

A Framework for Designing 3D Virtual Environments

Salvatore Catanese¹, Emilio Ferrara²,
Giacomo Fiumara¹, and Francesco Pagano³

¹ Dept. of Physics, Informatics Section, University of Messina, Italy

² Dept. of Mathematics, University of Messina, Italy

³ Dept. of Information Technology, University of Milan, Italy

Abstract. The process of design and development of virtual environments can be supported by tools and frameworks, to save time in technical aspects and focusing on the content. In this paper we present an academic framework which provides several levels of abstraction to ease this work. It includes state-of-the-art components we devised or integrated adopting open-source solutions in order to face specific problems. Its architecture is modular and customizable, the code is open-source.

Keywords: Virtual Environments, Games.

1 Introduction

Commercial games reach production costs up to millions dollars. During the process of development, teams spend a lot of time in prototyping of particular features, and, commonly, building technical frameworks to design the game on top of them. Designing a game from scratch requires a lot of time, fund investments, skills and resources. This process could be shortened by adopting existing platforms. In this paper we describe a fully developed framework, thought to ease the development of 3D virtual worlds, supporting state-of-the-art techniques. It could be adopted if programmers and artists would prefer to focus on aspects of the design process (e.g., gameplay, game mechanics, etc.), instead of technical aspects. In academia, exploiting an existing solution could ease researchers to carry out specific experimentations ignoring other aspects of the development of a platform. In fact, the source code of our framework has been released open-source¹. This represents a great advantage w.r.t. using different solutions, also freely available on the Web, whose code is closed, architecture is fixed and components are not replaceable, differently from our framework. Moreover, its functionality and potential have already been exploited: demonstrative virtual environments, designed using our platform, have already been presented to the scientific community [2].

¹ <http://informatica.unime.it/velab>

2 Related Work

Literature and academic interest in supporting videogames development is growing. There are some works sharing similarities with ours, e.g., Liu et al. [6] developed a Virtual Reality platform, designing server and client applications, including a rendering engine based on *OGRE*, combined with *RakNet* for networking features, and finally they developed an example application. Also Braun et al. [1] recently presented a platform, for simulating virtual environments where users communicate and interact each other, using avatars with facial expressions and body motions. Some interesting techniques of computer vision and physics effects have been implemented to increase the realism of the simulation. In a broader panorama, Hu-xiong et al. [5] discussed the design and implementation of a game platform capable of running online games, both from client and server perspective. Concluding, Graham and Roberts [4] analyzed the videogame development process from a qualitative perspective, trying to define attributes of 3D games and adopting interesting criteria to help achieving desired standards of quality during the design of academic and commercial products.

3 Architecture of the Framework

In this section we briefly illustrate the components usually integrated in those frameworks supporting the design of virtual environments. An important feature typically provided is the capability of rendering and managing 3D scenes, as 3D environments are usually enriched by a realistic behavior of elements populating them, by reproducing physics and detecting collisions. Input/output management includes, among others, reproducing music and sound, handling devices like keyboard and mouse, pads, etc. Characters populating 3D virtual worlds could be driven by artificial intelligence (another aspect optionally supported), by other humans (requiring networking features for multi-player aspects), or both. In this project we developed components, interfaces and plug-in for the integration of some existing open-source solutions. We devised a framework able of supporting teams in the process of design and development of virtual environments. Its modular nature ensures the possibility of integrating additional components to tackle specific requirements.

