Using a Didactic Game Engine to Teach Computer Science

Ricardo Nakamura João L. Bernardes Jr. Romero Tori
{ricardo.nakamura, joao.bernardes, romero.tori}@poli.usp.br

INTERLAB – Interactive Technologies Laboratory
Departamento de Engenharia de Computação e Sistemas Digitais
Escola Politécnica da Universidade de São Paulo
Av. Prof. Luciano Gualberto, travessa 3 no. 158 - CEP 05508-970 - São Paulo, SP

1. Introduction

This tutorial presents enJine, an open-source didactic game engine developed in Java, and discusses how to use it to aid in teaching game development and computer science or engineering topics, especially Computer Graphics. EnJine is developed at the Interactive Technologies Laboratory (Interlab) at University of São Paulo and is available as free software under the GPL License. It uses the Java 3D API and shares the multiplatform and object-oriented characteristics of the Java language. In its first version, this game engine was not a didactic tool in itself, but rather a tool for the creation of didactic games. All three of this tutorial's authors, however, have been involved for at least a few (and in one case quite a few) years in teaching courses in computer science and computer engineering programs and have experimented successfully with the use of game programming in this activity. The realization that enJine had a good potential as a didactic tool in itself was one of the main points behind the creation and release of a new version of it and has so far proven correct.

The following section describes enJine in detail, including a brief introduction to Java 3D, and a discussion of its architecture. Section 3 focuses in enJine's didactic use, exploring its potential for several courses, and sharing the authors' concrete experience in using it in Computer Graphics courses, providing useful tips and recommendations for educators interested in using the enJine in a similar fashion. Section 4 presents an example of how to create a relatively simple 3D game using it.

While this document focuses on the objective aspects of using enJine to serve as a reference and so dedicates more space to its general use than to its use in education, the actual tutorial will, after providing some basic knowledge of enJine, focus on the educational aspects in a more subjective way, discussing the authors' experiences and inviting a more active participation of the audience.

2. EnJine

As was just discussed, enJine is set of software components built over Java and Java 3D to provide services and a structure to the creation of games. The main features it offers are:

- A set of Java classes to allow flexible representation of game elements such as characters, objects and scenarios;
- Basic infrastructure for a game application, including the capability for complex interactions between game objects through a message exchange mechanism;
- 3D graphics rendering through the Java 3D API, so that students can have contact with Java 3D and especially the scene graph;
- Support for basic sound output and user input (currently only through keyboard, but easily extensible to other devices due to enJine's abstract input layer).

Other services, such as skin-and-bones animation (which will not be explored in this tutorial), multistage collision detection functionality or model loaders are also present in enJine's current version, and many other features are also planned for future versions. Among those currently being implemented is network support, for instance.

Unlike most of the large variety of game engines available today, enJine's main objective is to realize these functions in a didactic way. What this means is that it strives for simplicity, clarity and a correct (from a software engineering point of view) architecture. These goals are considered more important for enJine's didactic use than even computational performance, which is one of the most important factors for most other engines. That does not mean that performance with enJine is necessarily poor, however, but only that it is not the main priority, and since it has proven adequate to the tasks the tool has been put to so far, it has not even been thoroughly tested yet.

This text will focus on the didactic use of enJine, its main objective, but it bears mentioning that this tool is also used at Interlab to test new game-related technologies, such as Augmented Reality games. Before it is possible to delve deeper into enJine and discuss its use in teaching, however, at least a basic grasp of Java 3D is necessary.

2.1. An Overview of Java 3D

Java 3D is a 3D rendering API, initially developed by Sun. Since 2004 its development has turned to the open-source model under an active community. One of its most important and defining characteristics is the use of *Scene Graphs*, an hierarchical way to structure the virtual world that will be presented to the user. The scene graph is a directed acyclic graph. Its elements, or nodes, are connected arcs, which indicate the hierarchical relationship between them. If nodes A and B are connected, for instance, in the direction AB, it means that B is hierarchically subordinated to A and we say A and B are parent and child, respectively, in relation to each other. Because the graph is acyclic, the directed arcs can form no loops. A further restriction in the Java 3D scene graph is that every element can only have one parent. The scene graph is thus reminiscent of a tree structure, and the tree nomenclature is conventionally used to refer to it.

These hierarchical relationships are especially useful when describing a scene not only to organize it but specially to represent relative, hierarchical transformations between related scene elements. The 3D model of a car, for instance, may suffer translations or rotations as a whole to indicate its position, but its wheels must suffer not only that transformation, but two more rotations, to show that they are rolling and turning. This is indicated in a scene graph by making the global transformation (the car's position) the parent of the local transformations (the position of the wheels). The state of a leaf (a node without children) in the graph is thus obtained by the combination of the states of the nodes above it in the hierarchy, i.e. its parents, their parents and so on. Figure 1 illustrates a scene graph using their conventional representation.

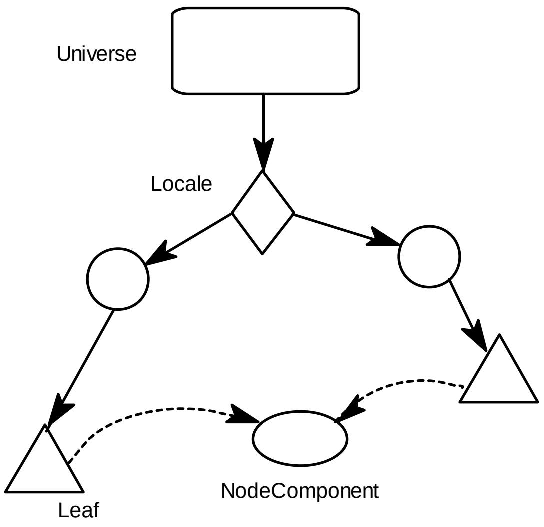


Figure 1 - A sample scene graph

The first nodes are the universe, which has no parent, and the locale. `Universes` may contain more than one locale (useful for large universes), but usually do not. The locale's children must always be groups (represented by the circles in figure 1), in fact, `BranchGroups`. Group nodes bring the other nodes together in the hierarchical structure. Aside from the `BranchGroups`, that do only that, there are also the `TransformGroups`, which have an associated 3D geometric transformation (a `Transform3D` object), applied to all the nodes below that `TransformGroup`. The leaves, represented by triangles, correspond to actual scene elements, such as 3D geometry, dynamic behaviors or visual and sound effects. Finally, node components are accessory to the other scene graph objects, serving to define some of their parameters, such as the `Appearance` component. They are not actually part of the graph and thus can be "shared" by more than one object, as shown in figure 1. To indicate this relationship, the arcs to node components are represented by dashed lines. Section 2.3 will show several examples of the use of Java 3D along with the enJine and its `Viewable` class.

2.2. Architecture

The architecture of enJine has evolved significantly from its first version, especially to make it more didactic. One example of such changes is that user now can (and in fact must) access Java 3D functionality directly (because that was considered worthwhile in a Computer Graphics course), while in the first version there was a layer above the Java 3D API, isolating it from the user.

Currently, at the highest level of abstraction, enJine can be divided into three layers, as shown in Figure 2. At the bottom are the external libraries, which are used by enJine to communicate with the operating system and devices. Some of these libraries also provide more complex services. This is the case of Java 3D which is used as the rendering solution for enJine.

The middle layer of this architecture consists of enJine's main modules, which implement the game services such as video rendering, sound output, collision detection and others. Many of the classes found at this layer are abstract and provide the structural organization, as well as algorithmic templates, which can be extended by users to fit enJine to their needs.

At the top layer lies the enJine Framework, which is a set of classes intended to provide a more complete and easier-to-use solution for the development of games. The framework includes concrete implementations of many enJine classes suited for specific uses. For instance, the `SinglePlayerGame` class implements the base code to aggregate the enJine service providers needed for a single-player computer game. The framework layer also includes utility classes and demonstration programs.

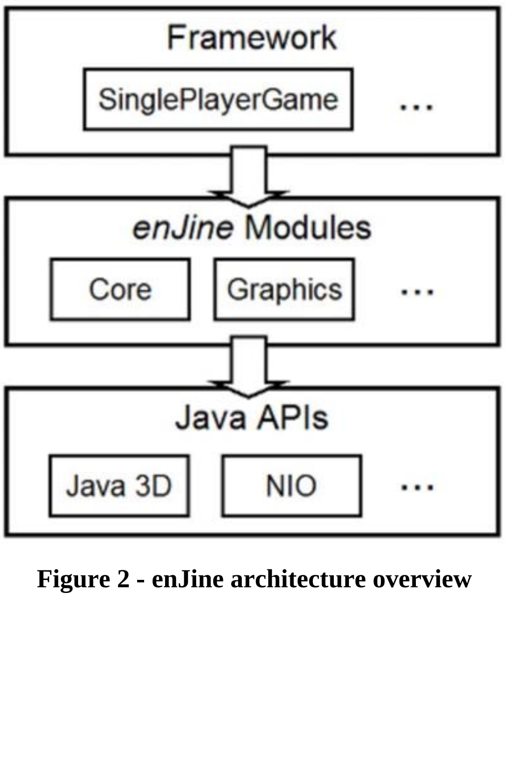


Figure 2 - enJine architecture overview

2.2.1. Packages

EnJine is implemented in Java, and is divided into a set of packages that help organizing the classes according to their purpose. Figure 3 shows the packages that correspond to the middle layer of the enJine architecture, discussed before.

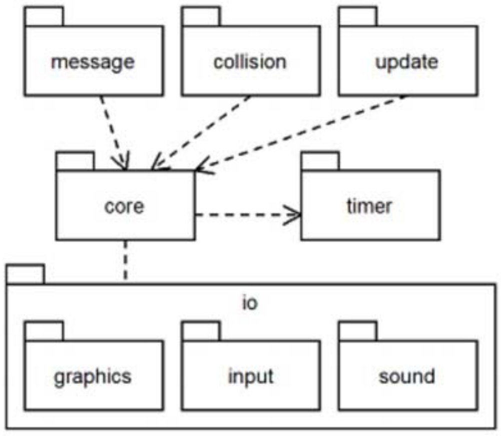


Figure 3 - The main enJine packages

The `core` package contains the classes that represent the fundamental pieces of implementation of a game using enJine, such as `GameObject`, `Game` and `GameState.` Given the importance of these classes to understanding enJine, they will be explained in more depth in later sections.

The `timer` package is a very small set of classes that are intended to decouple the source of timing information from the rest of the code. Timing information is useful in games to drive simulation and animation loops. There are many different classes provided with the Java Standard Edition, as well as other libraries, which can be used to obtain information about how much time has passed during the execution of a program. Each of these classes has their own characteristics and shortcomings, so this package attempts to hide their details under a uniform `Timer` interface.

The `io` package is devoted to the input and output services, and for this reason, it is further divided into the `graphics`, `input` and `sound` sub-packages. Each of them will be explained in a separate section of this text.

The `collision` package contains classes that work together to create a flexible collision detection system.

The `update` package is devoted to the classes that implement the game's logical simulation loop. EnJine attempts to make it very clear that the simulation and rendering loops can be decoupled, both conceptually and in implementation. This is one of the reasons to place the simulation classes in a separate package.

The `message` package offers a message exchange system that can be used in the implementation of games to create communication systems between game entities. This feature can be useful for complex interactions among objects, scripting systems and artificial intelligence for game characters.

Besides the main packages presented here, enJine also contains three other packages, which belong to the top architectural layer: `framework`, `demo` and the recently added `loaders` package. The `framework` package contains a set of classes that are aimed at making enJine easier to use, especially for new users and students. The `demo` package contains a set of demonstration programs that are meant to illustrate how to use basic and advanced features of enJine. And lastly, the `loaders` package is intended to become a repository of classes to aid in loading assets (models, sound samples, textures etc.) in different file formats into enJine.

