Introducing Real World Physics and Macro-Actions to General Video Game AI

Diego Perez-Liebana
University of Essex
Colchester, UK
dperez@essex.ac.uk

Matthew Stephenson
Australian National University
Canberra, Australia
matthew.stephenson@anu.edu.au

Raluca D. Gaina
University of Essex
Colchester, UK
rdgain@essex.ac.uk

Jochen Renz
Australian National University
Canberra, Australia
jochen.renz@anu.edu.au

Simon M. Lucas
University of Essex
Colchester, UK
sml@essex.ac.uk

***Abstract*—The General Video Game AI Framework has featured multiple games and several tracks since the first competition in 2014. Although the games of the framework are very assorted in nature, there is an underlying commonality with respect to the physics that govern the game: all of them are based on a grid where the sprites make discrete movements, which is not expressive enough to cover any meaningful physics. This paper introduces an enhanced physics system that brings real-world physics such as friction, inertia and other forces to the framework. We also introduce macro-actions for the first time in GVGAI in two different controllers, Rolling Horizon Evolution and Monte Carlo Tree Search. Their usefulness is demonstrated in a new set of games that exploits these new physics features. Our results show that macro-actions can help controllers in certain situations and games, although there is a strong dependency on the game played when selecting which configuration fits best.**

I. Introduction

When referring to games, these can come in many forms. We could be looking at puzzles, single or N-player games, 2D, 3D, card, board games and many other categories. Since 2014, the General Video Game AI (GVGAI [1]) Framework and Competition have benchmarked a total of 100 single-player and 40 two-player games [2]. These games do vary in many forms and characteristics (from puzzles to shooters, featuring labyrinths, dungeon crawlers, role-playing and arcade games).

However, all these games have at least one thing in common: the physics behind the games share certain characteristics that make them similar. All GVGAI games up to date were based on a grid structure that discretizes the game state, coarsening positions and velocities to certain values. For instance, the orientation of the sprites in this case could be either `Nil`, `Up`, `Down`, `Left` or `Right`, without providing a finer precision. Although this is good enough for certain games (i.e. *PacMan*), it lacks the required precision to build games such as *Scorched Earth* (Wendell Hicken, 1991), where the player shoots missiles with a specific inclination measurable in degrees.

The first aim of this paper is to describe the modifications carried out in the GVGAI framework to allow this change (among many others; see Section III-A), in order to showcase the enhanced possibilities of game design within the framework and its associated Video Game Description Language (VGDL). This paper also presents and describes the first set of 10 games designed around these features[1].

Being able to have such precision is an interesting feature from the game design point of view, but it additionally allows for more research in learning and playing with agents. Not only the state space of the games grows considerably, which can make search algorithms struggle, but it also modifies the way agents navigate through the level. For instance, in a grid physics game (like *Aliens*), the amount of pixels traveled by the agent for every lateral move is a constant, determined by the speed of the sprite. In continuous physics games (like *Mario*), however, due to the inclusion of inertia, the distance traveled per action applied depends heavily on the current speed.

Therefore, this opens an interesting research question for analysing how these agents perform in these new games in a general setting. The second aim of this paper is to provide an initial study about how the two most powerful sample controllers within the GVGAI framework (Monte Carlo Tree Search and Rolling Horizon Evolutionary Algorithm) play the games in this new set. In particular, this paper analyzes the effects and performance of using macro-actions and different lookaheads in real-world physics games.

In the rest of this paper, Section II describes related works on the subject, and Section III revises the original GVGAI framework and the modifications performed to include the new physics, including also the description of the new game set. Section IV describes the agents tested and the macro-actions system. Section V details the experimental work and discusses the results, to finally conclude the paper in Section VI.

II. Literature Review

Games used in competitions in the Computational Intelligence in Games (CIG) community come in different shapes and forms, but many bring the complexity of a continuous set of states and/or actions, as well as real-world physics at different degrees. The presence of these features clearly affects the approaches that are taken to tackle and play these games. The main contribution of this paper is to bring this type of games under the general framework, rather than addressing them separately.

[1]For simplicity, we refer to this as *Continuous* or *Real-World* Physics games and sets, in contrast with the original *Grid Physics* ones.

A popular competition in the field is the Mario (or Platformer, in its latest versions) AI competition [3]. In this game, the player controls Mario, whose objective is to clear the level, reaching the far end, while collecting bonus items and avoiding hazards. Although the set of actions is discrete - they are mapped from the buttons of a controller pad -, the game features a continuous state space and real-world physics such as gravity and inertia: if Mario is running, it won't stop automatically when directions are no longer supplied, but a few frames later. M. Nicolau et al. [4] took these considerations into account when creating their entry for the Mario AI competition, in a work that evolved Behaviour Trees with Grammatical evolution to improve navigation in the level.

Another popular competition that spanned through a few years was the Physical Travelling Salesman Problem (PTSP) challenge [5]. As in Mario AI, the physics of the game engine did take into account aspects such as inertia and elastic bouncing (up to different degrees in the Multi-Objective version of the competition [6]). In this game, where a ship needs to collect all the waypoints scattered in a maze as quickly as possible, the physics dynamics were crucial to determine an optimal path through the level, as shown in [7]. In this work, the authors show how taking into account the inertia of the ship permits to find better routes than only considering the distances between the waypoints.

With regards to physics, the Geometry Friends AI [8] occupies a similar position in the space of CIG game competitions: inertia and gravity are taken into account in a game where two different players must collaborate in order to achieve a common task. This competition, however, as well as the ones mentioned previously in this section, have a discrete set of actions. Effectively this means that all actions can be either *on* or *off*, not being able to specify an intensity for how much the agent moves in a specific direction with a single action.

