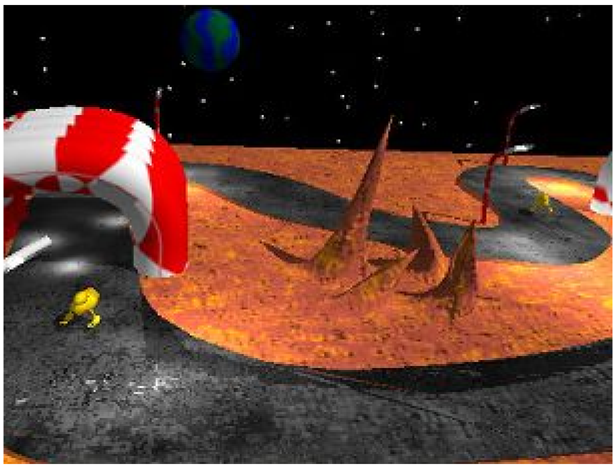
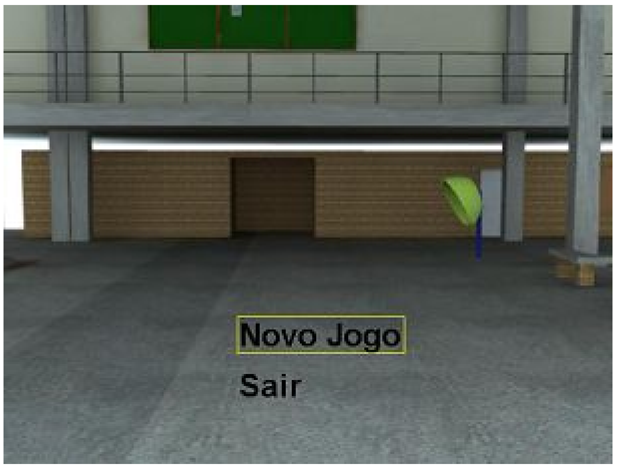
enJine: Architecture and Application of an Open-Source Didactic Game Engine

Ricardo Nakamura, João Bernardes, Romero Tori

Escola Politécnica da Universidade de São Paulo,
Department of Computer Engineering and Digital Systems, Brazil


a) Spaceship Racing



b) Lego Adventure

Figure 1: Screenshots from two sample games made with enJine by students

Abstract

In this paper we present enJine, an open-source didactic game engine, its architecture and some results of applying it as a support tool for teaching Computer Graphics in a computer engineering course. The evolution of the engine's requisites and architecture is discussed, as well as the plans for future development of the software.

Keywords: game engine, game development, educational software tools, computer graphics.

Author's contact:
`{ricardo.nakamura, joao.bernardes, romero.tori}@poli.usp.br`

1. Introduction

This paper presents "enJine" [EnJine Homepage 2006], an open-source game engine based on the Java 3D API and therefore sharing multiplatform and object-oriented characteristics of the Java language. EnJine's main purpose is to serve as a didactic tool to aid in teaching game design and computer science, especially computer graphics and software engineering subjects. For this reason, the project is aimed at providing an open, well-structured game engine that remains simple enough to allow the development of relatively complex games in a short time span. Given these characteristics, enJine may also be used as a test-bed for research in game development, virtual and augmented reality and other related fields.

The enJine project has evolved considerably along its nearly four years of existence. Currently, the following features are among the ones that may be found in enJine, despite its simplicity of use:

- 3D rendering based on Java 3D (which, in turn, currently uses DirectX or OpenGL);
- Stereo sound;
- Skin-and-bones animation;
- Multi-stage collision detection (currently only implementing subspace and bounding volume filters) and ray-casting collision detection;
- An abstract input layer to simplify the use of non-conventional input devices;
- 2D Overlay;
- .X and .OBJ model loaders;
- Core classes constituting a game's basic architecture: the game, its stages, game objects (which are very flexible) etc.;
- A framework package to facilitate the creation of certain games (currently only single-player games);
- Separation between graphical and logical update cycles.