2.2.2. Game and GameState

The `Game` and `GameState` classes are used to define the overall structure of a game built with enJine. The `Game` class implements a standard main loop in the form of a template method that calls several abstract methods at key points during the loop. While the `mainLoop` method of that class cannot be overridden in subclasses, those abstract methods provide the necessary opportunities for customization of the game's behavior. The `Game` class also implements code to manage objects of `GameState` subclasses. Each `GameState` represents a different mode of operation of the game. The most common usage for `GameStates` is to implement title screens and game levels. However, they can also be used for internal modes of operation in a game, such as an equipment management interface for an adventure game.

At any time during the execution of a game with enJine, there is one active `GameState` that corresponds to the current mode of operation for the game. The `GameState` class contains a set of abstract methods, which are called from the game's main loop when that state is active. Figure 4 shows a sequence diagram to illustrate some of the calls performed by the `Game` to the current `GameState`.

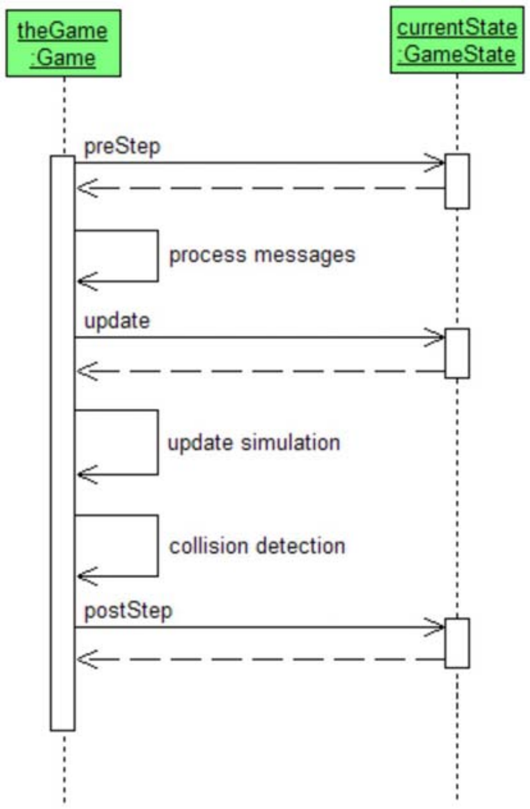


Figure 4 - interaction between Game and GameState

2.2.3. GameObject

The `GameObject` class is used to represent any entity in a game. This includes parts of the scenario, player-and computer-controlled characters, interactive items, projectiles etc. To represent such diversity, enJine adopts an aggregation model, in which the `GameObject` class can be connected to one or more objects, which provide specific services. In the current version of enJine there are four service providers: `Viewable`, `Updater`, `Collidable` and `Messenger`. Each of them is an abstract class, which defines the public interface and general behavior of the corresponding service. In general, these classes work according to the Observer design pattern: one or more of their methods will be called by a subsystem in enJine to notify the object about a specific event.

The `Viewable` class is used by game entities that require a visual representation. Therefore, it might be the most frequently implemented service provider. The public interface for this class consists of two methods: `getView` and `updateView`. The first one is called by the rendering subsystem when the object is added to the game. It must return a Java 3D scene branch containing the desired graphical representation for the object. The other method is called every time the rendering subsystem is about to generate a new view of the scene, so that the objects can update their visual representation.

The `Updater` class is used to indicate that a game entity has some kind of dynamic behavior. These include player-controlled objects, characters controlled by artificial intelligence or objects responding dynamically to stimuli through a physical simulation system, for instance. This class contains an `update` method that is called at each iteration of the game's main loop and should be used to perform updates to the logical state of the entities (while the Viewable should be responsible for updating the visual model of the object according to its logical state).

The `Collidable` class indicates that the object must be included in the collision detection tests. This class includes a method that gets called whenever the object collides with another.

Objects that intend to communicate through the message exchange mechanism implemented in enJine use the `Messenger` class. This mechanism allows objects to exchange information without the need to store direct references to each other. It also allows sending "broadcast" messages that are received by all entities that have a Messenger object. These features make the message exchange system suitable to the development of scripting systems, artificial intelligence implementations and other complex interactions between objects.

The `GameObject` class does not contain any attributes to represent the internal state of the game entities. Instead, it is assumed that each game will create an appropriate subclass with its optimal set of attributes. For instance, objects in a racing game might need only store a pair of coordinates and their orientation on the track, whereas a flight simulator might require that objects store 3D coordinates and an orientation vector.

Figure 5 illustrates all that has been explained about the `GameObject` class. In that diagram, `MyObject` is a subclass, which extends `GameObject` by adding the attributes necessary to a specific game. At the same time, it inherits the `GameObject`'s ability to connect to one or more service provider classes.

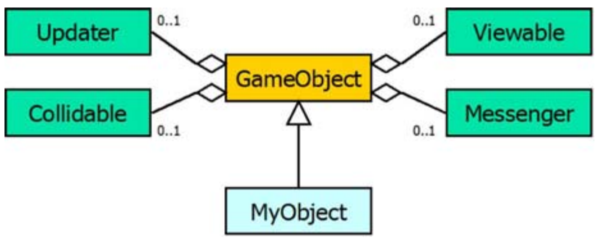


Figure 5 - the GameObject structure

2.2.4. Graphics

EnJine graphics are implemented using the Java 3D API. Therefore, all 3D models presented in the game must be built as Java 3D scene graphs, either through procedural generation of meshes or by loading and processing 3D model files.

The main graphics class in enJine is called `GameDisplay`. It is responsible for initializing the basic Java 3D objects and managing the game window. EnJine also uses a non-public class called

ViewBehavior to drive the visual update loop of the game. The ViewBehavior class is a subclass of the Java 3D Behavior class and operates under the Java 3D behavior model. It stores a list of all Viewable objects in the game and calls their updateView method before a new frame is rendered.

The Camera class encapsulates a Java 3D view branch, configuring the necessary components and providing services to customize parameters such as front and back clipping plane distances.

EnJine supports multiple simultaneous game views, to implement split screens, by having two distinct classes that implement display functionality. The GameView class corresponds to a viewport into the game world. Each object of this class must be associated to a Camera object. The GameWindow class is a container that can hold one or more GameViews in a user-specified format.

The graphics subsystem of enJine also includes an overlay system. It is used to draw 2D graphics on top of the rendered scene. Its main use is to create user interfaces and display game information to the players. The overlay system is implemented through the OverlayCanvas class and the Overlay interface. A game that wishes to display an overlay need only implement that interface in a class and register a corresponding instance with the setOverlay method of the GameView.

2.2.5. Collision

Finding all the collisions occurring in a set of objects is a problem with computer-intensive solutions. The naïve approach to this problem requires at least N^2 comparisons between objects, where N is the total number of objects in the set. For this reason, there are a number of possible optimizations that attempt to reduce the number of comparisons made between objects. The main goal is to reject possible pairs of objects as soon as possible, before doing the more expensive computations.

EnJine implements a collision detection system that is based on multi-stage filter architecture. At the top level of this architecture, the game's virtual space is divided into "subspaces" which represent closed volumes in space. The collision detection system first groups the objects by the subspaces, which they intersect, and then builds a list of candidate pairs of colliding objects. As Figure 6 illustrates, a careful partitioning of the space may reduce the number of pairs that need to be tested.

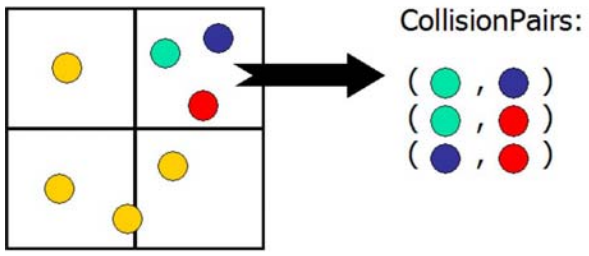


Figure 6 - creating collision pairs based on subspaces

Each collision pair created during this process of subspace grouping is then submitted to a chain of collision filters. Each filter performs a successively more accurate test on the objects of the collision pair. Pairs that are rejected by a filter are removed, so that fewer objects are sent to the more complex and time-consuming filters.

Currently, enJine implements only a bounding volume collision filter. This is the reason why each instance of a Collidable subclass must also be associated to an ObjectBounds instance, which contains a collection of volumes that represent a simplified version of the object. These volumes are used instead of the actual polygonal data of the 3D models when performing approximate collision detection. Figure 7 shows an example of this concept: the red outlined ellipse and boxes are used together as a simplified collision model for the helicopter.

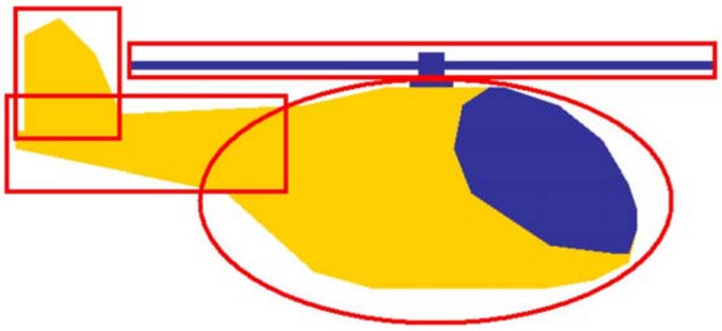


Figure 7 - ObjectBounds allow approximate collision detection

It is possible to extend the enJine collision detection system to implement more accurate filters (such as polygon intersection tests) or even more specialized collision filters for certain types of games.

2.2.6. Input

Input in enJine is handled through the `InputManager` class, which handles one or more `InputDevices`. Each object of an `InputDevice` subclass represents a concrete input device, such as a keyboard or joystick. This mechanism allows the extension of enJine's input model to support new and unconventional input devices. Each `InputDevice` object contains one or more `InputSensors`, which map the possible commands that can be issued by the user through that device. For instance, a keyboard `InputDevice` would have one `InputSensor` object for each key.

However, game entities do not access the input devices directly. Instead, the provide instances of the `InputAction` class which represent the commands they accept from the user. They may be on a higher semantic level than the input sensors. For instance, `InputActions` may represent commands such as "jump" or "pick up object". `InputActions` are bound to `InputSensors` through methods provided by the `InputManager`. This way, the game actions and specific device commands are decoupled and it becomes easier to create configurable input systems. Figure 8 shows a class diagram with the relationship between the different classes of the input subsystem.

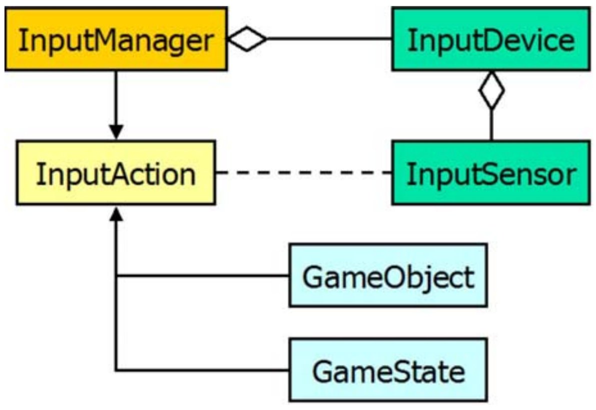


Figure 8 - enJine input system

2.2.7. Sound

Sound in enJine is managed by the `GameAudio` class, which is implemented as a singleton. `GameAudio` contains methods to register one or more `SoundElement` objects. `SoundElement` itself is an abstract class, with subclasses that represent audio clips of different formats, as well as MIDI tracks. Each registered object is associated to a unique identification tag, which may be a numerical value or a text string.

Playback of a sound in enJine is performed by calling methods of the `GameAudio` class, passing back the identification tag of a registered `SoundElement`. `GameAudio` also contains methods to control playback parameters such as volume.

2.2.8. enJine Framework

The enJine Framework consists of the `framework`, `demo` and `loaders` packages. Together, they provide examples and tools to aid learning and using enJine. One of the most interesting classes in the Framework is `SinglePlayerGame`. This class is meant to aid users learning how to develop games with enJine. It can also serve as a template for simple games. As the class diagram in Figure 9 shows, the `SinglePlayerGame` class already includes most enJine subsystems. It configures, initializes and manages all of them, according to parameters given by the user.