A clear counterexample for this kind of scenarios is Angry Birds [9], where the player must provide an angle and a strength for shooting birds into intricate structures that must be destroyed. A more complex example is the car racing game TORCS (The Open Racing Car Simulator), and its associated competition [10]. In this game, steering left or right and throttling are continuous actions, as they can be specified in a range of floating point values (from -1 to 1 and from 0 to 1, respectively). Coupled with a continuous state space (such as position, velocity and orientation) and real-world physics, treating the action and state space with care was necessary in order to develop an efficient controller. Popular approaches used for controllers in this contest were the evolution of fuzzy logic drivers [11] and complex heuristics [12].

The idea of using macro-actions (from a simple action repetition to the design of more complex variants) has been used multiple times when the size of the state space makes search a very costly task. Pioneered in the early days of Reinforcement Learning [13], macro-actions have been used in Real-Time Strategy games like Wargus [14] (applying them to simultaneous moves of variable duration), the artificial game P-Game [15] and the card game Dou Di Zhu [16], where the authors split actions in several consecutive decisions in order to reduce the branching factor at the expense of tree depth.

Last but not least, macro-actions have also been used in the PTSP game mentioned above [7], [17], both for tree search and evolutionary techniques, as explored in this paper. In their work, the authors propose a simple repetition of actions as a way to coarse the search and provide a longer thinking time for the agent in this real-time game. Results showed that there was an optimal amount of times an action should be repeated to maximize performance: shorter macro-actions would not allow for an effective exploration of the search space, while longer ones did not provide the agent with enough precision to navigate through the maze efficiently. A similar approach has been followed in this work when applying this concept to the new GVGAI games, aiming to investigate if the findings there extrapolate to multiple games at once.

III. The General Video Game AI Framework

The General Video Game AI Framework [1] is a Java implementation of the original *py-vgdl* benchmark developed by Tom Schaul [18]. The framework reads games described in VGDL and presents an object oriented interface to an agent that is able to play it, without providing the rules, objectives or the meaning of the different sprites present in the game.

The agent receives information about the game state by means of a Java object, which allows the agent to query the game state (score, timesteps and victory conditions), the avatar state (player's position, velocity, orientation), and the positions of different sprites in the level. These are provided as observations, identified with an integer *id* for its (anonymized) type. In the competition version of the framework, agents must comply with a thinking budget of 40ms per action, plus an initialization phase of 1s at the beginning of the game. During this time, the agents can use a forward model to simulate the effects of applying an action in a copy of the current game state. In order to present a fair setup, agnostic of the machines used to run extensive tests, it is possible to limit the thinking budget as a maximum number of uses of the forward model.

The version of VGDL that was run during the 2016 competitions allows for the definition of 2-dimensional single and 2-player games, with *sprites* defined by a type, a 2D position and a square size. Interactions can occur between sprites when they collide, and their consequences are determined within the game rules and the application of *effects*. There exists a mapping between the concepts that need to be specified in VGDL and the sprites, effects and terminations that are implemented in the framework. New additions to the ontology need to be addressed in both sides of this mapping, and the ones added for this work are described in the following subsections. For more information about the previous implementation of the framework, the reader is referred to [1].

A. Towards Real-World Physics Games

This section describes the updates on the GVGAI's ontology, avatars, effects and terminations required to facilitate the creation of games with real-world physical properties.

1) Physics Ontology: The most important modification to the GVGAI Framework is the addition of a new continuous physics ontology, which can be used to allow sprites to move in a more fine-tuned way throughout any given level. Sprites can either be assigned to use these new continuous physics, or the old grid-based physics. Sprites that use the continuous physics require three new parameters to be defined for them: *mass*, *friction* and *gravity*. Whilst it is typical for gravity have the same value for all sprites within a game, it can also be set separately for each sprite type.

Much like the original grid physics, sprites with continuous physics undertake movement through the use of an `active movement` function, which takes into account the direction and speed of the sprite. Unlike sprites with grid physics, these directions are not limited to one of four options, and the speed is not defined at the grid level, but rather at the screen pixel level. This allows the sprite to move in any possible direction and with any possible speed.

Given the input of a speed and direction, the movement of the sprite is updated as follows. A vector for this input is calculated, using the latter value to determine its direction and the former value, multiplied by the sprite's mass, to determine its magnitude. This vector is then added to the the sprite's current velocity, in order to update the sprite's direction and speed. The sprite's current velocity is used to change the position of the sprite at every tick. Using this method means that sprites will suffer from inertia when moving, as new movement commands do not necessarily override previous ones, but are instead added together.

In addition, the sprite is subject to `passive movement`, which occurs automatically every game tick. This function may be used to both slow down the sprite, due to friction, and to add a downwards force on the sprite, due to gravity. If the sprite is affected by friction, then, every tick, the sprite's current speed is multiplied by $1 - f(s)$ (where $f(s)$ is the sprite's friction). If the sprite responds to gravity, a gravity vector $\overrightarrow{g}$ is added to the sprite's current movement vector at every tick. $\overrightarrow{g}$ points straight down and its magnitude is equal to the sprite's mass multiplied by the sprite's gravity.

2) New Sprites: This section describes the new avatar and other sprites created for the continuous physics games. The MovingAvatar, FlakAvatar, OrientedAvatar, ShootAvatar and Missile types mentioned, which some of the sprites described are based on, are part of the old GVGAI Framework [1].

