Towards a Video Game Description Language

Marc Ebner[1], John Levine[2], Simon M. Lucas[3], Tom Schaul[4], Tommy Thompson[5], and Julian Togelius[6]

1 **Institut für Mathematik und Informatik, Ernst Moritz Arndt Universität Greifswald**
marc.ebner@uni-greifswald.de

2 **Department of Computer and Information Science, University of Strathclyde**
john.levine@strath.ac.uk

3 **School of Computer Science and Electrical Engineering, University of Essex**
sml@essex.ac.uk

4 **Courant Institute of Mathematical Sciences, New York University**
schaul@cims.nyu.edu

5 **School of Computing and Mathematics, University of Derby**
t.thompson@derby.ac.uk

6 **Center for Computer Games Research, IT University of Copenhagen**
julian@togelius.com

Abstract

This chapter is a direct follow-up to the chapter on General Video Game Playing (GVGP). As that group recognised the need to create a Video Game Description Language (VGDL), we formed a group to address that challenge and the results of that group is the current chapter. Unlike the VGDL envisioned in the previous chapter, the language envisioned here is not meant to be supplied to the game-playing agent for automatic reasoning; instead we argue that the agent should learn this from interaction with the system.

The main purpose of the language proposed here is to be able to specify complete video games, so that they could be compiled with a special VGDL compiler. Implementing such a compiler could provide numerous opportunities; users could modify existing games very quickly, or have a library of existing implementations defined within the language (e.g. an Asteroids ship or a Mario avatar) that have pre-existing, parameterised behaviours that can be customised for the users specific purposes. Provided the language is fit for purpose, automatic game creation could be explored further through experimentation with machine learning algorithms, furthering research in game creation and design.

In order for both of these perceived functions to be realised and to ensure it is suitable for a large user base we recognise that the language carries several key requirements. Not only must it be human-readable, but retain the capability to be both expressive and extensible whilst equally simple as it is general. In our preliminary discussions, we sought to define the key requirements and challenges in constructing a new VGDL that will become part of the GVGP process. From this we have proposed an initial design to the semantics of the language and the components required to define a given game. Furthermore, we applied this approach to represent classic games such as Space Invaders, Lunar Lander and Frogger in an attempt to identify potential problems that may come to light. Work is ongoing to realise the potential of the VGDL for the purposes of Procedural Content Generation, Automatic Game Design and Transfer Learning.

1998 ACM Subject Classification I.2.1 Applications and Expert Systems: Games

Keywords and phrases Video games, description language, language construction

Digital Object Identifier 10.4230/DFU.Vol6.12191.85

© Marc Ebner, John Levine, Simon M. Lucas, Tom Schaul, Tommy Thompson, and Julian Togelius; licensed under Creative Commons License CC-BY

Artificial and Computational Intelligence in Games. *Dagstuhl Follow-Ups*, Volume 6, ISBN 978-3-939897-62-0.
Editors: Simon M. Lucas, Michael Mateas, Mike Preuss, Pieter Spronck, and Julian Togelius; pp. 85–100

Dagstuhl Publishing
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Germany

1 Motivation

Recent discussions at the Dagstuhl seminar on Artificial and Computational Intelligence in Games addressed the challenge of extending the principles of General Game Playing (GGP) to video games. While the rationale and challenges facing research in General Video Game Playing (GVGP) is the focus of another report in these proceedings, the need for a Game Description Language (GDL) designed to suitably express the key concepts and mechanics inherent in video games became apparent. This has led to this report exploring the challenges and potential pitfalls faced in creating a Video Game Description Language (VGDL) that will work as part of the GVGP process.

The focus of traditional GDLs is to express components expected in the state of a game, and the rules that induce transitions, resulting in a state-action space. Typically, GDLs for use in GGP competitions conform to a set-theoretic language that expresses atomic facts in predicate logic. One of the more popular examples: the GDL component of the Stanford General Game Playing Competition employs this approach to define discrete games of complete information [4]. While this provides means to define and subsequently generate games that conform to this particular set, the language is ill-suited for defining video game environments. This is due to a number of factors:

- Nondeterministic behaviour in video games as a result of non-player characters (NPCs) or elements of chance are common.
- Decisions for actions in video games can be attributed to player input or prescribed NPC behaviour. These decisions are no longer turn-based and may occur simultaneously at any step of the game.
- Video game environments may employ continuous or temporal effects, real-time physics and context-based collisions or interactions between combinations of players and artefacts. Part of the player's task is learning to predict those dynamics.
- Typical video games use much larger environments than board games for the player to interact with. However, the player-environment interactions are sparse, meaning that a part of the environment complexity is rarely or never influencing decision making.