Rendering Engine. Our framework integrates an open-source rendering engine, namely *OGRE*. The *rendering* is the process of generation of an image from a model. A *model* is a description of a three-dimensional object defined by data structures containing geometry, textures, lighting, materials, etc. The rendering can be in real-time or not: the first approach is adopted for example in developing videogames, while the latter is typically used in those applications where the primary requirement is photorealism rather than performance (e.g., post-production effects in movies, medical image analysis, etc.). Rendering algorithms usually try to simulate optic phenomena to reproduce 3D environments. These techniques rely on rendering *primitives* (e.g., triangles and polygons for three-dimensional scenes), and not pixels. The process of transforming 3D scenes

into 2D images is implemented by rendering pipelines, supported by graphics hardware. A typical input for a graphics pipeline is a model of scene and its output represents a bi-dimensional raster. *OpenGL* and *Direct3D* are two examples of graphics pipeline implementations. The interaction between graphics pipeline and hardware is possible via direct access to resources or graphics libraries. Usually, rendering engines exploit existing graphics libraries to ease the process of development. Our platform supports both *Direct3D* and *OpenGL*.

Scene Manager. A *scene manager* is included in our framework to manage scenes representing 3D environments. This component organizes the objects on the scene and the relationships among them, in a hierarchical way. A scene manager could be designed in different ways and its implementation could affect the overall performance. Our scene manager adopts a “scene graph” whose nodes represent entities or objects on the scene and edges represent relations among them. Moreover, it manages *bounding volume hierarchies* (BVHs), trees adopted to represent bounding volumes (e.g., spheres, bounding boxes, etc.) containing objects. BVHs are also adopted for speeding up the collision detection among objects on the scene. In order to increase performance, minimizing the number of rendered elements, our framework supports techniques of spatial partitioning, such as the *Binary Space Partitioning* (BSP) [3] and *octrees* [7]. Some areas are not always visible in the frustum (i.e., the region of space visible from the character point of view), thus the scene manager, exploiting spatial relations among nodes, disregards their rendering.

Physics Engine. A valid framework should allow the simulation of the physics on the reproduced virtual environment. Several physics engines have been developed during last years to improve the degree of realism of videogames. *Havok* and *PhysX* are two examples. The latter is a robust solution; we integrated it inside our framework, developing an interface to introduce a good level of abstraction.

Collision Detection. The problem of detecting collisions among objects on the scene is complex and involves several aspects of the simulation of a virtual world. *PhysX* provides advanced directives for detecting collisions of characters with obstacles, to simulate the impact of objects with other elements on the scene and so on. *PhysX* adopts bounding volumes to surround objects inside shapes and checks for interactions among bounding volumes; it additionally exploits the ragdoll model for collision detection of characters. After detecting a collision, the physics engine simulates effects on involved objects.

Character Controller. One important aspect of the gameplay in a videogame is the strategy of control of the character. Players interact with the virtual world using their avatars. It is fundamental to reproduce a realistic behavior in order to avoid frustration during the game experience. Moreover, the character is central in the perception of the player, thus all the imperfections are extremely visible. An important aspect to be considered is the shape adopted for detecting the collisions. A simple choice is a “capsule”, embedding the character; this because, i) the shape is smooth, so the character could run on irregular terrains without

getting stuck; ii) its symmetry ensures the character could turn around itself without hitting obstacles; iii) no rough-edges could obstacle the character when going through narrow passages. Collateral effects are related to its simplicity. More complex models, such as the ragdoll [8], have been proposed. This technique produces procedural animations, instead of static ones. A ragdoll model is a collection of constrained-rigid-bodies (usually representing bones and joints of characters) adopted as a skeletal animation system. Animations are generated in real-time, without adopting predetermined ones, increasing the degree of realism. Moreover, the rigid-body system is subjected to rules of the physics engine; thus, the interaction between the character and the environment is more accurate. Our framework supports both these two control strategies.

4 Design and Characteristics of Our Platform

Our goal was to design a framework solution to support design and development of virtual environments, providing the following features: 3D rendering capabilities, physics simulation, collision management and character controller, I/O management, scene and camera management (i.e., loading and saving scenes representing virtual environments, with graphics and physics characteristics, designed with a specific application).

4.1 Adopted Tools

There are several open-source solutions which provide state-of-the-art features for many of the required tasks. Thus, the most of the time has been spent for the integration of these components together, in order to obtain a unique framework. All the components, interfaced each other, work as a substrate for developing an application on top of them. A short description of the chosen tools follows.