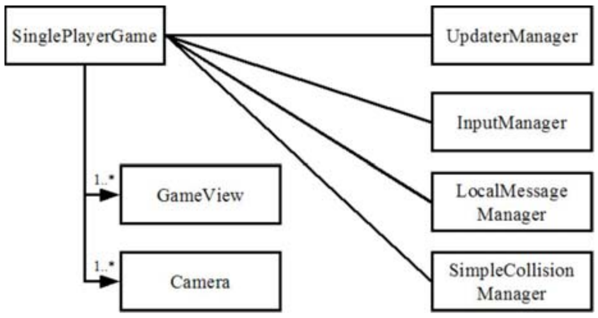


Figure 9 - SinglePlayerGame

3. Teaching with enJine

There are different ways in which enJine may be used as a support tool to teach computer science subjects. Its open and simple architecture allows students to examine its code or even modify or extend it. Besides this, when programming games with enJine, students may have contact with the following topics, among others:

- Object-oriented programming concepts: class design, inheritance, aggregation;
- Software engineering practices and concepts: architectural design, design patterns, modeling languages such as UML;
- Computer graphics concepts: scene graphs, 3D modeling and texturing, real-time rendering, shading and lighting, geometrical transformations;
- Game programming techniques: decoupling of logical and visual update loops, interaction techniques, collision detection.
- As with most 3D computer graphics applications, enJine also makes intensive use of vector and matrix mathematics. Therefore, it may be useful to demonstrate a practical application of those concepts to students.

Most of these concepts will actually be used in the simple game developed in the tutorial “Creating a Game with enJine” presented in the next section.

EnJine can be used to teach many different computer graphics topics. For instance, it can help understanding the usage of scene graphs in the organization of complex graphics applications. It can also serve as a test bed for experimenting with specific topics, such as viewpoint movement, shading models and texture mapping. However, we believe that the most significant use of enJine in this context is to serve as a platform to develop complete computer games under a project-driven teaching approach. For this reason, we will focus on this topic.

In order to discuss the application of enJine in the development of a game project by students, we will show some practical results. They result from the authors' experience in teaching an introductory computer graphics course in both computer engineering and computer science undergraduate programs. Here we will focus on the project development as planned in this course. A more detailed discussion about the whole course, the didactic tools and methodology can be found in more specific papers [1] [2] [3].

In this course the amount of time available to students for development of the project is very short, which should be the case of most of basic courses on computer graphics. The students attending this course have 30 hours at laboratory classes plus approximately 40 extra-class hours. So, in roughly 210 hours of work force (teams are generally formed by 3 students), a group of students has to develop a complete functioning 3D game, including the time devoted to learn the tools (3D modeler, enJine and Java 3D), design the game and do some "artistic" production (textures, sound etc). The focus of the course is on development, so little attention is given to "artisitic" issues (although it is well known the importance of that for the success of a game) and students are allowed to "reuse" third part free materials, being helped by colleagues from other departments, such as Arts and Design departments, or even by professionals.

We had to impose some limitations and requirements to the project in order to keep it feasible and useful for learning objectives without overloading students. Another issue addressed by those requirements is related with the uniformity among the activities developed by different groups of students. Today we use the following set of requirements:

- the game must be developed in Java, using enJine (and therefore Java 3D);
- the game must have a static 3D element, usually the environment, of some complexity and that has actual, not just esthetical, influence in the game;
- at least one element, usually the game's main character, has to be dynamic, meaning it is not a rigid body but has relative movement between its components;
- the game must not have a single fixed camera. The camera either follows the character somehow, or the player can choose between several cameras;
- the game design has to be approved by the teacher before the development starts;
- as much as possible all practical exercises developed in laboratory classes should result in assets to be used in the project;

These requirements do not imply in low variety of possible games. So we can provide some good level of uniformity in students activities and learning, without limiting their creativity and motivation. The following paragraphs, quoted from [1], explains how we organize and follow-up the students activities during the project development.

" The first step in the project is presenting the requirements to the students, letting them choose their groups and demanding a game proposal as soon as possible. While this proposal does not need to be very formal, it must clearly state what the students intend to do and what is the project's scope. In the author's experience, most students do not get project complexity right in this first proposal. Many want to make, in a few months, a game that would take a professional studio a couple years to finish. Some want to do as little as they think is possible. This is the reason why these proposals are delivered quickly. The teacher analyzes each of them with the students and proposes changes when necessary, remaining as faithful to the original idea as possible, so the complexity of the projects is more uniform and they are feasible but still challenging. This analysis is also useful to see how well each proposal conforms to the requirements. Once the proposal has been modified and accepted, students start working on a game specification. While the teacher does not immediately analyze this specification, it is useful for the students. To prepare the specification they sketch the game's architecture and estimate initial values for the games numerical parameters, such as how an action should be scored. These activities usually take up a lot of time if they are left to be done only after implementation starts.

After the proposal is accepted, even before the specification is finished, game implementation is initiated and integrated into the course's practical activities as show in Table 1. Using the modeling software, after completing its tutorial, they start building game environment and other static elements. Then they visually build a scene graph, using a didactic software tool designed for that, to represent the *game's character and other dynamic elements. Finally, using a didactic engine and the results from these previous assignments, they finish building the game.*

Interaction between different groups is encouraged up to a certain point. While discussion of problems and solutions is healthy, when these solutions are simply copied from another project it becomes a problem. This is usually minimized, however, by the variations between proposed games and software architectures. A certain competition mood usually develops in laboratory classes with a few groups of students trying to develop the "coolest" game. This competition, however, has always been healthy, because students know there is little advantage in "winning" aside from personal satisfaction, since all the "contenders" usually end up developing works that are more than good enough to get the maximum grade in the project evaluation.

Students must deliver a project presentation by the end of the course. Figure 2 shows some students term projects screenshots. Project evaluation is based on the resulting software, this presentation and project documentation. This documentation includes the specification elaborated at the beginning of the course and a report detailing project development: how and why it deviated from the original specification, what were the major technical issues in the project, which solutions were chosen and a general discussion of the project. This documentation is not evaluated as rigidly as it would be in a software engineering course, but since most students have already acquired that knowledge, they tend to put it to good use anyway. While originality and good playability in the developed game is rewarded, one thing that is not usually strongly considered in the evaluation is artistic achievement. As said earlier, most students are not of an artistic profile and evaluating them in that aspect usually results in frustration.” [1]

Theory	**Practice**
Introduction to course and CG: origins, classifications, and devices. **Assignment**: game proposal	**Use of a 3D modeler** **Concepts:** Solid modeling, curves and surfaces, polygon mesh, colors, transparency, textures, light sources, camera, shading and ray-tracing **Assignment**: Modeling the game's static components
Optics, human vision, color systems	
Solid and Scene Representation and Modeling	**Introduction to scene graphs** **Concepts:** Scene graphs, geometric transformations, animation **Assignment**: Building the game's dynamic components (usually characters)
Graphics Pipeline	
Local and global illumination; texture, light, bump and normal mapping	
Geometric transformations, projection etc.	**Game programming with an engine** **Concepts:** 2D overlay, camera manipulation, user interaction, polygon data structures **Assignment**: Game completion
Curves and surfaces	
Rasterization, anti-aliasing	
Advanced topics (animation, VR, AR, visualization...)	

Table 1: Course Structure (*extracted from [1]*)

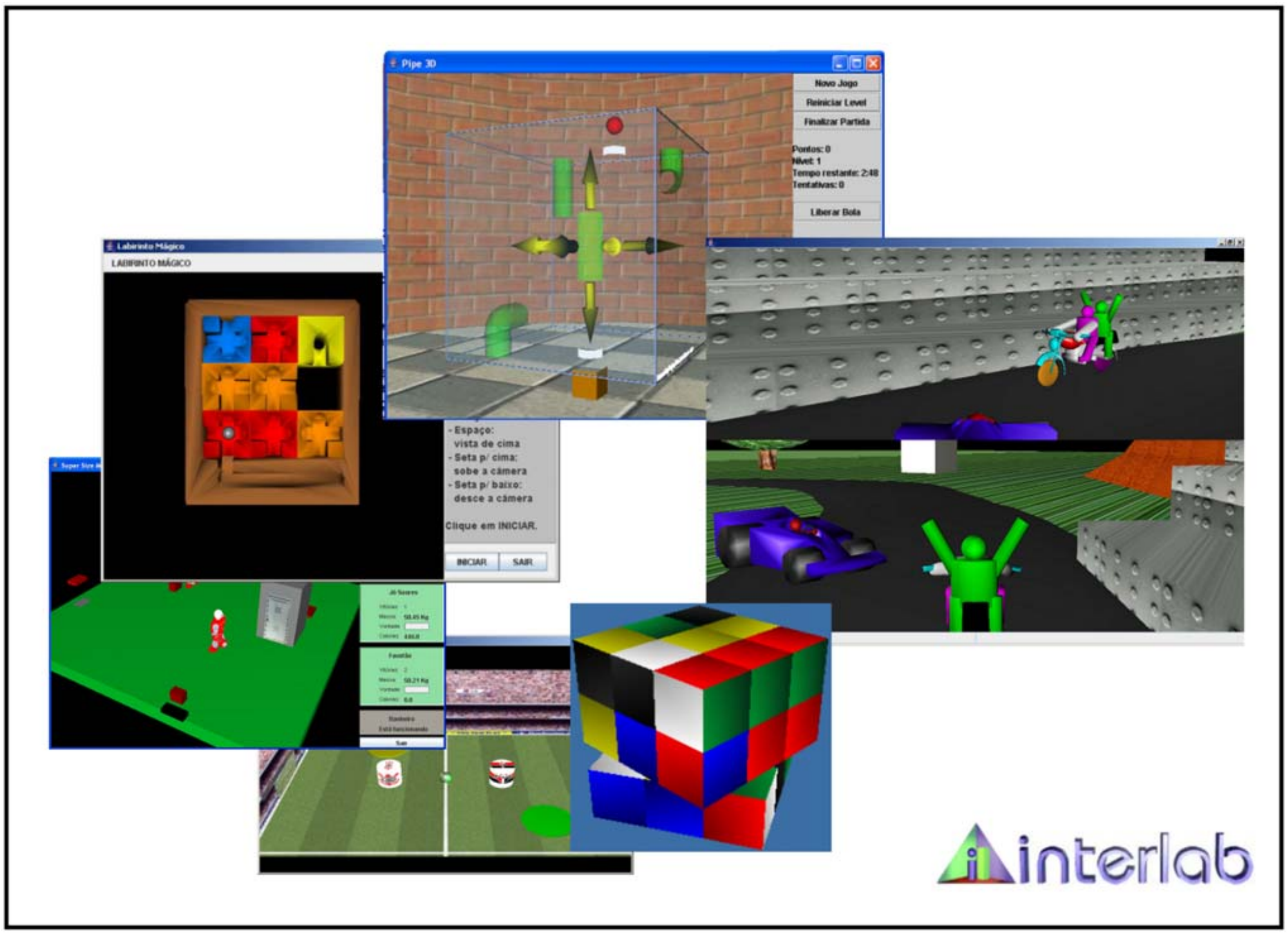


Figure 2 – Screenshots of some students´ projects *(extracted from [1])*

4. Creating a Game with enJine

Now that the architecture of enJine has been explained, it is possible to discuss how to create a game with it. This section will present a walkthrough of the implementation of a simple game called "FootBot Lite" using many of the features found in enJine. A compressed file containing the source code and resources (such as textures) for the project presented in this section should be available on the enJine website at http://enjine.iv.fapesp.br.

