grail

delivered by

QED SOFTWARE

Our Approach to Truly Advanced Intelligent Behaviors in Games

Introduction

At QED Software we build advanced Artificial Intelligence. Our position at the intersection of academia and business allows us to identify opportunities for applying the latest achievements of science and technology and push the boundaries of innovation. One such area is enhancing business productivity by applying AI to video games. Indeed, truly advanced AI makes it possible to dramatically improve the experience for both players and game developers.

On the one hand, the improved gaming experience for the player adds to the overall success of the product; on the other hand, a faster, more error-resilient game development cycle allows for achieving quicker and cheaper time-to-market rates for new game products.

About QED Software

QED Software (www.qed.pl) is an AI products company and a technological enablement partner for firms that are pioneers in various areas. We turn applied AI research into production-ready solutions, and we have the expertise to embed them into large software frameworks.

Our products are used to reduce the uncertainty of Machine Learning (ML) processes, to align ML with business priorities, to support humans in the ML loop and also to reduce the ML footprint in intensive data environments.

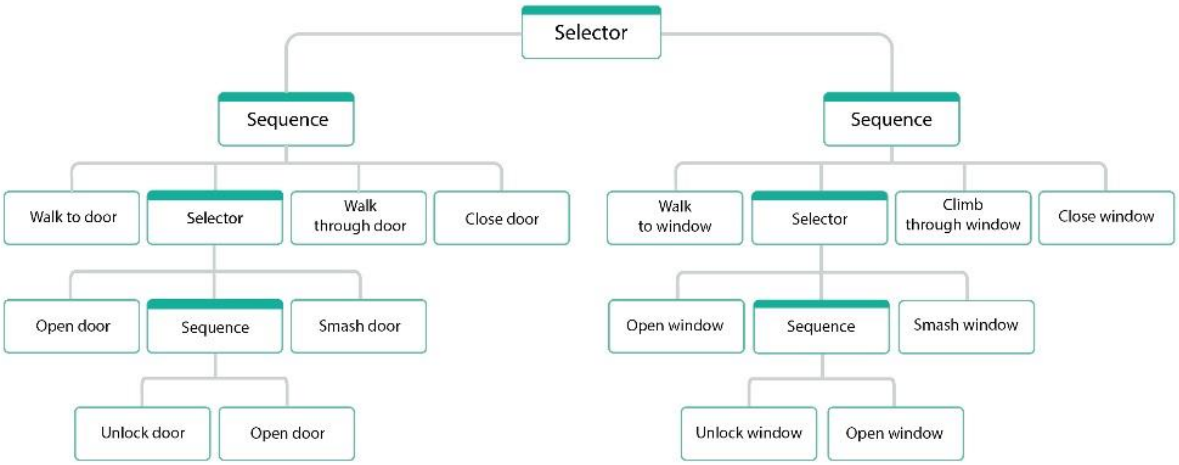
Our qualifications extend beyond solely ML to the broad AI fields of simulations, intelligent agents, approximate reasoning, and whatever comes next. This is why we can address a truly wide range of applications, from cybersecurity to entertainment, from BI to computer vision, and more.

Contents

1. What Do We Mean by Truly Advanced AI in Games?

The ultimate goal, the holy grail, of game development is to create the experience of believability and immersion for the human player. This is achieved through a wide range of techniques that pertain to the key aspects of game-playing: story, graphical fidelity, animations, music and sound effects, and voice acting, just to name a few. A big role in this mix is AI-controlled Non-Player Characters (NPCs, bots, or agents), that when poorly implemented, can break the immersion and impair user experience. The key is to design intelligent decision-making entities that will produce desired behaviors in complex virtual environments.

Typically, agent behaviors are implemented using branched conditional statements, forming so-called behavior trees. A behavior tree represents the reasoning process behind choosing a behavior. Behavior trees are great for simple AI-controlled agents. In more complex games, however, they tend to seem insufficient: they are hand-crafted, thus the game designer has to anticipate everything, and they encode lots of arbitrarily chosen conditions so are difficult to modify when the game changes. Behavior trees do not scale well with the game and they can grow very large. This all makes them hard to manage.



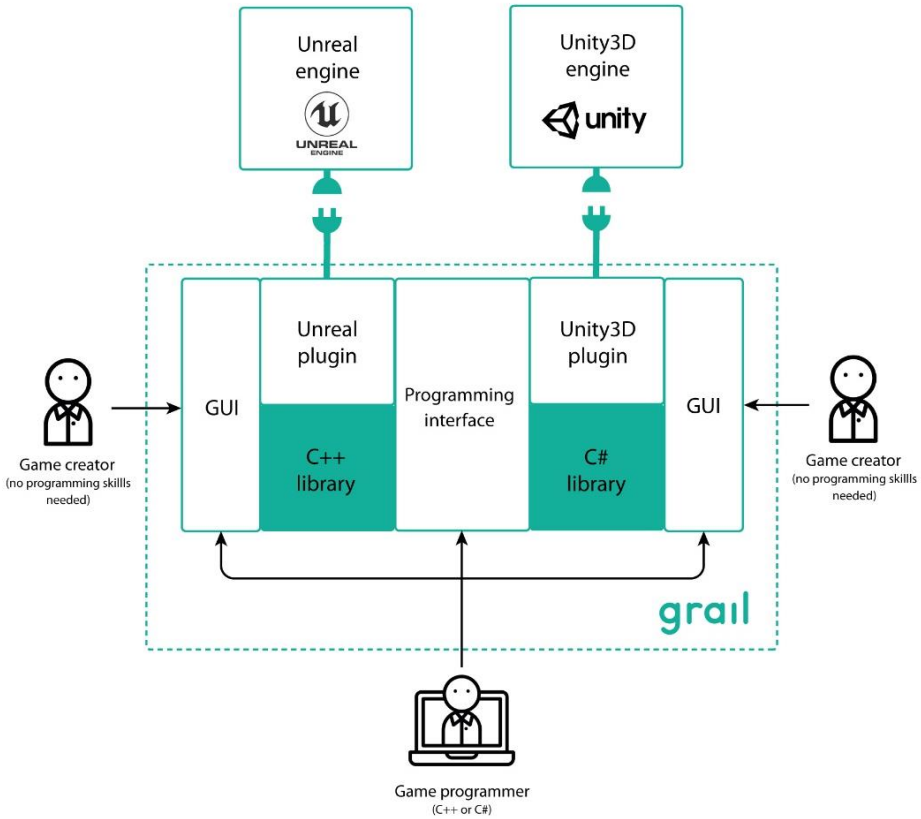
In the example above, the behavior tree represents an AI agent trying to enter a building. A **selector** node denotes the alternative options – it executes child nodes as long as the first one succeeds. In this example, the agent can either go through the door (the left subtree) or through the window (the right subtree). A **sequence** node must be executed from left to right. It fails as soon as its first child node fails. If a node fails, the tree execution returns to the parent node.

Nowadays, behavior trees are not really enough. We need to boost agents' perception, memory, and decision-making skills using truly advanced AI. To this end, we should leverage more refined techniques that are available to us thanks to scientific research. QED Software proposes a disruptive innovation to the standard decision-making AI technologies used in game development.

Our powerful tool, Grail, enables game developers to create advanced and nuanced intelligent behaviors and, at the same time, simplifies the development process. Importantly, there is no need to compromise between high quality AI and development ease. On the contrary – the paradigm shift that we propose provides better results for the end user (player) and at the same time facilitates the development process. This is science-fueled innovation in its purest form.

2. How Our System Works

Grail is an AI middleware for game development. You can think of it as a chip, implant, or plugin, that contains refined yet approachable AI techniques for implementing Advanced Intelligent Behaviors. Grail consists of a C++ code library , a C# library, and dedicated connectors for each of the two popular game engines: Unity and Unreal. As such, Grail can be plugged into any game development environment that interfaces with those two programming languages. In addition to the code and the plugins, we provide tools with a graphical user interface that can be used for configuration and debugging without the need for coding and recompiling of the game.



As a tool primarily used by game developers and quality assurance engineers, Grail provides two intuitive interfaces: a graphical user interface (GUI) and a programming interface of choice (C++ or C#).

Grail creators have been using Grail heavily since early development stages to implement advanced intelligent behaviors in Tactical Troops: Anthracite Shift, a game that was published in 2021.

There are two main Grail use cases. One is AI configuration during game creation, which can be achieved using the GUI and/or the programming interface. The other is live game AI debugging using the GUI.

Depending on the need, there are a few modes of working with Grail to choose from: using only the programming interface, using only the GUI, or using both. The most common modes are given in the table.

