# AynOmel 3D: A Pattern-Based Game Framework

Olia Michou, Maria Vamvaka, Apostolos Ampatzoglou

Department of Applied Informatics,
University of Macedonia, Thessaloniki, Greece
E-mail: ampatzoglou@doai.uom.gr

**Abstract**

The paper presents the structure of a game framework (AynOmel 3D). More specifically, the paper focuses on three major points, on the game framework modules, on the accompanying software tools, and on how design patterns proved beneficial in its design. The main idea behind the software architecture is to design a game framework completely based on patterns and investigate the potential benefits deriving from it. The effect of design patterns on the software is examined in respect to maintenance and flexibility.

**Keywords**
Design patterns, game engines, game frameworks, game development

## *1. Introduction*

Nowadays it is a common sense that games play a very important role in modern societies concerning economy, lifestyle and quite recently scientific research. The game industry is now considered to be one of the most powerful in the business spectrum [7, 13]. In [13] it is mentioned that worldwide game industry reached $33.5 billion in size in 2005, with expected growth to $58.4 billion by 2007. These numbers combined with the fact that computer games currently rivals television and films in both market size and cultural impact [7] can justify why game development is considered a placeholder in current lifestyle. Additionally, it is estimated that 90% of U.S. households have rented or purchased at least one video game, and that young people of the country spend an average of 20 minutes per day playing video games. Even though game development industry is considered extremely powerful worldwide, in Greece there are extremely few paradigms of commercial games. Most of them include minor game logic (plot) and graphical representation and in that sense cannot be competent to foreign products.

The game development industry growth has encouraged researchers to accept game development as a serious scientific field rather than a “not academic subject”

[3]. Recently, major journals and computer magazines host papers concerning game development. Additionally, some conferences include panels dealing with game programming methodologies or even refer exclusively to game development. The scope of these papers varies from the academic way of teaching game programming to analyzing complicated techniques concerning the actual methodologies of game development. In order to strengthen related research most universities have included game programming courses in BSc and MSc studies, and in some cases have even established MSc diplomas concerning game development. On the other hand, in Greece even though there are courses that describe the key functionality of a game (like programming, artificial intelligence, graphics, databases etc), there is no specific course dealing with assembling a game from the aforementioned parts.

Nowadays the game production timeline needs to be shrunk in order to be able to conform to modern market demands, in the sense that games are by nature extremely evolving software, since newer versions of a single game need to arrive in market in an extremely short interval. In order to achieve this expectation game industry employs game engines and game frameworks [4, 11]. Therefore, game production is divided into two concrete stages, the development of the engine/framework and that of the actual game using the aforementioned mechanisms (engine, framework). Both game engines and frameworks would prove beneficial only if they are “well-structured” so as to be easily maintained and appropriate for different game genres.

This paper introduces a game engine and a framework that is designed employing software engineering techniques that aim to provide flexibility and ease future adaption. The software under study (AynOmel 3D), helps developers of 3D virtual worlds (including game programmers) create scenes using powerful 3D packages (like 3D studio max) through user-friendly GUI’s (software tools). Additionally, the source code of the package is available to the developer through libraries that can be enhanced with specific requirements. In section 2, a description of design patterns will be presented accompanied with a literature review on how pattern can prove beneficial in games. Section 3, will analyze the structure of the system. Finally, conclusion and future work will be presented.

## *2. Design Patterns*

The initial idea of patterns as problem-solution pairs was introduced in the field of architecture, where it had been examined whether quality is an objective attribute [1]. With the term design patterns in software engineering one refers to identified solutions to popular programming problems. The use of such patterns effects the architecture of the software and in most cases reduces its complexity and increases its flexibility and reusability. Consequently, pattern application can prove beneficial, taking into account factors like maintenance time and cost [8].

Using design patterns helps a programmer avoid making common errors, solve problems and structure his code. It is supported in [12] that patterns are about sharing experience, so that everyone can learn from others and that instead of inventing new ideas; patterns capture what has been successful in the past. Additionally, in [11] it is argued that experienced developers resolve challenges by applying patterns, whether they are aware of it or not. The primary contribution of the patterns community is to capture, document, and refine these patterns so everyone can benefit.

Furthermore, design patterns are proven to support the design, analysis and comparison of games [5] as an alternative to the need of developing a common language, concepts and terminology for games. Since games are commonly big projects that require collaboration among staff with different expertises, patterns should be viewed as a tool to overcome communication differences in an effective and efficient way.

