGDL in a number of interesting ways, such as adding continuous physics, the language still imposes some hurdles to be overcome both in terms of the game rules and the modes of interaction. For example, by default a spaceship avatar can either move or shoot at any particular time, but a human player would often wish to do both, and currently GVGAI has no support for any point and click input devices such as a mouse or trackpad.

VGDL-based games are designed around a fixed number of levels to test the generalisation of the agents. An even better alternative is to allow random or other procedural level generation methods, which is easier to achieve in a high-level language.

Note that GVGAI aims for really general game AI that can be thoroughly tested on a number of dimensions, and has the potential to go far beyond the Atari 2600 games used in the Atari Learning Environment [1] both in terms of variability of challenge and depth of challenge.

The benefits of using high-level languages for GVGAI are therefore clear. For the game designer or AI researcher, the advantage of making a game GVGAI compatible is also significant: for a little extra effort it enables a great many agents to be tested on or to play-test the game. Following the efforts of [6], GVGAI games are now available in OpenAI Gym [2], something that would also be possible for the example games in this chapter, and future games that conform to the same interfaces.

Finally, each game we implement should really be thought of as a class of games with many variable parameters to control the gameplay. This helps to justify the extra effort involved in implementing a Java game since it may offer a wide range of test cases, with different parameter settings leading to very different gameplay.

2 Implementation Principles

In this section we outline the main considerations when implementing these games. The steps taken below are not difficult, and help lead to well-designed software. The approach is most easily taken when implementing a game from scratch - retrofitting these design principles to an existing implementation is significantly harder.

2.1 Copying the Game State

Among all the GVGAI tracks, the planning tracks have been the most popular, especially the single-player planning track. The main requirements for planning track compatibility are to be able to copy the current game state, and to advance it much faster than real-time. Of these, it is the copyability that requires the greatest programming discipline to achieve. For a well-structured object-oriented design, the game state will often comprise an object graph consisting of many objects of many classes. This does not present any great difficulty, but it does mean that care must be taken to make a deep copy of the game state. Also, static or global variables must be avoided. Circular references are also best avoided since they cause problems for serialising the game state to JavaScript Object Notation (JSON), as JSON does not support circular references.

There are two approaches to copying the game state: either a generic serialiser / deserialiser such as GSON[2] or WOX[3] can be used, although note that GSON requires that the game state object graph must not contain any circular references (which is a limitation of JSON rather than the GSON library *per se*). Note that serialising the game state is also useful in order to save particular states for future reference and also to allow agents to interact with the game via network connections. The generic options allow the copy function to be implemented in a couple of lines of code, but they are not very flexible. They are inherently slower than bespoke copy methods, as they use run-time reflection operators which are slower than regular method calls.

Therefore, it is often preferable to write bespoke game-state copy methods. In addition to providing greater speed they also have the flexibility to only make a shallow copy of large immutable objects if desired. For example, a game may contain a large geographical map which does not change during the course of the game. In such a case making a shallow copy (i.e. only the object reference to the map is copied) is sufficient and faster, and also saves on memory allocation / garbage collection[4].

[2] https://github.com/google/gson
[3] http://woxserializer.sourceforge.net/
[4] The potential value of writing bespoke copy methods is clear when using Java, but may be unnecessary when using Kotlin. Kotlin has special data classes which provide default copy methods, and it also has immutable types.

When implementing a game, there are often many parameters that significantly affect the gameplay. A particular combination of parameter settings defines one game in a potentially vast space of possible games. This game-space view has two benefits. We can test agents more thoroughly by selecting multiple games from the space to be sure that they are not over-fitting to a particular instance of the game. Also, we can use optimisation algorithms to tune the game to meet particular objectives. For maximum flexibility we recommend storing every game parameter in a single object, defined in a data-bound class. For example, suppose we have a PlanetWars game, we then define a class called PlanetWarsParams. Each object of this class defines a particular combination of game parameters, and hence a particular game among the set of possible games in this space.

When copying the game state, a deep copy must also be made of the parameters object. This enables experimenting with different game-versions concurrently. An interesting use-case is as follows: the SFP agent can be given a false FM i.e. one in which the parameters have been changed compared to the ones used in the game under test. Experimenting with false forward models provides valuable insights into how bad a model can be before it becomes useless. Interestingly, initial tests show the model can be highly inaccurate while still being useful.