Modes of working with Grail (components used)	**Game language** (or language interface)	**Game engine**
C++ library	C++	any
C++ library + GUI tools	C++	any
C++ library + UE plugin	C++	Unreal Engine
C++ library + UE plugin + GUI tools	C++	Unreal Engine
C# library	C#	any
C# library + GUI tools	C#	any
C# library + U3D plugin	C#	Unity3D
C# library + U3D plugin + GUI tools	C#	Unity3D

3. Advanced Intelligent Behaviors in Grail – a Paradigm Shift

Unlike traditional approaches based on behavior trees or finite state machines, where the game developer explicitly declares what an intelligent entity should do under certain conditions, in Grail, the underpinning idea is to equip the entity with options and let a reasoning mechanism choose one of them at a particular moment. Due to the way a behavior is assessed in Grail (e.g. using mathematical curves instead of or in addition to conditional statements), agents behave meaningfully even in situations that the programmer/designer did not anticipate, even if the AI has to take many different factors into account. For example, available behaviors are given a score, and the next behavior is selected based on that score. An intelligent agent chooses the behavior that is most useful in a given place and moment. Still, legacy **behavior tree-based** agents can be smoothly integrated with **Grail-based** agents.

In technical terms, a behavior is an abstraction of an action an agent may take. Such action is an atomic piece of execution logic. A behavior may be as simple as changing a value of some variable or it can also be as complex as moving an army of units from one place to another. What is important is that the agent

will choose a behavior over some other candidate behaviors. But how? A component called a reasoner (the "brain" of an intelligent agent) is responsible for intelligent behavior selection. This is the heart of Grail, where truly advanced AI technologies for intelligent behavior selection are implemented: **Utility-Based AI, Simulated Games AI, Planner AI** and **Evolutionary Optimizable Scripts.** Utility-Based AI is just one technique. Simulated Games AI is an example of another technique (implementing a Monte Carlo Tree Search algorithm), a technique of choice for combinatorial games, which has not often been applied in video games due to the high complexity and memory consumption. Our original adaptation of this technique to the specifics of video games in Grail aims to change this. Planner AI allows for a very flexible definition of actions using all the mechanisms of the chosen programming language: nested conditionals, loops, arithmetic operations, and a wide range of tools that are not typically available in off-the-shelf planning algorithms. The full power of Grail comes from the ability to combine several advanced AI techniques to equip artificial agents with truly Advanced Intelligent Behaviors[1].

4. Modularity, Communication and Hierarchy – Practical Aspects of Development

Grail has been designed and built by AI scientists, software engineers, and game developers, paying special attention to ease of use. Grail facilitates implementation of intelligent entities, communication between them, knowledge transfer (via shared blackboards), as well as logging and serialization. Thus, Grail is also a framework for creating multi-agent systems. Architecture-wise, our system consists of four main components:

AI Manager, which takes care of registered entities and shared blackboards. It is responsible for Grail's threading and update loop, which controls the order in which agents take actions. It provides sort and group entities features.

AI Entity: an AI-controlled agent is a basic object that can execute behaviors. Behaviors might be provided by the user via a setter or by an assigned reasoner. An AI Entity may correspond to a physical character in the game, such as a soldier in a shooter game, but it can also be virtual without any visible in-game manifestation. An AI Entity can choose one behavior at a given moment among one or more possible behaviors.

Reasoner is responsible for decision-making and assigning selected behavior to AI-controlled entities. It is possible to provide a custom implementation of the reasoner (e.g. behavior tree-based) or use one of the specific reasoners available in Grail.

4.1. Communication & Hierarchy

How Blackboard Architecture Facilitates Code Reusability

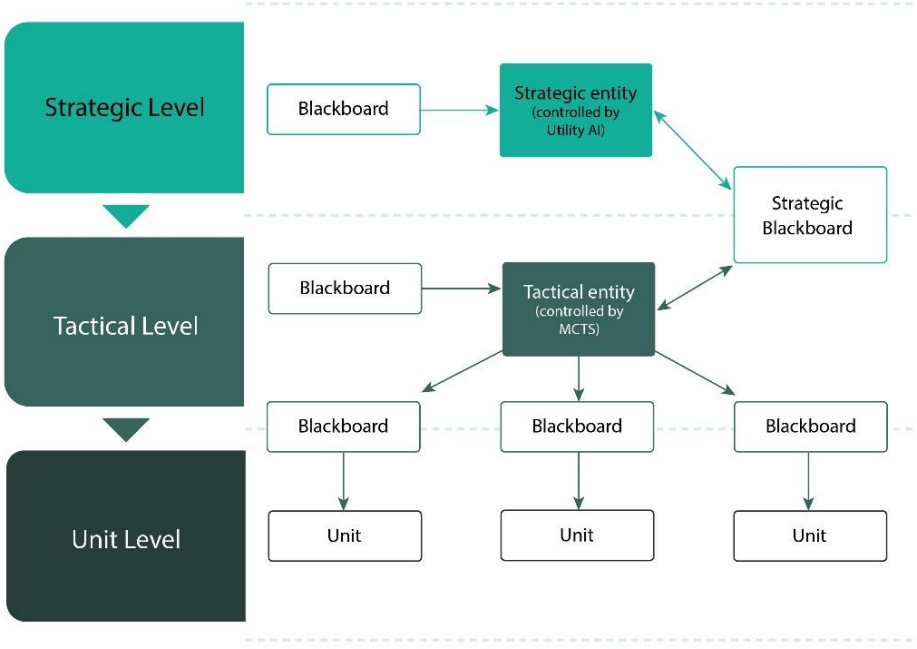
Using blackboards as a method of communication between agents allows for an easy replacement of individual character control modules, which in turn facilitates the reuse of the code by many types of opponents and further reduces programming time.
For example, in the Tactical Troops: Anthracite Shift game, there are three bot layers: strategic, tactical, and units. First, using Utility AI, the strategic layer selects high-level goals for each unit and puts it on a shared board. Then the tactical layer, after reading the board, constructs an adequate simplified game model and conducts MCTS simulations on it. The same bot configuration controls enemy groups trying to hunt down and kill a specific NPC, only the strategic layer here is replaced by a simple script writing to the same shared board.
This excludes the tactical layer from taking any action; it only expects a preserved communication protocol (i.e. the orders must be on the board, and it does not matter who delivers them).

Grail provides a built-in mechanism for exchanging data between entities based on Blackboards. Each entity manages one private Blackboard and may share any number of Blackboards with others.

An AI-controlled agent can be built in a hierarchical and component manner. It can consist of any number of components responsible for different aspects of behavior and arranged in any hierarchy, e.g.:

- high-level brain (strategy), arms (shooting) and legs (moving);
- minister (resource management), general (strategy), captain (squad tactics), unit (move, attack).

The system is flexible: you can, but you don't have to, create a hierarchy.



Implementing AI hierarchy in Grail using Blackboard Architecture

5. How Can Grail Help Game Developers?

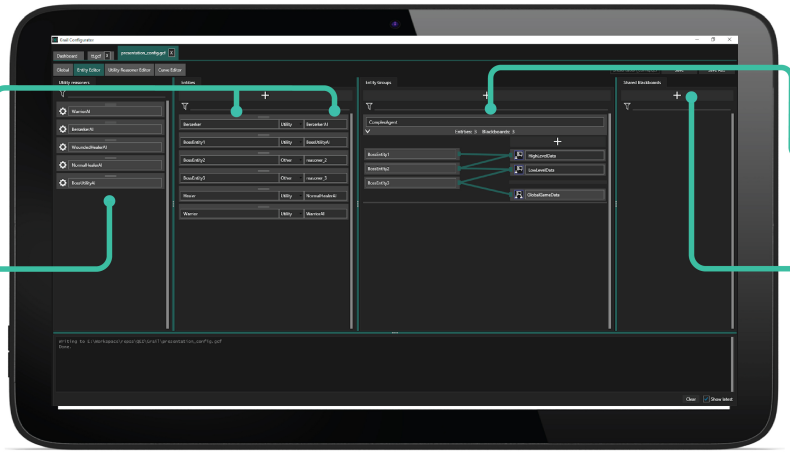
Grail enables developers to use sophisticated AI in implementing truly Advanced Intelligent Behaviors for AI-controlled bots. It has been designed and used by game developers with the following requirements in mind:

Grail has been designed and used by game developers.

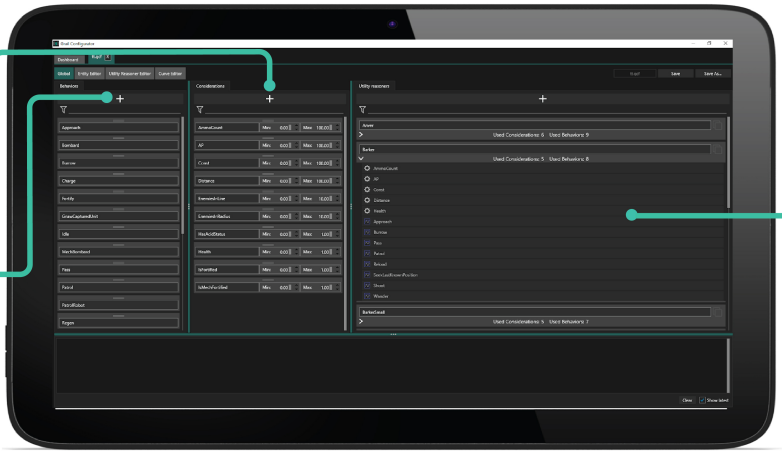
- Ease of modification during development. Almost at any time in the game, it is possible to replace the AI – either the entire bot or its individual components.
- Short time from making a change to seeing its result. Rich AI configuration options are available from the level of text files (as well as from the level of attached graphic tools) and do not require code compilation.