These factors among others present an interesting research challenge in its own right: to try and maintain a balance between simplicity and generality in a description language whilst ensuring it is suitably expressive. Driven by these factors and the challenge they represent, we have raised key goals that shape our aspirations:

- To define a GDL that supports the core mechanics and behaviour expected of classical 2D video games.
- To ensure the VGDL defined will support the work being developed in the GVGP discussion group.
- Define a language that is sufficient not only to represent a game to the human reader, but also for a compiler to generate an instance of the game.
- Provide a VGDL that is unambiguous, extensible and tractable that could provide opportunities for Procedural Content Generation (PCG).

In this report, we consolidate the key points raised by the authors as we explored the challenges and impact of expanding the current research in Game Description Languages for Video Games to further these design goals. We highlight the current state-of-the-art in the field, the core criteria and considerations for the design and subsequent construction of a VGDL, the challenges the problem domain will impose, the potential for future research and collaboration and finally highlight preliminary steps in developing a VGDL prototype.

2 Previous Work

Several attempts have been made in the past to model aspects of both simple arcade games and board games. Most of them had the purpose of being able to generate complete games (through evolutionary computation or constraint satisfaction) but there have also been attempts at modelling games for the purposes of game design assistance.

The Stanford General Game Playing Competition defines its games in a special-purpose language based on first-order logic [4]. This language is capable of modelling a very large space of games, as long as they are perfect information, turn-taking games; those games that can practically be modelled with the Stanford GDL tend to be board games or games that share similar qualities. The language is very verbose: an example definition of Tic-Tac-Toe runs to three pages of code. We do not know of any automatically generated game descriptions using this GDL, but even if they exist, their space is unlikely to be suitable for automatic search, as any change to a game is likely to result in an unplayable and perhaps even semantically inconsistent game.

One noteworthy attempt at describing and evolving board games is Browne's *Ludi* language and the system powered by this language that evolves complete board games [1]. The Ludi system is a high-level language, and expressly limited to a relatively narrow domain of two-player "recombination games", i.e. turn-based board games with a restricted set of boards and pieces, and with perfect information. The primitives of the Ludi GDL are high-level concepts such as "tiling", "players black white" etc. As the language was expressly designed to be used for automatic game generation it is well suited to search, and most syntactically correct variations on existing game descriptions yield playable games.

Also related is the Casanova language developed by Maggiore [6], now an open source project [2]. Casanova is a high-level programming language with particular constructs that make certain aspects of games programming especially straightforward, such as the update/display loop, and efficient collision detection. However, it is significantly lower level than the VGDL developed here and its complex syntax would means that it is unsuitable as it stands for evolving new games.

Moving from the domain of board games to video games that feature continuous or semi-continuous time and space, only a few attempts exist, all of them relatively limited in scope and capabilities. Togelius and Schmidhuber created a system which evolved simple game rules in a very confined rule space [11]. The game ontology featured red, blue and green things (where a thing could turn out to be an enemy, a helper, a power-up or perhaps something else), and a player agent. These could move about in a grid environment and interact. The rules defined the initial number of each thing, the score goal and time limit, how each type of thing moved, and an interaction matrix. The interaction matrix specified when things of different colours collided with each other, and with the player agent; interactions could have effects such as teleporting or killing things or the agent, or increasing and decreasing the score. Using this system, it was possible to generate very simple Pac-Man-like games.

Nelson and Mateas created a system that allowed formalisation of the logic of dynamical games, and which was capable of creating complete *Wario Ware*-style micro-games [9], i.e. simpler than a typical Atari 2600 game. That system has not yet been applied to describe or generate anything more complex than that.

Togelius and Schmidhuber's system inspired Smith and Mateas to create Variations Forever, which features a somewhat expanded search space along the same basic lines, but a radically different representation [10]. The rules are encoded as logic statements, generated using Answer Set Programming (ASP [5]) in response to various specified constraints. For

example, it is possible to ask the generator to generate a game which is winnable by pushing a red thing into a blue thing; if such a game exists in the design space, ASP will find it. Another system that took the same inspiration to another direction is Cook et al.'s ANGELINA, which expands the design space by generating not only rules but also other aspects of the game, particularly levels, concurrently [3]. The Inform system [8] permits specifying text adventure games, coded in natural language. Further, Mahlmann et al.'s work on a Strategy Game Description Language, used for representing and generating turn-based strategy games, should also be mentioned [7].

As far as we are of, no game description language can represent even simple arcade games of the type playable on an Atari 2600 with any fidelity. For example, the physics of Lunar Lander, the destructible bases of Space Invader or the attachment of the frog to the logs in Frogger cannot be modelled by existing GDL's.

Aside from generating rules, there are a number of applications of evolutionary computation to generate other types of content for games; see [12] for a survey.