- *LanderAvatar*: A special type of OrientedAvatar (avatar with a defined orientation) that can accelerate (action `Up`), decelerate (`Down`) and rotate (`Left` or `Right`). When the LanderAvatar accelerates, a set value is added to its speed, while its direction remains unchanged. Decelerating has a similar effect, but in the direction opposite to the sprite's current orientation. When the LanderAvatar rotates, it only changes its direction.
- *SpaceshipAvatar*: A special type of ShootAvatar similar to the LanderAvatar, but which can also spawn new sprites (`Use`). Sprites spawned have the same direction as the avatar.
- *CarAvatar*: A special type of OrientedAvatar that can move forwards (`Up`), backwards (`Down`) and turn (`Left` and `Right`). The CarAvatar must be moving either forwards or backwards at all times. The CarAvatar has a continuous movement of a set speed, with the direction being determined by its current orientation and whether it is moving forwards or backwards. When the CarAvatar rotates, it only changes its direction.
- *AimedAvatar*: A special type of ShootAvatar that can only rotate (`Left` and `Right`) and spawn a new sprite (`Use`). When the AimedAvatar rotates, its direction changes. Sprites spawned have the same direction as the avatar.
- *BirdAvatar*: A special type of OrientedAvatar that only has one action, jump (`Use`). When the BirdAvatar jumps, a vertical upwards movement of a set speed is applied.
- *PlatformerAvatar*: A special type of MovingAvatar that can move sideways (`Left` and `Right`) and jump (`Use`). The PlatformerAvatar is defined as "on the ground" if there is at least one sprite of a pre-defined set directly below it. When the PlatformerAvatar moves either left or right, a vector with a set speed and the desired direction is added to its current movement vector. This speed is greater when the PlatformerAvatar is on the ground. When the PlatformerAvatar jumps, a vertical upwards movement of a set speed is applied, if and only if the PlatformerAvatar is "on the ground".
- *WizardAvatar*: A special type of MovingAvatar similar to the PlatformerAvatar, but which can also spawn new sprites (`Down`). Sprites spawned have the same vertical position as the avatar and are placed either to the left or right of the avatar, based on whether it last moved left or right.
- *Walker*: A special type of Missile that moves either left or right. When the Walker is prevented from moving in its current direction, it reverses its direction.
- *WalkerJumper*: A special type of Walker for which, each tick, there is a random chance that a movement vector with a set speed in the upwards direction will be applied.

3) New Interactions: This section describes the new Interactions that were created for the continuous physics games.

- *WallStop*: Sets either the horizontal or vertical component of the first sprite's movement vector to zero, depending on the relative position of the second sprite.
- *WallBounce*: Multiplies the horizontal or vertical component of the first sprite's movement vector by -1, depending on the relative position of the second sprite.
- *BounceDirection*: Sets the direction of the movement vector of the first sprite to the direction of the vector formed by the centre points of the two sprites.
- *KillIfNotUpright*: Kills the first sprite if the difference between its current rotation and $\frac{3}{2}\pi$ (upwards direction) is greater than a set value.

4) New Games: Table I shows the new set of games that make use of the features described above and constitute the set employed for the experiments presented in this paper.

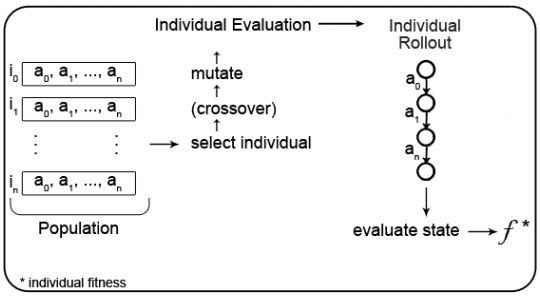

Figure 1: Rolling Horizon Evolutionary Algorithm steps.

These games present challenges at multiple levels, such as platformers (*Mario*, *Candy*), shooters (*Asteroids*), games that require long term planning (*PTSP*) and classic arcade games which require fast reaction times (*Lander* or *Pong*).

IV. Agents

This section describes the agents employed in this study, together with the macro-action handler and the value function used for both approaches.

A. Rolling Horizon Evolutionary Algorithm

Rolling Horizon Evolutionary Algorithms (RHEA) [19] are a type of EAs where individuals represent sequences of L actions (or action plans). The forward model (FM) provided by the GVGAI framework is used to evaluate the individuals by executing all the actions in the sequence. The final fitness is provided by the value function (see Section IV-C2) that evaluates the state reached at the end of the plan.

Figure 1 shows the basics steps of the RHEA agent. The initial population is formed of one or more individuals that are initialized at random (although seeding strategies have been explored in the domain of GVGAI [20]). Traditional genetic operators, such as selection, crossover and mutation are applied to generate new individuals, which, after being evaluated, may be kept in the population if their fitness is among the best N (with N being the population size).

After the budget has been consumed, the best individual of the population is identified and its first action returned as the move to make in the game. For more details about this algorithm and improvements on the vanilla version of this agent in GVGAI (which is the one used in this work), the reader is referred to [19], [20].

B. Monte Carlo Tree Search

Hundreds of research articles have been written about Monte Carlo Tree Search (MCTS) and its application to games. Not surprisingly so, as it has traditionally been one of the most efficient techniques to build players for the game of Go [21], and it has shown a high proficiency in General Game [22] and General Video Game Playing [23].

MCTS is a search technique that builds an asymmetric tree by performing random sequences of actions and gathering statistics about the actions and states visited [24]. The main algorithm is divided into four steps: *Selection* is guided by a tree policy, choosing actions from the root of the tree down to a non fully expanded node, balancing exploitation of the best actions and exploration of the others. Once a non fully expanded node has been reached, a new node is added as a child of this one to the tree and a random roll-out (or sequence of actions) is executed up to a certain depth L. The state reached at this point is evaluated with an heuristic (see Section IV-C2), and the visit count and average reward are updated in all visited nodes of the tree during this iteration.

As MCTS is an anytime algorithm, it can be stopped when the thinking budget is exhausted, and it returns the best action (in terms of average reward, number of visits, or other recommendation policies) from the root to be executed in the real game. For more information about the algorithm, variants and applications, the reader is referred to [24].

C. Macro-Actions and Value Function

Both algorithms employed in this study share two main components: the heuristic (or value function used to evaluate a given state) and the way actions are grouped in macro-actions.

1) Macro-Actions: We define *macro-action* as a sequence or repetition of an action, $\langle a_1, \ldots, a_m \rangle$, during M steps. Executing a macro-action consists of playing the sequence of actions contained within it. It is possible to argue that the macro-actions employed in this study are the simplest possible, but previous works on this area showed that this can be an effective method [17].

