# PALAIS: A 3D Simulation Environment for Artificial Intelligence in Games

**Patrick Schwab** and **Helmut Hlavacs**[1]

**Abstract.** In this paper we present PALAIS — a virtual simulation environment for Artificial Intelligence (AI) in games. The environment provides functionality for prototyping, testing, visualisation and evaluation of game AI. It allows definition and execution of arbitrary, three-dimensional game scenes and behaviors. Additionally, PALAIS incorporates a plugin system that supports swift integration of custom AI algorithms. As a result, PALAIS effectively reduces the effort necessary to research, develop, prototype and showcase behaviors used for non-player characters in games. Finally, we demonstrate the power of the provided plugin system by exemplarily extending the functionality of PALAIS with an external module. PALAIS is available at http://www.palais.io.

## 1 INTRODUCTION

The development of game AI typically requires a testbed environment to validate and visualise results in a virtual-world scenario. Game developers and researchers frequently employ either game engines or custom-coded game scenes as their testbed environments. Using these environments for simulation has several disadvantages: suboptimal code reuse, significant barriers to entry and increased development time over using a more domain-specific environment. PALAIS attempts to solve these issues by providing commonly required functionality, such as a graphical user interface (GUI), loading required assets, data visualisation, scripting, entity management and rendering, in an existing, accessible framework. Having this framework in place enables the user to focus her efforts on AI-related code.

Moreover, custom-built solutions are often not easily distributed. We propose a container format that stores all scene-related assets in standardised formats. In PALAIS these scene containers are called *scenarios*. Any instance of PALAIS can execute these scenarios. The scenario structure, which is further described in section 3, and its distribution process is depicted in figure 1. The scenario structure allows users to share their scene definitions, graphical assets and game AI. This simplified distribution process gives others the opportunity to learn from, and build on, existing work. Consequently, our tool is also suitable for use in game AI education. Teachers can utilise the provided environment to supply students with interactive demonstrations of game AI techniques. We believe this form of hands-on education, where students can monitor and adapt execution parameters in actual game scenarios, can significantly increase the accessibility of game AI. Similarly, the simulation environment can serve as a demonstration platform for researchers to showcase their algorithms and techniques.

[1] University of Vienna, Faculty of Computer Science, Research Group Entertainment Computing, Austria, email: a0927193@unet.univie.ac.at and helmut.hlavacs@univie.ac.at

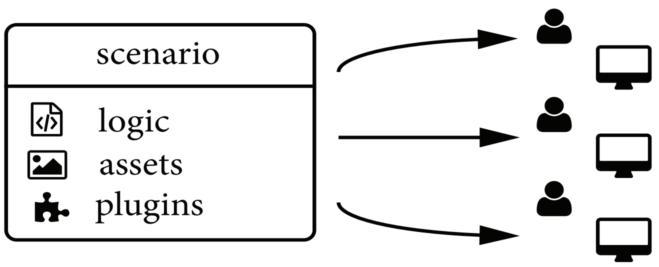


**Figure 1.** A schematic overview of the scenario structure and its distribution.

## 2 RELATED WORK

As mentioned, game developers and researchers commonly turn to commercial [15][7], open-source [14] or in-house engines for AI simulation. These general game engines overlap in functionality with PALAIS, particularly in the 3D rendering domain. PALAIS is more suitable for the simulation of game AI, because it provides the domain-specific functionality required for game AI development. Other toolkits, such as MASON [11], BREVE [10] and NetLogo [17], also provide full simulation environments. A significant drawback of some of the listed alternative simulation toolkits is the lack of extensibility via native code. Game developers strive to reach the maximum performance possible with the available computational resources. Thus, time-critical AI code for games is frequently written in native code. Our proposed simulation environment pays tribute to this by offering a plugin system [6] that allows extension through native, dynamically loaded libraries. The plugin system enables developers to test, prototype and evaluate the same native code that they use in their game engine. Ultimately, the ability to interface with native plugins also leads to more independent AI code compared to alternative simulation environments, because only the minimal necessary application programming interface (API) is exposed to plugins. Although the level of abstraction is not as high as it is with realisation-independent approaches. For example, [16] present such an realisation-independent approach.

Additionally, PALAIS provides a scripting API to increase its general accessibility and suitability for rapid prototyping. The scripting API is accessed via ECMAScript [5]. ECMAScript is one of the most widely-understood programming languages. Its most notable implementation is JavaScript, which is used to perform client-side scripting in Internet browsers. As a result of its prevalence, ECMAScript is a natural choice to provide scripting functionality in PALAIS.