3 Criteria and Considerations

During our initial discussions, we identified what core criteria would be required for any resulting VGDL. These criteria are instrumental in ensuring the original goals introduced in Section 1:

- **Human-readable:** We want to ensure a simplicity in the structure of the language that ensures a human reader can quickly formulate new definitions or understand existing ones through high level language.
- **Unambiguous, and easy to parse into actual games:** We intend for the software framework attached to the language to be able to instantly generate working prototypes of the defined games through a simple parsing process. The game mechanics and concepts transfer between games and, where possible, between classes of games.
- **Searchable/tractable:** A representation of the game components in a concise tree structure allows for generative approaches like genetic programming, specifically leading to natural cross-over operators.
- **Expressive:** The language must be suitably expressive to represent the core mechanics and objects one expects of classical 2D video games. Most of these aspects are by design encapsulated, simplifying the language and drastically reducing the code required to define them.
- **Extensible:** Many game types, components and dynamics can be added any time, by augmenting the language vocabulary with new encapsulated implementations.
- **Legal & viable for randomly generated games:** all game components and dynamics have 'sensible defaults', which make them as likely as possible to inter-operate well with a large share of randomly specified other components.

These criteria are considered as we move into the subsequent section, in which we discuss how we envisage the structure of the VGDL, and how components necessary to describe video games are expressed.

4 Language Structure

Our discussion focused on the core components required in order to represent a simple video game, this was separated into the following categories:

- **Map:** Defines the 2D layout of the game, and also the initialization of structures as obstacles or invisible entities such as end of screen boundaries and spawn points.
- **Objects:** Objects that exist within the game. These can be different types of entities, such as non-player characters (NPC), structures, collectibles, projectiles, etc.
- **Player Definitions:** Determines the number of players, and which ones are human-controlled.
- **Avatars:** Player-controlled object(s) that exist on the map. When a player passes input to the game, it will have an impact on the state of the avatar.
- **Physics:** Definition of the movement/collision dynamics for all or some object classes.
- **Events:** Control events sent from the player via input devices, timed or random events, and object collision triggered events that affect the game.
- **Rules:** Any event can be specified to lead to game-state changes like object destruction, score changes, another event, etc. The rules also specify the game objectives, and termination.

4.1 Map

The map of the game defines the (initial) two-dimensional layout, in particular the positioning of the obstacles and the starting positions of the objects (including the avatar). Initially, we see maps as string representations, that can be parsed directly into a game state by mapping each ASCII character to an instance of the corresponding object class at its position.

4.2 Objects

The fundamental components of a game are the *objects* that exist in the 2D space. Every object has a set of (x,y) coordinates, a bounding polygon (or radius), and is represented visually on-screen. Objects are part of a hierarchy that is defined by the VGDL and permits the inheritance of properties like physics, or collision effects with a minimal specification. The player-controlled avatar is a specific type of object, and most games dynamics revolve around the object interactions. A permanent static object, such as a wall, collectable (power pills, health packs, coins etc.), portals or spawn/goal location is referred to as a 'structure'. These static objects will not require any update logic for each game tick. Meanwhile dynamic objects, such as avatars, NPCs, elevators and projectiles will specify how they change state on subsequent ticks of the game.

4.3 Physics

The temporal dynamics of non-static objects are called their 'physics', which include aspects like gravitational pull, friction, repulsion forces, bouncing effects, stickiness, etc. Furthermore, more active NPC behaviors like fleeing or chasing are included under this header as well. Those dynamics operate on (specified or default) properties of the objects, like mass, inertia, temperature, fuel, etc. The player control inputs affect the dynamics of the avatar object(s), and those mappings to movement and behavior are encapsulated in the avatar class used (e.g. spaceship, car, frog, pointer).

4.4 Events

We recognise that in many games, there are instances of state-transition that are not the result of an established agent committing actions. As such, we have identified the notion of

an *event*: a circumstance that at an undetermined point in time will result in change to one or more objects on the next time step of the game.

To give a simple example, an NPC's behaviour is defined by its own rationale. As such, at any given time step, the action the NPC will commit is determined by pre-written code that may factor elements from the environment as well as whether the current time step bears any significance in-terms of the NPCs lifespan or any cyclical behaviour. Meanwhile, a human player's actions are not inferred from the entity itself. Rather, it is achieved by the system polling the input devices that are recognised by the game software. In the event that the input devices present a signal, we recognise this as an event, given that this signal will appear at an indeterminable point in time.

We have recognised a range of circumstances we consider events, namely:

- Signals from players via recognised input devices.
- Object collisions (with each other, or with map boundaries).
- Actions occur at specific points of time in the game (e.g. spawning).
- Actions that will occur with a given probability.

For any game we wish to define, we must identify the range of events that can occur at any point. As is seen later in Sections 6, we identify the list of events required for a handful of games.