EnJine is a stable tool, tested by a considerable number of users with concrete results (Figure 1 shows screenshots from two games made by students, for instance), but is, regardless of that stability, an ongoing project with much to be added. Features planned for future versions range from Networking, Augmented Reality and Shader support (being studied now) to Scripting, Artificial Intelligence and Physical Simulation (in a longer time frame).

The evolution of the enJine project and its requirements are presented in section 3, but first, section 2 discusses some related works. Section 4 discusses enJine's architecture and features. Section 5 gives a brief overview of the process of creating a game with enJine, while the last two sections discuss the results and conclusions obtained so far with this work.

2. Related Work

One of the main motivation for the development of enJine was that at the time the project started (2002) no similar work had been found either for engines for educational games or for didactic game engines. Even today there are few related initiatives and none of the ones found and discussed here has the same approach and objectives as those of enJine.

Coleman et al. [2005], for instance, have developed a didactic game engine for teaching game design and programming, called Gedi. This tool, although having similar didactic objectives, as is the case of enJine, is a 2D engine and is built on top of the Direct3D API.

Peternier et al. [2006] have developed a platform for teaching computer graphics, named MVisio, composed of a set of compact applications, used to teach specific computer graphics techniques and algorithms, and a pedagogical-oriented graphic engine, for practical developments. Unlike enJine, MVisio is not aimed to game development, being a more general-purpose 2D/3D graphics engine, instead.

Another approach can be found in Game Maker [Game Maker 2006], which is a didactic tool to teach game programming. It is focused on 2D games (although some features for 3D rendering are available) and presents both a visual programming interface and a customized object-oriented scripting language.

As stated by Peternier et al. [2006], there are a lot of free and open graphics engine, like Ogre [Ogre 2006] and Crystal Space [Crystal Space 2006], but these libraries are not designed for didactic purposes, consuming too much time in learning effort. Another drawback for these non-didactic tools is that their architectures often are not sufficiently clear, light, elegant and easy to extend for teaching purposes. That may be a consequence of an effort towards optimization or of such an architecture not being among their requirements.

3. Project History

EnJine was initially a tool to help in the creation of educational games. The need for such a tool was perceived in an earlier project, described by Nakamura et al. [2003a], which consists of using a commercial game engine to develop an educational "mod" game. In that work, Nakamura et al. [2003a] also discuss which requirements an engine should have to better aid in the creation of educational games.

When the enJine project began in March 2003 it had two main goals: to create a game engine satisfying the requirements established in the previous research and to study the usage of formal software development methodologies when creating a game engine. Nakamura et al. [2003b] describe the creation of enJine's first version.

At the same time the enJine was being developed, its first educational game, called FootBot, was also being designed and implemented. This game only reached its prototype stage, however. At that moment a few problems in enJine's first implementation were noticed, and it was decided to create a completely new version, based on the experience gained with the first one. Among these problems we highlight: an exceedingly complex architecture, especially in the multiplayer game-related classes, stemming from the will of providing a very flexible game engine, and architectural failures that led to little decoupling in the implementation of the input subsystem.

The second version of EnJine was developed with the main objective of providing a simpler, more focused architecture that would better fit the project's initial goals. During development of this version it was realized that the simplicity and clearness of enJine's new architecture, its object-oriented approach and the use of scene graph, provided by Java 3D API, made it a good tool for teaching Computer Graphics, Software Engineering and Game Development concepts. As a result it was decided that the main focus of enJine should be its application as a didactic tool in itself instead of as a didactic game development tool. Given this change of focus in the project, development of version 2 was interrupted and version 3, which is the subject of this article, was started. With these new goals in mind some requirements received less priority, such as the "models library" mentioned by Nakamura et al. [2003a] and network/multiplayer infrastructure. New requirements were also included, such as animation support and model loaders. Those features that were omitted in this third version should be present in the next one.