The benefits of using macro-actions is two-fold. Firstly, they permit the execution of longer evaluations before making the decision of the action to execute. While in a single-action scenario, the algorithm has 40ms to decide the next action, macro-actions of size M have $M \times 40$ms to choose a move. The reason is that the previous macro-action also needed M time steps to be executed[2]. Note that, for every time step, only one action (the next in the current macro-action) is returned and executed in the real game.

Secondly, the size of the problem is reduced and the ability to perform forward planning by the algorithms tested is improved, at the expense of losing granularity on the decision process. However, modifying a single action in a given sequence may not produce a big impact, while it may bring a better performance because of the farther lookahead.

The experiments described in this paper employ different macro-action lengths M, and a distinct number of macro-actions per tree play-out (for MCTS) or individual evaluation (RHEA). The number of macro-actions is referred to in this paper as L, thus the total algorithm lookahead is $L \times M$. Algorithm 1 shows the macro-action handler, common for both MCTS and RHEA.

2) Value Function: As the focus of the experimental work of this paper is set on the search capabilities of the algorithms, the heuristic function to evaluate states has been chosen in terms of simplicity. The value of the state is calculated as the score of the game at that given moment, plus a high number

[2]With the exception of the very first move, with no previous action.

Game	Description
Artillery	An AimedAvatar which shoots bullets affected by gravity. These bullets can break or bounce off certain special sprites and they can push boulders. The player wins if they are able to hit (and kill) all the devils within the level, with either bullets or boulders. They lose when the timer runs out. 1 point is given for every devil killed.
Asteroids	A SpaceShipAvatar which shoots bullets. The bullets can break certain special sprites and kill aliens. The player wins if they are able to kill all aliens within the level and loses if they crash into a sprite. 1 point is given for every alien killed.
Bird	A BirdAvatar affected by gravity. The player wins if the avatar touches the green section of the level (goal) and loses if they touch a red section (pipe). 1 point is given for every coin collected.
Bubble	A FlakAvatar (sideways movement only) which shoots bullets upwards. The bullets can be used to split bubbles that bounce around the level. Once a bubble reaches a small enough size, it will be destroyed when hit, rather than splitting. The player wins if they remove all bubbles from the level and loses if they touch a bubble. 1 point is given for every bullet that hits a bubble.
Candy	A WizardAvatar which can create or destroy blocks that are directly in front of it. The avatar can jump on top of these blocks to reach certain locations. There are enemies which will move back and forth along the ground. The player wins if they touch the goal and loses if they touch an enemy or fall outside of the level space. 1 point is given for each egg collected.
Lander	A LanderAvatar which is affected by gravity. The player wins if the avatar touches a blue section with a slow enough speed while facing upwards, and loses if they touch any other object. No points are given.
Mario	A PlatformerAvatar which is affected by gravity. Certain sections of the floor can carry the avatar and other objects upwards, similar to an elevator. There are zombies of sprite type Walker and sharks of type WalkerJumper. Zombies and sharks can both be killed by the avatar jumping on top of them. Boulders can be pushed to allow the player to cross fire. The player wins if they touch the mushroom and loses if they touch either an enemy (unless from above), fire, or fall off the level space. 1 point is given for each coin collected or enemy killed.
Pong	An avatar which can only move up or down. There are balls that bounce off the walls and the avatar when they hit these sprite types. The player wins if a ball touches the green section on the left of the level. They lose if a ball touches the green section on the right of the level. No points are given.
PTSP	An OrientedAvatar (only movement). There are enemy aliens that move randomly around the level. The player wins if they collect all the green orbs and loses if they touch an alien. 1 point is given for every green orb collected.
Racing	A CarAvatar. The player wins if they collect all blue squares and loses if they touch a red section of the level. 1 point is given for every blue square collected.

Table I: Games in the first continuous physics set of the GVGAI Competition.

Algorithm 1 Algorithm to handle macro-actions (from [17]).

```
1:  function GetAction(GameState : gs)
2:      if IsGameFirstAction(gs) then
3:          actionToRun ← DecideMacro(gs)
4:      else
5:          for i = 0 → remainingActions do
6:              gs.Advance(actionToRun)
7:          if remainingActions > 0 then
8:              if resetAlgorithm then
9:                  Algorithm.Reset(gs)
10:                 resetAlgorithm ← false
11:             Algorithm.NextMove(gs)
12:         else                          ▷ remainingActions is 0
13:             actionToRun ← DecideMacro(gs)
14:     remainingActions = (remainingActions − 1)
15:     return actionToRun
16:
17: function DecideMacro(GameState : gs)
18:     actionToRun ← Algorithm.NextMove(gs)
19:     remainingActions ← M
20:     resetAlgorithm ← true
21:     return actionToRun
```

(10^6) if the game is finished in a victory for the agent (-10^6 in case of a loss).

V. Experimental Work

An extensive experimental work has been put in place to study the performance of the algorithms described in Section IV in the games detailed in Table I. These algorithms are MCTS and RHEA, although a deeper study has been performed in the latter case, varying the size of the population for the EA. The population sizes P tried are 1, 5 and 10, and results are reported for these agents as RHEA-1, RHEA-5 and RHEA-10, respectively.

Different lookaheads have been explored for all algorithms, aiming to explore how farther simulations into the future affect the victory rates on these games. The lookahead values employed are 30, 60, 90 and 120 steps from the current game state. These lookahead values are reached by a combination of macro-action length M and simulation depth L (or individual length for the RHEA agents), and different configurations have been explored to measure the effect of M. Additionally, an extra configuration has been run for all controllers, where no macro-actions and a simulation depth of 10 has been used. This is a useful setting for comparisons, as it is the default depth of the controllers as provided in the GVGAI framework.