To summarise, compared with the mentioned, existing works, the key distinguishing features of PALAIS are domain-specific functionality, interactivity, accessibility and extensibility.

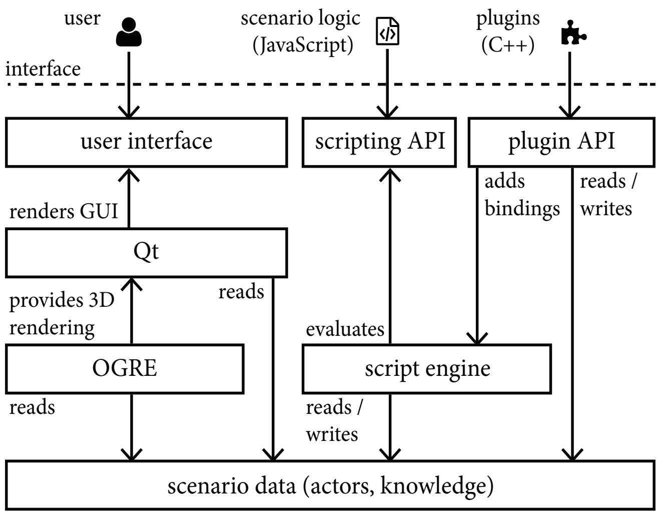
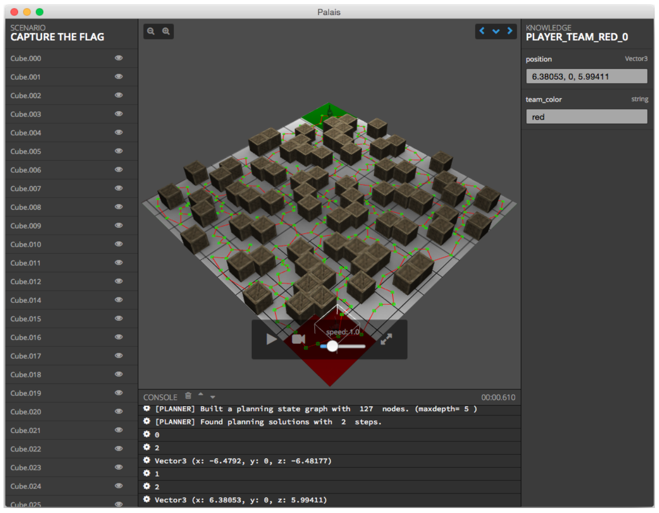


**Figure 2.** A schematic overview of the most significant interactions between the internal components of the simulation environment and its external accessors.



position (Vector3): 6.38053, 0, 5.99411

team_color (string): red

```
[PLANNER] Built a planning state graph with  127  nodes. (maxdepth= 5 )
[PLANNER] Found planning solutions with  2  steps.
0
2
Vector3 (x: -6.4792, y: 0, z: -6.48177)
1
2
Vector3 (x: 6.38053, y: 0, z: 5.99411)
```

**Figure 3.** The GUI of PALAIS after loading a scenario. The left panel lists all active actors in the game scene. The right panel shows the knowledge inspector. The center panel displays a rendering of the scene itself.

## 3 SCENARIO STRUCTURE

Scenarios are the entity corresponding to a given game scene in PALAIS. They encapsulate specific game situations defined by users. The common use case is to define scenarios that provide a minimal environment for evaluation of AI behaviors and algorithms. Essentially, these scenarios are self-contained packages that include the assets, logic scripts and plugins necessary to execute a game scene. The following sections describe the components of a scenario.

### 3.1 Assets

The term 'assets' in the context of scenarios refers to all scene-related data files that don't contain, native or interpretable, code. Typically, assets mainly consist of the files needed for rendering the scene, such as 3D mesh data, textures and materials. PALAIS can load scene files created with external 3D modelling tools like [1]. However, PALAIS currently only supports the scene and mesh formats native to OGRE.

### 3.2 Logic Scripts

Logic scripts are the files containing ECMAScript code. PALAIS interprets these files at runtime. Since no compilation is required, the user can simply reload scripts after changes. The ability to reload scripts allows for frictionless development of behaviors, as the results of code changes can be evaluated quickly.