2.2 Advancing the Game State

SFP algorithms typically make many more calls to the next state function than they do to the copy method. For example, rolling horizon evolution or Monte Carlo tree search with a sequence length (roll-out depth) of L will make a factor of L more next state calls than game state copies. For video games, values of L typically varies between 5 and 500, so this means care must be taken to implement an efficient next state function.

Beyond following normal good programming practice, the main ways to do this are as follows.

- Use efficient data structures for detecting object collisions.
- Pre-compute and cache data that will be regularly needed.
- In the object-update loop, it may be unnecessary or even undesirable to update every object on every game tick.

The simplest and most efficient data structure for detecting object collisions is an array which can be arranged as a 2d grid for 2d games and offers constant-time access (the time to access a grid element does not depend on the size of the grid or the number of objects in it). For 3d games Binary Space Partition trees are a common

choice, offering access which has logarithmic cost with respect to the number of entries.

An example of pre-computation may be found in the extended version of PlanetWars we implemented [4]. Each planet exerts gravitational pull on the ships in transit, but this is pre-computed as a vector field and stored in a 2d array, speeding the next state function by a factor of the number of planets N when there are many ships in transit.

Regarding object-updates, some games even use skipped updates to improve the game play. For example, the original Space Invaders by Taito (1979) would only move a single alien in each game tick. This led to two emergent and interesting effects. One is that the aliens move in a visually appealing dog-legged fashion, only lining up occasionally. The other is that the remaining aliens speed up as their compatriots are destroyed. The increase in speed, an effect which is most extreme when going from two to one remaining alien, makes the game much more fun to play and provides a natural progression in difficulty within each level.

3 Interfacing

The original GVGAI interface as seen by the AI agents seems over-complex. For the new Java games, we designed a simpler interface which is still suitable for the SFP agents and is exactly the same for single- and two-player games (or even n-player games). This is called AbstractGameState and is listed in Table 1. The `next` method takes an array of int as an argument where the ith element is the action for the ith player. The other methods are self-explanatory, although the `nActions` method assumes that each player will have an equal number of actions available in each state, something which is clearly not true in general and will be changed in future versions.

All the games presented in this chapter implement the `AbstractGameState` interface. All agents should implement `SimplePlayerInterface`. Note that the `getAction` method includes the id of the player: this is to support multi-player games (such as Planet Wars) within the same interface. The `reset` method is useful for Rolling Horizon Evolution agents that would normally use a shift-buffer to retain the current best plan between calls to `next`.

3.1 Running GVGAI Agents on the Java Games

The main purpose of GVGAI-Java is to expand and enrich the set of GVGAI games. To meet this requirement, we implemented a wrapper class that maps each AbstractGameState method to its equivalent standard GVGAI method. There are some

Table 1: The `AbstractGameState` interface.

```
interface AbstractGameState {
    AbstractGameState copy();
    AbstractGameState next(int[] actions);
    int nActions();
    double getScore();
    boolean isTerminal();
}
```

Table 2: `SimplePlayerInterface`: an interface which all compatible agents implement. The `reset` method is included because some agents otherwise retain important information between calls to `getAction`.

```
interface SimplePlayerInterface {
    int getAction(AbstractGameState gameState, int playerId);
    SimplePlayerInterface reset();
}
```

limitations: methods such as `getAvatar` are not available in this cut down version: calling a non-implemented method throws a runtime exception. This means that many heuristic methods will not work, and unfortunately excludes some leading GVGAI planning agents such as YOLO-Bot.

For some games this is almost unavoidable: games such as Planet Wars do not have an avatar. While it may be possible to modify a game to include an avatar in a meaningful way, the result would be a different game (possibly an interesting one).

For more typical arcade games, the solution is to extend the `AbstractGameState` interface with a new interface called (say) `ExtendedAbstractGameState`. This will have all the required GVGAI method signatures. All compatible games would then implement the extended interface.

Having implemented the wrapper class, we can now run the existing GVGAI agents on an extended set of games. Just to be clear, at the time of writing this only works for the agents which are solely reward based and do not use heuristics based on observable game features. As explained above, the extension to do this is straightforward for typical arcade games but has not yet been done.