OGRE. It is a cross-platform 3D rendering engine. This engine is scene-oriented and allows the graphical representation of 3D virtual environments, providing state-of-the-art techniques for visual effects, texturing and lighting. A key aspect is its modular nature. It is possible to extend the provided features by the inclusion of new components. We exploited both these aspects to extend and improve the engine itself. Its architecture supports both the *Direct3D* and *OpenGL* graphics pipelines. *OGRE* is constituted by a collection of classes and libraries. Three main classes (*scene manager*, *resources manager* and *render*) do the most of the work. It is possible to inherit and extend the default scene manager to implement new or different features; for example the “BSPSceneManager” is optimized to represent in-door environments, while the “TerrainSceneManager” better works with out-door scenes; finally, the “OctreeSceneManager” is a good compromise for general purposes.

OGREOggSound. We integrated this component in the framework; it is an audio library which acts as a wrapper for the *OpenAL* API. Its adoption allows to automatize the inclusion of audio features, e.g., to support wave and “Ogg

Vorbis” audio sources, static and dynamic environmental sounds, 2D/3D audio, etc. Sound elements can be included inside the scenes as positional audio sources to reproduce more realistic environments.

PhysX and NxOGRE. *PhysX* is part of our framework. It is in charge of the simulation of physics laws to increase the degree of realism of the reproduced virtual environments. Some key aspects managed by the engine involve a rigid-body and soft-body system simulation, an advanced character controller supporting different shapes (e.g., capsules, boxes, triangular and convex meshes, etc.), simulation of fluid dynamics, field strength, collisions, etc. *PhysX* is integrated via the *NxOGRE* class wrappers, which introduces a useful level of abstraction to ease the access to functionalities provided by the library.

Blender. *Blender* is a cross-platform open-source 3D graphics and modeling application. We adopted *Blender* because of the possibility of customizing this tool to include new features. *Blender* can manage multiple scenes. Each scene is represented by its structure, objects, textures, materials, sound sources, etc. We extended it to support also physics; some physical properties can be defined for each object, in order to better reflect its behavior in the virtual world and the type of interaction between the character and objects.

4.2 The Framework

The basic idea is to use *Blender* to design the scene and then pass its output to the appropriate manager. *Blender* originally deals just with graphics, but we extended and improved its open and powerful architecture. Using our framework, it is possible to add to each component of the scene a set of user-defined attributes and their values. This way, designers can specify values describing the physical properties of characters and objects on the scene. To represent this combination of graphical and physical attributes, we defined a novel XML file format via a DTD (Document Type Definition). Thus, we devised a new *Blender* exporter plug-in. The next step was to import these scenes into the framework. This task is delegated to the loader module which analyzes the imported file. It sends graphical information to *OGRE*, physical information to *PhysX* and sounds to *OGREOggSound*. *OGRE* is responsible to manage the scene and to collect user input from devices. Thus, input is passed to *PhysX*, which manages the movement of characters, determines possible collisions and sends back to *OGRE* the response, in order to update the graphical representation. Another task was the camera management. The final touch was sound management, to add soundtracks and dynamic sounds.

ExDotScene. The first step, adopting our framework, is to create the virtual environment, which could be designed by using *Blender*. Graphics are stored as meshes, scenes are represented adopting the designed scene graph, and physics is additionally included. The standard *DotScene* format does not contain meshes or textures, but just an XML description of elements on the scene. Our framework supports: i) the creation of physical objects with a graphical representation

