The Design of a Graphics Engine for the Development of Virtual Reality Applications

Silvano Maneck Malfatti [1]
Selan Rodrigues dos Santos [2]
Luciane Machado Fraga [1]
Claudia Marcela Justel [3]
Paulo Fernando Ferreira Rosa [3]
Jauvane Cavalvante de Oliveira [1]

Abstract: This work presents the design and the features of a flexible realtime 3D graphics engine aimed at the development of multimedia applications and collaborative virtual environments. The engine, called EnCIMA (Engine for Collaborative and Immersive Multimedia Applications), enables a fast development process of applications by providing a high level interface, which has been implemented using the C++ object-oriented programming paradigm. The main features of the proposed engine are the support to scene management, ability to load static and animated 3D models, particle system effects, network connection management to support collaboration, and collision detection. In addition, the engine supports several specialized interaction devices such as 3D mice, haptic devices, 3D motion trackers, data-gloves, and joysticks with and without force feedback. The engine also enables the developer to choose the way the scene should be rendered to, i.e. using standard display devices, stereoscopy, or even several simultaneous projection for spatially immersive devices. As part of the evaluation process, we have compared the performance of EnCIMA to a game engine and two scene graph toolkits, through the use of a testbed application. The performance results and the wide variety of non-conventional interaction devices supported are evidences that EnCIMA can be considered a real time virtual reality engine.

1 Introduction

Virtual reality (VR) applications aims to enable user to experience the sense of *presence* — casually defined as the feeling of being "in" a virtual environment — which is realized through factors such as immersion, involvement, and interaction [1, 2]. Allowing the user to experience a virtual environment (VE) through simultaneous and coordinated

[1]Laboratório Nacional de Computação Científica, Laboratório ACiMA, Av. Getúlio Vargas, 333, 25651-075 Petrópolis, RJ, Brazil, {malfatti,lmfraga,jauvane@lncc.br}
[2]Universidade Federal do Rio Grande do Norte, Departamento de Informática e Matemática Aplicada, Campus Universitário, s/n, 59072-970 Natal, RN, Brazil, {selan@dimap.ufrn.br}
[3]Instituto Militar de Engenharia, Seção de Engenharia de Computação SE/8, Praça General Tibúrcio, 80, 22290-270 Rio de Janeiro, RJ, Brazil, {{cjustel,rpaulo}@ime.eb.br}

sensory stimuli is the basic mechanism VR uses to afford presence. This is accomplished with special interaction devices (e.g. spatially immersive devices, head/face mounted displays, data gloves, haptic devices, trackers, etc.). Consequently, the process of designing a VR application is not an easy task, specially if the application is built directly on a graphics application programming interface (API). This situation requires the developer to be able to handle all the resources necessary to make the application work as intended.

To make things easier, in the early 90s the concept of "engine" appeared in the game industry. In general terms, an engine is a middleware, i.e. a group of library functions that abstract most of the low level implementation details by providing the developer with a high level programming interface [3]. These functions are organized in groups with specific functionality such as resource management, networked communication, 2D and 3D rendering, collision detection and response, audio rendering, physics simulation, and scene management [4]. As a result, the developers are able to generate applications that are independent of third-party APIs, provided that the programming is done through the high-level engine's functionality. In addition, this approach supports platform-independent VR applications and speeds up the overall development process. The success of this idea was responsible for making some VR application in areas like medical simulation, military training, education, art and entertainment, to adopt game engines as their basic development toolkit [5]. However, the use of game engines to develop VR applications has a major limitation: because engines usually supports only traditional interaction devices — i.e mice, keyboards, and joysticks — the type of virtual environment that can be generated is restricted to desktop VR [6].

According to Maia [7], another issue is that the resources of a game engine are targeted at supporting game related features. To obtain a general purpose game engine it would be necessary to re-design it entirely. For this reason, there have been an effort towards developing engines specialized for VR applications. For instance, the work done by Pinho [8] showed that VR engines should provide special features such as support to a wide range of special interaction devices, various 3D model loaders, schemes for management and optimization of 3D graphics environment, multi-platform support, and multi-screen display capability. Therefore, we set out to design a virtual reality engine that *i)* follows a modular approach, organizing coding in layers of abstraction; *ii)* affords a fast development process of applications with support to collaboration and multimedia features; *iii)* supports various types of VR interaction devices, in special multi-screen display with stereoscopy; *iv)* is independent of third-party libraries, and; *v)* delivers high-performance graphics in a more engaging environment with animated characters, better lighting, particle system, terrain modelling, collision detection, support to acceleration hardware and specialized display devices such as CAVE-like systems.

This paper presents a VR engine aimed at assisting the development of VR applications, which supports the aforementioned features in addition to the graphics resources

available in tools used in game design.

Section 2 presents some background on the usual components of an engine architecture. In Section 3 we present a few examples of similar graphics engines, focusing on their features, whereas in Section 4 we delve into the EnCIMA's architecture, exploring some of its features and functions. Section 5 describes two test applications developed with our engine and presents some performance results Finally, in Section 6 we present some conclusions and future work.

2 Graphics Engine Architecture

A graphics engine is a key component in a VR application, being responsible for performing important tasks such as accessing input devices, resource management, updating components of the virtual environment, rendering the 3D scene and presenting the final result through display devices [2, 5].