3.2 Running new agents on the VGDL Games

When developing agents for the new Java-based games it is also desirable to benchmark them on existing VGDL games. One reason is that the Java games are much faster and allow more effective tuning of the agent parameters. It is therefore interesting to test how well the newly tuned agents perform on the VGDL games.

The approach is simply the opposite of the previous one: we now take a standard VGDL game and provide a generic wrapper class that implements the AbstractGameState interface. Hence, with this one class we enable our new agents to play any of the VGDL games.

4 Sample Java Games

This section outlines some Java games that fit the model proposed here.

4.1 Asteroids

Asteroids, released to great acclaim in 1979, is one of the classic video games of all time, and Atari's most profitable[5].

The challenge for players is to avoid being hit by asteroids, while aiming at them accurately. There are three sizes of asteroids: large, medium and small. Each screen starts with a number of large rocks: as each one is hit either by the player, or by a missile fired by an enemy flying saucer, it splits in to a smaller one: large rocks split into two medium ones, medium into two small ones, small ones disappear. There is also a score progression, with the score per rock increasing as the size decreases. The arcade version features two types of flying that appear at various intervals to engage the player in combat. The version implemented here does not include the flying saucers, but still provides an interesting challenge.

A key strategy is to control the number of asteroids on the screen by shooting one large one at a time, then picking off a single medium rock and each small one it gives rise to, before breaking another large rock. Shooting rocks randomly makes the game very hard, with a potentially great many rocks for the player to avoid.

Figure 1 gives a screenshot of the game screen. Pink lines illustrate the simulations of the rolling horizon evolution agent using a sequence length of 100 and controlling the spaceship. These are shown for illustrative purposes only, and ignore the fact that each rollout involves other changes to the game state such as firing missiles and the rocks moving and splitting.

[5] `https://en.wikipedia.org/wiki/Asteroids_(video_game)`

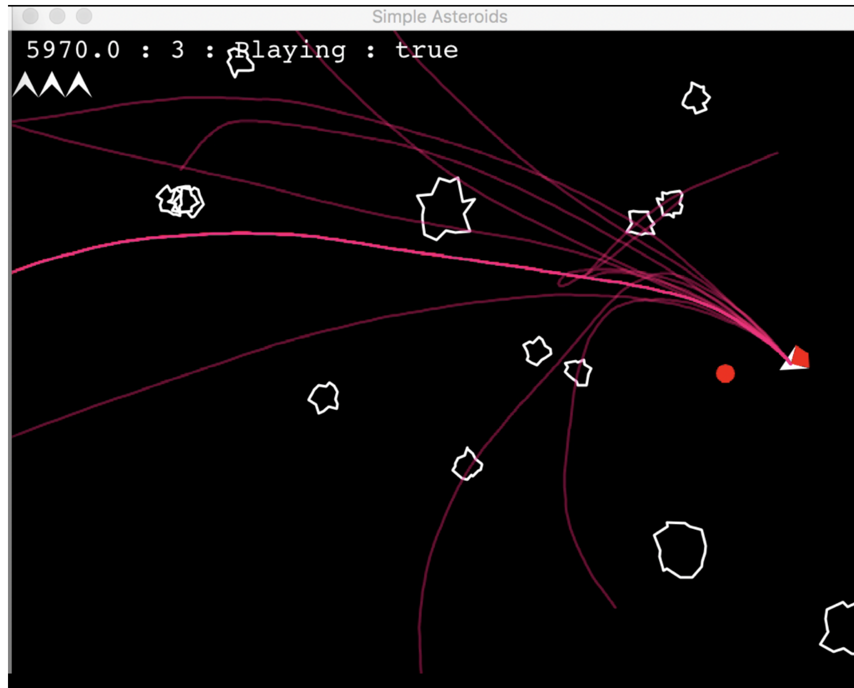

Fig. 1: Java version of Asteroids, compatible with GVGAI. The game runs at a high speed but includes legacy code which does not implement the recommended approach to parameterisation.

Comparison with VGDL Version Although VGDL has been extended to allow continuous physics and includes a version of Asteroids, the main advantages of the version presented here, enabled by the use of a high-level language are:

– much faster (Java version runs at around 600,000 game ticks per second).
– easy to generate random games (in fact every game starts with different random rock positions and velocities).
– easy to procedurally generate rocks, and rotate them as they move. This is mostly cosmetic, though for large jagged rocks it can affect the game play.
– overlays to show a key aspect of the rollouts. This helps provide insight for a researcher when tuning an agent: for example, if the projections all end in the same or very similar positions then the rollouts show insufficient exploration of the state space.[6]

[6] Perhaps counter-intuitively, uniform random rollouts may exhibit exactly this problem. When the ship is travelling forward at high speed it takes a more focused effort to turn it around than is afforded by uniform random rollouts.