Additionally, in [2] the authors have examined the way object-oriented design patterns can affect the structure and the maintainability of a game by analyzing existing systems. The results suggested that patterns can reduce complexity and coupling of a game, increase cohesion of the code but on the other hand increases the project's size concerning lines of code. In addition to that in [10] there was an attempt of creating a game that was based on patterns. The results suggested that design patterns should be considered an efficient way of properly achieve abstractions and decoupling in games.

## *3. AynOmel 3D Structure*

In this section, an extended description of the proposed structure of a game engine and framework will be presented. More specifically, in section 3.1 there is a description of the software tools. Section 3.2 deals with the modules that AynOmel 3D is decomposed to. Finally, section 3.3 concerns how object-oriented design patterns affected the design of the software.

### *3.1 AynOmel 3D Tools*

AynOmel 3D is a game engine that is accompanied with several tools that help game programmers create a game without extreme programming skills. Although the programmer does not have to be an expert in programming he must have a quite good understanding of 3D graphics, 3D studio max, Photoshop, artificial intelligence and game design. The tools of AynOmel 3D are:

- Actor Creator
- Actor Animation Creator

- Behaviour Creator
- Scene Creator
- Game Constructor

*Actor Creator* imports a moving model from package, attaches texture or material to model, provide photorealistic rendering of actor with default light and shading and saves actor information to a file for later use. *Actor Animation Creator* imports a skeletal model from specific moving patterns, specifies animation frames, creates animation from keyboard, saves information to a file, loads an actor and previews animation.

*Behaviour Creator* loads an actor from a file, attaches a list of animations to the actor and creates an artificial intelligence algorithm that handles the behaviour of the actor during each phase of the game. *Scene Creator* imports static moving actors from files, places animated actors into the scene, set scene start and exit points, sets lights for the scene and saves the scene to a file. *Game Constructor* sorts the scenes order, writes an event handler for every scene and finally, writes the information into a file.

The *Game Engine* reads next scene from a file, places the main actor at the start point and positions computer characters in the scene. Additionally, connects the scene event-handler to the openGL event-handler. On every move of an object, it runs the collision detection algorithm, and on every collision, it runs the physics engine algorithms. When the player reaches the scene ending, the engine loads and handles the next game scene.

### *3.2 Modules*

Designing and programming large-scale software is a very complicated task that requires many work-hours. Consequently, in most cases software is divided logically into subprograms that are autonomously designed, coded and tested by separate programmers' groups. These subprograms are called modules. Decomposing software to modules is an important decision that plays a vital role in the architecture and further designing of the program.

According to [9], the game engine consists of several interconnected components. All these components (modules) are part of a looping event handler. The writers divide the game engine to several modules as they are presented in Figure 1, and then analyze each part's functionality.

In the figure below, the black arrows represent input data flow and the gray arrows represent output data flow. The final output to users can be both sound and image that are produced by the audio, the graphics, the 2D/3D engine and the music/sound modules. The input received from hardware (keyboard, mouse etc) goes

to the controls module, the user-interface, the network and the client-control modules. The main core of the game engine consists of the four modules seen on the right of the figure, the main loop module, the timer, the event-handler module and the dynamic game-data. Those modules, schedule the transformation of the input signals to output, using the simulation component and the static game-data module.