- A complete AI framework combined with a graphical configuration and debugging tool significantly reduces the amount of work required to obtain high quality AI in games.
- Facilitate code reuse via blackboards. Grail uses blackboards (local and shared) to model agent knowledge and the method of its exchange. The use of blackboard as a method of communication between agents allows for easy replacement of individual character control modules, facilitating the reuse of the same code by many types of opponents, which further reduces programming time.
- Lightweight. Can run on a severely limited CPU budget.
- Flexibility. Our framework does not make any hard assumptions about the decision-making algorithms used by AI-controlled bots. If necessary, the developer can easily integrate his or her own techniques (or techniques from other libraries) by implementing a custom reasoner. The library supports the construction of AI in such a way that it has a range of behaviors that are dynamically selected at a given moment in accordance with the adopted game model, and the creator of the game does not directly program what the bot is supposed to do. The latter is also possible (if necessary), but is not the main way to work with the tool.

AI CONFIGURATION



AI configuration



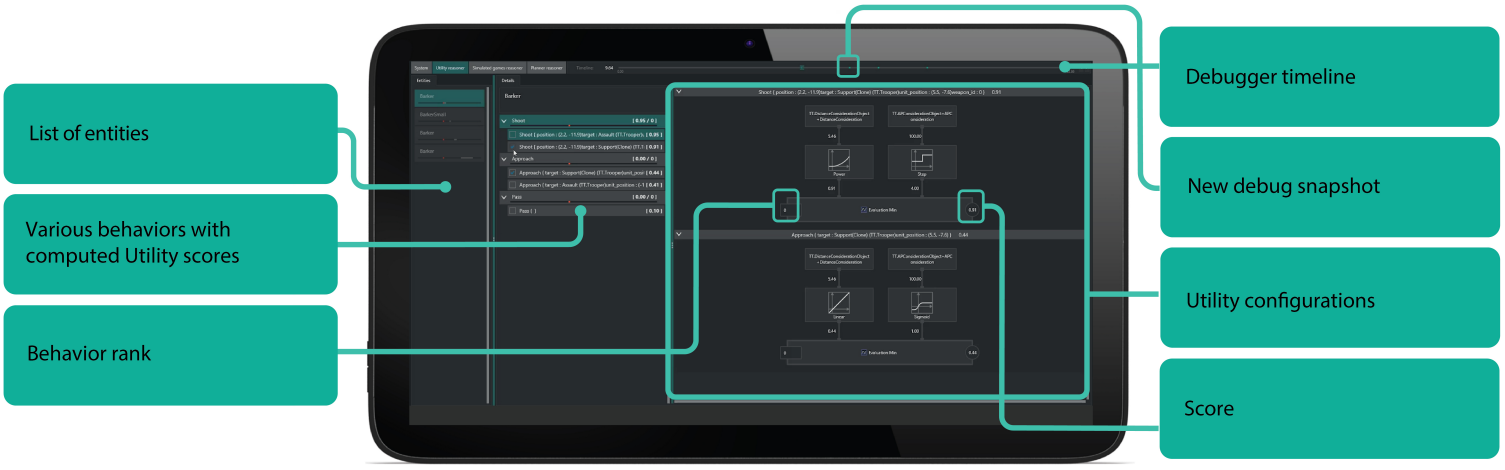
Adding Utility reasoners

Curve editor

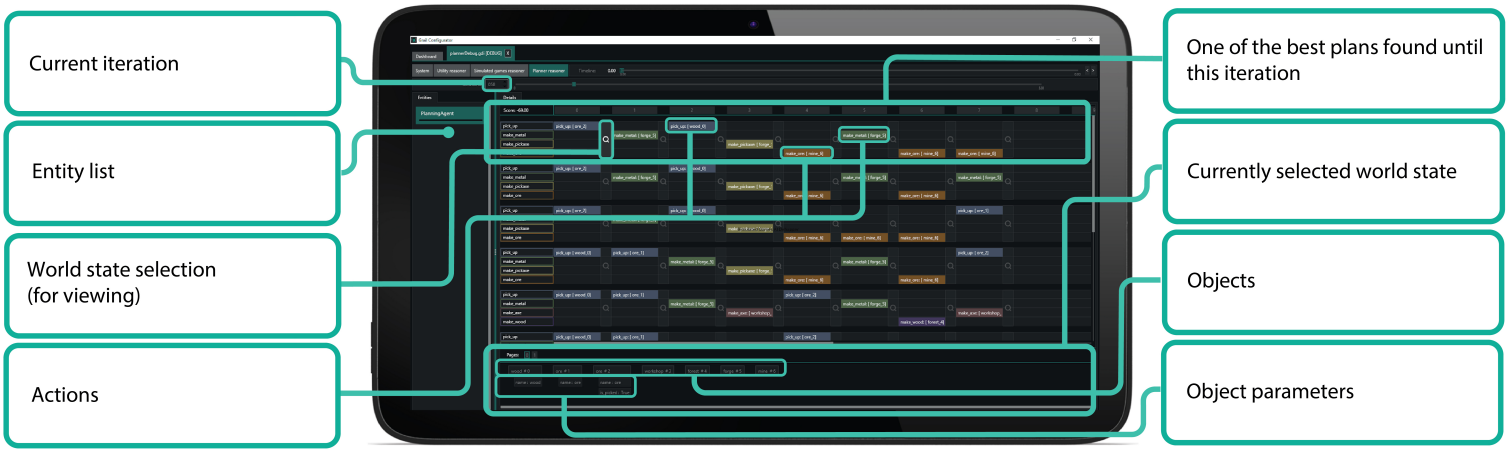
Configuration and fast prototyping

Utility configuration details on the example of Tactical Troops

DEBUGGING



Utility-Based AI Debugging



Planner AI Debugger

Simulated Games AI Debugger

6. How Can Grail Help Game Development Management?

Teams that use Grail for AI development and quality assurance can benefit from the above-mentioned flexibility, ease of modification, and quick turnaround. They can implement smarter behaviors more quickly, as compared to legacy techniques. At the same time, Grail can be well integrated with other techniques, ensuring backward compatibility with all legacy code. Importantly, a faster and more-robust-to-errors development cycle allows teams to iterate with more ideas and leads to tightening up that agile sprint.

Grail is the result of a several-year-long R&D program that QED Software carried out to build a tool for non-experts to enable them to create expert AI in a game.

Great research does cost. Grail is the result of a several-year-long R&D program that QED Software carried out to build a tool for non-experts to enable them to create expert AI in a game. Thus, Grail provides game development teams a way to benefit from the most important scientific achievements in creating the product, without the need to repeat the same research, without the cost of recruiting a team of AI experts, and without the costs of maintaining them.

Grail provides GUI tools for configuring AI (AI Entity Configurator and Utility Configurator) as well as debugging AI (Utility Debugger, Planner Debugger, Simulated Games Debugger). Moreover, our plugins add very simple GUIs to the Unity and Unreal game engines: to view debug files, load data from configuration files created in Grail, display basic debug data, such as the actions currently performed by agents, etc.

Grail's intuitive GUI tools can speed up teams' work in areas such as:

- AI configuration & debugging;
- flexible, easy-to-use Utility-Based AI;
- MCTS for solving complex adversarial games;
- goal driven behavior using planners;
- coordination of AI agents in challenging situations;
- complex AI behavior achieved by combining multiple algorithms;
- world state knowledge management using blackboards;
- easy integration of third-party reasoning algorithms.

7. Why Do We Need Truly Advanced AI in Games?

It will always be about believability and immersion. The goal of advanced intelligent behaviors in games is to create a world where the human player will have the best entertainment. What are the aspects of believability and immersion where great decision-making AI is pivotal?

Non-cheating AI. Especially AI that is not omniscient. Computer players should not have access to information a human player would not have in an equivalent situation. They should play according to the same rules. This is especially important in RPG games, strategy games, and all multiplayer games with bots:

- NPCs should not spawn out of nowhere (see the example of Cyberpunk police);
- computer players should not start with more resources, e.g. a prebuilt base in a strategy game.

Realistic skills of computer agents related to who they are. For example, a soldier that has perfect aim because a computer can calculate ballistics perfectly, is not believable and breaks immersion. On the other hand, a powerful archmage in an RPG game may be superhuman when it comes to skills. It must be appropriate.
This is a common issue – living creatures should not be able to calculate something in a computer-like style.

Ability of NPCs to adapt to a given situation:

- including the ability to react to player actions;
- avoids repetition of the same pattern all the time.