(bodies) and without (actors); ii) static and dynamic objects instantiation; iii) serialization of multiple 3D scenes. Actually, the last point was not natively supported by the built-in `DotSceneInterface` library. We developed an improved version, namely `ExDotSceneInterface`, to allow loading and saving multiple scenes supporting the following elements: i) nodes of the scene graph; ii) graphical entities (i.e., meshes, textures and materials); iii) lighting (i.e., lights on the scene with properties like positioning, direction, brightness, etc.); iv) camera (i.e., source node and target node pointed by the camera, its positioning and orientation); v) scene attributes (e.g., in/out-door, type of shading, clipping distance, etc.); vi) representation of the physics of the scene (e.g., serialization of static and dynamic actors and bodies, models of physics, etc.). According to requirements of the scene loader, which should support the physics, a DTD has been defined, namely `ExDotScene`, which extends the official `DotScene` DTD. A body element contains a shape element and an “actorParam” element. The shape element must contain one and only one element chosen among a list comprising cube, capsule, sphere, convex and triangular meshes. These elements are described by typical attributes like dimensions, etc. Moreover, it is possible to specify additional parameters for the shapes, using a “shapeParam” tag.

Blender Exporter Plug-in. This component recursively analyzes each node of the graph scene, considering information relative to its position, orientation and scale. Also specific elements, such as lights, cameras or meshes are taken into account during this process. Moreover, the framework gives the possibility to export only the properties of objects that satisfy some specific conditions. This way, designers can choose to export physical properties related to the entire scene, or not. Acting on the “Logic Properties” panel of *Blender*, they can set the following values in order to define physical properties of each object: i) *body*, if set to false the object is an actor; ii) *shape*, suggests the type of shape to use for the object; iii) *static*, sets if the physical object is static or dynamic; iv) *mass*, sets the mass -only for dynamic objects-; v) *skin*, suggests the value of the skin width that the physics engine will use during the simulation; vi) *file*, determines the .nxs file to be adopted as a physical representation of the convex or triangular shapes. At the end of the export phase, the scene is stored as an XML file, using the previously discussed `ExDotScene` DTD, which includes the physics.

Scene Importer. The core of the scene importer with physics support is composed of the `DotSceneProcessor` class, plus a number of other node processor classes. `DotSceneProcessor` contains a list of objects and allows several scenes to be loaded. Formats different from the `DotScene` are supported, albeit overriding the correct methods for loading the scene. Our extension supports the `ExDotScene` format.

Character Controller. This component, which was designed exploiting some key aspects of the physics engine, includes the following features: i) creation

of the principal and secondary characters (i.e., creating and tracking these instances, managing the interaction among them and between characters and the environment); ii) update of the character status (i.e., managing movements and actions, at each frame, and synchronizing the graphical mesh of the character w.r.t. the actual status); iii) character auto-stepping (to avoid characters get stuck in minor terrain bumps); iv) character walkable parts (i.e., definition of those areas of the environment that are not accessible to characters, acting like boundaries); v) modifiable bounding volumes (to simulate the crouching or groveling of the character); vi) character callback (i.e., response to collisions). The control system of the character is composed of three main classes: `GameCharacter`, `GameCharacterController` and `GameCharacterHitReport`. By using *PhysX*, it is possible to divide actors by groups, so as to manage and set different behaviors during the collision detection and response, accordingly.

Game Character Controller. The `GameCharacterController` implements the Singleton pattern so to ensure that only one instance of the class may exist inside the application. This class is interfaced with the *PhysX* library and deals with the instantiation and the management of the character. During the render loop of *OGRE*, the `GameFrameListener` class invokes a function of the `GameCharacterController` in order to execute the “simulate” and “render” methods over all characters recorded in the controller.

Game Character. The `GameCharacter` class represents the player inside the physical scene. It is possible to set the associated graphical mesh and the scene node in order to synchronize the visual and the physical representation. A specific method deals with setting a new movement direction: a vector representing the three-dimensional components of the velocity of the character is used. At a given moment the position of the character depends not only on the user input, but also on the velocity. During the simulation cycle, the “simulate” method is used to update the physical shape and the “render” method to update the visual representation of the character by setting the position of the mesh according to the position, expressed in global coordinates, of the physical shape.

Character Hit Report. As a consequence of a collision, *PhysX* generates a `HitReport` event. The `GameCharacterHitReport` class allows the setting of customized callback actions as a consequence of the occurrence of a collision, determining the actor with which the collision occurred and the group the actor belongs to. If the actor is a dynamic object and belongs to a group the character may interact with, the impulse that must be applied to the actor is calculated.