### 3.3 Plugins

Plugins are the other group of code attached to a game scene. Plugins, unlike logic scripts, contain compiled code. Plugins are standard shared libraries. Their specific file format depends on the operating system (OS) and the processor architecture for which the code was compiled. Relying on platform-specific formats impedes the portability of scenarios across platforms. However, we accept this price to support the integration of precompiled code. In practice, this means that a scenario must contain plugins compiled for every required target platform.

## 4 PROGRAMMING MODEL

We call programmable entities within a scenario in PALAIS *actors*. A generic key-value store, labeled *blackboard*, represents the individual knowledge of every actor. As the naming suggests, blackboard systems [3] inspired this form of knowledge representation. We chose a blackboard architecture because it offers flexibility and is conceptually easy to grasp and use for developers. To represent global knowledge, the game scene itself incorporates a blackboard as well. For visualisation, all actors must be connected to a rendered object in the 3D game scene. PALAIS implicitly makes all rendered objects within a game scene available as actors. Additionally, native or interpreted code can instantiate new actors at runtime.

### 4.1 Time Simulation

All code instances, native and interpreted alike, receive notifications of time advances. These tick events are independent of the frame rate of the simulation and represent fixed, simulated time steps. PALAIS adjusts the simulation speed by adapting the rate at which it emits these tick events relative to the passed time. This ensures the simulation results are the same, regardless of simulation speed.

## 5 INTERFACES

Figure 2 depicts a general overview of the interfaces of PALAIS. PALAIS exposes several external interfaces to fulfil the previously mentioned requirements.

### 5.1 Graphical User Interface

For users, the main external interface is the graphical user interface (GUI) provided by the runtime of PALAIS. Its main purpose is to display the data related to the currently active scenario. Most importantly, it displays the current state of the scenario in a 3D game scene. We integrated the open-source rendering engine OGRE [14] with the Qt framework [4] to provide a cross-platform GUI and 3D view. The GUI (figure 3) allows the user to configure certain rendering parameters, such as the camera's 3D orientation, zoom level and viewing direction. The user can also view blackboards of the scenario and actors in the knowledge inspector panel of the GUI.

## 5.2 Scripting API

The scripting API is another external interface of PALAIS. The scripting layer is primarily meant to enable definition of arbitrary scenario logic as well as to facilitate rapid prototyping of algorithms and behaviors. PALAIS integrates a scripting engine to interpret ECMAScript code. The scripting API provides access to the currently loaded scenario and its actors. Scripts are able to read and write knowledge to the blackboards of the scenario and the actors. Lastly, scripts can consume core functionality provided by the runtime environment, e.g. dynamic actor instantiation, destruction and ray casting.

## 5.3 Plugin API

The last external interface to access PALAIS is the plugin API. The plugin system allows dynamic loading of third-party code. This core feature makes PALAIS suitable for integration of existing, custom AI code. The plugin API offers the same functionality as the scripting API, plus some more advanced features. Also, plugins are able to expose their functionality to the scripting layer by installing custom bindings. Custom bindings allow the use of arbitrary interaction patterns between native code in plugins and interpreted code in scripts.

## 5.4 Using Interpreted or Native Code in PALAIS

In essence, either scripting or plugins can be used to implement the same resulting scene logic. In fact, internally, the scripting interface is simply another layer on top of the same functionality. There is a performance overhead associated with the use of the the scripting layer, due to the additional code interpretation. Practically, that overhead means that computationally intensive tasks and tasks that run multiple times per time tick are more suited for implementation as plugins. Thus, the suggested workflow is to make all computationally intensive tasks available to the scripting layer via bindings. The extended scripting API can then be used to orchestrate the scene-specific logic.

## 6 INTEGRATING AN EXTERNAL MODULE

To demonstrate the power of its extension system we extended PALAIS with an external pathfinding module. The module is based on the A* search algorithm [9]. Our implementation of the pathfinding system follows the one described in [12]. A* pathfinding is a technique for determining shortest paths. It allows non-player characters (NPCs) to navigate game worlds. In this role, A* pathfinding is part of the standard repertoire of AI in games. Therefore, it is well-suited to serve as an example for exhibiting the potential of PALAIS. In particular, adding the functionality of the pathfinding module to PALAIS shows how easily existing AI code can be integrated with its environment.