Traditionally, the engine's functionality is organized in layers. Maia in [9] defined a generic structure for engines, which is composed of three layers, as shown in Figure 1. The EnCIMA's design follows this approach, having three layers, namely *Core*, *Sub-systems*, and *Application*, as shown in Figure 5. These layers, in turn, are organized into modules with a strict hierarchical dependency.

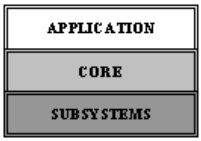

Figure 1. Generic architecture of a game engine, according to Maia [9].

2.1 The Sub-systems Layer

The sub-systems layer is composed of several modules that offer specific services to the core which, in turn, transforms them in high-level actions for the application. Therefore, every module of this layer is characterized by the application domain it is target for, the tasks assigned to the module, and the technology employed to execute these tasks.

Consider, for instance, the *Input* module which is responsible for recognizing the interaction devices plugged into the application. The *Input* module's main attribution is to receive user action entered through devices like mouse, keyboard, joystick, data gloves, and position tracking systems. For that purpose, the *Input* module must provide functions that recognize

buttons being pushed, requests for updating cursor position, understand tracker orientation and positioning in terms of VE's coordinate system, etc.

For this reason, every module in the sub-systems layer represents and interface between the core and APIs or low level driver that can have direct access to the underlying operational system. Figure 2 shows an example of a sub-systems layer composed of three modules with specific responsibilities.

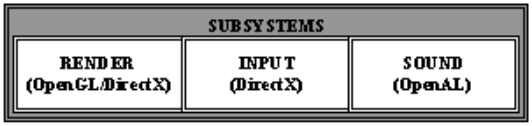

Figure 2. Example of a graphics engine's sub-systems layer.

2.2 The Core Layer

The core is responsible for providing the link between the application layer and the engine's available sub-systems. Johnston [10] defines the core as the engine's central component or server, responsible for invoking the appropriate functions with the right parameters in response to events generated by the user or the environment. Therefore, all modules from the sub-systems layer must register with the core, so that the core is able to initiate each registered module and coordinate the interaction and data exchange among the engine's components.

Similarly to the sub-systems layer, the core is also organized into several modules, called *managers*, each of which having specific responsibilities that are fundamental for the engine's proper functioning. These responsibilities may include access to the local file system, resource management, mathematics core library, activity log, and the specification of the manipulation interface between objects from the VE and the associated devices. Figure 3 presents an example of a core with six modules.

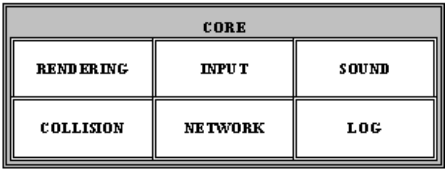

Figure 3. Example of a core layer comprised by six modules.

2.3 The Application Layer

The application is the target software that is supposed to use the functionality offered by the core. It is the application designer's responsibility to specify how the VE should look like, load and position all 3D models, set up sensors to trigger the appropriated animation and visual effects, as well as to define the VE's response based on either the engine's state machine or user interaction.

The usual strategy employed by VR application developers when they want to create a new VE is to derive through inheritance the engine's classes that offer support to scene resources. In doing so, the developer has access to high-level methods that encapsulate the functionality available within each of the various core's modules.

3 Related Work

In this section we briefly present and review several engines that can be organized in three groups: academic VR engines, 3D game engines, and graphic engines. We wanted to understand the functional aspects of the selected engines and how effectively they could be adapted to build VR applications.

3.1 Academic VR Engines

3.1.1 enJine

The enJine is an open source graphics engine developed entirely on Java and Java3D API. Consequently, one of the main features is the platform independence and the object-oriented paradigm, both inherent to Java applications [11].

The underlining motivation for enJine was to provide a tool with educational purposes, supporting the teaching of game programming, VR applications, and other computer related concepts such as software engineering and computer graphics.

The VR aspects of the enJine benefited from the Java 3D inherent features which affords, with a few lines of code, access to unconventional display devices such as head/face mounted displays, stereo rendering, and multi-screen projection. Despite the existence of comprehensive documentation and the facility in developing VR application, we believe enJine to have one noticeable limitation: it only supports the so-called traditional interaction devices, i.e. mice, keyboards, and joysticks.

3.1.2 SmallVR

SmallVR is a toolkit that supports the development of VR applications [8]. The conception of SmallVR was driven by the necessity of an educational tool that could be used in the practical lessons of Virtual Reality modules. SmallVR strongest advantage is to afford a fast development process of VR applications. This is accomplished because SmallVR provides all the basic core functionality, therefore the students do not need to program neither graphics device control nor functions to load 3D object models.

The third-party libraries used by SmallVR architecture are GLUT and OpenGL [12]. As a result, it is possible for the developer to keep utilizing the basic structure of GLUT programs, and still add objects and program actions without having to surrender the application's execution control. Other features of SmallVR are the support to scene graph, loaders for popular 3D model formats such as Wavefront's OBJ and Autodesk's 3DS, the implementation of rendering acceleration algorithms such as view frustum culling, collision detection, the existence of drivers to support motion tracking devices and head/face mounted displays.

Although SmallVR offers a range of valuable features and proper documentation, the toolkit does not provide other important graphics resources such as support for animated 3D models, shadow casting, particle systems, as well as audio rendering — all these features are left to the developer to program.

3.1.3 CGE

The CGE, CRAb Graphics Engine, is one of the most complete academic VR engines available. CGE was developed based on the CRAbGE framework, proposed by Maia [9].