Camera. Once a 3D virtual scene is designed and imported inside the framework, developers define the way users, using their characters, explore this world, the so called *camera system*. In general, the core of the camera system is composed of two scene nodes acting as point of view and target point of the camera. We suppose that the node connected to the camera always points to the target node. This way, the movement of the target node produces the movement of the camera node. Moreover, it is possible to move the camera around the target

object directly moving the camera node, thus obtaining particular cameraworks. This simple system allows three cameras, namely *chasing*, *fixed* and *first-person*. In the *chasing camera*, the target node is associated to the point the character is looking at. A more distant view node makes the character as appearing displaced w.r.t. the center of the scene. This way, the character is still visible but we obtain the effect of a side view of the scene. The *fixed camera* is similar. The target node is the point the character is observing, but the camera is fixed and cannot be moved. This system is used in several videogames (e.g., third-person games). In *first-person camera*, the scene node of the character is the camera node. The camera is independent from any other object on the scene, thus the scene coordinates are used to update the position of camera and target nodes.

4.3 Modular Structure

The framework has a modular structure, being composed of a series of packages each dealing with a certain functionality, as shown in Figure 1. The binding process is obtained developing the required interface classes. Sometimes, this introduces a useful level of abstraction in the implementation of the functionalities provided by each component. Each package could be replaced with other components to satisfy particular requirements. In other words, the framework can be customized without re-implementing all the functionalities, if the exposed interface of the replaced package is properly refactored. This is one of the advantages while using our framework w.r.t. other common frameworks (e.g., Unity). A brief description of the main packages follows.

GameSystem. It contains a series of classes dealing with the management of the rendering cycle. Within *OGRE*, it is possible to define some classes which detect changes before and after a frame has been rendered on the screen. To exploit this feature, it is necessary to register the various frame listeners in the object of *OGRE*. The class *GameFrameListener* controls the order of execution of frame listeners, stored as an ordered list. An instance of *GameFrameListener* is then registered in the root object, thus ensuring that each frame listener is executed in the correct order.

GameIO. It manages I/O devices using the Object-Oriented Input System (*OIS*) library. *OIS* provides two modes to manage the input, namely unbuffered and buffered. The latter input mode is safer, because the former could not reveal an event. *GameIO* implements the buffered mode using the *GameInputManager* class. *GameKeyListener* and *GameMouseListener* implement the *OGRE* corresponding interfaces and define the actions executed as a consequence of events produced by the input device(s).

GameAudio. It is developed as an interface with the *OpenAL* library in order to manage audio. To do so, we developed a wrapper class using *OGREOggSound*, which provides methods to integrate *OpenAL* features (see Section 4.1).

GameSceneLoader. It provides those methods needed to load and save a scene. It could be adopted to load *ExDotScene* based scenes, or, eventually, methods could be overloaded to manage different formats (see Section 4.2).

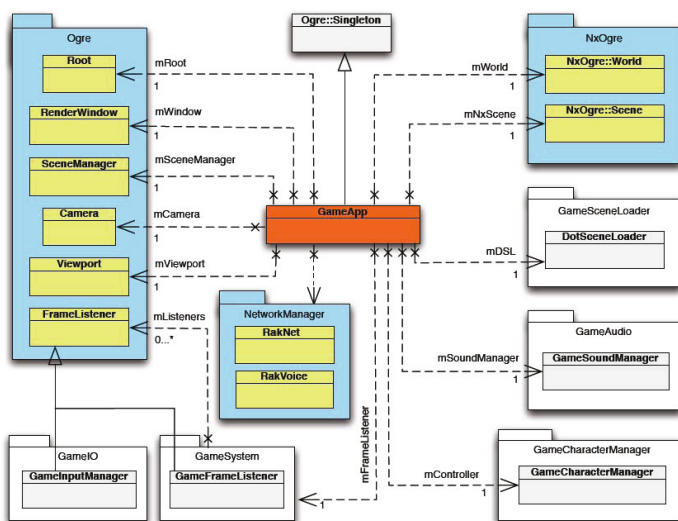


Fig. 1. The modular structure of the framework

GameCharacterController. It is the interface with the control system of the character developed with *PhysX*. It adopts a wrapper class to reproduce physics effects via *PhysX* and to manage the character controller (see Section 4.2).