FootBot Lite is a simplified version of FootBot, the game that was implemented with the first version of enJine. In this case, two players compete against each other in the same computer. Each player controls a small robot placed on opposing sides of a courtyard, which is divided by a "neutral zone" where they cannot enter. Players can steer and move their robots to hit a ball and send it to the opponent's goal. Every time the ball hits a player's goal, a point is awarded to the other player. The first player to score five points wins the game. Figure 10 shows an image of the implemented game.

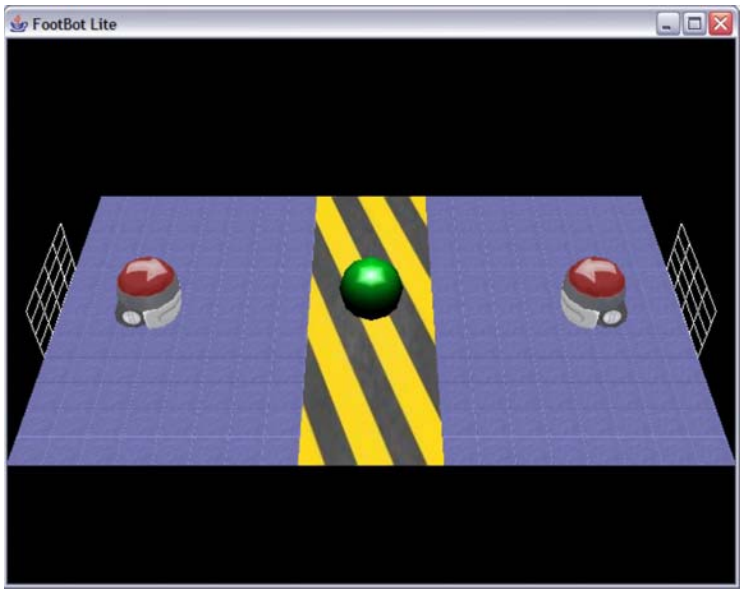


Figure 10 - FootBot Lite

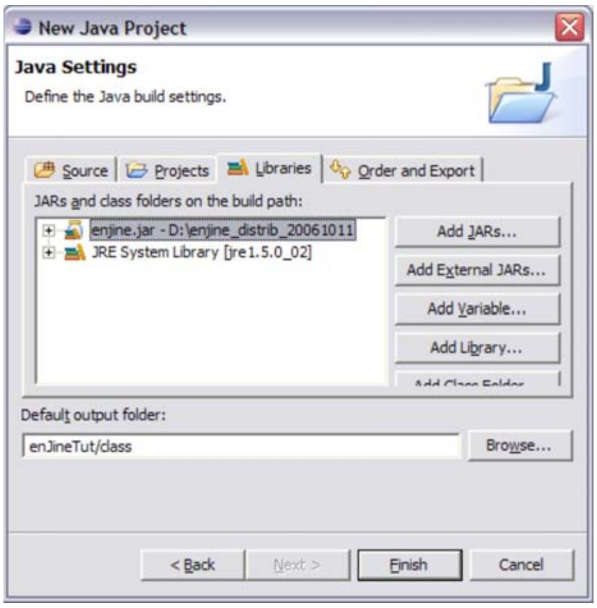
4.1. Configuring a project to work with enJine

To be able to work with enJine, it is necessary to install the Java 2 Standard Edition SDK, version 1.4 or newer, and the Java 3D SDK version 1.3.1 or newer. The Java SDK is needed to compile Java classes and enJine depends on Java 3D to perform graphics rendering. The installation packages for these can be found at http://java.sun.com/ and https://java3d.dev.java.net/, respectively. The use of an Integrated Development Environment (IDE) such as Eclipse [4] or NetBeans [5] is also recommended, although not mandatory.

Once the steps above have been taken care of, it is necessary to download the enJine distribution and set up a project. The enJine distribution can be obtained at http://enjine.iv.fapesp.br, under the "enJine distribution" link at the "downloads" section. The distribution is a ZIP file containing the source code and a JAR file with the pre-compiled classes. It should be uncompressed to a directory of the user's choice, preserving directory structure.

After the distribution package has been uncompressed, it is necessary to configure a project so that it includes the enjine.jar file on the build path. For instance, using the Eclipse IDE [4], the steps are:

1. Create a new project using the menu item "File → New Project… → Java Project"
2. Fill in the necessary information such as project name and location
3. On the third configuration screen (Java Settings), switch to the Libraries tab and click on the "Add External JARs" button. Then navigate to the directory containing the enJine distribution and select the enjine.jar file. See Figure 11.
4. Click on the "Finish" button to end the project configuration.



JARs and class folders on the build path:

enjine.jar - D:\enjine_distrib_20061011
JRE System Library [jre1.5.0_02]

Default output folder:

enJineTut/class

Figure 11 - adding enjine.jar to the build path

4.2. The game structure

Let’s have a look at the game structure before we begin writing the source code. For this project we will use the `SinglePlayerGame` class from the framework package. This will aid in the development process because that class already configures many of the basic components of enJine. Note that the class name refers to a game played in a single computer, as opposed to a networked game. However, FootBot Lite will be actually played by two people sharing a single keyboard.

A game created with enJine is structured around game states, which represent different modes of operation of the game. Each game state is implemented as `GameState` subclass. Figure 12 shows a diagram with the game states for FootBot Lite. The diagram only shows a transition to the terminal state from the Victory state, for clarity. However, it should be possible to quit the game from any state.

In the `TitleScreen` state, the game will only display its title and wait until the “space” key is pressed, to start the game. The `Setup` state will be used to prepare the playing field of the game, and when this is done it will immediately switch to the `GamePlaying` state. The `GamePlaying` state will be responsible for setting up the robots and ball in their initial positions and controlling the game, until a goal is scored. When this happens, the game will switch to the `GoalScored` state. In this state, the game will be momentarily paused while the players are informed of the new score. If neither player has reached five points, the game will move back to the `GamePlaying` state, resuming play. Otherwise, it will switch to the `Victory` state, where the game will display information acknowledging the winner and wait until the “space” key is pressed again, to return to the `TitleScreen` state.

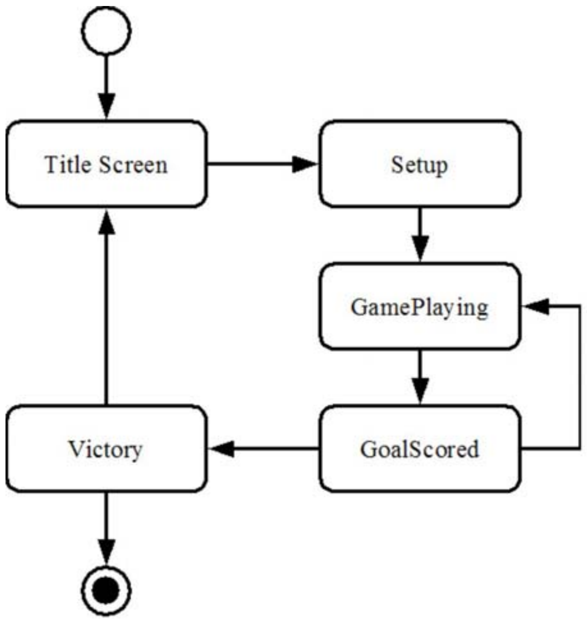


Figure 12 - game states for FootBot Lite

Looking at the game description, we can identify four different types of entities: robots, ball, goals and the courtyard. In enJine, each game entity is represented as an aggregation of an instance of `GameObject` (or subclass) and its service providers – such as subclasses of `Viewable` and `Updater`. The implementation of each of these entities will be discussed in later sections.

4.3. Game setup and states

After this quick overview of the main elements that will make up FootBot Lite, we can proceed to discuss its implementation. We begin with the creation of a subclass of `SinglePlayerGame` called `FootBotGame`. Doing this will allow us to add objects that represent specific game information, such as the players' scores, as well as game states and game entities. Listing 1 shows the `FootBotGame` class at this point. Notice that the player scores have been implemented as private data members, and the access methods prevent illegal operations on them, like setting a negative score.

Listing 1 - the FootBotGame class

```
package fblite;

import interlab.engine.framework.SinglePlayerGame;

public class FootBotGame extends SinglePlayerGame {
    private int player1Score;
    private int player2Score;

    public void initialize() {
        super.initialize();
        resetScores();
    }

    public void resetScores() {
        player1Score = 0;
        player2Score = 0;
    }

    public void increasePlayer1Score() {
        player1Score++;
    }
    public void increasePlayer2Score() {
        player2Score++;
    }

    public int getPlayer1Score() {
        return player1Score;
    }

    public int getPlayer2Score() {
        return player2Score;
    }
}
```

The next step is to define a class that will act as the entry point for the game – that is, a class with a static `main` method. It will also be responsible for the instantiation of the main game components. While it would be possible to do this on the `FootBotGame` class, this is a better design as it separates different responsibilities in different objects. The first version of the `FootBotLite` class is presented in Listing 2. The calls in the `run` method specify that the game must run in a window with an internal size of 640x480 pixels and a title "FootBot Lite".

Listing 2 - the entry point for the game

```
package fblite;

public class FootBotLite {

    private FootBotGame game;

    public FootBotLite() {
    }

    public void run() {
        game = new FootBotGame();
        game.setWindowParameters(640, 480, "FootBot Lite");
        game.initialize();
    }

    public static void main(String[] args) {
        FootBotLite game = new FootBotLite();
        game.run();
    }
}
```

However, at this point executing the program will simply result in it creating and initializing an instance of `FootBotGame` and then exiting. To be able to start the game's main loop, it is necessary to define an initial game state. This should be the `TitleScreen` state, according to Figure 12. However, we will skip the development of the title screen for now and start with the `Setup` and `GamePlaying` states. These are created as concrete subclasses of `GameState` with empty methods. We also add instances of these classes as public attributes of the `FootBotGame` class, as shown in Figure 13. This way, these attributes may be used to request state changes to the game, without the need to create new instances of each state every time.

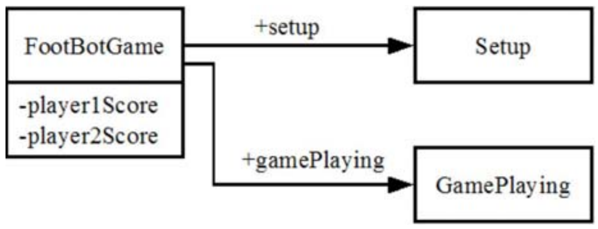


Figure 13 - adding attributes to the FootBotGame class

Listing 3 shows the definition of the `Setup` class, with empty methods omitted. The idea is to have an object of the `Setup` class creating the game entities in its `initialize` method. After that task is completed, it requests a state change to the `GamePlaying` state. At this moment, the `initialize` method is still empty so the game will immediately switch to the next state.

Listing 3 - the Setup class

```
package fblite;

import interlab.engine.core.Game;
import interlab.engine.core.GameState;

public class Setup extends GameState {
    …
    public void postStep(Game game, long delta) {
        // it is safe to downcast here
        FootBotGame g = (FootBotGame)game;
        // request a switch to the GamePlaying state
        g.changeState(g.gamePlaying);
    }
    …
}
```

Now that a game state has been defined, the `run` method of the `FootBotLite` class can be changed to start the game's main loop. The new run method of the `FootBotLite` class is shown in Listing 4. The first parameter to the `mainLoop` method is the desired logic update rate – the number of updates that should happen per second. The second parameter is an instance of a `GameState` subclass, which will be used as the game's initial state.

Listing 4 - the completed run method of FootBotLite

```
    public void run() {
        game = new FootBotGame();
        game.setWindowParameters(640, 480, "FootBot Lite");
        game.initialize();
        // run the game at 25 updates per second
        game.mainLoop(25, game.setup);
    }
```

Right now, if the game is executed, a blank window will be displayed, as there is no behavior defined for the game. To conclude the implementation of the basic structure of the game, we can create the classes representing the other possible game states. At this point, this means creating the `TitleScreen`, `GoalScored` and `Victory` classes as empty subclasses of `GameState` and adding one attribute of each type to the `FootBotGame` class.