CGE is an open source, platform-independent engine whose rendering system is done with the OpenGL API. One of the hallmark features of the CGE is a highly customizable interface, via scripts and plugins. In terms of graphics resources, CGE provides several scene rendering acceleration techniques such as hierarchical frustum culling, level-of-detail (LOD), billboards, and particle systems.

Furthermore, CGE also supports collision detection, spatialized sound, the loading of static and animated 3D models such as 3DS and MD2, and accepts some types of special interaction devices like positioning tracker and dataglove. However, CGE does not handle simultaneous and coordinated renderings for multi-projection spatially immersive devices, like the CAVE system, nor provide support for haptic devices.

3.2 3D Game Engines

3.2.1 Delta 3D

Delta3D [13] is a fully fledged game engine that can be understood as a middle layer between the application and a set of many open source products such as OpenAL (audio), Python (scripting), fltk (GUI), OpenSceneGraph (rendering), Open Dynamics Engine – ODE (Physics simulation), Cal 3D (character animation), etc.

Delta 3D provide a high-level, cross-platform C++ API, whose main feature is its modularity, or the ability to replace any module of the lowest level for a new option without having to change the application code. Delta3D also handles networking, supports the rendering to multiple simultaneous displays, and provides some auxiliary tools to help the application development process, such as an object viewer, a particle editor, binary space partition (BSP) compiler, etc.

In terms of supported devices, it handles keyboard, mice, joystick, and trackers. However, it does not provide support to haptic devices and rendering to multiple displays has to be coordinated and adjusted by the programmer.

3.2.2 IRRLICHT

The IRRLICHT engine [14] is another open source, cross-platform, high-performance game engine, developed in C++, whose rendering is done in either OpenGL or Direct X. Its strongest feature is the rendering capability, with a large set of visual special effects, such as dynamic shadows, particle systems, light maps, environment mapping, and character animation.

IRRLICHT imports common mesh formats such as Maya, 3DStudio, COLLADA, Milkshape, Quake 3, etc. Differently from other engines, IRRLICHT also supports 2D drawing functions like alpha blending, color key based lighting, font drawing and mixing 3D with 2D graphic.

Nonetheless, it does not handle audio properly and only supports conventional interaction devices, which clearly compromises its use in the development of VR applications.

3.3 Unreal Engine 3

This is one of the most complete and popular game engine. It also is coded in C++ and can be ported to several PC based architecture as well as game consoles.

The highlight of the Unreal is the high quality graphic results that can be achieved with this engine, without compromising performance. Even though it is a commercial en-

gine, its special license allows the free use of the engine for training purposes or in research projects [15]. However, the applications created with this license cannot be distributed, unless a commercial license is purchased. The main disadvantages of this engine is the lack of support to VR interaction devices and complex API.

3.4 Graphics Engines

3.4.1 OGRE

The OGRE 3D (Object-Oriented Graphics Engine) is considered one of the most popular open-source graphics engine. It is well known for its large community of users and good documentation [16]. It is developed with C++, built atop of OpenGL or DirectX, and is also cross-platform. OGRE supports most of the traditional computer graphics techniques and algorithms, as well as vertex and fragment programs. It has its own mesh format which is exported from most of the popular professional modelling software.

In terms of adequacy to VR applications, OGRE offers support to different types of stereoscopic rendering, rendering to head-mounted displays, CAVEs, and projection walls.

Nonetheless, it has a poor performance when running the OpenGL version in linux. Another limitation is that it handles only its own mesh format, therefore all models have to be converted to that format.

3.4.2 OpenSceneGraph

The OpenSceneGraph (OSG) is not actually an engine but a high performance 3D graphics toolkit that provides the middleware necessary to support a graph representation of a scene. OSG interfaces the application layer and the low-level rendering API (e.g. OpenGL). It is a cross-platform API developed in Standard C++ and OpenGL, and has been used in applications in the field of scientific visualization, VR, video game industry, and simulations.

OSG supports rendering features such as view-frustum culling, level of detail (LOD), vertex array, OpenGL Shader language, a wide range of 3D database loaders and writers, and can be interfaced with popular GUI toolkits such as Qt, FLTK, SDL, wxWidgets, etc.

One of the main features is the built-in support to special movements of camera such as airplane mode, or even wheeled vehicles, all that with collision detection. However, because OSG is not a proper engine, it lacks supports to special interaction devices and some of the traditional visual effects commonly found in other graphics engines.

4 The EnCIMA Engine

The EnCIMA engine was designed with the purpose of enabling the developer to get his application up and running as quick and simple as possible. The engine offers an easy way to use object-oriented interface designed to minimize the effort required to render 3D scenes, in such a high level that the application becomes independent of third-party 3D graphics rendering API (e.g. Direct3D or OpenGL). For that reason, the developer does not need to have previous or specific knowledge on how to program a given API nor the details necessary to allow interaction with special input/output devices. In terms of graphics resources, EnCIMA provides the loading of both static and animated 3D models, the procedural generation or loading of terrain meshes, environment effects such as skydome, sprites, static and animated billboards, and particle system effects.

The engine also provides high-level functions to control 2D and 3D audio playback, and access to spatialized sound that can be affected by a series of dynamic simulation effects such as Doppler, environment volume, attenuation, and movement of the sound's source location.

Additionally, the engine supports 3D and 2D input devices. Figure 4 shows a variety of devices handled by the engine: data gloves, 3D mice, tracking systems, joysticks (with or without force feedback), and Sensable's Phantom.