Today, enJine version 3 is being applied as a support tool for teaching Introductory Computer Graphics in undergraduate Computer Science and Computer Engineering Programs. Some of its early results in this field are discussed in Tori et al. [2006a]

and Tori et al. [2006b], however those works are dedicated to the teaching methodology rather the architecture and technical details of enJine. Figures 1 and 8 show screenshots from a few sample games made by students (in approximately 6 weeks only, from the time they were introduced to enJine to game completion) during these courses.

4. Architecture

Conceptually, enJine's architecture is divided in three layers, as illustrated in Figure 2. At the bottom lay the Java Standard Edition API classes, as well as libraries such as Java 3D, upon which the game engine's functionality is built. The middle layer contains the different enJine modules, which are responsible for providing services such as graphics rendering, player input, sound output, and general game management to user applications. The topmost layer contains a game framework that implements some patterns of use of the enJine services. For instance, the framework includes a `SinglePlayerGame` class that contains code to set up the enJine components necessary to create a single-player game. The game itself is in a layer just above the framework and is capable of accessing all the three layers beneath its own.

The framework layer is intended to reduce development time and simplify initial contact with the enJine API, but its use is not mandatory. It is expected that experienced users will extend the classes present in the standard enJine modules to fit their specific needs, and most of those classes have been designed to allow for such flexibility and extensibility.

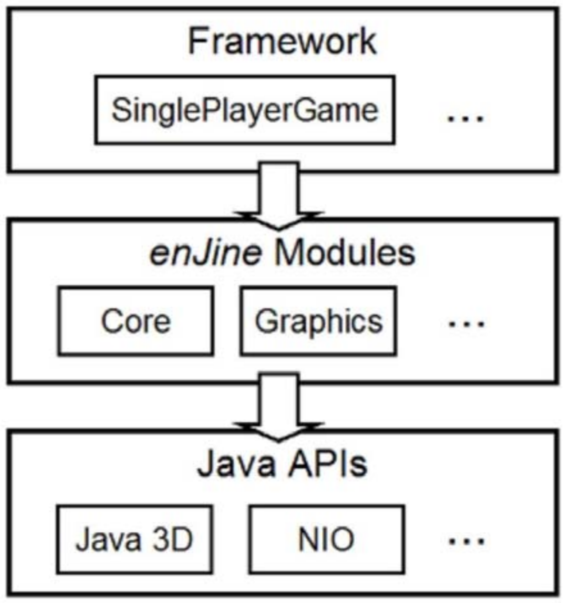


Figure 2: enJine layered architecture

The different enJine modules are implemented as Java class packages. The existing packages, as well as the dependencies existing between them, are shown in Figure 3.

The packages that are planned but not yet implemented in enJine are: a network package, which is being implemented at this time, a haptics package within the I/O module, Scripting, Artificial Intelligence and Physics packages.

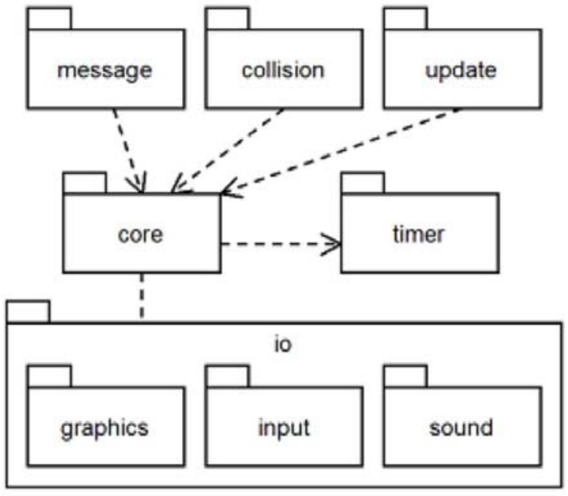


Figure 3: enJine packages and dependencies

4.1 Core Classes

The core package contains the main entities of the game engine: the classes `Game`, `GameState` and `GameObject`. Together, these classes can be used to model and implement the game's structure and logic, relying on the other modules for services such as graphics rendering.