How difficulty is implemented. For example:

- enemies who are inadequately hard to beat will not be perceived as believable;
- difficulty scaling should not be implemented by making opponents deal more damage with the same weapon or gather more resources per second. This is not believable. Instead, they should be more intelligent, use more sophisticated strategies, tricks, etc.;
- being too strong or too weak (related to the level of intelligence the player expects from a virtual character) usually breaks immersion.

Realistic senses of computer agents (perception, hearing, memory). Whether they can hear and see in a similar fashion to living creatures.

- For example, police officers should not chase you if you did something wrong completely unnoticed when nobody was around.
- Remembering facts you told NPCs and referring to them in future conversations.

Using "all senses" in a consistent fashion. For example, attacking and shouting "charge", or running away and being silent, being scared, and sweating. The ability to combine behaviors, e.g. talking, walking through the room, looking through the window.

Feel that computer agents try to accomplish goals we expect them to pursue. A soldier, a businesswoman, an alien, a dog – each of these entities may have various goals, and we as humans intuitively know which of them are believable.

Learning and reasoning. NPCs should learn from mistakes. They should also be able to draw conclusions based on what they observe.

NPCs showing emotions, e.g. expressing gratitude to the main character, or healing a wounded ally in a battle. Also, computer players may be "ruthless"

in non-human ways in their actions. For example, if the computer player needs to kill one opponent unit to win, they may even shoot through an ally soldier (who happens to be in the line-of-sight) and kill them too, because the computer player knows it will win anyway.
This is non-human-like behavior that breaks immersion.

Various personas. NPCs with distinct personalities make for a more believable game compared to a game in which all NPCs are similar.

Smooth behaviors. NPCs are more believable if they are persistent in their behaviors. Think of a pathfinding algorithm from A to B. Going along a smooth line is more believable than zigzagging, even if a distance metric is defined in such a way that both paths have equal length. This thinking can be extrapolated onto general behavior of computer bots. Going back and forth between two states is not believable.

8. What Makes Games Built with Grail Better Value for Players?

With Grail, AI behaves less schematically, because the developer does not define behavior in the form of rules such as "*If you have hp <50, hide*", etc.

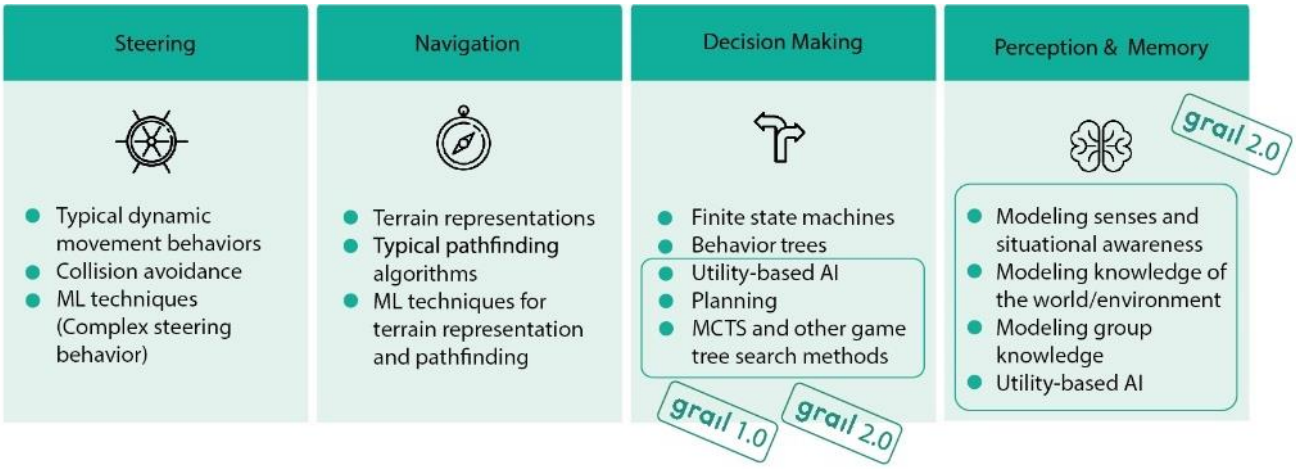
Opponents use advanced tactics (flanking, setting ambushes, etc.), thanks to ability to simulate enemy movements (MCTS).

Strong, interesting opponents in combinatorial games (e.g. in card games).

An NPC capable of carrying out long sequences of actions and achieving long-term goals (through the use of a planner) appears more intelligent and "alive."

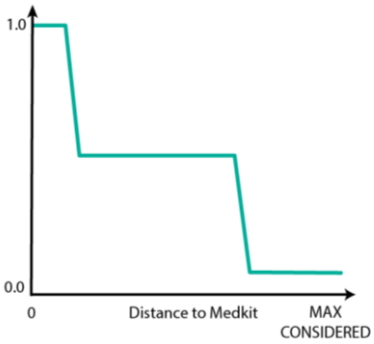
9. Core Technical Contribution

AI in games in general can be understood broadly as a collection of methods pertaining to steering, navigation, perception and memory modeling, and decision-making. Grail is centered on decision-making (in version 1.0) as well as perception and memory modeling (in upcoming version 2.0). Below are the details of Grail reasoners and their intended usage scenarios: Utility-Based AI, Simulated Games AI, Planner AI, and Evolutionary Optimizable Scripts.



9.1. Utility-Based AI

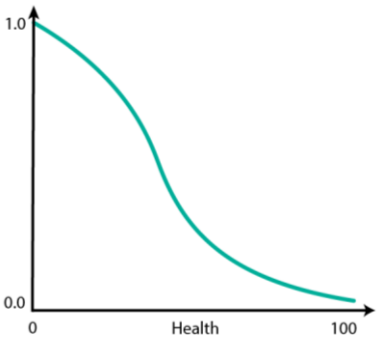
Utility-Based AI is a method of assigning scores to behaviors based on some heuristic, usually involving curves. Then the algorithm analyzes those scores and chooses the most suitable behavior.



Intelligent entities choose what to do based on the perceived utility of each action.

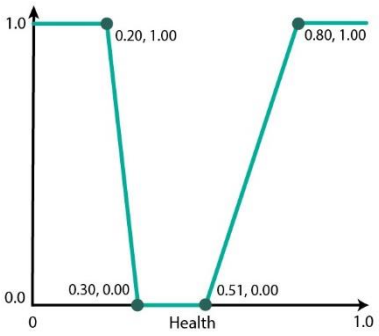
The concept of modelling behavior with utility scoring originates from economics and psychological science. It has laid the foundations of what became known as utility theory for game AI[2]. In the context of games, there are many names for this theory used interchangeably, such as utility system, utility-based AI or just utility AI. Utility-based AI has also become popular[3] thanks to its advantages such as scalability, extendibility, and ease of design.

The main premise is that intelligent entities choose what to do based on the perceived utility of each action. In the context of game AI, each action is associated with so-called utility curves (also referred to as response curves) that define relationships between decision factors in the game and measures of the action being appealing for the agent. Such a factor is called a consideration.



The figures on the left depict two sample curves defined for the action of looking for a med-kit in a shooter game. There is no limit to the types of curves that can be used. All of the membership functions commonly used in fuzzy logic are useful here as well. It is often said that utility-based AI provides a fuzzy-logic-quality compared to case-based scripted approaches such as behavior trees.

What is new, compared to many traditional techniques such as behavior trees or finite state machines, is that you can focus on defining the decision factors one by one and their interplay is an emergent behavior provided by the utility-based AI. This way, you can avoid combinatorial explosion and spaghetti code, which we often end up with when trying to combine all decision factors within a single script.



The results from processing utility curves defined for the same action but for various considerations are combined into one score that becomes the final perceived utility of the action. This process is performed by the so-called evaluator in the utility-based AI. There are many ways of performing such an aggregation of scores, as they are many ways to model a particular game using utility-based AI.

Finally, an action is chosen based on the utility values of all actions. This operation is performed by the so-called selector. Depending on the particular game and scenario, the agent may use various selectors. The most common options are (1) choosing the action with the highest utility score, (2) pseudo-roulette wheel selection, or (3) choosing a random action among a few top-scored actions.

9.1.1. Grail Implementation

Commonly, utility systems evaluate and select only a behavior type, which is next parametrized with heuristically chosen data.

Example 1:
Behavior types: attack or stay idle.
Characters:

- AI character;
- troll - strong and far from AI character;
- goblin - weak and close to AI character.

Let's assume that the AI character has selected the attack action. It is now determined that the AI character should attack the goblin because of a user-provided heuristic that the closest enemy should be attacked first.

Behavior Instantiation in Grail makes action selection much more clever. The utility-based AI reasoner allows users to evaluate and select contexts – abstract data that allows you to instantiate proper instances of a demanded behavior.

Example 2:
Considering behavior types and characters from Example 1, let's instantiate all behaviors. Thus, a list of all behaviors appears as follows:

- attack(troll);
- attack(goblin);
- stay idle.