- $L \times M = 10$; $(L, M) = (10, 1)$.
- $L \times M = 30$, with: $(30, 1)$,[3] $(5, 6)$, $(3, 10)$, $(2, 15)$.
- $L \times M = 60$, with: $(60, 1)$, $(6, 10)$, $(4, 15)$, $(2, 30)$.
- $L \times M = 90$, with: $(90, 1)$, $(9, 10)$, $(6, 15)$, $(3, 30)$.
- $L \times M = 120$, with: $(120, 1)$, $(12, 10)$, $(8, 15)$, $(4, 30)$.

Each one of these configurations is tested with our 4 agents, playing 100 times per game: 20 repetitions of the 5 levels in the 10 available games. A budget of 900 forward model calls is assigned for each game tick an agent must return an action.

[3]Note that the first configuration per lookahead is equivalent to no macro-actions but longer individual length or simulation depth.

Game	RHEA-1	RHEA-5	RHEA-10	MCTS
Artillery	39.00 (4.88)	35.00 (4.77)	32.00 (4.66)	41.00 (4.92)
Asteroids	3.00 (1.71)	10.00 (3.00)	17.00 (3.76)	31.00 (4.62)
Bird	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
Bubble	23.00 (4.21)	53.00 (4.99)	72.00 (4.49)	77.00 (4.21)
Candy	3.00 (1.71)	2.00 (1.40)	2.00 (1.40)	4.00 (1.96)
Lander	0.00 (0.00)	2.00 (1.40)	3.00 (1.71)	8.00 (2.71)
Mario	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
Pong	69.00 (4.62)	68.00 (4.66)	63.00 (4.83)	67.00 (4.70)
PTSP	1.00 (0.99)	4.00 (1.96)	3.00 (1.71)	4.00 (1.96)
Racing	30.00 (4.58)	52.00 (5.00)	55.00 (4.97)	66.00 (4.74)
Total	16.80 (2.27)	22.60 (2.72)	24.70 (2.75)	29.80 (2.98)

Table II: Results of the controllers with their default $L = 10$.

A. Average Victory Rates

Table II shows the results with the default depth of 10 single actions. Attending to the last row of the table (average victory rate over 100 games played, plus standard error between brackets), it seems MCTS perform better than the rolling horizon methods on this setting and that longer higher population sizes help RHEA to achieve a better performance, very close to MCTS in $P = 10$. Furthermore, these results offer a glimpse in the complexity of having games of different nature. Most of the games are dominated by MCTS, although in some games the $P = 10$ version of RHEA achieves a similar performance. However, other games like *Bird*, *Lander*, *Mario* and *PTSP* have a very low victory rate, in some cases with no games won in none of the configurations.

Table III shows the results of all configurations averaged across games, for all the runs that span across macro-action configurations, lookaheads and agents. The first conclusion quickly observed is that the results with $M = 1$ are far better than their counterparts. This is true for most cases, and, in fact, there is a regular trend that can be observed showing victory rates dropping as the length of the macro-action increases.

There are other interesting remarks to make about these results, especially in comparison with Table II. In general, it can be observed that RHEA usually improves performance when adding macro-actions of small lengths (second column of Table III), with the level of improvement decreasing as M gets higher. Regarding the change in population size for RHEA, it can be seen that performance typically increases when P is higher (which has also been observed in [19]), but only if the length of the macro-action is short. Larger populations with longer macro-actions reduce the performance of the algorithm to its worst results in this comparison. These results seem to suggest that a finer control for macro-actions benefit the agents at gameplay.

Similarly, MCTS also obtains the best average results with shorter macro-actions, typically $M = 1$ and $M = 10$, and results get worse when M increases beyond that point. It would be easy to rule out the macro-actions idea if it was not because of the improvement made in RHEA, but also if the results on a game by game basis were ignored. Next, a deeper analysis considering the results per game is given.

L × M:	30×1	5×6	3×10	2×15
RHEA-1	14.50 (2.10)	21.80 (3.29)	19.60 (3.16)	14.60 (2.46)
RHEA-5	31.60 (3.12)	32.90 (4.18)	22.40 (3.51)	14.70 (2.62)
RHEA-10	**36.50 (3.15)**	28.80 (4.06)	20.40 (3.43)	14.10 (2.67)
MCTS	**46.00 (3.02)**	**48.40 (4.28)**	**41.30 (3.72)**	20.50 (2.74)
L × M:	60×1	6×10	4×15	2×30
RHEA-1	14.10 (1.93)	22.70 (3.30)	19.10 (2.88)	11.80 (2.15)
RHEA-5	33.40 (3.05)	31.50 (4.08)	27.30 (3.74)	12.20 (2.42)
RHEA-10	**39.50 (3.10)**	24.90 (3.73)	23.00 (3.62)	10.00 (2.23)
MCTS	**46.90 (3.01)**	**46.40 (4.40)**	26.70 (2.60)	5.10 (1.45)
L × M:	90×1	9×10	6×15	3×30
RHEA-1	12.00 (2.11)	19.40 (3.09)	17.70 (2.88)	14.00 (2.31)
RHEA-5	32.80 (3.06)	30.90 (4.04)	28.70 (3.76)	17.70 (2.74)
RHEA-10	**37.30 (3.10)**	25.70 (3.89)	24.60 (3.70)	13.70 (2.52)
MCTS	**46.90 (3.08)**	**45.80 (4.56)**	25.90 (3.28)	9.00 (1.82)
L × M:	120×1	12×10	8×15	4×30
RHEA-1	14.10 (2.13)	20.50 (3.18)	17.10 (2.79)	12.80 (2.14)
RHEA-5	33.40 (3.07)	30.50 (3.92)	27.30 (3.64)	20.00 (2.74)
RHEA-10	**36.00 (3.03)**	25.10 (3.69)	24.20 (3.53)	15.90 (2.64)
MCTS	**44.40 (3.14)**	**48.40 (4.55)**	24.40 (3.15)	11.00 (2.15)

Table III: Results for all algorithms and configurations. Indicated values are the average of victories across all games, with the standard error between brackets. Results in bold mark the best performances for RHEA and MCTS.