4.5 Rules

The game engine detects when the bounding boxes of two objects intersect and triggers a collision event. Besides the default effect of ignoring the collision, common effects that could occur include: bouncing off surfaces/objects, destruction of one or both objects, sticking together, teleportation and the spawning of a new object. Outside of game entities, collisions can also refer to the player leaving the 'view' of the game, as such we require rules for recognising instances of scrolling, panning and zooming of the game view. Lastly, collisions can also trigger non-local events, this can affect the current score, end a given level either as a result of success or termination of the players avatar, move the game to the next level or indeed end the game entirely.

5 Representing the Language: Preliminary Syntax

We propose a preliminary syntax as close to human-readable as possible, in order to invite game contributions from as broad an audience as possible, but definitely including game designers and other non-programmers. Our proposed VGDL needs to suitably represent a tree structure, which we propose to encode in Python-like white-space syntax. The other syntax features are arrows (>) representing *mappings*, e.g. from tile characters to objects names, from object names to object properties, or from collision pairs to their effects. Object properties are assigned by terms combining property name, an '=' symbol, and the value. We require naming properties to make the code more readable and options ordering-independent. Example: "Rectangle (height=3, width=12)". Parentheses and commas can be added for readability, but are ignored by the parser.

6 Examples of Classic Games

The preceding sections discussed the purpose, structure and format of the VGDL that the authors have devised. In this section we showcase examples that were conceived during

Figure 1 A screenshot of the Space Invaders game [15].

discussion. For this language to work in conjunction with the GVGP research, it is desirable to suitably express the entities and mechanics of games that one would expect to find on the Atari 2600. This section therefore focuses on three games reflective of that period: Space Invaders, Lunar Lander and Frogger; chosen for their dissimilar mechanics and design. These examples were selected for two purposes: to provide context to the design decisions we made during discussion – thus providing definitions that reflect that we had envisaged – and secondly as an attempt to identify potential complications that would need to be resolved. What follows is our attempt to define the entities, physics, collision matrix and event table that reflect the game as succinctly as possible.

6.1 Space Invaders

Space Invaders, originally released in 1978 in arcades and in 1980 for the Atari 2600, is one of the most influential games in the industry both critically and commercially [15]. The mechanics of Space Invaders are relatively straightforward; the player must defend against waves of enemy characters (the space invaders) who are slowly moving down the screen. The enemies begin at the top of the screen and are organised into rows of a fixed size. The group will collectively move in one direction along the x-axis until the outmost invader collides with the edge of the screen, at which point the group moves down a set distance and alternates their horizontal direction. While moving down the screen, enemies will at random open fire and send missiles towards the bottom of the screen. These missiles either eliminate the player upon contact, or can damage one of bunkers situated between the invaders and the player. The player can hide behind these bunkers to strategically avoid being destroyed out by the invaders. In addition to these invaders, a flying saucer can appear at random intervals which traverses along the top of the screen, opening fire on the player below. To retaliate against these threats, the player controls a cannon at the bottom of the screen which can be moved left or right within the bounds of the game world. The player can fire missiles at the invaders that will eliminate them upon contact. The player must eliminate all enemy invaders before they reach the bottom of the screen in order to win. Figure 1 shows a screenshot from the Space Invaders game. As previously described, we can see the player-controlled cannon at the bottom of the screen as the waves of enemies move down in their grouped 'typewriter' formation.

Referring to the initial language structure in Section 4, we identify the core components of Space Invaders as follows:

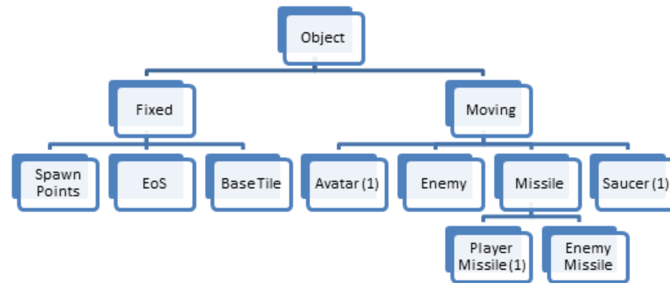

Figure 2 An object hierarchy of the entities found within the Space Invaders game. Note that the Avatar, Player Missile and Saucer have been identified as singletons.

Table 1 The Space Invaders collision matrix, identifying the behaviour of the game in the event that two entities collide in the game.

	Avatar	Base	Enemy Missile	Player Missile	Enemy	End of Screen	Saucer
Avatar			life lost		game over	avatar stops	
Base			base damaged; missile disappears	base damaged; missile disappears	base tile removed		
Enemy Missile				both destroyed		missile destroyed	
Player Missile					destroys enemy	missile destroyed	saucer destroyed
Enemy						moves down, all enemies change dir.	
End of Screen							disappears
Saucer							

Table 2 The event listing for Space Invaders, showcasing not only instances which happen with a certain probability or time, but also the relation of input devices to avatar control.