4.4. Creating the courtyard

Now that the game structure is implemented, we can create the first game entity: the courtyard, which is the scenario of FootBot Lite. The courtyard is divided into two player fields and a neutral zone that can only be crossed by the ball. Notice that in our implementation, the goals are considered separate entities, not part of the courtyard. Figure 14 shows an object diagram that corresponds to the courtyard game entity. It consists of a standard `GameObject` instance connected to a `CourtyardView` object.

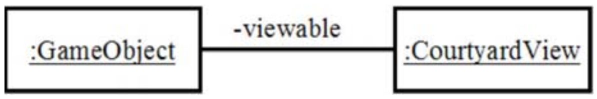


Figure 14 - the courtyard game entity

`CourtyardView` is a `Viewable` subclass that is responsible for building the graphic model of the courtyard. This is accomplished by constructing the scene graph shown in Figure 15. A single textured mesh is used to build both player fields, with a gap in the middle. A second mesh, with a different texture, is used to create the neutral zone.

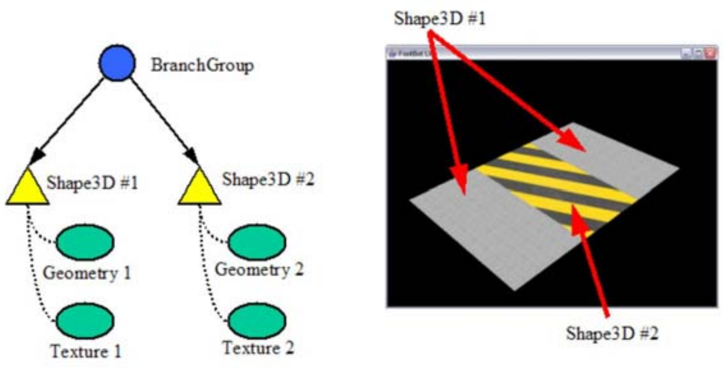


Figure 15 - scene graph for the scenario

Listing 5 shows the details of the `CourtyardView` class. The `buildPlayerFields` and `loadPlayerFieldTex` methods contain code to create simple meshes for the courtyard and load the textures that will be applied to them. The `buildNeutralField` and `loadNeutralFieldTex` methods are similar and for this reason, have been omitted. As the courtyard is completely static, the `updateView` method is empty for this class.

The mesh-constructing methods from `CourtyardView` use numerical constants that have been added to the `FootBotGame` class (see Listing 6) to define the dimensions of the courtyard. It is a good practice to define named constants for the game parameters, as this avoids the occurrence of undocumented “magic numbers” in the code. Large projects may even benefit from the creation of separate classes just to organize groups of related constants.

Listing 5 - the CourtyardView class

```
package fblite;

import javax.media.j3d.*;
import com.sun.j3d.utils.image.*;

import interlab.engine.core.Viewable;

public class CourtyardView extends Viewable {
    private BranchGroup root;
    public void updateView(long delta) {
    }

    public BranchGroup getView() {
        root = new BranchGroup();

        Shape3D playerFields = new Shape3D(buildPlayerFields(),
                loadPlayerFieldTex());
        root.addChild(playerFields);

        Shape3D neutralField = new Shape3D(buildNeutralField(),
                loadNeutralFieldTex());
        root.addChild(neutralField);

        return root;
    }

    private Geometry buildPlayerFields() {
        int vertexCount = 8;
        int[] strips = {4,4};

        TriangleStripArray field = new
TriangleStripArray(vertexCount,
                GeometryArray.COORDINATES | GeometryArray.NORMALS |
                GeometryArray.TEXTURE_COORDINATE_2,
                strips);

        float[] normals = {0, 1, 0, 0, 1, 0,
                0, 1, 0, 0, 1, 0,
                0, 1, 0, 0, 1, 0,
                0, 1, 0, 0, 1, 0};
        field.setNormals(0, normals);
        normals = null;

        float[] texCoords = {
                0, FootBotGame.PLAYER_FIELD_LENGTH, 0, 0,
                FootBotGame.COURTYARD_WIDTH,
FootBotGame.PLAYER_FIELD_LENGTH,
                FootBotGame.COURTYARD_WIDTH, 0,
                0, FootBotGame.PLAYER_FIELD_LENGTH, 0, 0,
                FootBotGame.COURTYARD_WIDTH,
FootBotGame.PLAYER_FIELD_LENGTH,
                FootBotGame.COURTYARD_WIDTH, 0};
        field.setTextureCoordinates(0, 0, texCoords);
        texCoords = null;

        float xmax = FootBotGame.COURTYARD_WIDTH/2.0f;
        float zmax = FootBotGame.PLAYER_FIELD_LENGTH +
            FootBotGame.NEUTRAL_ZONE_LENGTH/2.0f;
        float zmed = FootBotGame.NEUTRAL_ZONE_LENGTH/2.0f;
        float[] geometry = {-xmax, 0, -zmax, -xmax, 0, -zmed,
                xmax, 0, -zmax, xmax, 0, -zmed,
                -xmax, 0, zmed, -xmax, 0, zmax,
                xmax, 0, zmed, xmax, 0, zmax};
        field.setCoordinates(0, geometry);

        return field;
    }

    private Appearance loadPlayerFieldTex() {
        Appearance app = new Appearance();
        TextureLoader loader = new TextureLoader("floor.jpg", null);
        app.setTexture(loader.getTexture());
        return app;
    }
   …
}
```

Listing 6 - defining game constants in FootBotGame

```
public class FootBotGame extends SinglePlayerGame {
    public static final float COURTYARD_WIDTH = 12;
    public static final float PLAYER_FIELD_LENGTH = 8;
    public static final float NEUTRAL_ZONE_LENGTH = 4;
   …
}
```

When working with enJine, it is advised to create one or more factory classes, which can be used to construct the game entities by assembling `GameObject` (or subclasses) instances with the appropriate subclasses of `Viewable`, `Updater` and other service providers. Doing so helps in code organization, avoids the duplication of creation code and makes it easier to identify reusable components.

The `GameObjectFactory` class will be created for this purpose. At first, it contains a single object-creation method, called `createCourtyard`, which returns a game entity that represents the courtyard (see Figure 14). Furthermore, that class uses a simple implementation of the Singleton design pattern, which makes sense, as there is no need for more than one instance of the object factory in this case. Listing 7 shows the entire factory class.

Listing 7 - a very simple object factory

```
package fblite;

import interlab.engine.core.GameObject;

public class GameObjectFactory {
    private static GameObjectFactory theInstance;

    private GameObjectFactory() {
    }

    public static GameObjectFactory getInstance() {
        if (theInstance == null) {
            theInstance = new GameObjectFactory();
        }
        return theInstance;
    }

    public GameObject createCourtyard() {
        GameObject obj = new GameObject();
        CourtyardView view = new CourtyardView();
        obj.setViewable(view);
        return obj;
    }
}
```

After all these steps, it is possible to rewrite the `initialize` method of the `Setup` class so that it creates the game courtyard and adds it to the game, as shown in Listing 8. However, if we run the game, the window will remain blank. The reason for this is that we still have to position the camera in a way that it points to the courtyard.

Listing 8 - initialize method for the Setup class

```
    public void initialize(Game game) {
        GameObject obj =
GameObjectFactory.getInstance().createCourtyard();
        game.addObject(obj);
    }
```

We will implement the camera change in the `initialize` method of the `GamePlaying` class. This way, the camera will only be adjusted after all objects have been created in the previous game state. Listing 9 shows the complete `initialize` method, which will be explained now: The `SinglePlayerGame` class already contains a `Camera` object that controls the in-game camera, and our `FootBotGame` class inherits that feature. We cast the `Game` reference received as a parameter down to this subclass, so that we can access its specialized methods, such as `getCamera`. Once the reference to the camera is available, the `lookAt` method is used to position the camera towards the courtyard. At this point, running the game should result in a window displaying the game courtyard.

Listing 9 - initialize method for the GamePlaying class

```
    public void initialize(Game game) {
        FootBotGame g = (FootBotGame)game;

        Camera camera = g.getCamera(0);
        // look FROM point (20, 20, 0) TO point (0, 0, 0)
        // using the Y axis (0, 1, 0) as the up vector
        camera.lookAt(new Point3d(20, 20, 0),
                      new Point3d(0, 0, 0),
                      new Vector3d(0, 1, 0));
    }
```

4.5. Creating the robots

Now that the courtyard is in place, we will proceed to the creation of the player-controlled robots. This will occur in two steps. First, the robots will be made visible and then, in the next section, their behavior will be added.

The first thing to notice is that the robots must store information about their position, speed and orientation, but the standard `GameObject` class does not contain any such attributes. The solution is to create a `GameObject` subclass named `FootBotObject`, shown in Listing 10, that includes the needed attributes. Although the code is straightforward, it is important to notice that the attributes have been declared with private scope, to preserve encapsulation. The orientation is represented as a single value because in this game, the only possible change in orientation will be a rotation around the vertical axis (the Y axis in the standard Java 3D coordinate system).

Listing 10 - the FootBotObject class

```
package fblite;

import interlab.engine.core.GameObject;
import javax.vecmath.*;

public class FootBotObject extends GameObject {
    private Point3f position;
    private float orientation;
    private Vector3f speed;

    public FootBotObject() {
        super();
        position = new Point3f();
        orientation = 0;
        speed = new Vector3f();
    }

    public void getPosition(Point3f retValue) {
        retValue.set(position);
    }

    public void setPosition(Point3f newValue) {
        position.set(newValue);
    }

    public float getOrientation() { … }

    public void setOrientation(float newValue) { … }

    public void getSpeed(Vector3f retValue) { … }

    public void setSpeed(Vector3f newValue) { … }
}
```

To be able to view the robots on the game, it is necessary to implement a `Viewable` subclass. In this case, it is called `RobotView, and it` includes code to load a 3D model from file and place it in a Java 3D scene graph. The `updateView` method of this class contains code to update the model's position and orientation according to the associated `FootBotObject`. Listing 11 shows the code for the entire class.

Listing 11 - the RobotView class

```
package fblite;

import java.io.FileNotFoundException;

import javax.media.j3d.*;
import javax.vecmath.*;

import com.sun.j3d.loaders.*;
import com.sun.j3d.loaders.objectfile.ObjectFile;

import interlab.engine.core.Viewable;

public class RobotView extends Viewable {
    private FootBotObject parent;

    private BranchGroup root;
    private TransformGroup placement;

    public RobotView(FootBotObject parent) {
        this.parent = parent;
    }

    public void updateView(long delta) {
        // load the current position and convert to a vector
        Point3f pos = new Point3f();
        parent.getPosition(pos);
        Vector3f posVec = new Vector3f(pos);
        // generate a translation matrix corresponding
        // to the position
        Transform3D positionTransf = new Transform3D();
        positionTransf.setTranslation(posVec);
        // generate another rotation matrix corresponding
        // to the robot's orientation around the Y axis
        Transform3D rotationTransf = new Transform3D();
        rotationTransf.rotY(parent.getOrientation());
        // compose the two transforms and apply to the model
        positionTransf.mul(rotationTransf);
        placement.setTransform(positionTransf);
    }

    public BranchGroup getView() {
        root = new BranchGroup();

        placement = new TransformGroup();
        placement.setCapability(
          TransformGroup.ALLOW_TRANSFORM_WRITE);
        root.addChild(placement);

        try {
            ObjectFile loader = new ObjectFile();
            Scene scene = loader.load("robot.obj");
            placement.addChild(scene.getSceneGroup());
        }
        catch(FileNotFoundException ex) {
            ex.printStackTrace();
        }

        return root;
    }
}
```

Now, it is possible to add a new `createRobot` method to the `GameObjectFactory` class, to help with the creation of robot entities in the game, as show in Listing 12. This method can be called from the `initialize` method of the `Setup` class to create the two robots, in a similar way as the courtyard was created before.