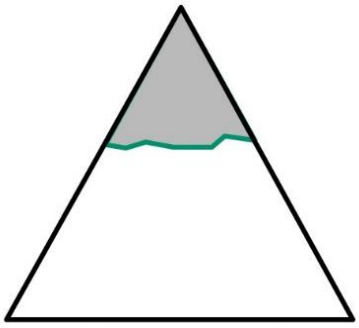
Each attack behavior can be evaluated independently giving users more conflict resolving abilities during action selection. If we provide the reasoner with curves indicating that the character should attack closer enemies, attack(goblin) will be

selected. However, if the curves indicate that stronger enemies should be attacked with priority, the reasoner will select attack (troll).

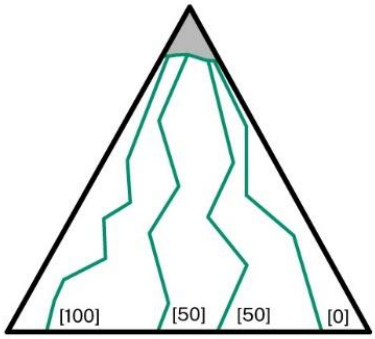
Persistence is an optional reasoner parameter determining a score bonus for the currently executed behavior. The purpose of persistence is to allow you to force entities to swap behaviors only if new behavior options are significantly better than the current one, to prevent erratic behavior.

Operation stack. While using Grail's Utility-Based AI, suspended behaviors are put on top of the stack and can be retrieved later, if they are still legal. Behaviors from the stack also get a persistence bonus to their score.

9.2. Simulated Games AI



Alpha-beta tree search



Monte Carlo Tree Search

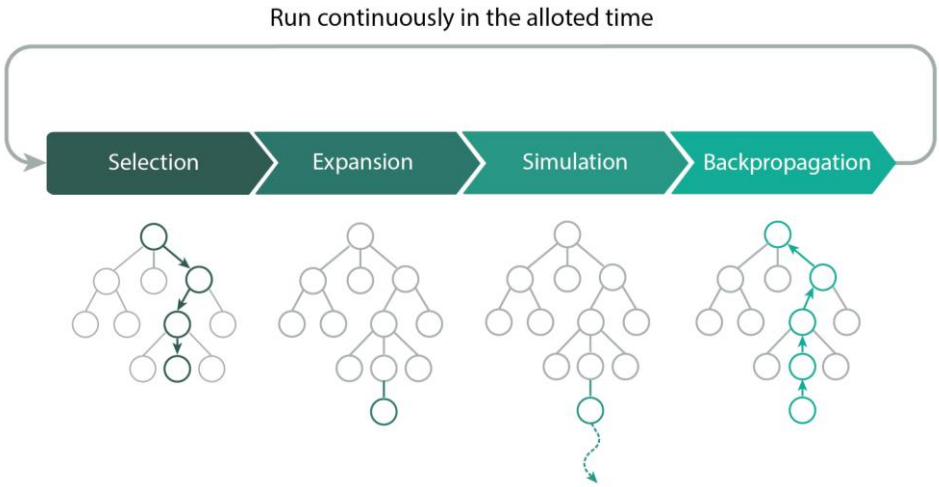
The idea is to learn how to play the game in the most effective way by gathering statistical evidence through simulations.

Grail's Simulated Games AI implements a Monte Carlo Tree Search (MCTS) algorithm. MCTS is a state-of-the-art technique for implementing AI in combinatorial games. It is a simulation-based technique and, as such, it requires a model that can simulate a game that includes:

- determining all possible actions in a given state;
- applying the effects of actions that result in a new state (frame update);
- making sure that equal actions applied in an equal state result in the same state;
- determining whether the game is finished (termination condition);
- determining a numerical outcome of the game.

MCTS builds the so-called game tree iteratively. In such a tree, nodes represent possible game states, whereas edges correspond to possible transitions between states. Both nodes and edges may contain additional data associated with states and actions, respectively. Although it is said that a game tree is searched and not constructed (because the tree is an abstraction that already exists as a defined mathematical model) we will use the verb 'construct' in relation to the game tree to express the process of building the data structure in a computer memory. The idea is to learn how to play the game in the most effective way by gathering statistical evidence through simulations. The algorithm contains a formula that balances the exploitation vs. exploration trade-off. Think of a player – e.g. a chess grandmaster – that can think for a certain time and play virtual games in their mind before making the actual move in the real game. In such hypothetical games, they may check the most promising moves in more depth (exploitation) and test various new promising moves (exploration). MCTS does exactly that before recommending the actual action to make in the game.

Each simulation iteration uses the knowledge gathered so far and consists of four phases: selection, expansion, simulation, and backpropagation.



MCTS scales better than traditional game-tree search algorithms such as alpha-beta. As it relies on the real outcome of the game, it is more robust and asymptotically converges to perfect play with respect to the given goal.

Other game-tree search algorithms are breadth-first and cannot be applied without a heuristic evaluation function in non-trivial games. Despite being the technique of choice for combinatorial games, MCTS has not often been applied in video games. Grail aims to change this by adapting this technique to the nature of video games.

9.2.1. Grail Implementation

Grail's Simulated Games AI implements an MCTS algorithm in its core. The approach is based on simulations of certain simplified games within your game. Such a simplified game can, for instance, be the strategy layer in a 4X strategy game.

We made certain aspects of MCTS easier to manage and we introduced a few modifications.

We made certain aspects of MCTS easier to manage and we introduced a few modifications[4].

1. The game state is not stored in the tree. In the MCTS algorithm, the state of the game is usually stored explicitly in tree nodes. This can hamper video games due to huge, unpredictable memory usage.
2. We do not require any complex definition of a state in the game and, in particular, we do not require the implementation of a state comparison function, which may be inefficient for large game states.
3. Instead, we use intuitive interfaces for units in the game. You define available actions for the units, their reset procedure (when the simulation starts from the current state again), and their team alignment. Units that share a team alignment are considered by the MCTS to be working towards the same goal.
4. There are two interfaces for units – ThinkingUnit and StochasticUnit. The ThinkingUnit will perform actions according to the MCTS exploration

vs. exploitation paradigm. The StochasticUnit will perform actions randomly using the provided random distribution.

5. You define the results of actions and which unit goes next.
6. However, we require actions to have repeatable deterministic effects in a given state. Randomness is still totally possible, but the StochasticUnit must perform random actions and for each random outcome a separate action needs to be defined. For example, the StochasticUnit may roll a 6-sided die by having 6 available actions. They will be chosen randomly instead of 'intelligently' because of how StochasticUnits work.
7. It is possible to have both discrete and continuous actions. The former have immediate results, whereas the latter can have delayed results. Both are possible in Grail, which is shown in the documentation.
8. Each unit can be assigned a heuristic reasoner that, based on the state in the simulation, may choose an action heuristically instead of the one proposed by the MCTS.

This way, it is easy to introduce expert knowledge or scripted behavior precisely where we want to. In other situations, the unit will still be simulated in the spirit of the MCTS.

9.3. Planner AI

The role of planning algorithms is to find the optimal plan according to hard-coded or user-provided requirements.

Planning is a branch of AI research concerned with finding appropriate sequences of actions needed to achieve stated goals given an initial world state. The action sequences leading to that goal are called plans. The role of planning algorithms is to find the optimal plan according to hard-coded or user-provided requirements. This might mean finding the shortest sequence of actions, but it might also be necessary to minimize a more abstract action execution cost metric.

All planning algorithms require the concept of a state (or world state). The world state contains all information needed to determine if a given action may be performed.

Another key concept of planning algorithms is action. An action is represented by a list of preconditions and a list of effects. Preconditions determine whether the action can be executed, given a world state. Effects define world state transformations applied after choosing the action.

Planner goals are defined as a set of conditions, in a way similar to action preconditions. Whenever the planner stumbles upon a state that satisfies all of the conditions, the best plan leading to this state is returned. The set of all possible states is called a state space. In any state there can be (and usually there is) more than one action available.

9.3.1. Grail Implementation

In planning algorithms, states are typically modeled as sets of Boolean variables corresponding to certain facts about the world, for example:

```
present(crane, room_1) = true
adjacent(room_1, room_2) = true
```

We found such representation to be too limiting in the context of video games, so we designed our own way to describe states.

Parameterized Object. In Grail Planner AI, the basic world state building block is a Parameterized Object. Parameterized Objects are containers for arbitrary data types, with entries identified by string keys (they're very similar to Blackboards in this regard). Additionally, Parameterized Objects can hold multiple collections of unsigned integers. They are used mostly to represent references to World Objects (to model an inventory, connections between rooms, etc.).

World Object is a special type of Parameterized Object, representing an entity that can be subject to various actions. Each World Object is characterized by its type, such as "monster," "potion," "room" (World Object types also support multiple inheritance), and a unique reference id.

World State. In Grail, the World State is a container for all existing World Objects. Additionally, it is also a Parameterized Object, giving you the ability to describe state variables that are not part of any World Object. Typically, you will have to create only the initial World State and the planning algorithm will carry on from there.

Representing Actions. To tell the planner what it can do, we have to define a set of objects called action templates. An action template contains a name, an argument list, action preconditions, effects, and cost computation logic.