– better controls for a human player (can thrust while firing and rotating)

4.2 Fast Planet Wars

Planet Wars (also called Galcon) is a relatively simple Real-Time Strategy (RTS) game that has many free versions available for iOS and Android. The Java version shown in Figure 2 is described in more detail by [4]. The game involves many features that would lie beyond the current capabilities of VGDL, such as the following.

– Gravity field (shown by vector lines on the map) causes ships to follow curved trajectories
– Allows plug and play of different actuators (i.e. different ways for a human player or AI agent to control the game). The simplest actuator involves clicking a source then destination planet to transfer ships between.
– An alternative actuator uses the spin of the planets to add an additional skill to the game. This involves clicking and holding on the source planet. While the planet is selected, ships are loaded on to a transit vessel. When the planet is de-selected (e.g. the mouse button is released) then the transit departs in the current direction it is facing. Combined with the gravity field, this adds a significant level of skill to the act of transferring ships between the intended planets.
– The implementation has also been designed for speed, and runs at more than 800,000 game ticks per second for typical game setups.

The game includes many parameter settings as described in [4] including the number of planets, the ship growth rates, the range of planet radii, the size of the map, the ship launch speed and the strength of gravity. All of these parameters have significant effects on the gameplay.

4.3 Cave Swing

Cave Swing is a simple side-scrolling casual game where the objective is to swing through the cave to reach a destination blue zone at the rightmost end, while avoiding all the pink boundaries. It is a one-touch game suitable for a mobile platform. The avatar is the blue square. Pressing the space bar attaches an elastic rope to the nearest anchor point, releasing it disconnects the rope. Both connection and disconnection happen with immediate effect. Anchor points are shown as grey disks, apart from the nearest one which is shown as a yellow disk. Points are scored for progressing to the right and upward, but each game tick spent loses points. There is also a penalty for crashing and a bonus for completing the game within the allotted time. Hence,

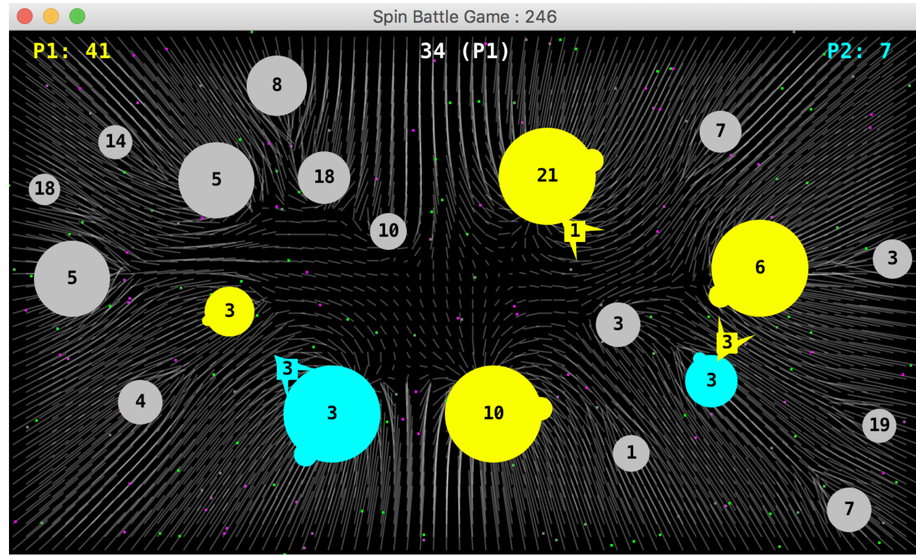

Fig. 2: A fast implementation of Planet Wars with spinning planets and a gravity field, both of which have significant effects on the game play. The game is compatible with GVGAI and adopts the recommended approach to parameterising the game.

the aim of the game is to travel to to the right as quickly as possible and finish as high up the blue wall as possible.