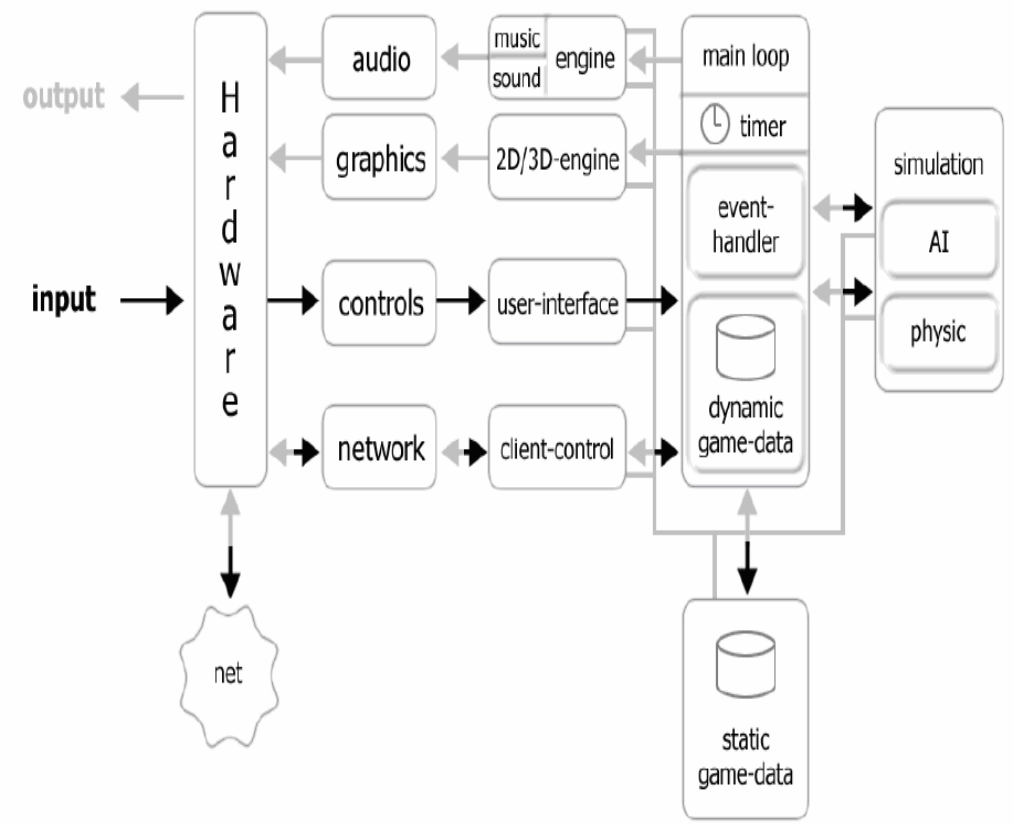


**Figure 1: The game engine modules [9]**

In AynOmel 3D the approach of module decomposition was based on the actual development of the C++ libraries. In that sense every library represents a logical module of the program. The logical parts of AynOmel 3D are presented in Figure 2.

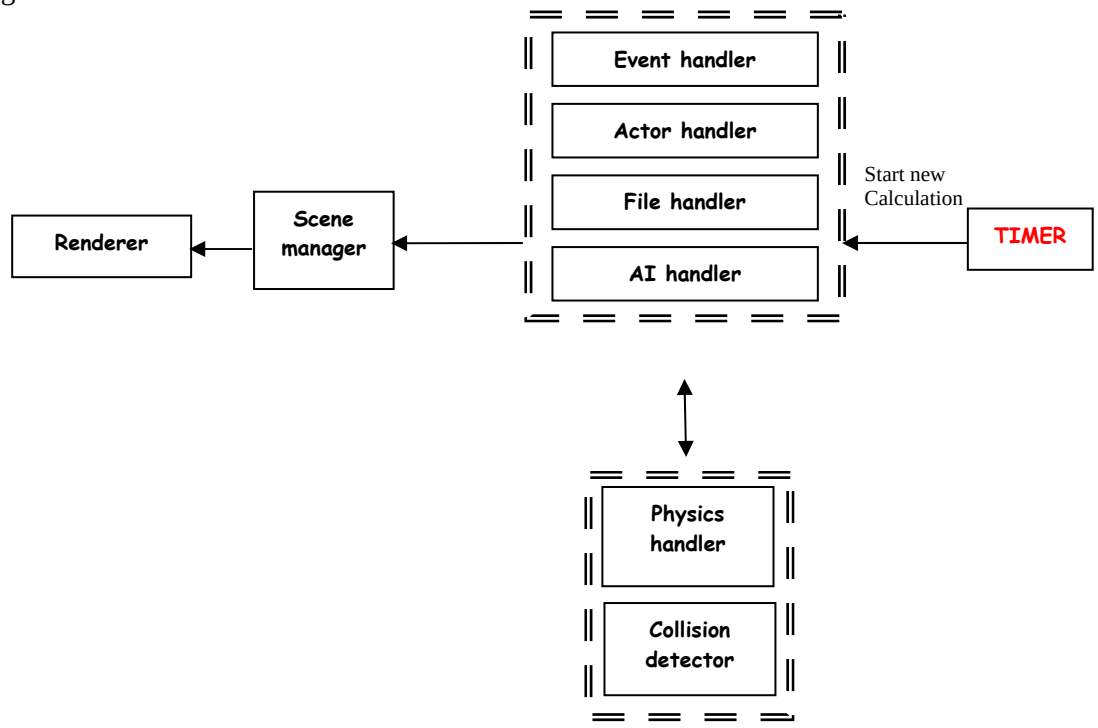


**Figure 2: AynOmel 3D modules**

The mechanism of the game engine is based on a timer that initiates the drawing of the scene, the behaviour selection for computer and player characters. For every clock interval, the engine checks if the player has triggered an event from an input device (mouse, keyboard etc). If he has, he performs the actions mentioned in the event-handler module. Next, the engine simulates all computer controlled actors' behaviour through the AI-handler module. In case that there is any movement of an actor (computer or user) the collision detection module checks whether there are any colliding objects in the scene and if any exists, it runs the appropriate physics-handler module's algorithm. The scene-handler module is responsible for collecting the above data and passing all the information needed (actor appearence, animation, scene lights etc) for visualization to the renderer module. The renderer is a set of functions that generates the final optical output of the scene. The renderer performs operations such as hidden-surface removal, clipping, camera position, light calculations etc.