Action logic as functions. Preconditions, effects, and cost computation are modeled as user-provided functions written in a programming language (instead of using a special description language, as is usually done), allowing for a very flexible definition of actions. Thanks to this approach you can use all the mechanisms of your chosen programming language inside action logic: nested conditionals, loops, arithmetic operations, and all sorts of stuff that's not typically possible in planning algorithms. For example, you can procedurally compute an action cost, depending on the World State. You can apply variants of action effects, based on various parameters like health, fatigue etc., unrestricted by typical limitations of planning domain description languages.

You can use all the mechanisms of your chosen programming language inside action logic.

Goal representation. In Grail, planner goals are simply represented as collections of functions with bool return type. As with actions, this approach allows for very flexible goal definitions. Why not just a single function? If you define your goal as a list of subgoals, it's much easier to come up with a sensible planner heuristic

– you may use Hamming distance between the current state and the goal or penalize unsatisfied goal conditions in a different way.

9.3.2. Performance Considerations

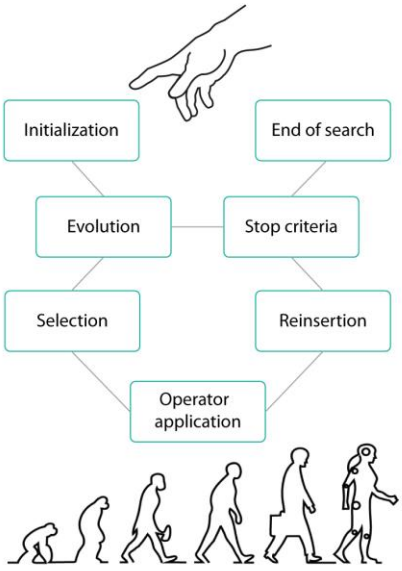
Planning can be really computationally demanding, and to make its use practical in video games, we certainly don't want to take too much CPU time when computing the plan. To address this issue, Grali's Planner AI works in an iterative manner – this means that you can easily spread the computation over multiple frames.

Max depth. You can significantly improve computational complexity of your planning problem by limiting maximum search depth or – in other words – maximum plan length. In most video games, there will be no need to consider plans longer than a few actions, and setting a reasonable depth limit will cost you nothing while drastically shrinking the search space graph.

Max plan cost. Another way to limit search depth is to limit maximum total plan cost. This way you can still obtain long sequences of cheap actions while still getting some performance boost, especially in domains with large action cost disparities.

Max iteration count. Sometimes finding a plan is simply not possible and there's no point searching the state space on and on. Grail's Planner AI gives you the option to limit the maximum iteration count after which planning will be considered a failure.

9.4. Evolutionary Optimizable Scripts



The evolutionary algorithm (EA) is a representative of the evolutionary computation family that encompasses algorithms such as genetic algorithms, genetic programming, differential evolution, evolution strategies, memetic algorithms, and several others. The evolutionary computation is part of the so-called Computational Intelligence (CI). In general, most techniques in computer science that are referred to as "intelligent" fall into either Computational Intelligence, Machine Learning, or Artificial Intelligence.

Evolutionary algorithms are population-based. Each individual in the population, also termed a phenotype, encodes a specific candidate solution to the given optimization problem.

Their idea has been inspired by nature – how natural organisms evolve over many generations. The standard EA is inspired by Darwinian natural selection; however, there are also variants that draw inspiration from Lamarckian evolutionary theory.

At the start, the initial population usually consists of random solutions, possibly even invalid or poor quality ones. The idea is to employ the survival of the fittest process to "breed" new solutions until an acceptable one is found (or we run out of time). This is done using selection pressure and genetic operators. The selection, as the name implies, selects the best individuals from the current generation to survive to the next generation based on their fitness computed by the so-called fitness function, which is typically provided by the programmers. The standard genetic operators are crossover and mutation. Crossover is the way parent solutions are combined to create a child solution. Mutation affects a chromosome of an individual to maintain genetic diversity. Mutation is usually applied with a small probability for each chromosome.

However, there are both numerous variants of particular evolutionary operators (mutation, crossover, and selection) as well as various strategies and heuristics used. EAs can be used instead or together with machine learning. Examples of the latter case include using EAs to optimize the weights of neural networks, or evolving neural networks (e.g. the NEAT method – NeuroEvolution of Augmenting Topologies).

EAs can avoid local optima more easily than gradient-based methods. Because the whole population undergoes the process, trends can be observed and the best solutions, if shared by many individuals in the population, tend to be more stable.

There are many articles that analyze the advantages of EAs. For instance, they can be applied to non-differentiable problems. They can avoid local optima more easily than gradient-based methods. Because the whole population undergoes the process, trends can be observed and the best solutions, if shared by many individuals in the population, tend to be more stable. Finally, EAs are an ideal method to which parallel computing can be applied.

9.4.1. Grail Implementation

Grail comes with implementation of a highly configurable evolutionary algorithm. It is designed to optimize what we call "EvoScripts".

Anything can subclass the EvoScript and become immediately optimizable by our EA. You choose which parameters are optimizable using a very straightforward interface.

Instead of writing:

```
int minimumHp = 2;
if(HP > minimumHp)
    // do something
```

You will write:

```
EvoParam<int> minimumHp = 2;
if(HP > minimumHp)
    // do something
```

The value of "2" is the initial value, but you can provide the domain of the available values for your parameter.

For the evaluation – the survival of the fittest process – described in the theoretical introduction, we introduce the so-called Arena.

Arena is an interface for you to implement that will serve as a battle between EvoScripts. The goal of it is to assign a score (the higher the better) that denotes how good each EvoScript is. In Arena, you will typically start your game, provide the starting state (scenario), run it, and then assign a score to each participant.

We provide helpers for a tournament-style contest, e.g. how many points/frags a player represented by an EvoScript gets. However, if you have a custom scenario – for example you do not want to optimize the players but the game, you may ignore it and just provide the function that will assess the quality of the current settings.

Another example: you may create an AI that can play a puzzle game. Then the arena may use such a player and test how much time it takes for it to solve the game. If the parameters of the puzzles are represented within the EvoScript, the algorithm will aim to optimize them to the desired time of solving the game.

We provide a default parameterization of the EA based on typical scenarios. Even mutation and crossover operations are provided, by default, thanks to the "EvoParam" interface. However, you can change, for example:

- the crossover rate;
- how individuals are chosen for the crossover;
- crossover operation;

- mutation rate and range;
- how individuals are chosen for the mutation;
- mutation operation;
- the type of selection (from one generation to another);
- elitism in the selection.

If you want to read more on this topic, please refer to our documentation pages at **https://www.grail.com.pl/doc.html.**

9.5. Grail Integration with External Algorithms

In some cases, it is important to enable integration of legacy AI-controlled agent code with new tools. This type of combination will work when we have some complex, scripted behaviors, but want the scripted behaviors to be chosen wisely, based on the observed state of the world. For example, suppose we have two behaviors: attacking an opponent and picking up a first aid kit. The developer wants the attack to proceed as follows each time: The agent positions itself using some heuristics independent of the decision-making process.

1. If the agent has found a good position, it goes to it; if not, it goes back to point 1.
2. The agent turns towards the opponent and takes aim.
3. The agent waits a predefined amount of time (to simulate reaction time).
4. The agent shoots.

This rigidly scripted sequence of behaviors with branching depending on simple conditions is ideal for the use of behavior trees – we have simple rules and a predictable situation, and the creator has full control over all parameters and conditions.

But the choice of whether we should attack the opponent (and which one) or go get the first aid kit (and which one) is so complex that it will be much better to model it with the Utility-Based AI. Thus, we get the following flow:

1. Utility-Based AI selects the most promising behavior.
2. The behavior tree associated with the behavior is triggered.
3. Utility-Based AI selects the next behavior and (if it can) abandons the previous one.

And so on.

Grail makes use of very abstract data structures, and because of that, integrating Grail with external algorithms such as behavior trees, finite state machines, etc. is quite simple.

Grail makes use of very abstract data structures, and because of that, integrating Grail with external algorithms such as behavior trees, finite state machines, etc. is quite simple. All that the user needs to do is to encapsulate their chosen algorithm in Grail's Behavior. To this end, they need to provide 3 things:

1. A way of executing the Behavior defined by methods Start, Update, and Finish. For example, when using behavior trees, this should consist of actually running a behavior tree.
2. A way of knowing whether the Behavior has finished, defined by method IsFinished. For example, by checking if the behavior tree has finished its calculations.
3. A way of knowing whether the Behavior can be executed in the current game state, defined by method IsLegal.
   Having done that, a user can effortlessly embed their chosen algorithms in Grail reasoners.