The avatar falls under the force of gravity, but this can be more than counteracted by the tension in the elastic rope when attached. The tension force is calculated using Hooke's law using the assumption that the natural length of the rope is zero, so the tension is proportional to the length of the rope.

Cave Swing follows the recommended approach of having all the game parameters bundled into a single object (of type `CaveSwingParams`). Significant parameters include: the size of the cave (width and height), the gravitational force (specified as a 2D vector to allow horizontal as well as vertical force), Hooke's constant, the number of anchor points, the height of the anchor points and various score-related factors.

4.4 Speed

Table 3 shows the timing results for the three games outlined in this chapter running single-threaded on an iMac with core m5 processor. For comparison we also include two typical VGDL games running on the same machine. The speed-up achieved with the Java games is clearly significant.

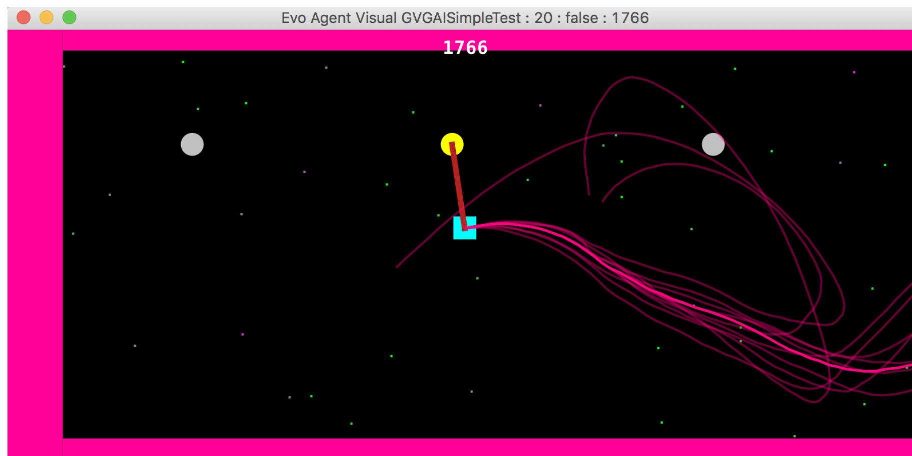

Fig. 3: The simple but fun casual game we call *Cave Swing*. This runs at high speed and adopts our preferred approach to bundling all parameters into a `CaveSwingParams` object.

Table 3: Speed of the `next` method in units of thousands of operations per second (iMac with 3.4 GHz Intel Core i5 CPU running single-threaded).

Game	kop/s (1)
Asteroids	600
Planet Wars	870
Cave Swing	4,800
Aliens (VGDL)	65
SeaQuest (VGDL)	61

5 Conclusions

In this chapter, we outlined three games that are compatible with GVGAI but written in Java rather than VGDL to enable faster operation, more flexible gameplay and additional bespoke visualisations (the pink overlaid lines showing expected agent trajectories for each rollout). Authoring games in a high-level language offers some clear advantages and disadvantages compared to using VGDL, but we believe the advantages are significant enough to make this an important future direction for general game AI research. The high speed makes the approach especially well-suited to SFP methods such as MCTS and RHEA.

Each of the games presented in this chapter has a number of parameters to vary, many of which significantly affect the gameplay. Each game is therefore one of many possible games and we encourage testing agents on many instances of the game in

order to provide more robust results. The game parameters can also be optimised to achieve particular aims, such as being particularly hard or easy for certain types of agent, or to maximise skill-depth. Due to the speed of the Java games and the sample efficiency of the N-Tuple Bandit Evolutionary Algorithm (NTBEA) [5], games can now be optimised in less than 10 seconds in some cases.

References

1. M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: an evaluation platform for general agents," *Journal of Artificial Intelligence Research*, vol. 47, no. 1, pp. 253–279, 2013.
2. G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai Gym," *arXiv preprint arXiv:1606.01540*, 2016.
3. D. Jemerov and S. Isakova, *Kotlin in Action.* Manning Publications, 2017.
4. S. M. Lucas, "Game AI Research with Fast Planet Wars Variants," in *IEEE Conference on Computational Intelligence and Games*, 2018.
5. S. M. Lucas, J. Liu, and D. Perez-Liebana, "The n-tuple bandit evolutionary algorithm for game agent optimisation," *arXiv preprint arXiv:1802.05991*, 2018.
6. 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.