B. Game Victory Rates

When looking at the results per game from the experimental work described here, it is noticeable that there is a strong dependency on the game in terms of algorithm, lookahead and the use of macro-actions. Table IV shows the results for configuration $L \times M = 30$ in all games separately. Results for other configurations are not included here for the sake of space, but similar trends showcase in all cases.

In many games, as expected after presenting the overall results, average victory rates are higher with macro-action lengths of $M = 1$ and $M = 10$. However, some games show a different behaviour. The most interesting result is the performance shown for those games that were unable to achieve a decent number of victories with no macro-actions. As seen previously in Table II, these games are *Bird*, *Lander*, *Mario* and *PTSP*, and in all of them the use of macro-actions increases the victory rate in a greater or lesser extent. For instance, in *Bird*, *Lander* and *PTSP*, the use of macro-actions improves the performance from approximately 0% victory rate to values between 30% and 50% in different settings. Both games present scenarios where a longer lookahead may be beneficial, but not without a finer degree of control. A lookahead of 30 (rather than 10, as in the results of Table II) does not guarantee an improvement in performance, but using macro-actions to simultaneously reduce the search space does (i.e. RHEA-5 increasing to 45% of victories). In fact, it's not surprising that *PTSP* is a game where macro-actions work well, it was particularly in that game where these macro-actions were originally introduced [7]!

There is also a small performance raise in *Mario*, in this case the victory rates increase from 0% to less than 20%. Although this is a noticeable improvement, it's smaller than in other

Algorithm:	**RHEA-1**				**RHEA-5**			
L × M:	30 × 1	6 × 5	3 × 10	2 × 15	30 × 1	6 × 5	3 × 10	2 × 15
Artillery	**34.00 (4.74)**	12.00 (3.25)	3.00 (1.71)	3.00 (1.71)	**52.00 (5.00)**	11.00 (3.13)	3.00 (1.71)	6.00 (2.37)
Asteroids	0.00 (0.00)	***3.00 (1.71)***	***2.00 (1.40)***	0.00 (0.00)	**36.00 (4.80)**	19.00 (3.92)	0.00 (0.00)	0.00 (0.00)
Bird	0.00 (0.00)	***22.00 (4.14)***	***30.00 (4.58)***	***15.00 (3.57)***	0.00 (0.00)	**45.00 (4.97)**	22.00 (4.14)	13.00 (3.36)
Bubble	18.00 (3.84)	**37.00 (4.83)**	27.00 (4.44)	13.00 (3.36)	**68.00 (4.66)**	45.00 (4.97)	35.00 (4.77)	14.00 (3.47)
Candy	1.00 (0.99)	**4.00 (1.96)**	0.00 (0.00)	0.00 (0.00)	6.00 (2.37)	**22.00 (4.14)**	7.00 (2.55)	0.00 (0.00)
Lander	1.00 (0.99)	***7.00 (2.55)***	***7.00 (2.55)***	0.00 (0.00)	4.00 (1.96)	**29.00 (4.54)**	16.00 (3.67)	0.00 (0.00)
Mario	0.00 (0.00)	0.00 (0.00)	***5.00 (2.18)***	***4.00 (1.96)***	0.00 (0.00)	2.00 (1.40)	**16.00 (3.67)**	8.00 (2.71)
Pong	***60.00 (4.90)***	***61.00 (4.88)***	42.00 (4.94)	52.00 (5.00)	**79.00 (4.07)**	65.00 (4.77)	45.00 (4.97)	35.00 (4.77)
PTSP	1.00 (0.99)	***36.00 (4.80)***	***42.00 (4.94)***	***35.00 (4.77)***	13.00 (3.36)	***50.00 (5.00)***	***48.00 (5.00)***	***42.00 (4.94)***
Racing	30.00 (4.58)	36.00 (4.80)	38.00 (4.85)	24.00 (4.27)	**58.00 (4.94)**	41.00 (4.92)	32.00 (4.66)	29.00 (4.54)
Algorithm:	**RHEA-10**				**MCTS**			
L × M:	30 × 1	6 × 5	3 × 10	2 × 15	30 × 1	6 × 5	3 × 10	2 × 15
Artillery	**57.00 (4.95)**	13.00 (3.36)	9.00 (2.86)	3.00 (1.71)	**51.00 (5.00)**	33.00 (4.70)	4.00 (1.96)	0.00 (0.00)
Asteroids	**48.00 (5.00)**	12.00 (3.25)	1.00 (0.99)	1.00 (0.99)	**85.00 (3.57)**	36.00 (4.80)	24.00 (4.27)	4.00 (1.96)
Bird	0.00 (0.00)	**36.00 (4.80)**	22.00 (4.14)	8.00 (2.71)	0.00 (0.00)	30.00 (4.58)	27.00 (4.44)	20.00 (4.00)
Bubble	**79.00 (4.07)**	28.00 (4.49)	17.00 (3.76)	12.00 (3.25)	**97.00 (1.71)**	50.00 (5.00)	87.00 (3.36)	38.00 (4.85)
Candy	6.00 (2.37)	**17.00 (3.76)**	5.00 (2.18)	1.00 (0.99)	9.00 (2.86)	**56.00 (4.96)**	8.00 (2.71)	1.00 (0.99)
Lander	10.00 (3.00)	**17.00 (3.76)**	9.00 (2.86)	0.00 (0.00)	49.00 (5.00)	**58.00 (4.94)**	35.00 (4.77)	0.00 (0.00)
Mario	0.00 (0.00)	***8.00 (2.71)***	***9.00 (2.86)***	***8.00 (2.71)***	0.00 (0.00)	***18.00 (3.84)***	***14.00 (3.47)***	1.00 (0.99)
Pong	**80.00 (4.00)**	69.00 (4.62)	52.00 (5.00)	35.00 (4.77)	80.00 (4.00)	**100.00 (0.00)**	94.00 (2.37)	62.00 (4.85)
PTSP	15.00 (3.57)	***48.00 (5.00)***	***47.00 (4.99)***	***40.00 (4.90)***	16.00 (3.67)	47.00 (4.99)	**61.00 (4.88)**	36.00 (4.80)
Racing	**70.00 (4.58)**	40.00 (4.90)	33.00 (4.70)	33.00 (4.70)	**73.00 (4.44)**	56.00 (4.96)	59.00 (4.92)	43.00 (4.95)