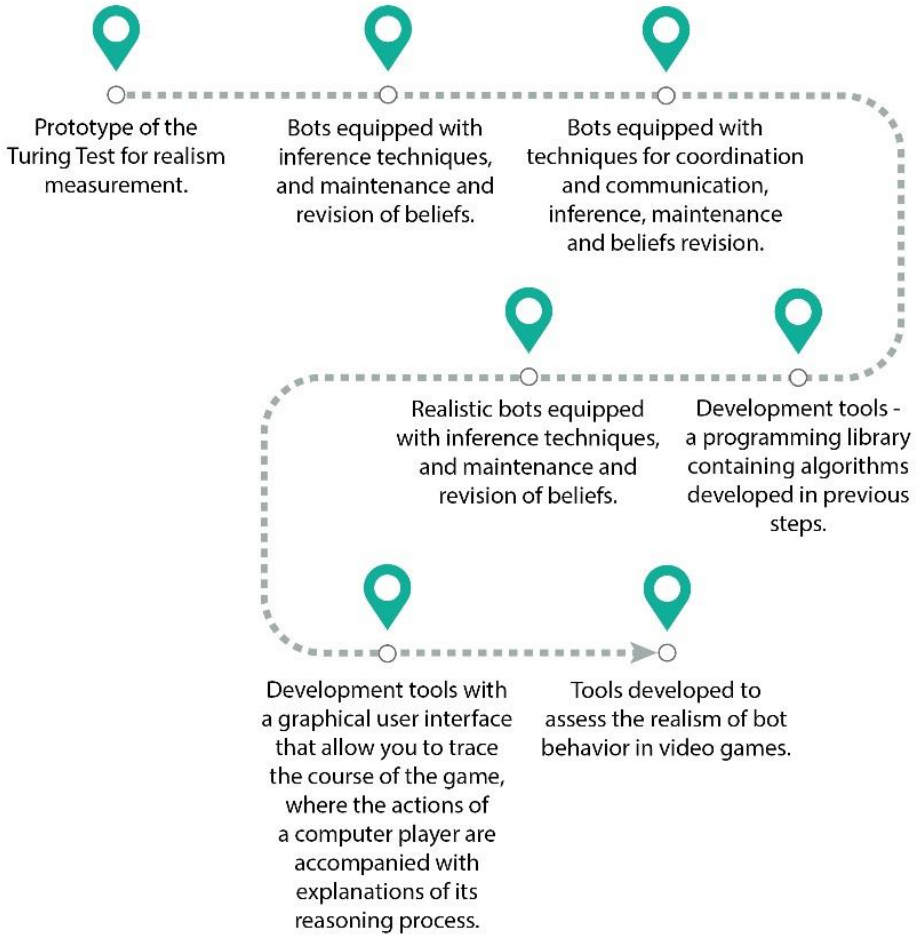
   Of course, it is entirely possible to do that in reverse and embed Grail's reasoners in external algorithms, but that does not differ from the usual way of using Grail. Users need to simply declare a reasoner inside a scope they wish to use it in and update it as often as needed.

9.6. What Type of Games Would Benefit Most From Which Grail Techniques?

Simulated Games AI The technique is suitable for 4X, real-time strategy (RTS), real-time tactics (RTT), turn-based strategy (TBS), turn-based tactics (TBT), economic, simulation, and tower defense games, as well as any games containing such elements.	**Planner AI** Especially good for games where the world can be modeled in the form of requirements for action and action results. Originally used in F.E.A.R and Killzone 2, so proven in games of this type. Planners are good for aspects such as queuing game production, crafting, planning the order of visiting checkpoints, logic puzzles, and turn-based games with a large number of actions to be performed during one turn.
Utility-Based AI Suitable for any game but especially good for action games, survival shooters, first-person shooters (FPS), and role-playing games (RPG) where you need to make a quick decision. This technique has also been successfully applied to games such as The Sims for modeling the characters' preferences. It is a very good method for making strategic decisions when the quality of the decision is influenced by relatively few factors and the relationship between the factor and the usefulness of the decision can be modeled. If dependencies are more complex (and, for example, require simulation), there are better methods, e.g. Simulated Games AI.	**Evolutionary Optimizable Scripts** This technique has two uses. The first is simply scripted behavior, so it will work well in any game where AI is written as a specific heuristic. The second is to optimize selected parameters that ultimately influence the AI performance if only a method of its evaluation is provided. Grail's evolutionary algorithm is a meta-optimizer used as a parameter tuning tool.

10. Grail Roadmap

How to prevent situations that break immersion? Some of the aspects above have been tackled before by AI, while some are only now receiving attention. Here is what we have planned for Grail's future development.



1. Development of an effective method of measuring the realism of NPCs' behavior (based on the Turing Test). Our goal is to measure the realism of NPCs in the AI production loop, therefore this test needs to be relatively fast and accessible.
2. Enable NPCs to infer and dynamically create missing knowledge in the game with incomplete and uncertain information. The goal of such reasoning is to enable game playing at a sufficiently strong level while maintaining the limitations on the information that human players have. Inference techniques will be developed in terms of the game techniques used to make decisions in the game. Therefore:
    a) inference techniques must produce the knowledge needed to make a decision
    b) inference techniques can produce input data for decision-making or can be integrated with the decision-making algorithms themselves

(e.g. extend the Utility-Based AI technique to include uncertain and incomplete information)

3. Representing the beliefs of the bot, especially as inferred by the above-mentioned inference techniques. A particular problem is the method of maintaining and revalidating these beliefs in a dynamically changing environment.

11. You Are in a Good Company: the History of Applying Advanced AI in Video Games

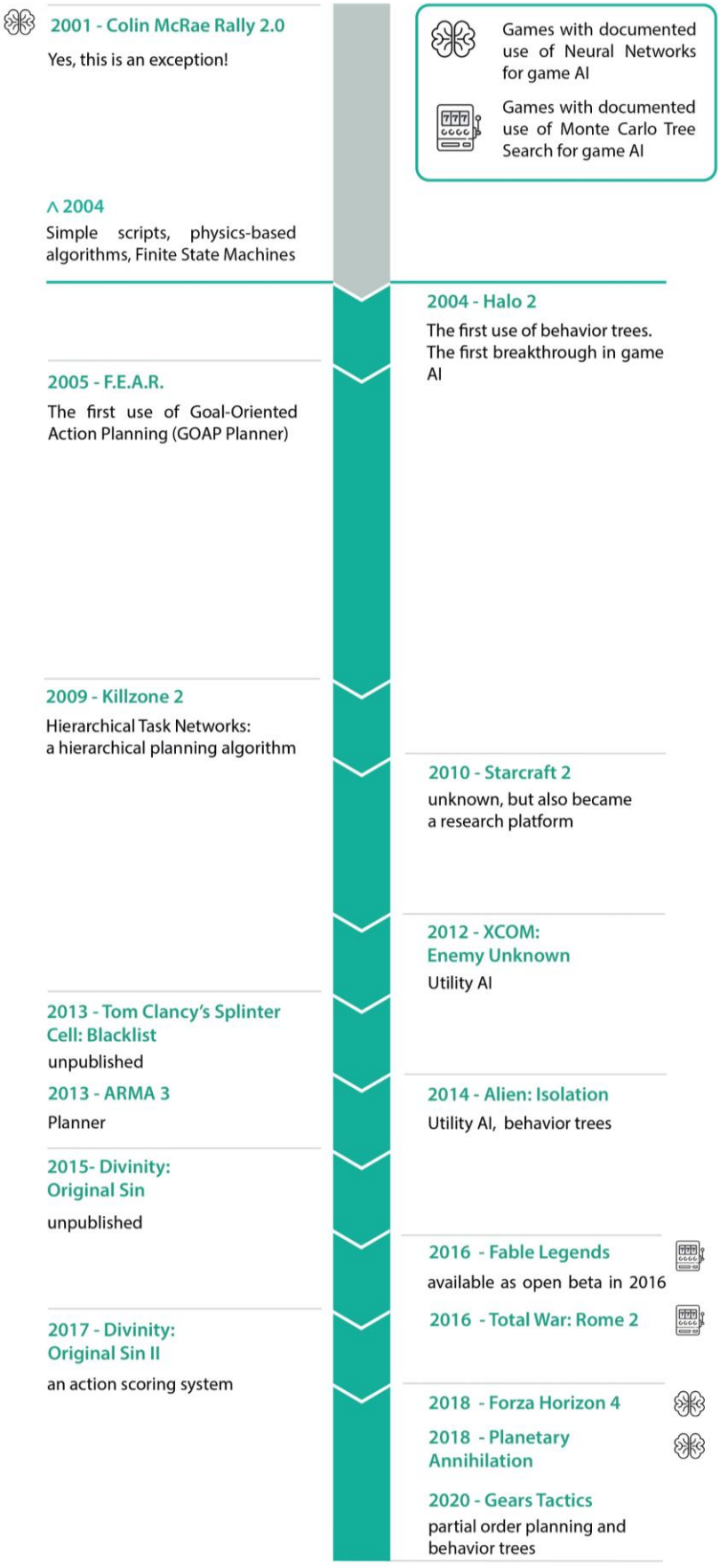
Considering what's ahead, let's reflect on the accomplishments of the game development industry with respect to applying advanced AI to date. In summary: the game development industry has been using rather simple methods to create AI. Despite enormous progress in scientific AI/ML, gamedev has not adopted the latest achievements. There is a distinctive gap between scientific AI and gamedev AI for the purpose of decision-making, and perception and memory modeling.