Event	Action
Joystick left	Avatar left
Joystick right	Avatar right
Joystick button	Avatar fires missile
Time elapsed	Saucer appears
Probability event	Random enemy fires a missile

- **Map:** An empty map save for the tiles that represent the bases the player uses for cover. Locations are annotated for spawn points NPC waves, the human player and the structures that present the boundary of the screen.
- **Objects:** These are separated in 'Fixed' and 'Moving' objects. The former is indicative of spawn points and the tiles that represent a base. Meanwhile, the latter denotes entities such as NPCs and missiles. A hierarchy of these objects is shown in Figure 2.
- **Player Definition:** At present we consider this a one-player game, with the controls identified in Table 2.
- **Avatars:** Player controls one sole avatar, the missle cannon.
- **Physics:** Each moving object in the game moves a fixed distance at each timestep. There are no instances of physics being employed beyond the collisions between objects.
- **Events:** The events of the game can be inferred both from the collision matrix in Table 1 and the event listings in Table 2.
- **Rules:** Rules of the game, such as player scoring, the death of either enemy or player, and winning the game can be inferred from the event table and collision matrix.

We feel that this information, which was gathered as part of our groups discussion, is representative of the Space Invaders game. The object tree in Figure 2 identifies each type of object which is used in the game. From this we can begin to dictate rules for collision as well as assign events to these objects. Note that we have identified some objects as singletons, given that only one of these objects at most should exist in the game world at any given timestep. The collision matrix in Table 1, highlights the behaviour and events that are triggered as a result of objects colliding within the game. Meanwhile the events shown in Table 2 not only shows the actions that the user can have via the avatar, but circumstances such as enemy saucers or missiles that are triggered by time or probability.

6.2 Lunar Lander

Lunar Lander is an arcade game released by Atari in 1979. This game was selected due to its greater emphasis on precision when controlling the avatar, which proved popular at the time of release [14]. Control of the avatar is confounded by gravitational forces and momentum gathered from movement, which is an element that we did not have to consider in Space Invaders. As shown in Figure 3, the player is tasked with piloting the lunar module towards one of the predefined landing platforms. Each platform carries a score multiplier and given the rocky terrain of the moon surface, they are the only safe places to land the module; landing the module in any location other than the pads results in death. The player can rotate the lander within a 180 degree range such that the thrusters of the lander can range from being perpendicular and parallel to the ground. The aft thruster must be used to slow the descent of the lander given the gravitational forces bearing down upon it. However, the player has two further considerations: the control must be applied proportionally to provide adequate strength to the thrusters. Secondly, applying the thrusters will consume the limited fuel supply. The player scores multiples of 50 points for a successful landing, with the multiplier identified on the landing platform. Failed attempts also receive score based on how smoothly the lander descended.

Once again we discussed how the game could be formulated within the proposed VGDL, summarised as follows:

- **Map:** The map file would identify the terrain of the world as structures, with unique identifiers for landing platforms based on their score multipliers. The map will also

Figure 3 A screenshot of Lunar Lander taken from [14] showing the player trying to land on either of the highlighted platforms.

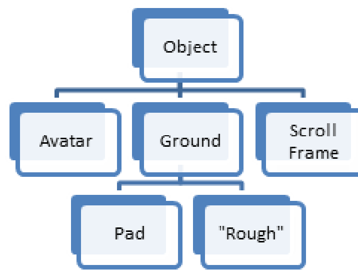

Figure 4 An object hierarchy of the entities found in Lunar Lander, complete with reference to the scroll pane used for viewing the game.

Table 3 Lunar Lander collision matrix.

	Lunar Lander	Landing Pad	Rough	Close to EOS
Lunar Lander		IF upright and low speed then win else ship explodes	ship explodes	scroll/wrap

Table 4 Lunar Lander events.

Event	Action
Joystick Left	Rotate Left Stepwise
Joystick Right	Rotate Right Stepwise
Joystick Up	Increase Thrust
Joystick Down	Decrease Thrust
Landing Altitude Reached	Zoom Scroll Pane

identify where the avatar will spawn in the game world at the beginning of a landing (typically the top-left).