The `Game` and `GameState` classes implement the "state" design pattern [Gamma et al. 1995]. This way, subclasses of `GameState` can be used to represent game behaviors such as different game levels or different game modes (such as a world map and an inventory management screen). `Game` is the class that encapsulates the game's main loop.

The `GameObject` class can be used to represent different game entities such as player and computer-controlled characters and other interactive elements.

Each instance of `GameObject` aggregates one or more objects that implement specialized services provided by the enJine, as shown in Figure 4. This modular design allows the creation of game entities with very different characteristics and behavior through the reuse of the same common components.

The choice of using classes instead of interfaces for these services was made to stress the distinction between logical and visual behavior in game entities. For instance, with interfaces, the user might be able to implement all services in a single class in a way that they are tightly coupled and any sense of independence between logical behavior and rendering is lost.

Besides, this choice encourages the use of object aggregation, which is considered a good practice in object-oriented programming.

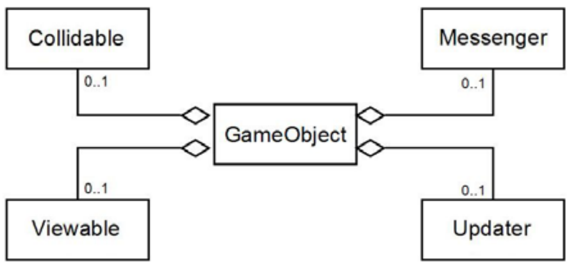


Figure 4: GameObject architecture

When instantiated within a `GameObject`, the `Viewable` class, indicates that that game entity requires graphics rendering services. All visual changes to objects (including animation) should be encapsulated in a `Viewable` object.

The `Updater` class indicates the object has some sort of dynamic behavior and thus requires that its logical state be constantly updated. This dynamic behavior may be the result of player input, some Artificial Intelligence algorithm or Physical Simulation, for instance.

An instance of `Collidable` within a game entity indicates that that object should be included in collision detection computations. This class encapsulates the entity's bounding volume and its response to collisions.

Finally, the `Messenger` class allows the exchange of messages between game objects, one of the main requirements for educational games (or simply for games with a more complex narrative or interaction) mentioned by Nakamura et al. [2003a].

Currently, elements of game scenery are also represented by subclasses of `GameObject`. While this solution is correct in many cases, one of the future extensions of enJine is to support more efficient game map structures.

4.2 The I/O Subsystem

The I/O subsystem actually comprises three packages, which are responsible for the services of graphics rendering, sound output and user input handling in enJine.

The graphics package is built on the functionality provided by the Java 3D API in retained mode. By design, the Java 3D classes are exposed through the enJine, so that it can be used as a tool for didactic activities related to that graphics library. Moreover, this allows users to explore any new capabilities that are added to Java 3D in future versions.

Rendering in the enJine is performed through the `Viewable` abstract class, following the Observer design pattern [Gamma et al. 1995]. Objects that subclass the `Viewable` class must provide a Java 3D scene graph branch that corresponds to the object's visual representation. Whenever the game view is rendered, each `Viewable` is notified so that they can update that representation.

The ability to load and render skinned meshes has been recently added to enJine. Users can create models in software packages that support this style of animation and export them in the ".X" format, and then load and use these models in their games. Currently there are plans to support different model formats and to develop a tool to optimize model representations for enJine.

Currently, enJine provides basic functionality regarding sound and input. This includes playback of popular sound file formats, with pan and volume controls, and support for keyboard and mouse inputs. However, as the input system decouples specific input methods and the intended actions, it is possible to extend it to other input devices.