Table IV: Results per game in the configuration $L \times M = 30$. Averages and standard errors of the measures indicated in bold when better than the other variants (*italics* where the best result is shared).

cases. A possible reason for this is that *Mario* requires longer planning in conjunction with fast reaction to avoid enemies and hazards in the levels. A deeper research in platformer games may be required to tackle this type of games in a general context, which is also a new kind in the GVGAI framework.

Dependency on algorithms and macro-action lengths is also present in other games. For instance, in the game *Bird* (see Figure 2), there's a clear difference between using distinct lookaheads and macro-action lengths. The best performance is achieved when the lookahead is 90 and 120, with 60% victory rate, achieved by MCTS. However, all RHEA approaches are better than MCTS with the shorter lookahead ($L \times M = 30$), and $M = 1$ shows practically no victories for all agents.

VI. Conclusions and Future Work

This paper has introduced a new type of physics for the General Video Game AI (GVGAI) Framework, which aim to enhance the possibilities of the benchmark to reproduce real-world physics. Elements such inertia, gravity or friction are now part of the framework and a set of new games has been presented with some initial experimentation. The effects of these elements may have an impact on the way agents play these games, thus an initial study has been performed and presented in this paper with several agents. Additionally, this work extends the GVGAI agents to include the concept of macro-actions. An extensive experimental study has been conducted and reported here, in order to analyze the behaviour of two different agents, Rolling Horizon Evolutionary Algorithm (RHEA) and Monte Carlo Tree Search (MCTS), with and without macro-actions and different lookaheads.

One of the main conclusions that can be drawn from the results is that there is no algorithm that dominates all the others. This finding is typically seen in GVGAI, as the best configurations depend highly on the game at stake. In the default case (with no macro-actions and shorter simulation depths), MCTS performs better on average in the games of this set. The use of macro-actions boosts the performance of both algorithms, and it is worthwhile highlighting that in some of these cases (4 out of the 10 games used) victories can't be achieved without macro-actions. Generally, shorter macro-actions produce better results than longer ones. Additionally, it seems plausible that macro-actions help the algorithms perform better in games that are more complex because of a need of more fine tuned control. However, there's the necessity of finding the appropriate value of the macro-action length for each game, as, depending on the game characteristics, finer or coarser control may be preferred.

The results suggest that it would be interesting to investigate how to decide the appropriate macro-action length in-game, given that it is not possible to provide a general length that works well in all scenarios. This would work on the lines of meta-heuristic controllers, which have been showcased in winners of the past editions of the GVGAI Competition [25]. It seems sensible that developing a meta-agent able to tweak its internal parameters and algorithms based on the game played should be the natural next step of this research.

But there are also different avenues for future work. For instance, no work has been done in the use of macro-actions in grid-physics games in GVGAI, although it seems natural that different macro-action lengths will be also needed per game. It is our intuition (corroborated by some preliminary tests) that the lengths would actually vary significantly in the grid-physics games, as only very short macro-actions have shown to provide good results. Although it is possible that better

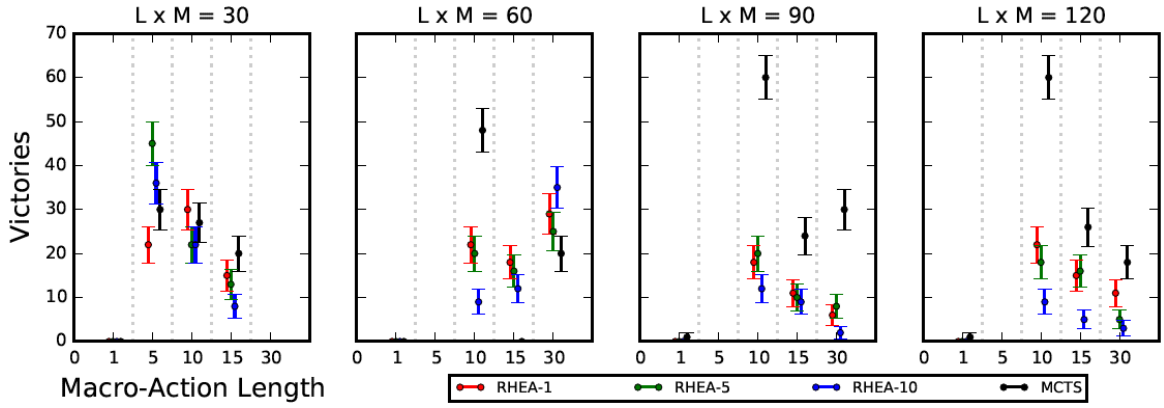

Figure 2: Victory rate (with standard error bars) in the game *Bird*, for all algorithms and configurations.

results can be obtained with macro-actions more complex than a simple repetition of actions. Ideally, an approach that could bring good performance to the agents would be one where each action is more involved, performing moves such as path-finding to the closest sprite of a given type, escape from a given location, or simply provide a specific sequence of actions derived from the current state of the game.

Last but not least, the addition of real-world physics may still be enhanced, for instance adding continuous actions for the agent. In this cases, the controllers would provide an action within a given range (i.e. $[-1, 1]$ for steering, or $[0, 1]$ for throttling in a racing game), which would open the framework to not only a new dimension of games, but also adding interesting aspects to the other tracks of the competition [25].