Listing 12 - the createRobot method

```
    public GameObject createRobot(float x, float y,
     float z, float angle) {
        Point3f pos = new Point3f(x, y, z);
        FootBotObject obj = new FootBotObject();
        obj.setPosition(pos);
        obj.setOrientation(angle);
        RobotView view = new RobotView(obj);
        obj.setViewable(view);
        return obj;
    }
```

4.6. Adding dynamic behavior to the robots

At this point, the robots are visible, but they still won't respond to player commands. To fix this, it is necessary to set up the input component of enJine. This process begins by registering the appropriate input devices (in this case, the keyboard) with the `InputManager` that already exists in the `SinglePlayerGame` class. Listing 13 shows the `run` method of the `FootBotLite` class with the added method calls.

Listing 13 - registering the keyboard with the InputManager

```
    public void run() {
        game = new FootBotGame();
        game.setWindowParameters(640, 480, "FootBot Lite");
        game.initialize();
        InputManager m = game.getInput();
        m.registerDevice(Keyboard.getInstance());
        game.mainLoop(25, game.setup);
    }
```

The next step consists in creating an `Updater` subclass, which contains `InputActions` corresponding to the acceptable user commands, as well as the code to treat those commands. In this case, this is the `RobotUpdater` class shown in Listing 14. To illustrate different implementation techniques, this class uses simple aggregation of the `InputAction` objects, exposing them as public members of the class. It is important to notice that those members are declared as `final`, thus protecting them from being overwritten by mistake when accessed by other objects.

The `update` method of the `RobotUpdater` class deserves a more detailed explanation, as it contains the code for the movement of the robots. It can be divided into four sections: at the beginning of the method, there are a few conditional statements that verify if the user has issued any commands to the robot. Afterwards, the current state of the associated `FootBotObject` is retrieved. What follows then is the simulation code, which recalculates the robot's orientation, speed and position, and clamps these values as needed. The code also prevents the robots from leaving their assigned area of the courtyard. The last lines of the update method write the new state values back on the `FootBotObject`. This way, the next iteration of the rendering loop may use them to update the graphical view of the robot.

Listing 14 - the RobotUpdater class

```java
package fblite;

import interlab.engine.core.Game;
import interlab.engine.core.Updater;
import interlab.engine.io.input.InputAction;

import javax.media.j3d.Transform3D;
import javax.vecmath.*;

public class RobotUpdater extends Updater {
    // many constants to constrain robot movement
    // omitted here for brevity
    public static final float ROBOT_RADIUS = 1;
      …
    private FootBotObject parent;

    public final InputAction turnLeft;
    public final InputAction turnRight;
    public final InputAction moveForward;

    public RobotUpdater(FootBotObject parent) {
        super();
        this.parent = parent;

        turnLeft = new InputAction(1, 0, "left");
        turnRight = new InputAction(2, 0, "right");
        moveForward = new InputAction(3, 0, "move");
    }

    public void update(Game g, float delta) {
        float turnSpeed = 0;
        float moveSpeed = 0;

        // 1. verify commands
        if (turnLeft.getIntensity() > 0) {
            turnSpeed = 1.57f;
        }
        if (turnRight.getIntensity() > 0) {
            turnSpeed = -1.57f;
        }
        if (moveForward.getIntensity() > 0) {
            // acceleration
            moveSpeed = 2.0f * delta;
        }
        // 2. get the current state
        float orientation = parent.getOrientation();
        Point3f position = new Point3f();
        parent.getPosition(position);
        Vector3f speed = new Vector3f();
        parent.getSpeed(speed);

        // 3. simulation
        orientation += turnSpeed * delta;
        // add speed caused by the acceleration
        speed.x += (float)(moveSpeed * Math.cos(orientation));
        speed.z += (float)(-moveSpeed * Math.sin(orientation));
        float currentSpd = speed.length();
        if (currentSpd > 0) {
            // limit maximum speed
            if (currentSpd > 5) currentSpd = 5;
            // reduce speed by friction
            float newSpd = currentSpd - 0.6f * delta;
            // avoid friction from moving the robot backwards!
            if (newSpd < 0) newSpd = 0;
            speed.scale(newSpd/currentSpd);
            position.x += speed.x * delta;
            position.z += speed.z * delta;
        }
        // keep the robot within the playing field
        if (position.x < MIN_X) {
            position.x = MIN_X; speed.x = 0;
        }
        else if (position.x > MAX_X) {
            position.x = MAX_X; speed.x = 0;
        }

        if (position.z > 0) {
            if (position.z < MIN_Z_P1) {
                position.z = MIN_Z_P1; speed.z = 0;
            }
            else if (position.z > MAX_Z_P1) {
                position.z = MAX_Z_P1; speed.z = 0;
            }
        }
        else {
            if (position.z < MIN_Z_P2) {
                position.z = MIN_Z_P2; speed.z = 0;
            }
            else if (position.z > MAX_Z_P2) {
                position.z = MAX_Z_P2; speed.z = 0;
            }
        }

        // 4. update the state
        parent.setOrientation(orientation);
        parent.setPosition(position);
        parent.setSpeed(speed);
    }
}
```

Once the `RobotUpdater` class is completed, it is necessary to modify the `createRobot` method so that an instance of that class is created and attached to the `FootBotObject` that represents a robot in the game. Otherwise, the robots will remain without any behavior. The object diagram in Figure 16 shows the objects that make up a robot entity that is constructed by `createRobot`.

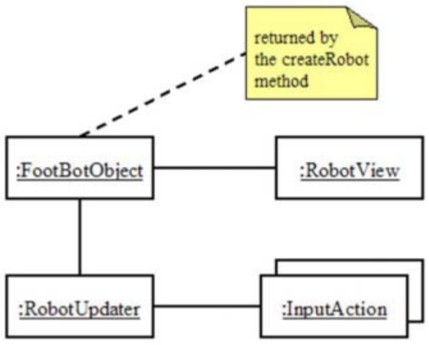


Figure 16 - object diagram of an instantiated player robot

One last step must be taken to complete the implementation of player control over the robots: the `InputActions` declared on the `RobotUpdater` class must be bound to `InputSensors` of an input device (in this case, keys of the keyboard). This is done through the `bind` method of the `InputManager` class. Listing 15 shows the relevant part of the updated `initialize` method of the `Setup` class, which now creates the two player robots and connects their actions to specific keyboard keys.

Listing 15 - binding actions to keys in the initialize method

```
    public void initialize(Game game) {
        FootBotGame g = (FootBotGame)game;
        …
        obj = GameObjectFactory.getInstance().createRobot(
                FootBotGame.ROBOT1_POS.x,
                FootBotGame.ROBOT1_POS.y,
                FootBotGame.ROBOT1_POS.z,
                FootBotGame.ROBOT1_ORI);
        RobotUpdater updater = (RobotUpdater)obj.getUpdater();
        InputManager input = g.getInput();
        Keyboard key = Keyboard.getInstance();
        input.bind(key.getSensor(KeyEvent.VK_W),updater.moveForward);
        input.bind(key.getSensor(KeyEvent.VK_A),updater.turnLeft);
        input.bind(key.getSensor(KeyEvent.VK_D),updater.turnRight);
        g.addObject(obj);
        …
    }
```

4.7. Adding the ball and the goals

The implementation of the ball and goal entities is very similar to what has been done with the robots. For simplicity, we will adopt the `FootBotObject` class as the base for these entities, although the goals do not require a speed attribute. Figure 17 shows the object diagrams for the ball and goal.

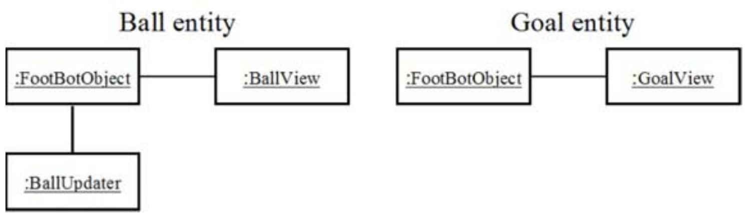


Figure 17 - the ball and goal entities

We will start by creating the ball. Like the robots, it requires both a `Viewable` and an `Updater` subclass, which will be named, obviously enough, `BallView` and `BallUpdater`. To make the example more interesting, the ball model uses the `Sphere` utility class from Java 3D, with associated shading and lighting attributes. This means that the ball will require a light source to be visible – otherwise it will be rendered as a flat, black circle. For this reason, the `BallView` actually instantiates a light source that will illuminate the ball as it moves around the courtyard. However, the scope mechanism of Java 3D is used so that the light only affects the ball – this way, the processing cost of the dynamic lighting is limited. Listing 16 shows the details of this technique, and Figure 18 shows the scene graph that is constructed on that method.

Listing 16 - getView method of the BallView class

```
    public BranchGroup getView() {
        root = new BranchGroup();

        placement = new TransformGroup();
        placement.setCapability(
          TransformGroup.ALLOW_TRANSFORM_WRITE);
        root.addChild(placement);

        // create a material that depends on shading and lighting
        Appearance ap = new Appearance();
        Material mat = new Material();
        mat.setAmbientColor(0.2f, 0.2f, 0.2f);
        mat.setDiffuseColor(0.0f, 0.8f, 0.1f);
        mat.setSpecularColor(1.0f, 1.0f, 1.0f);
        mat.setShininess(48.0f);
        ap.setMaterial(mat);

        Sphere ball = new Sphere(1.0f, ap);
        placement.addChild(ball);

        // attach a point light that will only affect the ball
        PointLight light = new PointLight();
        light.setPosition(0, 8, 0);
        BoundingSphere bs = new BoundingSphere();
        bs.setRadius(50);
        light.setInfluencingBounds(bs);
        light.addScope(placement);
        light.setEnable(true);
        root.addChild(light);

        return root;
    }
```

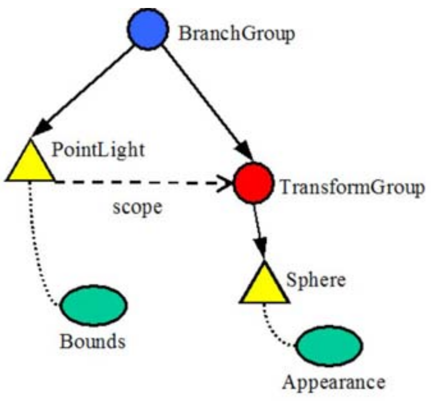


Figure 18 - the ball scene graph with a scoped light

The `BallUpdater` class must implement the behavior of the ball when it is moving, after a collision against a robot. It must also include code to bounce off the limits of the courtyard, so that it remains inside the playable area. Listing 17 shows the `update` method of that class. The simulation code also includes a special condition, which avoids slowing down the ball (due to friction) while it is in the neutral zone. Without it, the ball might stop in a place where neither robot can push it again.

Listing 17 - the update method of BallUpdater

```
    public void update(Game g, float delta) {
        // fetch the current state
        Point3f position = new Point3f();
        parent.getPosition(position);
        Vector3f speed = new Vector3f();
        parent.getSpeed(speed);

        float oldSpeed = speed.length();
        if (oldSpeed > 0) {
            float newSpeed = oldSpeed;
            // only apply friction if outside the neutral zone
            if (position.z > NEUTRAL_MAX_Z ||
                position.z < NEUTRAL_MIN_Z)
                newSpeed -= 0.1f * delta;
            // avoid friction from moving the ball backwards!
            if (newSpeed < 0)
                newSpeed = 0;
            speed.scale(newSpeed/oldSpeed);
            position.x += speed.x * delta;
            position.z += speed.z * delta;
        }
        // make the ball bounce off the courtyard limits
        if (position.x < MIN_X) {
            position.x = MIN_X; speed.x = -speed.x;
        }
        else if (position.x > MAX_X) {
            position.x = MAX_X; speed.x = -speed.x;
        }

        if (position.z < MIN_Z) {
            position.z = MIN_Z; speed.z = -speed.z;
        }
        else if (position.z > MAX_Z) {
            position.z = MAX_Z; speed.z = -speed.z;
        }
        // save the updated state
        parent.setPosition(position);
        parent.setSpeed(speed);
    }
```

Once the components of the ball are implemented, we can create the `GoalView` class. In order to present another option available for the 3D models in the game, it implements a 3D model represented by a mesh of lines, created using a Java 3D `LineArray` object. In this case, the lines are drawn with default attributes and are not affected by lighting, although it would be possible to set rendering attributes to them.