The EnCIMA's architecture (show in Figure 5) follows a multi-layer design that integrates the traditional components mentioned in Section 2: Sub-systems, Core, and Application. In the coming subsections we describe in more details each of the EnCIMA's three layers.

4.1 The Sub-systems Layer

The sub-systems layer integrates well-known APIs, and proprietary drivers that communicate directly with the Windows-based operational system. EnCIMA renders using OpenGL API, whereas the audio is handled by DirectX. The graphics user interface, interaction devices driver, and network connection are all handled by native Windows API.

4.2 The Core Layer

The core layer is composed by a set of manager modules. Each manager is defined as a C++ class whose main attribution is to communicate with the sub-systems and offer services to the application layer. Figure 5 presents the seven modules that form the EnCIMA's core layer. The next subsections describe the functionality of all seven modules.

Figure 4. Example of devices supported by the EnCIMA engine (left to right, top to bottom): data glove, 3D mouse, 3D tracker, wireless joypad, force feedback joystick, and phantom haptic device.

4.2.1 Graphics Manager

The *graphics manager* is responsible for the allocation of graphics resources, the loading of several image formats (including support to transparency), texturing, and the ability to capture screenshots. The singleton graphics manager is available to all classes that require any image related action, for instance texture loading, heads up displays (HUDs), terrain generation, and particle systems (e.g. fire, smoke, etc.). The graphics manager also provides an automated garbage collection system that shares graphics resources, and uses reference-counted memory to avoid unnecessary duplication of resources and memory leaks. This simple approach avoids memory waste and optimize its usage and access. The code shown in Code 1 presents the `LoadImage` method of the `GraphicsManager` that is responsible for the image loading process.

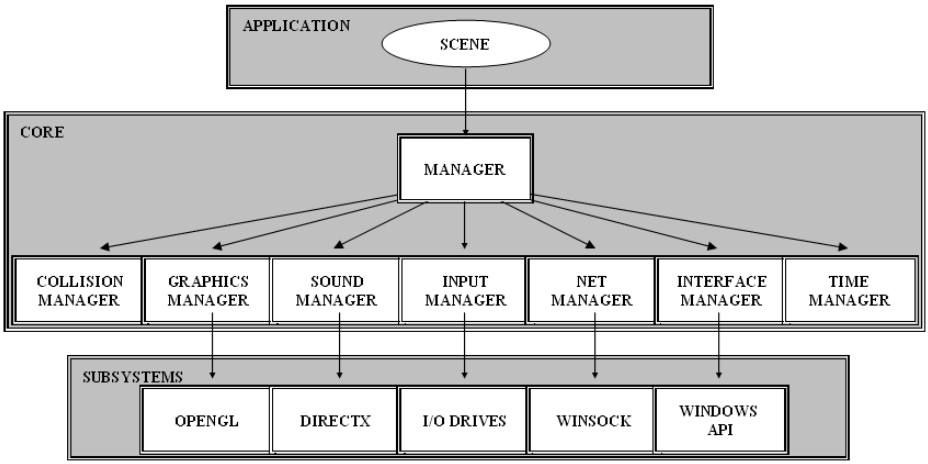

Figure 5. The EnCIMA's architecture diagram.

Code 1 `GraphicsManager`'s method to load images.

```
1  CVRImage*
2  CVRGraphicsManager::LoadImage( char* fileName, bool transparency ) {
3    // Verify whether the image has already been loaded.
4    for ( int iIndex=0; iIndex < (int)vImages.size(); iIndex++ ){
5      for ( strcmp( fileName, vImages[iIndex]->GetImageName() ) == 0 ) {
6          vImages[ iIndex ]->iReferences++;
7          return vImages[ iIndex ];
8      }
9    }
10
11   // Not loaded yet, thus we create a new one.
12   CRVImage* pNewImage = new CRVImage;
13
14   if ( pNewImage ) {
15       pNewImage->LoadImage( fileName, transparency );
16       if ( pNewImage->GetImageData() ) {
17           vImages.push_back( pNewImage );
18           return pNewImage;
19       }
20   }
21
22   // Image not found.
23   delete pNewImage;
24   return NULL;
25 }
```

4.2.2 Time Manager

The *time manager* controls all timers used in animation and special effects. Its main task is to guarantee that timers have a consistent temporal behavior in machines with various

processing capacities. The main advantage in centralizing time management is that the low level function that retrieves the system time is called only once every frame and used to update all registered timers.

4.2.3 Input Manager

The *input manager* communicates with every input device supported by the engine. This module detects, initiates, and uniquely identifies all input devices plugged into the application. The Code 2 snippet shows the usage of the input manager to initiate and obtain the angles from one of the 3D positioning tracker's sensors.

Code 2 Establishing communication with 3D positioning tracker through the input manager.

```
1  void SceneTest::Init() {
2     // Creates a 2D bitmap font.
3     fAngle = 0.0f;
4     Font   = CreateFont2D( "EnCIMA", 30, true );
5     Font->SetColor( 1.f, 0.f, 0.f );
6     Font->SetPosX( 50.f );
7     Font->SetPosY( 50.f );
8
9     // Initialize the tracker.
10    pManager->cInputManager.cTracker.Init();
11
12    // Get the angle with Y axis from sensor 1.
13    fAngle=pManager->cInputManager.cTracker.GetAngleY(1);
14 }
```

4.2.4 Sound Manager

The *sound manager* coordinates the loading of 2D and 3D sounds, allowing the application to set up various sound parameters such as volume, 3D position, area of influence, and attenuation.