RakNet. We integrated a cross-platform library for networking, which provides support for TCP and UDP communications. It also includes *RakVoice*, a toolkit for VoIP support and for real-time communications during game sessions which relies on the sound engine to reproduce sounds.

4.4 The Final Touch

The virtual world we earlier described still lacks of some features: first of all, logic has to be completely defined. The framework does not substitute developers in this aspect; this because we believe that a valid product should be designed even in details and the game logic can not be a surrogate of preset patterns. Moreover, the artificial intelligence of not-playing characters must be defined from scratch. The framework supports AI scripts coded in *Python*. The framework provides default implementation for the management of I/O devices but is not necessarily the optimal solution for any requirement. Similar arguments hold for other aspects, like the character control system or the camera system. Concluding, the strength of our platform is its modular nature, which ensures that developers could take the best from each component, but could also replace functions using other components, providing better, or simply different functionalities.

5 Conclusions

In this paper we presented the design and implementation of an academic framework to support the development of 3D virtual environments, helpful in those cases in which teams would like to save time in technical aspects, to focus on the contents. Our contribution can be summarized as follows: we presented a novel approach to create 3D virtual worlds enriched by physics effects. Our solution introduces the definition of physical properties of elements appearing on the scene, within the scene itself. These directives are interpreted by the physics engine, which is strictly interfaced with the rendering engine, reproducing physics effects. We additionally defined an extension of DTD for the *DotScene* format, introducing the support for the physics definition. This work represents the basis for future extensions and has already been adopted to show some techniques of design of virtual environments [2], e.g. the inclusion of environmental effects such as weather, day/night simulation and particle effects, exploiting techniques for terrain generation, realistic management of the water and fluid dynamics and the adoption of new rendering algorithms. Future work includes its adoption in different fields of application such as i) virtual/augmented reality; ii) virtual tours, reconstructions and museums; iii) engineering and architectural simulations.

References

1. Braun, H., Hocevar, R., Queiroz, R., Cohen, M., Moreira, J., Jacques, J., Braun, A., Musse, S., Samadani, R.: VhCVE: A Collaborative Virtual Environment Including Facial Animation and Computer Vision. In: Games and Digital Entertainment, pp. 207–213 (2010)
2. Catanese, S., Ferrara, E., Fiumara, G., Pagano, F.: Rendering of 3d dynamic virtual environments. In: Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques (2011)
3. Fuchs, H., Kedem, Z.M., Naylor, B.F.: On visible surface generation by a priori tree structures. *Computer Graphics*, 124–133 (1980)
4. Graham, T.C.N., Roberts, W.: Toward Quality-Driven Development of 3D Computer Games. In: Doherty, G., Blandford, A. (eds.) DSVIS 2006. LNCS, vol. 4323, pp. 248–261. Springer, Heidelberg (2007)
5. Hu-xiong, L., Guan-dong, X., Zhang, Y.: Solution for developing data communication module on networking game platform. *Comp. Eng. and Design* 11 (2005)
6. Liu, X., Du, H., Wang, H., Yang, G.: Design and development of a distributed Virtual Reality system. In: International Conference on Machine Learning and Cybernetics, vol. 2, pp. 889–894 (2009)
7. Meagher, D.: Geometric modeling using octree encoding. *Computer Graphics and Image Processing* 19(2), 129–147 (1982)
8. Witkin, A.: Physically Based Modeling: Principles and Practice Particle System Dynamics. SIGGRAPH Course notes (1997)