Figure 5 shows the relationship between objects from the different classes of the input system. Each input device (such as a joystick or keyboard) is represented by a subclass of `InputDevice`. `InputDevices` have a set of `InputSensor` objects that represent elements such as joystick buttons or axes and keyboard keys. On the game side, instances of the `InputAction` class are used to model player commands, such as "jump", "move left" or "open menu". The class `InputManager` is used to bind `InputActions` to `InputSensors`.

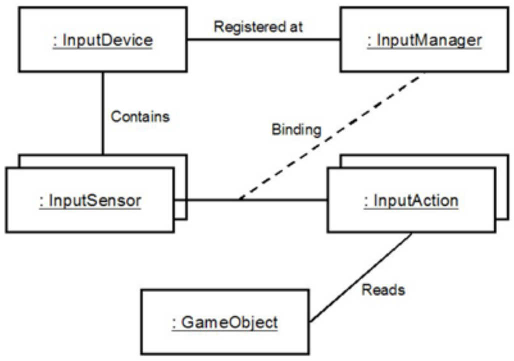


Figure 5 - Input System

4.3 enJine Framework

As the enJine was designed to be used as a didactic tool, it was necessary to provide features that would help learning to use it. This is the primary function of the enJine Framework layer.

The Framework provides a set of classes that exemplify how to use the other enJine packages to construct a game. Furthermore, it contains a `SinglePlayerGame` class, which implements much of the code needed to integrate and initialize the objects needed to create a single-player game.

Figure 6 shows how the `SinglePlayerGame` class contains a set of four managers, all singletons [Gamma et al. 1995], to control the update of individual objects.

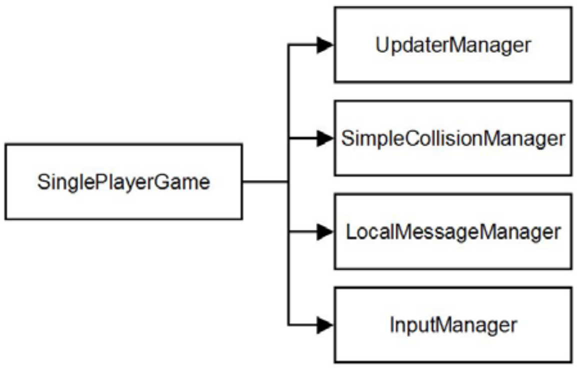


Figure 6: SinglePlayerGame

4.4 Future Development of the enJine

There are many plans to extend the functionality of enJine, while still maintaining it useful as a didactic tool for game development. Two of the most interesting additions are support for augmented reality games and networked games.

Previous versions of the enJine featured multiplayer game support that was versatile but very complex. This was found to go against the goals of the project and for this reason, it was removed. Now a new module for network support, which will be restricted to a client-server model and present other restrictions to make it simpler, is being implemented by undergraduate students and will be integrated in one of enJine's future versions, as soon as it is properly tested. Once the network package is available, classes to help creating multiplayer games will be added to the enJine Framework.

At the same time, there has been development of new enJine functionality to support the processing of video captured from a webcam using JarToolKit [2006] that will be integrated in enJine's future versions. This video data can be used both as a new input method, by interpreting user movements and gestures, and to allow the inclusion of the player's image in the game through extensions of both the input and graphics packages.

The use of shaders in enJine is another feature planned for future versions in a very near future, making use of the shader functionality in Java 3D available in versions 1.5 and higher. The integration of Scripting, Artificial Intelligence and Physical Simulation packages still has no estimated time frame.

5. Using enJine

This section presents a sample scenario of use of the enJine, to illustrate how its components interact. To create a single player game, users have two options. They may subclass the core `Game` class or use the `SinglePlayerGame` class in the framework package. For new users of the enJine and for most simple games, the latter choice is preferable.

Assuming the `SinglePlayerGame` class is adopted, the user must create one or more `GameState` subclasses to represent the different modes and levels of the game. A simple game with a title screen and one level might require two of those classes. Each `GameState` implementation requires code to initialize and remove the necessary game resources. Figure 7 illustrates a sample set of game states and transitions between them, represented as a finite state machine.