4.2.5 Network Manager

The *net manager* is responsible for establishing network connections between servers and clients, for collaborative applications. The data exchange follows a multicast-based client-server model. This means that every server is responsible for both managing groups of client applications and delivering modifications to all the participants of a given group.

4.2.6 Interface Manager

The *interface manager* is responsible for handling every detail related to the display device utilized by the application. For traditional displays, it is possible to set window size, position, background color, stereoscopic rendering, graphics resolution, and fullscreen mode.

4.2.7 Collision Manager

The *collision manager* avoids virtual objects to penetrate one another. A pre-stage or broad phase in collision detection is to identify the moment that objects get close enough to be considered for collision tests or narrow phase [17]. For the broad phase it is possible to chose among methods that focus on the space occupation, such as regular grids, or BSPTree; or a new method (in development) that focuses on the virtual object itself, called area of interest. The later method employs the same principles found in the area of Collaborative Virtual Environment to reduce message exchange among participants [18] to reduce the number of collision tests among dynamic objects. The main feature of the collision manager is the use of a dedicated physics processing unit to accelerate the narrow phase of the collision detection process.

4.3 The Application Layer

The *application layer* is the starting point for any VR application developed on EnCIMA. The EnCIMA engine was designed with the purpose of enabling the developers to get their application up and running as quick and simple as possible. The engine offers an easy to use object-oriented interface that reduces the effort required to render 3D scenes, in such a high level that the application becomes independent of third-party 3D graphics rendering API (e.g. Direct3D or OpenGL). For that reason, the developer does not need to have previous or specific knowledge on how to program a given API nor the interaction with special input/output devices.

Through this layer an application has access to all the engine's functionality and resources. The class `Scene`, shown on the top of Figure 5, is the realization of this layer. This class contains a high-level manager (`Manager`) that has a reference to every manager located in the core.

This centric approach facilitated the implementation of the `Scene` class where several high-level functions are available to the developer. The only requirement to start off an application is to derive through inheritance the `Scene` class and override the `Init()` and `Execute()` methods. The `Init()` method, as the name implies, initiates the application's objects and resources and is called only once in a run, whereas the `Execute()` is called every frame and is responsible for the application loop that contains the appropriate sequence

of actions. The class `CVRScene` handles all the other details, such as resource allocation, the rendering of 2D and 3D objects, and the release of system resources when the application finishes.

Other methods available in the `Scene` class are:

- `CreateFont2D()`: creates bitmap fonts to be used as HUDs.
- `CreateBillboard()`: creates a textured polygon whose front-face is always facing the virtual camera. This resource is specially useful in particle systems.
- `CreateMd2()`: loads an animated object in Md2 format. This alleviates the developer's burden of having to program animation, since this can be done in a pre-stage, using specialized modelling and animation software.
- `CreateObj()`: loads an static object in Wavefront's OBJ format. The object can be represented by a mesh of triangles or n-sided polygons, associated to material or texture. This is specially useful when loading pre-designed scenarios.
- `CreateBmpTerrain()`: generates a height-field-based terrain from a greyscale bitmap image.
- `LoadSound()`: loads a 3D sound that can be positioned anywhere within the VE.
- `LoadMusic()`: loads a 2D sound to be played regardless of the user's location within the VE.
- `CreateSkyBox()`: creates a background for the environment based on a texturized cube that encapsulates the entire VE.
- `CreateSkyDome()`: creates a background for the environment based on a texturized sphere that surrounds the entire VE.

In addition to the above mentioned functions, the `Scene`'s high-level manager grants access to all the core's modules, thereby allowing direct use of important underlying behavior when needed. Table 1 summarizes a comparison of functionalities provided by EnCIMA and the reviewed engines of Section 3.

5 Case Study

In this section we present two case studies in which the EnCIMA engine has been successfully applied in order to generate a VR application. The goal of the first application (shown in Figure 6) was to support navigation and exploration of an oil platform's installations. Furthermore, the application simulates two types of situations: the execution of evacuation and rescue procedures in case of a serious accident, and a training scenario for firefighters.

Table 1. Comparison of engine's functionality in terms of support to VR applications.

	VR REQUIRED FEATURES			
ENGINES	Multi-screen display	3D Sound	Stereoscopy	Haptic devices
enJine	•	•	•	–
SmallVR	•	–	•	–
CGE	–	•	–	–
Delta 3D	•	•	•	–
IRRLICHT	–	–	–	–
Unreal Engine 3	–	•	–	–
OGRE 3D	•	–	•	–
OpenScenGraph	•	–	•	–
EnCIMA	•	•	•	•

Obs: '•' means feature supported.

The environment for this application is represented, basically, by a static 3D model of the oil platform in conjunction with a skybox. The code fragment presented in Code 3 shows the `Init()` method that instantiates the application resources. To create the simulation's graphical interface the developer only needs to declare reference variables for the objects needed and invoke the methods from both the `Scene` class and the core's modules to allocate resources.

The second application, called AVIDHa (*Atlas Virtual Interativo Distribuído Háptico* or Distributed Virtual Human Atlas with Haptic Sense), is a 3D human body atlas for the purpose of anatomy study. AVIDHa allows students to interactively explore the several human body systems through the senses of touch and stereoscopic vision. The human body systems are available as high definition 3D models with photo-realistic textures acquired from Anatomium[4], as shown in Figure 7.