The AI techniques that are present in solutions that are on the market and available to game developers are currently insufficient to fulfill our goal in full. Plausible reasons for this rather slow adoption rate of more advanced AI methods are beliefs that limit progress, such as:

- "Advanced AI is hard to design…"
- "… and to control."
- "It can be overpowered (in an inhuman way)."
- "Advanced AI can produce unpredictable behaviors."
- "The methods are inefficient (too hard for a dedicated computational budget)."
- "Difficult to integrate with current methods."

We developed Grail with these aspects in mind, to address the possible issues upfront.

GAMES PARTICULARLY PRAISED FOR THEIR AI

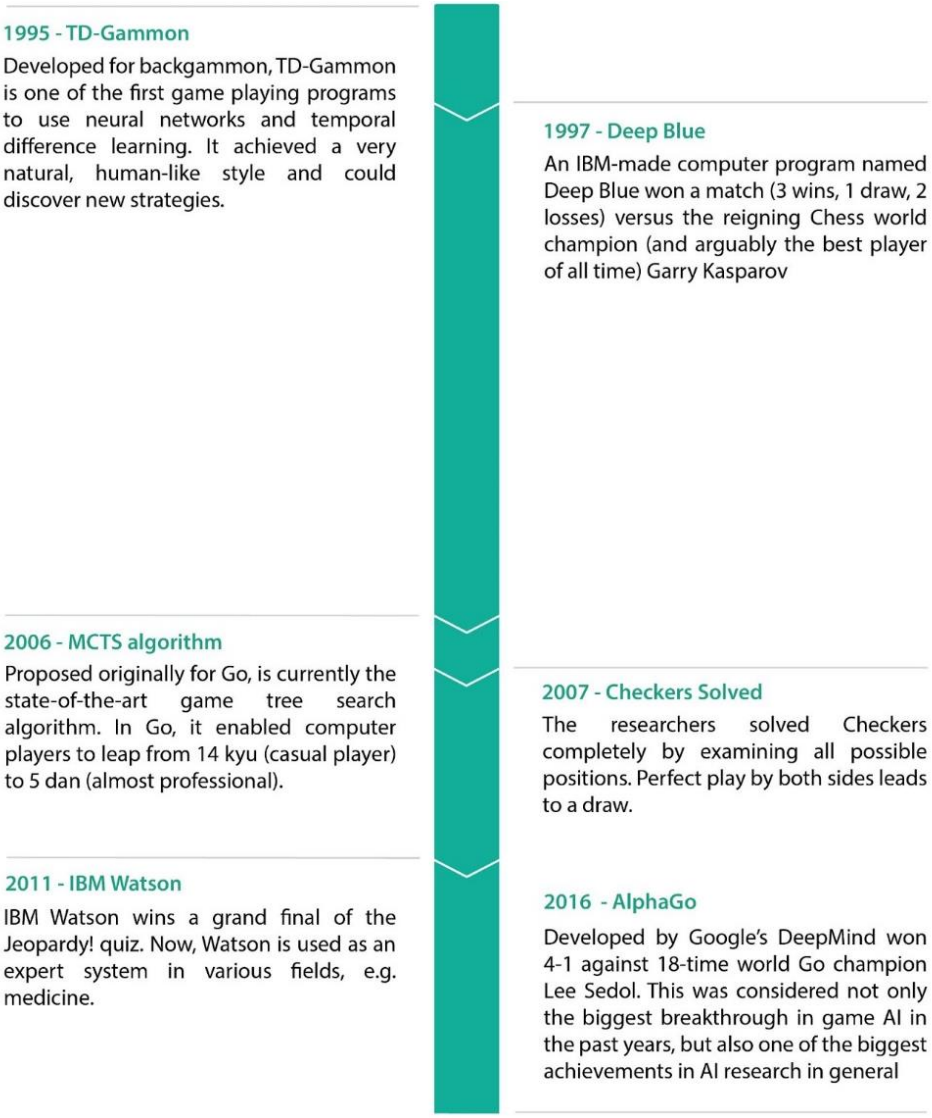


12. Games as a Sandbox for Developing Cutting-Edge Decision-Making AI

Chess was named the **"drosophila of AI"** because it was similarly popular in AI research as is this type of fruit fly in biological research.
Go was named the "**Grand Challenge of AI"** because it has been identified as a game that is hard to tackle through algorithms and would require a more human-like AI approach.
Arimaa is a game specifically designed in 2003 to be (1) extremely difficult for AI players and (2) playable with a chess set.

Games have been serving as a medium in Artificial Intelligence research since its inception in the time of pioneers such as Alan Turing, Arthur Samuel, and John von Neumann. In 1959, Samuel wrote a paper titled "Checkers playing program," and since then Game AI has been used to measure the progress of Artificial Intelligence.

Games are often chosen in research as a test environment for algorithms, techniques, or sophisticated approaches not necessarily aimed at playing games as the ultimate goal. Game environments can test aspects such as decision-making, goal-based optimization, search and control, reasoning – including spatial and temporal reasoning, knowledge representation, dealing with hidden information, and uncertainty.



Historically, research in AI in games has been synonymous with the goal of achieving optimal play in combinatorial games. These can be formally represented by game trees of various state-space and action-space complexities. Therefore, lots of algorithms proposed for game AI are search-based algorithms such as min-max, alpha-beta pruning, or MTD(f). Popular combinatorial games used for research are: Chess, Checkers, Backgammon, Shogi, Hex, Othello, Gomoku, Havanaah, Arimaa, Lines of Actions, Chinese Checkers, Amazons, Stratego, Scrabble, Bridge, and Limit Poker.

Much of the research in game AI revolves around competitions, which are hosted periodically, e.g. annually. Competitions are efficient ways of measuring the progress of AI in a given area. Moreover, they provide an objective benchmark for comparison of approaches[5], e.g. General Game Playing, General Video Game Playing, Arimaa Challenge, Starcraft AI, microRTS, Hearthstone AI, or VizDoom. Currently, research in decision-making AI has many niches and flavors and is carried out in all kinds of games, e.g. combinatorial games, serious games, modern video games, old video games (e.g. Atari, NES). The most iconic, cutting-edge approaches to AI in games seem to be attributed to the works of DeepMind and OpenAI research laboratories.

Firstly, DeepMind has provided us with increasingly sophisticated evolutions of AlphaGo in the form of AlphaGoZero (mastering Go without initial knowledge), AlphaZero (adds Chess and Shogi to games it mastered) and MuZero (adds Atari games). These programs can learn how to play games extremely well only through self-play, i.e. without any expert games to learn from. Secondly, DeepMind proposes AlphaStar. It is, In simple words, an AlphaZero-inspired approach to Starcraft II. It was the first computer program to master such a complex video game[6].

OpenAI has given us OpenAI Five, which is a super-strong AI player for Dota 2. In 2019, OpenAI Five became the first computer agent to defeat the world champions in an esports game[7]. Both AlphaStar and OpenAI Five are similar from a technological point of view. They combine simulated self-play with deep reinforcement learning, a set of neural networks for various aspects of the game, and the powerful computing machines they run on.

At QED Software Team, we are a mix of researchers and professionals with hands-on experience in the industry. We have a strong record of participating in research in game AI as well, which is reflected in many publications. Just to name a few, research in Hearthstone AI and participation in the International Hearthstone AI Competition, research in the field of general game playing, game-based data mining, and scientific-level approaches to specific games.

References

1. QED Software, Tactical Troops bots reason with Utility-Based AI and MCTS combined, upcoming paper, [online] Available at: **https://qed.pl/publications/** [Accessed 8 June 2021] GO BACK
2. Rabin S., 2017, Game AI Pro 3, CRC Press, [online] Available at: **http://www.gameaipro.com/** [Accessed 8 June 2021] GO BACK
3. Gamasutra.com. 2021. *Are Behavior Trees a Thing of the Past?* [online] Available at: **https://www.gamasutra.com/blogs/JakobRasmussen/20160427/271188/Are_Behavior_Trees_a_Thing_of_the_Past.php** [Accessed 8 June 2021] GO BACK
4. Grail.com.pl. 2021. *Simulated Games - defining a game in 10 steps: Grail Documentation*. [online] Available at: **https://grail.com.pl/documentation/documentation/0.1/manual/simulated_game/steps.html** [Accessed 8 June 2021] GO BACK
5. Świechowski, M., 2020, Game AI Competitions: Motivation for the Imitation Game-Playing Competition, [online] Available at: **https://annals-csis.org/Volume_21/drp/pdf/126.pdf** GO BACK
6. Deepmind. 2021. *AlphaStar: Mastering the Real-Time Strategy Game StarCraft II*. [online] Available at: **https://deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii** [Accessed 8 June 2021] GO BACK
7. OpenAI. 2021. *OpenAI Five*. [online] Available at: **https://openai.com/projects/five/** [Accessed 8 June 2021] GO BACK