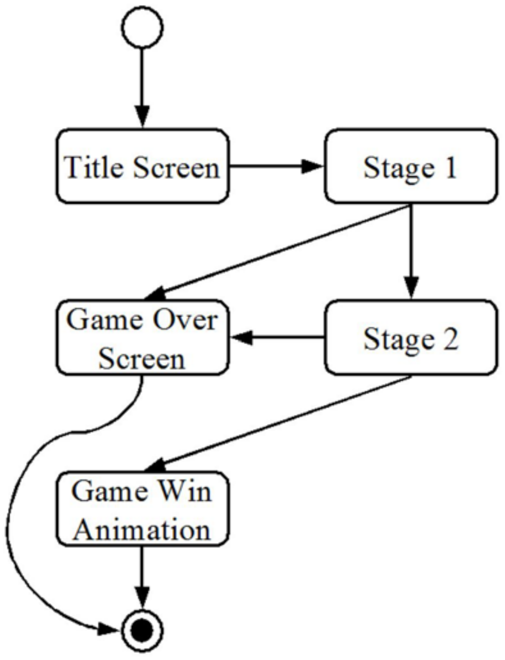


Figure 7: A sample set of game states

The game developer must also create one or more subclasses of `GameObject` to represent different types of entities in the game. In many cases, a single subclass containing common state information is enough, since appearance and behavior is defined through aggregate objects. Therefore, the developer must create classes that instantiate objects from classes such as `Viewable` and `Updater`, which will be used to build the complete game entities. Except in very simple games, it is advisable that an object factory [Gamma et al. 1995] be implemented to perform the creation of those entities.

It is also necessary to create one subclass of the `Viewable` for each type of object with different visual behavior. For instance, if many objects in the game use skinned models that are loaded with enJine's .X loader, a single class can be created to load and animate any .X model. Different `GameObjects` can use objects of this class.

For most games, it will be necessary to create one or more subclasses of the `Collidable` class. These classes are responsible for defining the collision volumes for the objects and for providing their collision-response behavior.

Lastly, the game developer must add code (usually to a `GameState`) that binds an input device to the game actions that can be performed by the player. This way, one or more game entities can receive and process the input actions.

Once the basic game structure is completed, it is possible to add features like an overlay to display game data superposed to the game scene. This can be accomplished by creating a class that implements the `Overlay` interface from the io.graphics package and binding it to one of the active game views. This is typically performed at the initialization stage of a game state.

Sound effects and background music can be added to a game in enJine using the `GameAudio` and `SoundElement` classes. Each `SoundElement` object represents a single data file of either sound effects or music. These objects are added to an instance of `GameAudio`, which provides methods for playback and control of those elements.

6. Results

EnJine 3.0 has been successfully used as a support tool for teaching Computer Graphics [Tori et al. 2006a] [Tori et al. 2006b]. Its open and easy-to-learn architecture, constructed over the object-oriented scene graph data structure provided by Java 3D, has proved to be an efficient didactic platform, allowing students to develop relatively complex game projects in as few as four laboratory classes. Figures 1 and 8 show some of these projects. Using a game engine in computer graphics classes is a good way for applying to practical students activities almost all fundamental concepts, algorithms and techniques seen in theoretical classes. Conversely, problems and questions arisen during game project activities, as well as questions concerning the low level computer graphics implementation of the game engine itself can be discussed back in theoretical classes.

Besides its didactic applications enJine has also been used as a platform for researches in game technology. Its open, simple and extensible architecture makes it easy to experiment with new techniques, algorithms and solutions for game development. Currently some researches in augmented reality technology applied to game development are taking place at INTERLAB. Eventually the results of these experiments can be incorporated in future versions of enJine.