After the `BallView`, `BallUpdater` and `GoalView` classes have been defined, new `createBall` and `createGoal` methods are added to the `GameObjectFactory` class so that they can be used to create the new entities. At this point, the game is visually complete, but the dynamic behavior of the objects is still lacking. In particular, although the ball contains simulation code, it will not move – for that to happen, the game must detect collisions between objects.

4.8. Collision detection

Now that the game elements, such as the courtyard, robots, ball and goals have been created, it is necessary to set up the collision detection system of enJine so that the entities can interact with each other. The `SinglePlayerGame` class already instantiates a `SimpleCollisionManager` object, which will be used to detect collisions between the objects.

The first step to activate the collision system is to call the `enableCollisions` method of the `SimpleCollisionManager`, with a value of `true` passed as parameter. We will do this on the `initialize` method of the `GamePlaying` class.

The next step to use `SimpleCollisionManager` is to define the subspaces that will partition the game universe. The collision detection system in enJine only tests collisions between objects that lie in the same subspace, in order to reduce the number of tests that must be performed. If there are no subspaces defined, then no collision detection happens at all. A subspace is represented by an object of the `Subspace` class, which in turn makes use of the `Bounds` objects defined in Java 3D to represent volumes in space.

Two subspaces will be defined for this game. Each of them is box-shaped and encompasses one of the player fields, as illustrated in Figure 19. There is no need to create a subspace for the central area of the courtyard, as only the ball may cross that region and there are no objects to collide with it there. For simple games such as this, a single subspace involving the whole courtyard might have been enough. However, the choice of two subspaces was made to better explain this feature of the collision detection system.

The creation of the subspaces is also performed at the `initialize` method of the `Setup` class. Listing 18 shows the code to configure and add the subspaces. The values used to define the bounding boxes are based on the size of the courtyard defined for this game. One important point to observe is that the subspaces should be defined **before** any objects are added through the `addObject` method. This way, each object is already inserted in the appropriate subspaces.

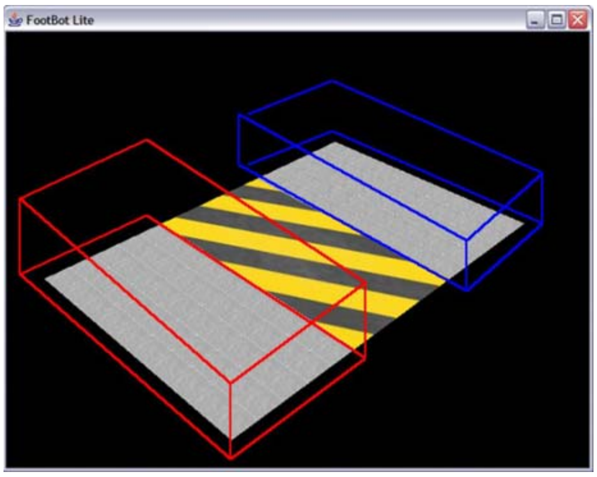


Figure 19 - collision subspaces

Listing 18 - creating the collision subspaces

```
FootBotGame g = (FootBotGame)game;

SimpleCollisionManager scm = g.getCollider();
BoundingBox sub1 = new BoundingBox(
        new Point3d(-FootBotGame.COURTYARD_WIDTH/2 - SUBSPACE_BORDER,
                -SUBSPACE_BORDER,
                FootBotGame.NEUTRAL_ZONE_LENGTH/2 - SUBSPACE_BORDER),
        new Point3d(FootBotGame.COURTYARD_WIDTH/2 + SUBSPACE_BORDER,
                FootBotGame.COURTYARD_HEIGHT + SUBSPACE_BORDER,
                FootBotGame.NEUTRAL_ZONE_LENGTH/2 +
                FootBotGame.PLAYER_FIELD_LENGTH + SUBSPACE_BORDER));
Subspace sp1 = new Subspace(sub1);
scm.addSubspace(sp1);
BoundingBox sub2 = new BoundingBox(
        new Point3d(-FootBotGame.COURTYARD_WIDTH/2 - SUBSPACE_BORDER,
                -SUBSPACE_BORDER,
                -FootBotGame.NEUTRAL_ZONE_LENGTH/2 -
                FootBotGame.PLAYER_FIELD_LENGTH - SUBSPACE_BORDER),
        new Point3d(FootBotGame.COURTYARD_WIDTH/2 + SUBSPACE_BORDER,
                FootBotGame.COURTYARD_HEIGHT + SUBSPACE_BORDER,
                -FootBotGame.NEUTRAL_ZONE_LENGTH/2 +
SUBSPACE_BORDER));
Subspace sp2 = new Subspace(sub2);
scm.addSubspace(sp2);
```

The other necessary step to enable the collision detection system is to attach a `Collidable` subclass to each object that can collide with others. The classes `RobotCollidable`, `BallCollidable` and `GoalCollidable` will be created for this purpose. In the next section, they will be used to implement collision response behavior for the objects, but for now they are just concrete subclasses of `Collidable` with empty methods.

As discussed in the previous chapter, each instance of a `Collidable` subclass must be associated to an `ObjectBounds` that represents the simplified collision model of the game entity. In this sample game, a single sphere will be used as the collision model for each robot and the ball, while a box will represent each goal.

All of the mentioned elements come together when creating the game entities. To illustrate the whole process, Listing 19 presents the updated version of the `createRobot` method, from the

GameObjectFactory class. Now this method also instantiates a RobotCollidable, sets its ObjectBounds and associates it with the newly created FootBotObject.

Listing 19 - adding a RobotCollidable to the robots

```
    public GameObject createRobot(float x, float y, float z, float
angle) {
        Point3f pos = new Point3f(x, y, z);
        FootBotObject obj = new FootBotObject();
        obj.setPosition(pos);
        obj.setOrientation(angle);
        RobotView view = new RobotView(obj);
        obj.setViewable(view);
        RobotUpdater updater = new RobotUpdater(obj);
        obj.setUpdater(updater);
        // create the new RobotCollidable
        RobotCollidable collidable = new RobotCollidable();
        // create an ObjectBounds object corresponding
        // to a single sphere, and attach it to the
        // collidable object
        BoundingSphere bs = new BoundingSphere();
        bs.setRadius(1.0);
        ObjectBounds bounds = new ObjectBounds(bs);
        collidable.setBounds(bounds);
        // lastly, attach the collidable to the robot
        obj.setCollidable(collidable);
        return obj;
    }
```

One last, but vital step must be taken to complete the collision detection setup. By default, the ObjectBounds assigned to the GameObjects are located on the origin of the coordinate system. That is, they are placed at coordinates (0, 0, 0). If the objects are actually located somewhere else, code must be added to keep the position and orientation of the collision models updated. Since the robots and ball will move with time, this code must be added to the update method of their Updater objects.

Listing 20 shows, in bold text, the code added to the update method to synchronize the collision model with the logical attributes of the entity. It is interesting to note that, as the collision models for the robots and the ball are all simple spheres, the code is the same for both of them. Also, only the position of the collision model must be updated, as it is completely symmetrical. Otherwise, additional operations would be necessary to update its orientation as well.

Listing 20 – synchronization of the collision model in the update method

```
    public void update(Game g, float delta) {
        // fetch the current state
        float orientation = parent.getOrientation();
        Point3f position = new Point3f();
        parent.getPosition(position);
        Vector3f speed = new Vector3f();
        parent.getSpeed(speed);

        // perform all calculations and update the state
        …

        // update the collidable object
        Vector3f vec = new Vector3f(position);
        Transform3D t3d1 = new Transform3D();
        t3d1.setTranslation(vec);
        parent.getCollidable().getBounds().setTransform(t3d1);
    }
```

For the goals, a simpler solution can be achieved, by positioning the `ObjectBounds` in the correct location when they are created, as shown in Listing 21. Since the goals do not change position during the game, there is no need to write update code for them.

Listing 21 - positioning the ObjectBounds in the createGoal method

```
public GameObject createGoal(float x, float y, float z) {
    Point3f pos = new Point3f(x, y, z);
    FootBotObject obj = new FootBotObject();
    obj.setPosition(pos);
    GoalView view = new GoalView(obj);
    obj.setViewable(view);
    GoalCollidable collidable = new GoalCollidable();
    // create a bounding box with the offset passed to
    // this method. This way, the goal collision bounds
    // are already in the right position.
    BoundingBox bb = new BoundingBox(
            new Point3d(
                  x - FootBotGame.GOAL_WIDTH/2.f, y, z - 0.1),
            new Point3d(x + FootBotGame.GOAL_WIDTH/2.f,
                    y + FootBotGame.GOAL_HEIGHT, z + 0.1));
    ObjectBounds bounds = new ObjectBounds(bb);
    collidable.setBounds(bounds);
    obj.setCollidable(collidable);
    return obj;
}
```

4.9. Collision response

Now that the collision detection system has been prepared, it is possible to write code to treat the collision events. When the `SimpleCollisionManager` detects a collision between two objects, it calls the `getCollisionHandlingPriority` method of both objects. The object that returns the highest value (thus having higher priority) gets its `handleCollision` method called.

In the case of FootBot Lite, the following collision response scheme will be adopted: the `BallCollidable` class will have highest collision priority, and will be responsible for handling collisions with both robots and goals. To set up this scheme, the `getCollisionHandlingPriority` of the `BallCollidable` class is changed to return a value of 1, while the corresponding method of `RobotCollidable` and `GoalCollidable` will still return a priority of zero.

Listing 22 shows the collision handling method of the `BallCollidable` class. It detects what kind of object has collided with the ball by accessing its run-time type information. Then, it calls an appropriate method to handle each type of collision.

Listing 22 - collision handling for ball/robot collisions

```
public void handleCollision(Collidable cl) {
    // handle collision against a robot
    if (cl instanceof RobotCollidable) {
        handleRobotCollision((RobotCollidable)cl);
    }
    // handle collision against a goal
    else if (cl instanceof GoalCollidable) {
        handleGoalCollision((GoalCollidable)cl);
    }
}
```

When a collision between the ball and a robot happens, the `handleRobotCollision` method calculates the new speed vectors for both objects. In FootBot Lite, we assume an elastic collision between objects of same mass – although it may not be very realistic, it causes interesting effects such as the ability of the ball to push the robots back. Both the current speed and position of both colliding objects are necessary to perform these calculations. For this reason, we add a `parent` attribute to the `RobotCollidable` and `BallCollidable` classes, along with a `getParent` access method. The `parent` attribute is set to the `FootBotObject` that contains the `Collidable` subclass, when the objects are constructed in the `GameObjectFactory` methods. After this change, it is possible to write the collision response code in Listing 23.