- **Objects:** The object tree, shown in Figure 4 indicates the avatar itself, the ground being separated into landing pad and the 'rough': i.e. all other parts of the terrain. One unique object in this tree is the Scroll Frame. Having played Lunar Lander we recognised the scroll frame's view follows the lander should it move towards the left and right edges of the screen. Furthermore, once the lander has descended to an altitude of approximately 500, the scroll frame zooms closer to the lander to allow for more precise control.
- **Player Definition:** This a one-player game, with controls as shown in the event list in Table 2.
- **Avatars:** The lander module.
- **Physics:** The game must model the gravitational force that exerts upon the lander, and the continuous variables needed to calculate the thrust speed. We would intend for the language's multiple physics models to contain some that represent these phenomena. These models will be parameterised to customise the model for the game.
- **Events:** With exception of the player input, the only event shown in Table 2 is the zooming of the scroll pane once the final descent altitude is reached.
- **Rules:** Rules of the game, such as player scoring, the death of the player, and winning the game can be inferred from the event table and collision matrix.

Again we have encapsulated the core game mechanics to be represented in the VGDL. We believe these definitions can be expressed succinctly within the proposed language. One component of the game that has not been explored as yet is the varying difficulties that Lunar Lander offers to the player. There are four difficulty settings, Training, Cadet, Prime and Command, differ in the strength of gravitational force applied, applying atmospheric friction to the movement of the lander and in some cases added rotational momentum. This could be achieved given our intention to apply parameterised physics models as part of the language.

6.3 Frogger

The last game covered in our initial discussions was Frogger, an arcade game developed by Konami and published by Sega in 1981 [13]. The game's objective is to control a collection of frogs one at a time through a hazardous environment and return to their homes. In order to reach the goal, the player must navigate across multiple lanes of traffic moving at varying speeds, followed by a fast flowing river full of hazards. While the first phase of the trip relies on the player using any available space on the road whilst avoiding traffic, the latter constrains the users movements to using floating logs and turtles – strangely, despite being an amphibian entering the water proves fatal. What proved interesting to players at release and subsequently to the discussion group is the range of hazards that the player must overcome. Each 'lane' of hazards moves independently across the screen at its own speed. Furthermore, some groups of floating turtles in the river would submerge after a set period of time, meaning the player had to quickly move to another safe zone.

Below we highlight the key features of the game that we would implement in the VGDL:

- **Map:** The map identifies where the initial location of the vehicles and logs. Furthermore, it must assign the goal locations, the surrounding screen edge to prevent the player moving off the level and the water hazards.

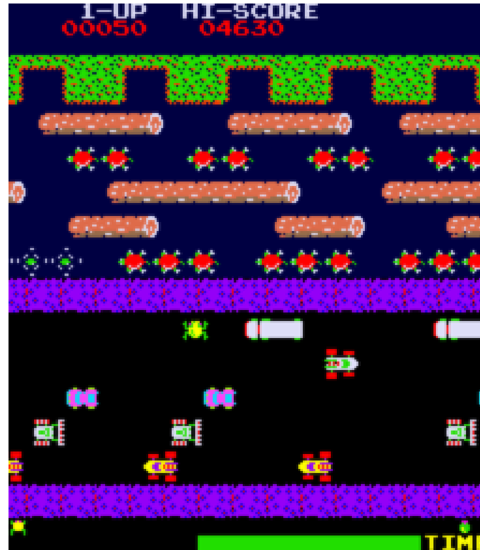

Figure 5 A screen of the Frogger game from [13], with the frog avatar crossing the freeway with only the river of logs and turtles keeping him from the lilypad goals at the top of the screen.

- **Objects:** The object tree in Figure 6 indicates the range of fixed and moving objects that exist in the game. While this tree looks similar to that shown in Figure 2, we create a hierarchy for the floating platforms in the game, using the Log as the supertype. This is of relevance to the collision matrix (Table 5), as we only define the event of colliding with the supertype. This has been introduced as collisions with the subtypes are conditional and driven by context. This issue would be addressed as a rule in the game.
- **Player Definition:** Frogger is a two-player game, however only one player is in control of the frog at any given time. The controls for the player are shown in Table 6.
- **Avatars:** The frog currently on screen.
- **Physics:** Frogger does not adopt any physics models in the movement or behaviour of any objects in game. The game is driven by the collisions between the frog and other objects, be they hazard or otherwise.
- **Events:** Table 6 shows that outside of player controls, we must consider the appearance of a fly, which provides bonus score, and the creation of new moving vehicles or logs on screen.
- **Rules:** Outside of standard scoring and life counts, the rules must carry extra information regarding the collisions that take place in game. While the collisions shown in Table 5 are similar to those shown previously, there are conditions on whether this results in death for the player. Firstly, should the player land on a floating turtle, then it is safe and as the collision matrix shows, the frog will stick to that surface. However should the turtle be submerged then the frog collides with the water, resulting in death for the player. Secondly, it is safe for a frog to collide with a crocodile provided it is not near its mouth. In the event of landing on the mouth, the player dies. In addition, there are rules that define how the player scores: should the frog reach the lily pad having landed on either a fly or female frog, then the score earned has a bonus of 200 points.