## 6.1 Pathfinding Module

The pathfinding module provides methods for constructing and searching shortest paths on navigation graphs. As is typical for game middleware, the module is implemented in C++. The compiled, executable code is in binary form. It contains native code that depends on the processor architecture. Consequently, to integrate the module, we must exploit the ability of PALAIS to load native code as plugins.

## 6.2 Plugin Integration Workflow

A shared library must conform to a simple, well-defined interface to be loadable in the plugin system of PALAIS. In the current version of PALAIS, said interface consists of just 5 methods. Specifically, it consists of two methods corresponding to the loading and tear-down of the plugin, two methods corresponding to the loading and tear-down of a scenario and one method realising the time tick notification. The methods for the loading and tear-down of plugins give plugins an opportunity to initialise and destroy any general setup structures they require. Similarly, the methods for the loading and tear-down of scenarios can be used to initialise and destroy per-scenario bookkeeping information and to install script bindings with the script engine of the scenario. Finally, the time tick event initiates all time-dependent or regularly scheduled functionality. As a complementary measure, the user can register script bindings to define additional entry points.

### *6.2.1 Example*

As is the case with most custom AI code, our pathfinding module does not conform to the plugin interface. Adapting existing code to the defined interface is the integration effort required to make the functionality of a plugin available to PALAIS. We employ the adaptor design pattern [8] to adapt the interface of our pathfinding module to the interface required by the plugin system of PALAIS. The following steps are necessary to integrate the pathfinding module:

1. First, we use the method corresponding to the initialisation of a scenario to load the navigation mesh of the currently active scenario. A navigation mesh [12] is a continuous representations of the walkable area in a game scene. After loading, the pathfinding module constructs a navigation graph from this navigation mesh. The resulting navigation graph can be searched in response to navigation requests. Furthermore, we install a script binding to make the pathfinding functionality available to scripts. These are the per-scenario steps necessary to provide a pathfinding service.
2. Next, we implement the process of searching a path. The first step in this process is initiated by script code calling the plugin via the binding registered previously. In response, the pathfinding system writes the shortest path to the blackboard of the actor that requested the shortest path.
3. Lastly, we add the actual actor movement according to the plans stored in their blackboards. For this, we use the time tick event: We sequentially check the blackboard of every actor for remaining paths to determine which actors in the current scenario must be moved. Finally, we remove a path node from the blackboard, once the actor that it belongs to reaches it.

This example demonstrates the potency of the blackboard architecture used in PALAIS. Due to the blackboard architecture the plugin system requires only a minimalist plugin interface. As a result, the blackboard architecture effectively decreases the effort required to integrate existing AI code with PALAIS.

## 6.3 Data Visualisation

Procedures for the in-scene visualisation of data are part of the core functionality of PALAIS. In addition to providing rendering

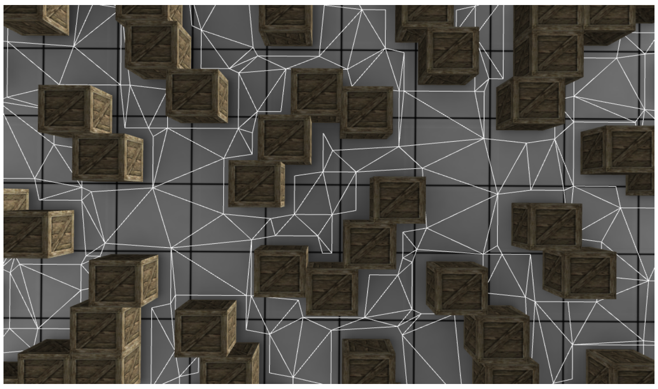


**Figure 4.** A rendering in PALAIS showing the navigation mesh used by the pathfinding module.

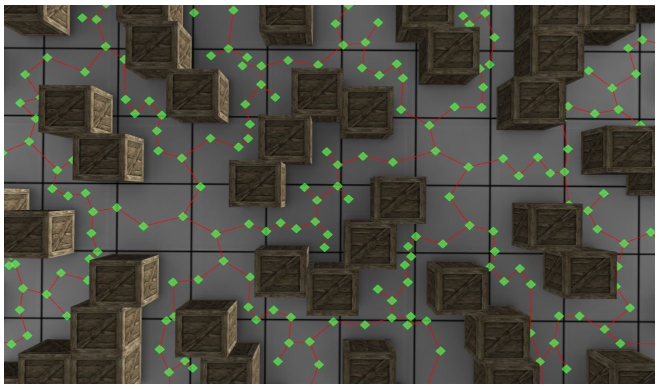


**Figure 5.** A rendering in PALAIS showing the navigation graph constructed from the navigation mesh in figure 4.

of arbitrary textured meshes, PALAIS provides means for rendering coloured primitives, such as lines, circles, quads, cuboids and spheres. As an example, the pathfinding module renders the navigation graph using the visualisation primitives of PALAIS. Figure 4 and figure 5 depict renderings of the navigation mesh and the navigation graph in PALAIS.