Listing 23 - ball / robot collision response

```
    private void handleRobotCollision(RobotCollidable rc) {
        FootBotObject rcParent = rc.getParent();
        // calculate the vector BR between the centers
        // of the ball and the robot
        Point3f robotPos = new Point3f();
        rcParent.getPosition(robotPos);
        Point3f ballPos = new Point3f();
        parent.getPosition(ballPos);
        Vector3f BR = new Vector3f(
                ballPos.x - robotPos.x,
                ballPos.y - robotPos.y,
                ballPos.z - robotPos.z);
        BR.normalize();
        // create a perpendicular vector to BR on the plane XZ
        Vector3f BRP = new Vector3f(BR.z, 0, -BR.x);
        // retrieve the current robot and ball speeds
        Vector3f robotSpd = new Vector3f();
        rcParent.getSpeed(robotSpd);
        Vector3f ballSpd = new Vector3f();
        parent.getSpeed(ballSpd);
        // with objects of same mass, the objects will
        // swap the component of speed in the direction of
        // vector BR and maintain the component in the
        // direction of BRP
        Vector3f ballSpdBR = new Vector3f(BR);
        ballSpdBR.scale(ballSpd.dot(BR));
        Vector3f ballSpdBRP = new Vector3f(BRP);
        ballSpdBRP.scale(ballSpd.dot(BRP));

        Vector3f robotSpdBR = new Vector3f(BR);
        robotSpdBR.scale(robotSpd.dot(BR));
        Vector3f robotSpdBRP = new Vector3f(BRP);
        robotSpdBRP.scale(robotSpd.dot(BRP));

        ballSpd.add(ballSpdBRP, robotSpdBR);
        robotSpd.add(robotSpdBRP, ballSpdBR);

        setNewSpeed(ballSpd);
        setCollisionFlag(true);
        rc.setNewSpeed(robotSpd);
        rc.setCollisionFlag(true);
    }
```

Notice that instead of immediately updating the speed values for the objects, they are stored in the `Collidable` objects along with a flag that indicates that a collision has happened (through the `setNewSpeed` and `setCollisionFlag` methods). As may be apparent from Listing 23, the `RobotCollidable` and `BallCollidable` classes must to be extended to add these attributes and the respective access methods.

Afterwards, the `BallUpdater` and `RobotUpdater` classes must be modified to check on these attributes and make the necessary changes to the objects' speeds. This approach ensures that all actual state changes are performed from the `update` method of the `Updater` subclasses. Although not mandatory for simple games such as this, implementing this behavior adds predictability and stability to the game.

Listing 24 shows the changes in the `update` method of the `BallUpdater` class, to take into account the collisions between ball and robots. It includes the creation of a new attribute called `lastPosition`, which simply stores the last position calculated for the ball. When a collision occurs, the ball is put back to that position (where presumably it is collision-free) and its speed is set to what was calculated in the collision handling method of the `BallCollidable` class. The same modifications are also applied to the `RobotUpdater` class.

Listing 24 - the update method of BallUpdater with collision response

```
    public void update(Game g, float delta) {
        //retrieve the current position, speed etc.
        …

        BallCollidable c = (BallCollidable)parent.getCollidable();
        if (c.getCollisionFlag()) {
            // return to the last position
            position.set(lastPosition);
            //override speed after collision
            c.getNewSpeed(speed);
           // reset the collision flag
            c.setCollisionFlag(false);
        }
        // save the last valid position
        lastPosition.set(position);
        …
    }
```

Collisions between ball and goal are handled by the `handleGoalCollision` method of the `BallCollider` class. We want to award a point to the scoring robot and switch to `GoalScored` state. To do this, we first add the code presented in Listing 25 to the `BallUpdater` class. This code creates a `goalScored` attribute that can receive three values: 0, 1, and 2. A value of 0 means nothing happened, while values of 1 and 2 indicate that one of the players has scored a point.

Listing 25 - modifying the BallUpdater to handle a goal scored

```
…
    private int goalScored;
…
    public void setGoalScored(int who) {
        goalScored = who;
    }

    public void update(Game g, float delta) {
        if (checkGoalScored((FootBotGame)g)) return;
        …
    }

    private boolean checkGoalScored(FootBotGame g) {
        if (goalScored != 0) {
            FootBotGame fbg = (FootBotGame)g;
            if (goalScored == 1) {
                fbg.increasePlayer1Score();
            }
            else {
                fbg.increasePlayer2Score();
            }
            goalScored = 0;
            fbg.changeState(fbg.goalScored);
            return true;
        }
        return false;
    }
```

After that code is in place, the handling of collisions between ball and goal becomes a simple matter. To determine which goal was hit, we can simply verify the position of the ball in the courtyard. Listing 26 shows the completed handleGoalCollision method.

Listing 26 - the complete handleGoalCollision method

```
    private void handleGoalCollision(GoalCollidable gc) {
        Point3f pos = new Point3f();
        parent.getPosition(pos);
        BallUpdater u = (BallUpdater)parent.getUpdater();
        if (pos.z < 0)
            u.setGoalScored(1);
        else
            u.setGoalScored(2);
    }
```

4.10. The GoalScored state

In the last section, we implemented code that would increase a player's score and send the game into the `GoalScored` state if the ball hit a goal. So far, that game state is completely empty. In this section we will implement the following game rules on it:

1. First we will disable collision checking, to avoid having the ball / goal collision be counted more than once. Remember that collision checking is enabled in the `initialize` method of the `GamePlaying` state, so it will be reactivated when the game resumes.
2. Then we will move the robots and ball to their starting positions. The ball will be left in the field of the player who suffered the goal, so that he can start playing now.
3. Lastly, we will check if a player has reached a score of five points. If so, the game will switch to the `Victory` state. Otherwise, it will switch back to the `GamePlaying` state.

To be able to implement rule #2, it is necessary to have direct access to the ball and robots. For this reason, we will add the methods shown in Listing 27 to the `FootBotGame` class. The `setRobot1`, `setRobot2` and `setBall` methods will be called from the `initialize` method of the `Setup` class, when the respective game entities are created.

Listing 27 - modifications to FootBotGame

```
public class FootBotGame extends SinglePlayerGame {
   …
    private FootBotObject robot1, robot2, ball;
    public void setRobot1(FootBotObject obj) {
        robot1 = obj;
    }

    public void setRobot2(FootBotObject obj) {
        robot2 = obj;
    }

    public void setBall(FootBotObject obj) {
        ball = obj;
    }

    public void resetPositions() {
        Vector3f zeroSpeed = new Vector3f(0, 0, 0);
        robot1.setPosition(ROBOT1_POS);
        robot1.setOrientation(ROBOT1_ORI);
        robot1.setSpeed(zeroSpeed);
        robot2.setPosition(ROBOT2_POS);
        robot2.setOrientation(ROBOT2_ORI);
        robot2.setSpeed(zeroSpeed);

        Point3f pos = new Point3f();
        ball.getPosition(pos);
        if (pos.z > 0)
            ball.setPosition(BALL_POS1);
        else
            ball.setPosition(BALL_POS2);
        ball.setSpeed(zeroSpeed);
    }
```

Listing 28 shows the relevant sections of the `GoalScored` class. Essentially, rule #1 is implemented in the `initialize` method, rule #2 in the `preStep` method and rule #3 goes in the `postStep` method.

Listing 28 - the GoalScored class

```
package fblite;

import interlab.engine.core.Game;
import interlab.engine.core.GameState;

public class GoalScored extends GameState {

    public void initialize(Game game) {
        // disable collisions
        FootBotGame g = (FootBotGame)game;
        g.enableCollisions(false);
    }
…
    public void preStep(Game game, long delta) {
        FootBotGame g = (FootBotGame)game;
        g.resetPositions();
    }

    public void postStep(Game game, long delta) {
        // switch to either Victory or GamePlaying state
        FootBotGame g = (FootBotGame)game;
        int s1 = g.getPlayer1Score();
        int s2 = g.getPlayer2Score();
        if (s1 >= 5 || s2 >= 5)
            g.changeState(g.victory);
        else
            g.changeState(g.gamePlaying);
    }
…
}
```

4.11. The Victory state

For completeness, Listing 29 shows the code for the Victory state. Essentially, it displays a message box informing which player won the match, and after that the scores are reset and a new game begins.

Listing 29 - the Victory class

```
package fblite;
```

```
import javax.swing.JOptionPane;

import interlab.engine.core.Game;
import interlab.engine.core.GameState;

public class Victory extends GameState {

    public void initialize(Game game) {
        FootBotGame g = (FootBotGame)game;
        if (g.getPlayer1Score() >= 5)
            JOptionPane.showMessageDialog(null, "Player 1 Wins!");
        else
            JOptionPane.showMessageDialog(null, "Player 2 Wins!");
    }
   …
    public void preStep(Game game, long delta) {
    }

    public void postStep(Game game, long delta) {
        // it is safe to downcast here
        FootBotGame g = (FootBotGame)game;
        g.resetScores();
        // request a switch to the GamePlaying state
        g.changeState(g.gamePlaying);
    }
   …
}
```

4.12. Creating an overlay

Now that the game is mostly complete, we can add an overlay to display the scores for the players. To do this, we create a `ScoreOverlay` class, which implements the `Overlay` interface. A reference to the `FootBotGame` instance is passed to it when it is created, so that it can use the score access methods. Listing 30 shows the entire class. To enable the overlay, the call to `getView(0).setOverlay(new ScoreOverlay(g))` is added to the end of the initialize method on the Setup class.

Listing 30 - a very simple overlay

```
package fblite;

import java.awt.Color;
import java.awt.Graphics2D;

import interlab.engine.io.graphics.Overlay;

public class ScoreOverlay extends Object implements Overlay {

    private FootBotGame game;

    public ScoreOverlay(FootBotGame game) {
        this.game = game;
    }

    public void draw(Graphics2D g2d) {
        g2d.setColor(Color.RED);
        StringBuffer s = new StringBuffer("Player 1: ");
        s.append(Integer.toString(game.getPlayer1Score()));
        s.append(" / Player 2: ");
        s.append(Integer.toString(game.getPlayer2Score()));
        g2d.drawString(s.toString(), 10, 20);
    }
}
```

4.13. Adding sound

As a last step of this tutorial, we will add a sound effect to the ball collisions against the robots. To do so, we must first register the sound effect with the audio manager class, called `GameAudio`. When a sound is registered, a text string is associated to it. This string can be used to reference the sound on other methods of `GameAudio`. Listing 31 shows the code added to the initialize method of the Setup class for this purpose.

Listing 31 - registering a sound effect

```
        GameAudio gameAudio = GameAudio.getInstance();
        gameAudio.addClipSoundEffect("bounce", "bounce.wav");
```

After the sound is registered, it can be played through the `playSoundEffect` method of `GameAudio`. Listing 32 shows the modified handleCollision method of BallCollidable that plays a sound effect when the ball hits a robot.

Listing 32 - playing a "bounce" sound effect

```
    public void handleCollision(Collidable cl) {
        // handle collision against a robot
        if (cl instanceof RobotCollidable) {
            GameAudio.getInstance().playSoundEffect("bounce");
            handleRobotCollision((RobotCollidable)cl);
        }
        // handle collision against a goal
        else if (cl instanceof GoalCollidable) {
            handleGoalCollision((GoalCollidable)cl);
        }
    }
```

4.14. Extending the game

This tutorial has presented all the steps to set up a simple game with enJine. There are many possibilities to enhance the game created so far. Some of them are:

- Implement the `TitleScreen` class to display the game's title, using either a textured mesh or an overlay;
- Enhance the `ScoreOverlay` class to display the player scores with a more interesting font or add decorative graphics;
- Use the other features of the `GameAudio` class to include more sound effects or background music;
- Include a time limit to the game, and the player with the highest score when the time runs out wins.

References

[1] R. Tori, J. L. Bernardes Jr. e R. Nakamura, "Teaching Introductory Computer Graphics Using Java 3D, Games and Customized Software: a Brazilian Experience", *Proceedings of ACM SIGGRAPH 2006 Educator´s Program (Conference Select CD-ROM).* 8p.

[2] R. Tori, J. L. Bernardes Jr. e R. Nakamura, "Ferramentas e metodologia para ensino de fundamentos de computação gráfica em cursos da área de computação", *Proceedings of SIBGRAPI 2006 Workshop Computer Graphics & Education" (CD-ROM).* 7p.

[3] J. L. Bernardes Jr., R. Nakamura and R. Tori, "Using Game Development and Java 3D to Teach Computer Graphics", Game Career Guide, CMP Group. Available at:
http://www.gamecareerguide.com/features/286/features/286/features/286/using_game_development_and_java_3d_.php

[4] Eclipse IDE. Available at http://www.eclipse.org

[5] NetBeans. Available at http://www.netbeans.org