The application allows the anatomy student to fly through and inside the human body. The flight and exploration modes are done with either 3D mouse[5] or joypad. The student may also choose to investigate each system separately, change organ's opacity to exam internal parts, capture screenshots for later examination, manipulate clipping planes to explore the inner parts of a given system, or even use a haptic device, such as the Sensable's Phantom Omini[6], to feel the organ's density and contours.

The AVIDHa may also run as a distributed collaborative application, allowing users geographically apart to each other to interact through the EnCIMA's network support. In this case a mediator, possibly an expert in anatomy, may drive the simulation and share his/her

[4] `http://www.anatomium.com/n-p1.html`
[5] SpacePilot/SpaceNavigator from `http://www.3dconnexion.com`
[6] `http://www.sensable.com/`

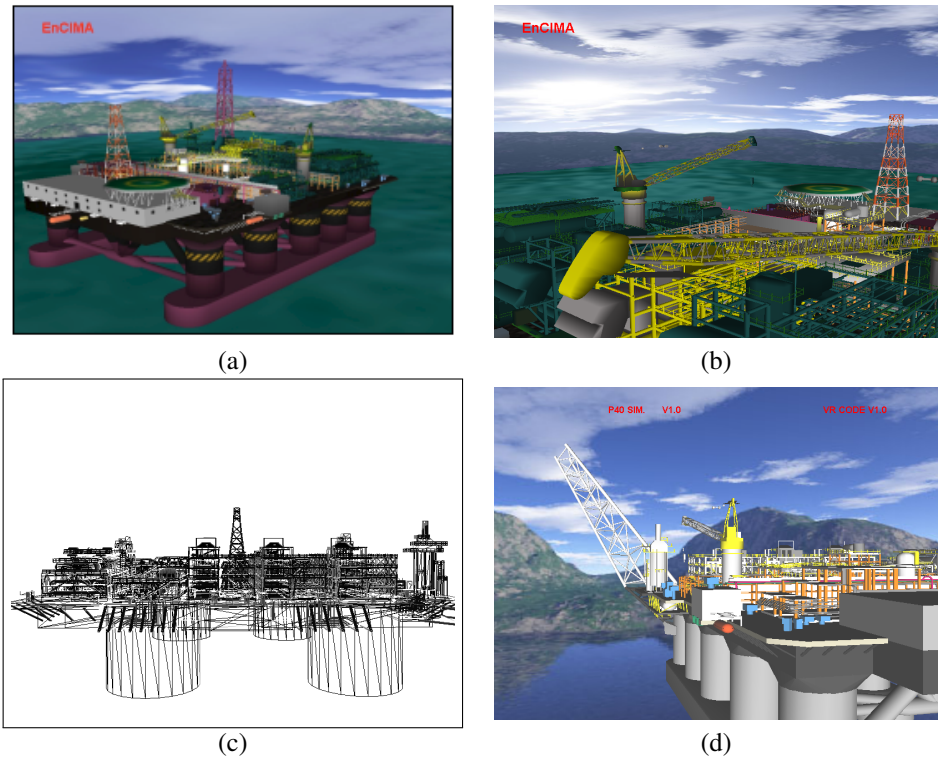

Figure 6. Using EnCIMA to render and simulate an oil platform: (a) overview, (b) detail, (c) the platform rendered in wire-frame mode, and (d) the platform is fully rendered with textures and surrounded by a skybox.

knowledge with other participants. When started, the application needs to load all 3D body systems. After that the rendering starts, which can either be monoscopic, displayed on a typical desktop monitor, or stereoscopic, displayed on a multi-screen projection or CAVE-like display.

In terms of performance for the AVIDHa application, the engine delivered a refresh rate of approximately 30 frames per second, for a 3D models with 5.3 million polygons and 87.8 MBytes of texture, running on a Intel Pentium D (3.0 GHz and 2.0 GBytes of RAM) with a NVIDIA GeForce 8800 GTX. Table 2 summarizes a comparison of performance obtained with the same application developed with other engines.

The results from Table 2 shows that EnCIMA has achieved a rendering performance

Code 3 Initializing the oil platform simulation application.

```
void SceneTest::Init() {

   // Creates a 2D bitmap font.
   fAngle = 0.0f;
   Font   = CreateFont2D( "EnCIMA", 30, true );
   Font->SetColor( 1.f, 0.f, 0.f );
   Font->SetPosX( 50.f );
   Font->SetPosY( 50.f );

   // Set the camera's position.
   pManager->cView.SetPosition( -500, 90, 0 );
   pManager->cView.RotY( 5.f );
   pManager->cView.UpdateView( );

   // Loads the platform model stored as an obj file.
   obj = this->CreateObj("p40t.obj","objects\\",true);

   // Loads sound.
   CVRListener* list = GetListener();
   CVRSound* sound = LoadSound("\\Sounds\\SANDSTRM.wav");
   sound->SetRepeat( -1 );
   sound->PlaySound( );
   sound->SetVolume( -800 );
   sound->SetMaxMin( 300, 100 );
   sound->SetPosition( -160, 30, 0 );

   // Creates a skybox.
   skybox = CreateSkyBox( true );
   skybox->SetCenterBox( 0.f, 0.f, 0.f );
   skybox->SetBoxSize( 3000, 1000, 3000 );
   skybox->SetFrontTexture( "\\Images\\front.bmp" );
   skybox->SetBackTexture( "\\Images\\back.bmp" );
   skybox->SetLeftTexture( "\\Images\\left.bmp" );
   skybox->SetRightTexture( "\\Images\\right.bmp" );
   skybox->SetDownTexture( "\\Images\\water3.bmp" );
   skybox->SetUpTexture( "\\Images\\up.bmp" );
   skybox->SetDrawGround( true );
   skybox->SetFloorHeight( 400 );
}
```