### 6.4 Accessing the Pathfinding Module

The plugin installs its script bindings when a scene is loaded. In our pathfinding example, all scripts in a scenario, that includes the pathfinding plugin, can invoke the process to navigate an actor to a goal along a shortest path. The script delegates the computation and handling of the movement to the plugin. This abstraction provided by plugins also allows the reuse of plugins in different scenarios.

## 7 CONCLUSION

PALAIS is a powerful environment for the simulation of AI in games. It caters specifically to the needs of game developers by granting access to its programming interface via interpreted and native code. Our exemplary integration of an external pathfinding module demonstrates that PALAIS is an apt choice for the simulation of scenes that depend on third-party AI libraries. Additionally, the ability to extend PALAIS with plugins lowers the barrier to entry for the usage of the simulation environment, since the same native code, that is used for the simulation in PALAIS, can easily be shared with game engines.

## 8 FUTURE WORK

The work on the simulation environment PALAIS is part of a larger, ongoing project to build a unified framework for game AI development. The framework includes functionality for each of the layers of the game AI model proposed in [12]. Particularly, it encompasses algorithms that facilitate the implementation of movement, decision making and strategy for non-player characters in games. Pathfinding, Behavior Trees [2] and Goal-Oriented Action Planning (GOAP) [13] are among the standard techniques the framework implements. These techniques will be integrated with PALAIS in the form of plugins to provide users with a solid foundation that allows the rapid development of AI behaviors. On the feature side, future work on PALAIS could involve refinement by adding support for physics-based dynamics and statistical evaluation of behaviors.

## REFERENCES

[1] Blender Online Community. Blender - a 3D modelling and rendering package. Retrieved from http://www.blender.org.

[2] Alex Champandard, ‘Behavior trees for next-gen game AI’, in *Game Developers Conference, Audio Lecture*, (2007).

[3] Daniel D Corkill, ‘Blackboard systems’, *AI expert*, **6**(9), 40–47, (1991).

[4] Digia Plc. Qt: cross-platform application and UI framework, 2012.

[5] ECMA International, *Standard ECMA-262 - ECMAScript Language Specification*, 5.1 edn., June 2011.

[6] Martin Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[7] Epic Games. Unity engine documentation. Retrieved from https://www.unrealengine.com/.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design patterns: elements of reusable object-oriented software*, Pearson Education, 1994.

[9] Peter E Hart, Nils J Nilsson, and Bertram Raphael, ‘A formal basis for the heuristic determination of minimum cost paths’, *Systems Science and Cybernetics, IEEE Transactions on*, **4**(2), 100–107, (1968).

[10] Jon Klein, ‘Breve: a 3d environment for the simulation of decentralized systems and artificial life’, in *Proceedings of the eighth international conference on Artificial life*, pp. 329–334, (2003).

[11] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, Keith Sullivan, and Gabriel Balan, ‘Mason: A multiagent simulation environment’, *Simulation*, **81**(7), 517–527, (2005).

[12] Ian Millington and John Funge, *Artificial intelligence for games*, CRC Press, 2009.

[13] Jeff Orkin, ‘Applying goal-oriented action planning to games’, *AI Game Programming Wisdom*, **2**(2004), 217–227, (2004).

[14] Torus Knot Software. Object-oriented graphics rendering engine (OGRE) Engine documentation. Retrieved from http://www.ogre3d.org/.

[15] Unity Technologies. Unity documentation. Retrieved from http://unity3d.com/.

[16] Marco Vala, Guilherme Raimundo, Pedro Sequeira, Pedro Cuba, Rui Prada, Carlos Martinho, and Ana Paiva, ‘ION framework–a simulation environment for worlds with virtual agents’, in *Intelligent virtual agents*, pp. 418–424. Springer, (2009).

[17] Uri Wilensky, ‘Netlogo’, http://ccl.northwestern.edu/netlogo/, Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL, (1999).