7. Conclusion

This paper presented enJine, an open-source didactic game engine developed at INTERLAB, Interactive Technologies Laboratory, at Escola Politécnica da Universidade de São Paulo. EnJine has been used as a support tool for teaching introductory computer graphics undergraduate courses and as a platform for researches in new techniques and technologies applied to game development, especially augmented reality technology and software engineering for game development. Its source code, pre-compiled distribution, documentation and samples are available at the website http://enjine.iv.incubadora.fapesp.br.

Some current issues on enJine project are: providing more complete support for sound effects and music; including more utility classes in the framework package; and optimizing the code, especially for model loading, animation and collision detection.

Future plans for enJine project include: new features such as shaders, network and multiplayer functionality; incorporation of research results, such as augmented reality capabilities, gesture recognition and real time video avatar; application of enJine as a didactic tool for other disciplines in computer science and computer engineering programs, such as software engineering, artificial intelligence and human-computer interface design; use of enJine in game design courses.

Besides the integration of new features and expansion of enJine's didactic use, future developments include the optimization of parts of the code, such as the skin-and-bones animation algorithm, for greater speed, without complicating enJine's architecture. Stress and performance tests still need to be executed in different hardware configurations to better quantify enJine's reliability, performance and dependence on hardware such as graphics accelerator cards. The creation of an interactive and graphical IDE (integrated development environment) is also being studied. Such tool would be aimed to increase the productivity of implementations based on enJine, without preventing users from direct accessing all desired levels and resources of enJine architecture.

Acknowledgements

The authors would like to thank Escola Politécnica da Universidade de São Paulo and Centro Universitário Senac for their support, and Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), for hosting the project "enJine" in its "Incubadora Virtual de Conteúdos". The authors would also like to thank

all their former students and the researchers from Interactive Technology Laboratory (Interlab) that directly or indirectly participated in the development of “enJine” software.

References

COLEMAN, R., ROEBKE, S. AND GRAYSON, L., 2005. Gedi: A Game Engine for Teaching Videogame Design and Programming. *Journal of Computer Science in Colleges,* 21(2), 72-82.

CRYSTALSPACE 3D HOMEPAGE [online]. Available from: http://www.crystalspace3d.org/ [Accessed 28 August 2006].

ENJINE HOMEPAGE [online]. Available from: http://enjine.iv.fapesp.br [Accessed 28 August 2006].

GAME MAKER [online]. Available from: http://www.gamemaker.nl/ [Accessed 10 October 2006].

GAMMA, E., HELM, R. JOHNSON, R. AND VLISSIDES, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. USA: Addison-Wesley.

JARTOOLKIT HOMEPAGE [online]. Available from: http://jerry.c-lab.de/jartoolkit/ [Accessed 28 August 2006].

NAKAMURA, R., TORI, R., BERNARDES JR., J. L., BIANCHINI, R. AND JACOBER, E., 2003. A Practical Study on the Usage of a Commercial Game Engine for the Development of Educational Games. *In: Proceedings of the Games and Digital Workshop, 4-5 November 2003 Salvador*. [S.l.]: SBC, 1 CD-ROM.