that approximates those of well established open source engines, which was one of the primary goals of this project. EnCIMA had the lowest loading time, mostly because our scene graph is simpler if compared to, say OSG, having only the basic OpenGL culling as its main acceleration algorithm. In contrast, the OSG based application had the worst loading time, taking almost 19 minutes to finish. Delta 3D, an open source game engine with modular architecture, uses other open sources projects in its sub-systems layer. Delta 3D renders with OpenGL and OSG, hence the same performance for both Delta 3D and the stand-alone OSG application. Notice, however, that none of the tested engines offers the same set of features as EnCIMA, specially the support to specialized interaction devices typical of VR applications, as shown in Figure 8. Delta 3D is the engine with the closest functionality to EnCIMA's.

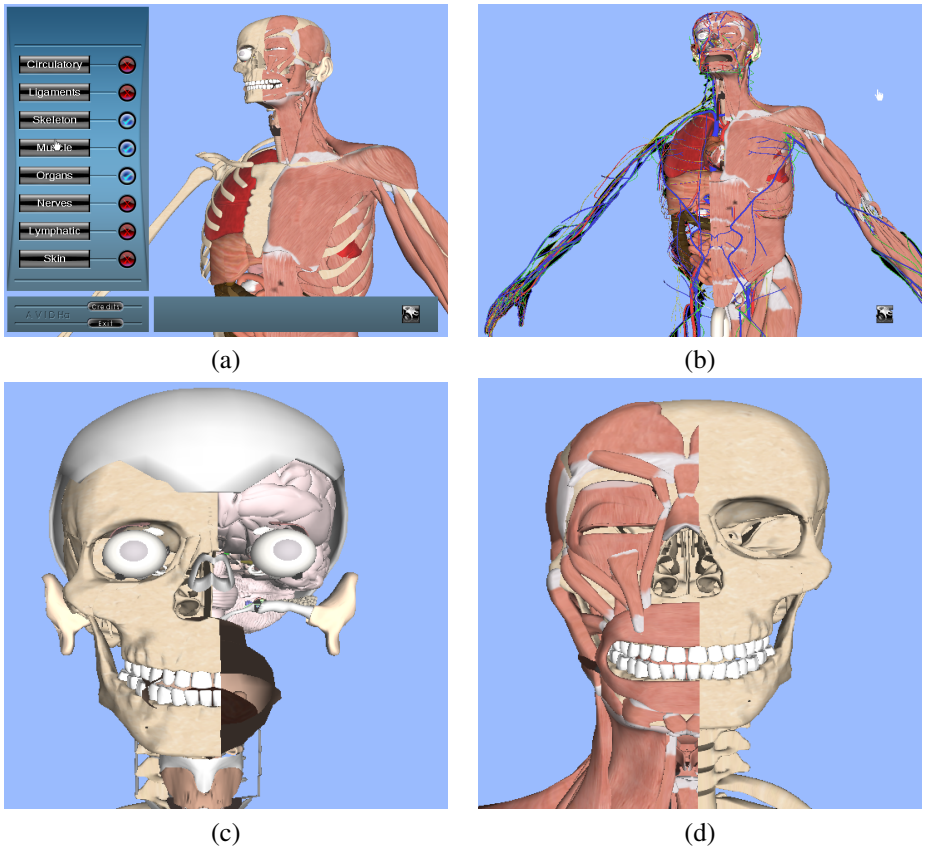

Figure 7. The AVIDHa application: (a) the interface is composed of user controls rendered as HUDs, and a 3D rendering space in which the human body model is shown; in (b) to (d) we show a detailed view of 3D models of human body systems. Some of the renderings were generated with transparency, clipping planes, and combination of body systems.

6 Conclusion and Future Work

The EnCIMA's flexible API offers the basic elements needed by all visualization and simulation applications, without requiring the user to have any knowledge on OpenGL or DirectX libraries. Consequently, applications built upon EnCIMA are up and running in a short development time, with considerably fewer lines if compared to traditional coding process.

Table 2. Comparison of engine's performance when loading a 5.3 million polygon model with 87.8 MBytes of texture on a Intel Pentium D 3.0 GHz, 2.0 GBytes of RAM, and a NVIDIA GeForce 8800 GTX.

ENGINE	APPROXIMATE FPS	LOADING TIME (MILLISECONDS)
Delta 3D	60	470,890
OSG v2.0	60	1,135,047
OpenSG v1.8	34	274,186
EnCIMA	31	145,422

In terms of performance, we achieved a frame rate sufficient to grant EnCIMA real time rendering status, which was one of our objectives established at the onset of this project. But the strongest feature, the one that distinguishes EnCIMA from traditional game engines, is the various support to specialized interaction devices, such as 3D mice, data gloves, haptic devices, and multi-screen display systems.

The next step is to port EnCIMA to other platforms, as well as the creation of navigation aids (such as maps, signs, and automatic camera control) that would help users to perform wayfinding in complex virtual environments. We also intend to develop an object deformation module, which is important in providing a consistent visual and haptic feedback.