Acknowledgments

This work was partly funded by the EPSRC Centre for Doctoral Training in Intelligent Games and Game Intelligence (IGGI) EP/L015846/1.

References

[1] D. Perez-Liebana, S. Samothrakis, J. Togelius, T. Schaul, S. Lucas, A. Couetoux, J. Lee, C.-U. Lim, and T. Thompson, "The 2014 General Video Game Playing Competition," in *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 8, no. 3, 2016, pp. 229–243.

[2] R. D. Gaina, D. Perez-Liebana, and S. M. Lucas, "General Video Game for 2 Players: Framework and Competition," in *Proceedings of the IEEE Computer Science and Electronic Engineering Conf.*, 2016.

[3] J. Togelius, N. Shaker, S. Karakovskiy, and G. N. Yannakakis, "The Mario AI Championship 2009-2012," *AI Magazine*, vol. 34, no. 3, pp. 89–92, 2013.

[4] M. Nicolau, D. Perez-Liebana, M. O'Neill, and A. Brabazon, "Evolutionary Behavior Tree Approaches for Navigating Platform Games," *IEEE Trans. on Computational Intelligence and AI in Games*, 2016.

[5] D. Perez, P. Rohlfshagen, and S. M. Lucas, "The Physical Travelling Salesman Problem: WCCI 2012 Competition," in *IEEE Congress on Evolutionary Computation (CEC)*, 2012, pp. 1–8.

[6] D. Perez, E. Powley, D. Whitehouse, S. Samothrakis, S. Lucas, and P. I. Cowling, "The 2013 Multi-Objective Physical Travelling Salesman Problem Competition," in *IEEE Congress on Evolutionary Computation (CEC)*, 2014, pp. 2314–2321.

[7] D. Perez, E. J. Powley, D. Whitehouse, P. Rohlfshagen, S. Samothrakis, P. I. Cowling, and S. M. Lucas, "Solving the Physical Traveling Salesman Problem: Tree Search and Macro Actions," *IEEE Trans. on Computational Intelligence and AI in Games*, vol. 6:1, pp. 31–45, 2014.

[8] R. Prada, P. Lopes, J. Catarino, J. Quiterio, and F. S. Melo, "The Geometry Friends Game AI Competition," in *IEEE Conference on Computational Intelligence and Games (CIG)*, 2015, pp. 431–438.

[9] J. Renz *et al.*, "AIBIRDS: The Angry Birds Artificial Intelligence Competition," in *AAAI*, 2015, pp. 4326–4327.

[10] D. Loiacono *et al.*, "The 2009 Simulated Car Racing Championship," *IEEE Trans. on Computational Intelligence and AI in Games*, vol. 2, no. 2, pp. 131–147, 2010.

[11] D. Perez, G. Recio, and Y. Saez, "Evolving a Fuzzy Controller for a Car Racing Competition," in *IEEE Symposium on Computational Intelligence and Games (CIG)*, 2009, pp. 263–270.

[12] E. Onieva, D. A. Pelta, J. Godoy, V. Milanés, and J. Pérez, "An Evolutionary Tuned Driving System for Virtual Car Racing Games: The AUTOPIA Driver," *International Journal of Intelligent Systems*, vol. 27:3, pp. 217–241, 2012.

[13] R. S. Sutton, D. Precup, and S. Singh, "Between MDPs and Semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning," *Artificial intelligence*, vol. 112, no. 1-2, pp. 181–211, 1999.

[14] R.-K. Balla and A. Fern, "UCT for Tactical Assault Planning in Real-Time Strategy Games," in *IJCAI*, vol. 40, 2009, p. 45.

[15] G. Eyck and M. Müller, "Revisiting Move Groups in Monte-Carlo Tree Search," in *Springer Advances in Computer Games*, 2011, pp. 13–23.

[16] E. J. Powley, D. Whitehouse, and P. I. Cowling, "Determinization in Monte-Carlo Tree Search for the Card Game Dou Di Zhu," *Proc. Artif. Intell. Simul. Behav*, pp. 17–24, 2011.

[17] D. Perez, S. Samothrakis, S. Lucas, and P. Rohlfshagen, "Rolling Horizon Evolution Versus Tree Search for Navigation in Single-Player Real-Time Games," in *Proceedings of Genetic and Evolutionary Computation Conference*. ACM, 2013, pp. 351–358.

[18] 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.

[19] R. D. Gaina, J. Liu, S. M. Lucas, and D. Perez-Liebana, "Analysis of Vanilla Rolling Horizon Evolution Parameters in General Video Game Playing," in *Springer Lecture Notes in Computer Science, EvoApplications*, 2017.

[20] R. D. Gaina, S. M. Lucas, and D. Perez-Liebana, "Population Seeding Techniques for Rolling Horizon Evolution in General Video Game Playing," in *IEEE Congress on Evolutionary Computation (CEC)*, 2017, pp. 2314–2321.

[21] D. Silver *et al.*, "Mastering the Game of Go with Deep Neural Networks and Tree Search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[22] M. Genesereth, N. Love, and B. Pell, "General game playing: Overview of the AAAI competition," *AI magazine*, vol. 26, no. 2, p. 62, 2005.

[23] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The Arcade Learning Environment: An Evaluation Platform for General Agents," *J. Artif. Intell. Res.(JAIR)*, vol. 47, pp. 253–279, 2013.

[24] C. B. Browne *et al.*, "A Survey of Monte Carlo Tree Search Methods," *IEEE Trans. on Computational Intelligence and AI in Games*, vol. 4:1, pp. 1–43, 2012.

[25] D. Perez-Liebana, S. Samothrakis, J. Togelius, T. Schaul, and S. M. Lucas, "General video game ai: Competition, challenges and opportunities," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.