The features defined are similar to that which we have seen in the previous two games, but we addressed the unique elements of this game by defining rules for specific contexts, whilst retaining a simplicity for the design. This issue of context-specific collisions is one

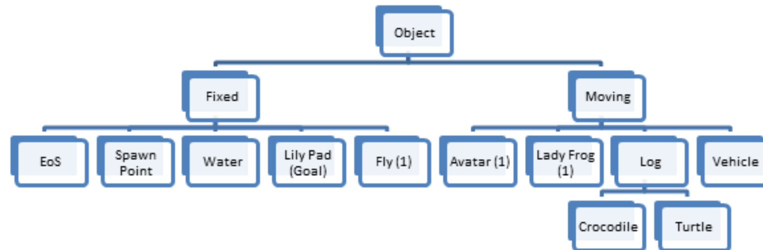

Figure 6 The proposed object hierarchy for the Frogger. Note the separation of fixed and mobile objects used previously in Figure 2. Furthermore, the crocodile and turtle objects have been placed under logs in the hierarchy given that they are also moving platforms for the frog. The difference is they carry conditions that dictate whether the frog has safely landed upon them.

Table 5 The Frogger collision matrix. Note we only consider collisions with a log type, as collisions with the log subtypes are determined at runtime based on their current state.

	Frog	Vehicle	Log	Fly	Goal Pad	Water	EOS
Frog		dies	sticks	frog eats fly	fills position, new frog created	player dies	blocks movement
Vehicle							disappears/wraps
Log							disappears/wraps
Fly							disappears/wraps
Goal Pad							
Water							
EOS							

Table 6 Frogger events.

Event	Action
Joystick Left	Jump Left
Joystick Right	Jump Right
Joystick Up	Jump Up
Joystick Down	Jump Down
Probability Event	Fly Appears
Probability Event	Lady Frog Appears
Time Elapsed	New Logs/Vehicles appear

that must be addressed early on in the language. Games that we would wish to define in the VGDL have collisions with enemies that are dependent upon state and/or location. For example, in the game Mario Bros. the player will be killed should they collide directly with a moving crab. Should the player land on the top of a 'Koopa' tortoise, then the enemy is stunned; while side-on contact proves fatal. Given our efforts on Frogger we believe this will not prove taxing when working in the formal language.

7 Prototype and Roadmap

Upon reaching the end of our time in Dagstuhl we had reached a consensus on what we expected of the Video Game Description Language. This of course is reflected by the concepts and examples shown throughout Sections 3 to 6. At the time of writing, a working prototype of the language has been developed in Python.[1] The language incorporates many of the considerations raised in Section 3 and is expressed in accordance with our expectations set out in Section 4. The language is implemented atop Pygame: a collection of Python modules that are designed to support creation of games.[2] The language compiler can take a user-defined game written in VGDL and construct a working pygame implementation. While a work in progress, the prototype is already capable of constructing games that mimic those discussed throughout our sessions. Figure 7 shows a working implementation of Space Invaders that has been developed using the prototype.

Referring back to the core language components in Section 4 and how we envisaged Space Invaders to be represented using this approach in Section 6.1, the game is currently implemented as follows:

- **Map:** The map is defined separate from the game logic and is shown in Figure 8. As discussed in Section 6.1, we have represented the screen bounds, the avatar, base tiles and spawn points for the aliens. The map definition assigns specific symbols to each of these objects, with the corresponding mapping in the game definition listing in Figure 7 (lines 11-13).
- **Objects:** Game objects are defined in the *SpriteSet* section of Figure 7. Note that upon defining the type of sprite and the class it adheres to, we introduce properties that are relevant to that entity. A simple example is the base tiles which are identified as immovable white objects. Meanwhile the aliens express the probability of fire, their movement speed and the type of behaviour they will exhibit. Note keywords such as 'Bomber and 'Missile' which are pre-defined types in the language. The indentation in the language can be used to denote hierarchical object definitions, e.g. both *sam* and *bomb* inherit the properties from *missile*, which allows statements like in line 22 that indicate the collision effect of any subclass of missile with a *base* object. Also, the *sam* missiles which are launched by the avatar are rightfully declared as singletons.
- **Player Definition & Avatars:** The player is acknowledged in as the avatar (line 3). Furthermore, it is classified as a FlakAvatar, a pre-defined type, that provides a working implementation of the control scheme and movement constraints required for Space Invaders play.
- **Events:** The collision matrix shown in Table 1 has been implemented in the *InteractionSet* within Figure 7. Note that in the majority of instances, the behaviour is to kill the sprites