7 Acknowledgement

The authors acknowledge financial support from Fundação Carlos Chagas Filho de Amparo à Pesquisa do Estado do Rio de Janeiro (FAPERJ, grant number 170.332/2006), Rede Nacional de Ensino e Pesquisa (RNP Giga, number 2439), PCI/LNCC/MCT, and Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq, grant number 305106/2004-0).

References

[1] J. G. R. Maia, J. B. Cavalcante Neto, H. O. O. Gomes, and C. A. Vidal, "CRAbRender: Um sistema de renderização para aplicações de rv," in *Proceedings of the VII Symposium on Virtual Reality, SVR2004*, vol. 1. São Paulo, SP: SBC/Faculdades Senac, 2004, pp. 380–382.

[2] G. C. Burdea and P. Coiffet, *Virtual Reality Technology*, 2nd ed. Wiley-Interscience, 2003.

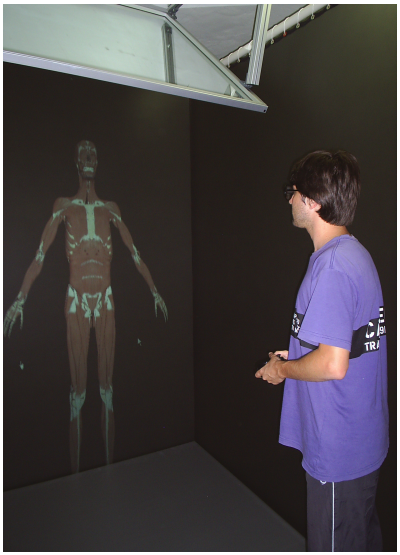

Figure 8. AVIDHa running on a stereo CAVE-like system.

[3] M. Lewis and J. Jacobson, "Game engines in scientific research - introduction," *Communications of the ACM*, vol. 45, no. 1, pp. 27–31, 2002.

[4] K. C. Finney, *3D Game Programming All in One*, 1st ed., ser. Course Technology PTR Game Development Series, André LaMoth ed., A. LaMoth, Ed. Thomson Course Technology PTR, April 2004.

[5] W. R. Sherman and A. B. Craig, *Understanding Virtual Reality: Interface, Application, and Design*, 1st ed. Morgan Kaufman Publishers, 2003.

[6] B. Stang, "Game engines: Features and possibilities," Institute of Informatics and Mathematical Modeling at The Technical University of Denmark, Tech. Rep., 2003.

[7] J. G. R. Maia, J. B. Cavalcante Neto, and C. A. Vidal, "CRAbGE: Um motor gráfico, customizável, expansível e portável para aplicações de realidade virtual," in *Proceedings of the VI Symposium on Virtual Reality, SVR2003*, vol. 1. Ribeirão Preto, SP: SBC/Faculdades COC, 2003, pp. 3–14.

[8] M. S. Pinho, "SmallVR: Uma ferramenta orientada a objetos para o desenvolvimento de aplicações de realidade virtual," in *Proceedings of the V Symposium on Virtual Reality, SVR2002*, vol. 1. Fortaleza, CE: Porto Alegre: SBC, 2002, pp. 329–340.

[9] J. G. R. Maia, "CRAbGE: Uma arquitetura para motores gráfico, flexíveis, expansível e portável para aplicações de realidade virtual," Master's thesis, Departamento de Computação, Universidade Federal do Ceará, Março 2005.

[10] D. Johnston, "3d game engines as a new reality," in *Proceedings of the 4th Annual CM316 Conference on Multimedia Systems*. Southampton University, UK: http://mms.ecs.soton.ac.uk/mms2004, November 2004, pp. 1–8.

[11] R. Nakamura, J. Bernardes, and R. Tori, "enJine: Architecture and application of an open-source didactic game engine," in *Digital Proceedings of the V Brazilian Symposium on Computer Games and Digital Entertainment*, B. Feijó, A. Neves, E. Clua, L. Freire, G. Ramalho, and M. Walter, Eds. Last access: 22/11/2007: http://www.cin.ufpe.br/~sbgames/proceedings/files/enjine.pdf, November 2006, pp. 1–7.

[12] D. Shreiner, M. Woo, J. Neider, T. Davis, and A.R.B. OpenGL, *OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL(R), Version 2*, 5th ed. Addison-Wesley Professional, August 2005.

[13] R. Darken, P. McDowell, and E. Johnson, "The delta3d open source game engine," *IEEE Comput. Graph. Appl.*, vol. 25, no. 3, pp. 10–12, 2005.

[14] N. Gebhardt, "Irrlicht game engine," http://irrlicht.sourceforge.net/, 2008.

[15] J. Jacobson and Z. Hwang, "Unreal tournament for immersive interactive theater," *Commun. ACM*, vol. 45, no. 1, pp. 39–42, 2002.

[16] G. Junker, *Pro OGRE 3D Programming (Pro)*. Berkely, CA, USA: Apress, 2006.

[17] G. van den Bergen, *Collision Detection in Interactive 3D Environments*, ser. The Morgan Kaufmann Series in Interactive 3D Technology. Morgan Kaufmann, October 2003.

[18] J. C. Oliveira and N. D. Georganas, "Velvet: an adaptive hybrid architecture for very large virtual environments," *Presence: Teleoperators and Virtual Environments*, vol. 12, no. 6, pp. 555–580, November 2003.