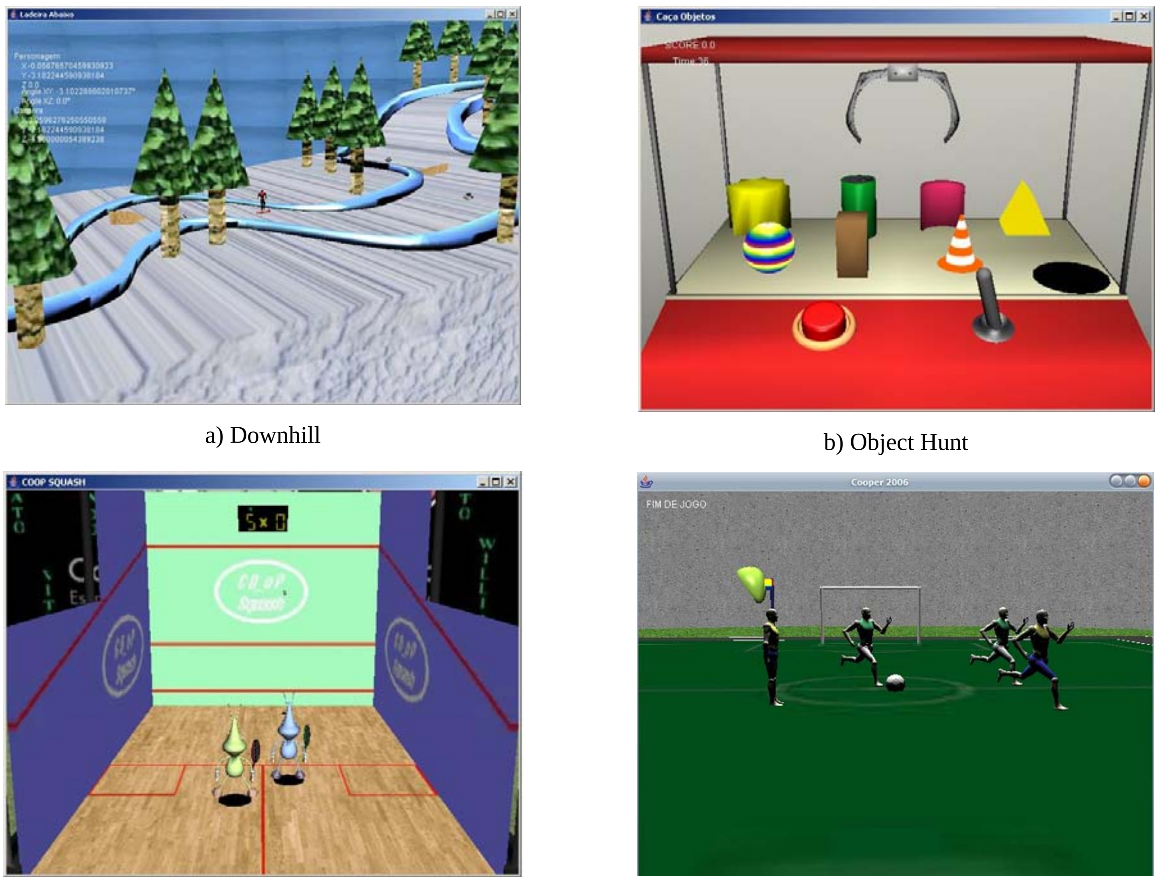
NAKAMURA, R., TORI, R., JACOBER, E., BIANCHINI, R., AND BERNARDES JR., J. L., 2003. Development of a Game Engine Using Java. *In: Proceedings of the Games and Digital Workshop, 4-5 November 2003 Salvador*. [S.l.]: SBC, 1 CD-ROM.

OGRE 3D HOME [online]. Available from: http://www.ogre3d. org/ [Accessed 28 August 2006].

PETERNIER, A., THALMANN, D. AND VEXO, F., 2006. Mental Vision: a Computer Graphics teaching platform. *Lecture Notes in Computer Science: Technologies for E-learning and Digital Entertainment,* 3942, 223-232.

TORI, R., BERNARDES JR., J. L. AND NAKAMURA, R., 2006. Teaching Introductory Computer Graphics Using Java 3D, Games and Customized Software: a Brazilian Experience. *In: Proceedings of Siggraph 2006 Educators Program, 30 July – 3 August 2006 Boston*. New York: ACM Press.

TORI, R., NAKAMURA, R. AND BERNARDES JR., J. L., 2006. Ferramentas e metodologia para ensino de fundamentos de computação gráfica em cursos da área de computação. *To be presented at the II Workshop de Computação Gráfica e Educação, 8-11 October 2006 Manaus.*



a) Downhill

b) Object Hunt

c) Coop Squash

d) Cooper 2006

Figure 8: Screenshots from a few more sample games made with enJine by students