[1] The source code is open-source (BSD licence), and the repository is located at `https://github.com/schaul/py-vgdl`.
[2] Pygame is freely available from `http://www.pygame.org`.

```
BasicGame
    SpriteSet
        avatar   > FlakAvatar   stype=sam
        base     > Immovable    color=WHITE
        missile  > Missile
            sam  > orientation=UP    color=BLUE singleton=True
            bomb > orientation=DOWN  color=RED  speed=0.5
        alien    > Bomber stype=bomb prob=0.01 cooldown=3 speed=0.75
        portal   > SpawnPoint   stype=alien  delay=16   total=20
    LevelMapping
        0 > base
        1 > avatar
        2 > portal
    TerminationSet
        SpriteCounter   stype=avatar    limit=0 win=False
        MultiSpriteCounter stype1=portal stype2=alien limit=0 win=True
    InteractionSet
        avatar  EOS   > stepBack
        alien   EOS   > turnAround
        missile EOS   > killSprite
        missile base  > killSprite
        base missile  > killSprite
        base    alien > killSprite
        avatar alien  > killSprite
        avatar bomb   > killSprite
        alien   sam   > killSprite
```

Figure 7 A definition of the Space Invaders game from a working prototype of the Video Game Description Language. While a work in progress, this definition satsifies many of the considerations from our discussions in Sections 4 and 6.1.

```
wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww
w                              w
w2                             w
w000                           w
w000                           w
w                              w
w                              w
w                              w
w                              w
w    000      000000     000   w
w   00000    00000000   00000  w
w   0   0    00    00   00000  w
w                1             w
wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww
```

Figure 8 A representation of the Space Invaders environment. Each character represents either the screen boundary, a component of a bunker base, the player and spawn points for the enemy players. While the game map is constructed here, the relation to the game entities is defined in the *LevelMapping* section of the game definition in Figure 7.

that are involved in the collision. Naturally this would need to handle more complex interactions in future games. Referring back to the event table shown in Table 2, the player inputs have been modelled within the *FlakAvatar* type in the language, meanwhile the probability that drives enemy fire has been declared as a property of the alien entity.

- **Rules:** The *TerminationSet* defines the conditions for winning and losing the game. The game states that having zero avatars will result in failure. Meanwhile should the number of aliens and unused spawn points reach zero,the game will declare the player as winner.

While this is a work in progress, the prototype already provides playable implementations of Space Invaders and Frogger, with definitions of games with continuous physics such as Lunar Lander are in their early stages. It is our intent to continue developing this prototype prior to developing the build that will operate in the GVGP framework. It is encouraging that we have reached this stage within a relatively short period of time since our discussions in Dagstuhl, and we will continue to expand the language until it satisfies our goals.

References

1 C. Browne. Evolutionary game design. *IEEE Transactions on Computational Intelligence and AI in Games*, pages 11–21, 2011.
2 Casanova. Casanova project page. `http://casanova.codeplex.com`, 2012.
3 Michael Cook and Simon Colton. Multi-faceted evolution of simple arcade games. In *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG)*, 2011.
4 Michael Genesereth and Nathaniel Love. General game playing: Overview of the aaai competition. *AI Magazine*, 26:62–72, 2005.
5 Vladimir Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138(1):39–54, 2002.
6 Giuseppe Maggiore, Alvise Spanò, Renzo Orsini, Michele Bugliesi, Mohamed Abbadi, and Enrico Steffinlongo. A formal specification for casanova, a language for computer games. In *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems*, pages 287–292. ACM, 2012.
7 T. Mahlmann, J. Togelius, and G. Yannakakis. Towards procedural strategy game generation: Evolving complementary unit types. *Applications of Evolutionary Computation*, pages 93–102, 2011.
8 Graham Nelson. *The Inform Designer's Manual*. Placet Solutions, 2001.
9 Mark Nelson and Michael Mateas. Towards automated game design. In *Procedings of the 10th Congress of the Italian Association for Artificial Intelligence*, 2007.
10 Adam M. Smith and Michael Mateas. Variations forever: Flexibly generating rulesets from a sculptable design space of mini-games. In *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG)*, 2010.
11 Julian Togelius and Jürgen Schmidhuber. An experiment in automatic game design. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games (CIG)*, 2008.
12 Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley, and Cameron Browne. Search-based procedural content generation: a taxonomy and survey. *IEEE Transactions on Computational Intelligence and Games*, 3:172–186, 2011.
13 Wikipedia. Frogger. `http://en.wikipedia.org/w/index.php?title=Frogger`, 2012.
14 Wikipedia. Lunar lander. `http://en.wikipedia.org/w/index.php?title=Lunar_lander_(arcade_game)`, 2012.
15 Wikipedia. Space invaders. `http://en.wikipedia.org/w/index.php?title=Space_Invaders`, 2012.