### *3.3 Patterns Used*

The main idea of the AynOmel 3D architecture was to create software that would be based on the use of design patterns. In order to provide support to the aforementioned claim two well-known metrics have been employed [Chidamber]. More specifically, it has been calculated how many classes (NOC) of the system participate in patterns and how many lines of code (LOC) compose those classes. The results suggest that the project is 81,1% pattern-based concerning NOC and 87,9% LOC as shown in Table1. In the table the number out of the parenthesis represent absolute values, while the values in the parenthesis represent the percentage of pattern participating lines of code and pattern participating classes.

**Table 1: Metrics on pattern-based**

| | **actor handler** | **file handler** | **renderer** | **TOTAL** |
|---|---|---|---|---|
| **NOC** | 15 (73,3%) | 8 (75,0%) | 14 (92,8%) | 37 (87,9%) |
| **LOC** | 1075 (97,5%) | 680 (68,7%) | 328 (96,0%) | 2083 (81,1%) |

The project has not been yet completed and until now has been created three libraries. In these libraries three patterns have been applied. More specifically, it is observed that the project has employed one instance of the State pattern, two instances of the Strategy pattern and two instances of the Bridge pattern. For every pattern identified in the project, there will be an analysis of the utility of one instance.

The State pattern has been used in implementing the camera positioning in openGL. The camera in most 3D environments can be either Free or Target. The free camera is defined only by one reference point (camera position) since the direction of the camera is fixed. On the other hand, the target camera is defined by two points (camera position, camera target). The application of State pattern is presented in Figure 3.

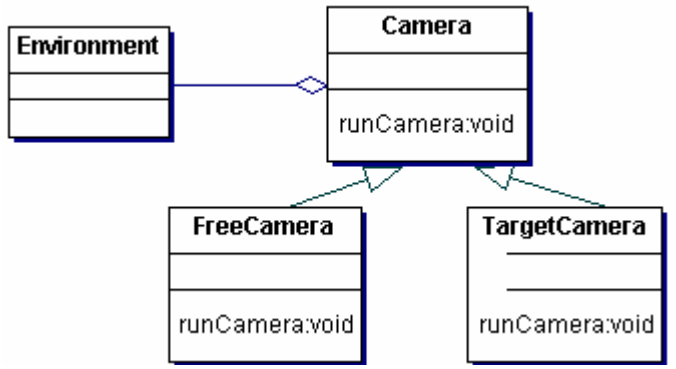


**Figure 3: State pattern - Camera**

The key idea in this pattern is to introduce an abstract class to represent the different states of an object. It is used either when an object's behaviour is dependant on its state or it must change its behaviour at run-time depending on that state or when operations include large, multipart conditional statements that are as well dependand on the object's state. Including the pattern in the code, the programmer states specific behaviour and partitions behaviour for different states are localized. Additionally, state transitions are made explicated, state objects can be shared and finally there is elimination of the conditional statements as it is shown below:

**Without state pattern:**

```
void Environment::runCamera() {

      if (cameraType=='TARGET') {
            glutLookAt(…); // Specify target camera arguments
      } else if (cameraType=='FREE'){
            glutLookAt(…); // Specify free camera arguments
      } else {
            printf("Not specified camera type");
      }

}
```

**With state pattern:**

```
void Environment::runCamera() {

    cam_->runCamera();

}
```

**Figure 4: State pattern – source code**

The extensibility of the pattern can be proved as a new camera can be added without major changes in the code of the Environment class. More specifically, if a new camera (CameraA) is added, the only change in the Environment class will appear in the constructor of the class where the programmer will have the option to initialize an instance of a camera of type CameraA.

Additionally, the Strategy pattern has been employed in the implementation of the functionality of lighting. The openGL renderer provides the programmer the option to choose between executing all lighting reflection calculations of the scene and executing only the necessary ones. In that sense, two algorithms are available for enlightening the scene. Furthermore, the system enables the developer to select between three distinct well-known lighting types (Spot, Directional and Positional light). The above description is depicted in Figure 5.

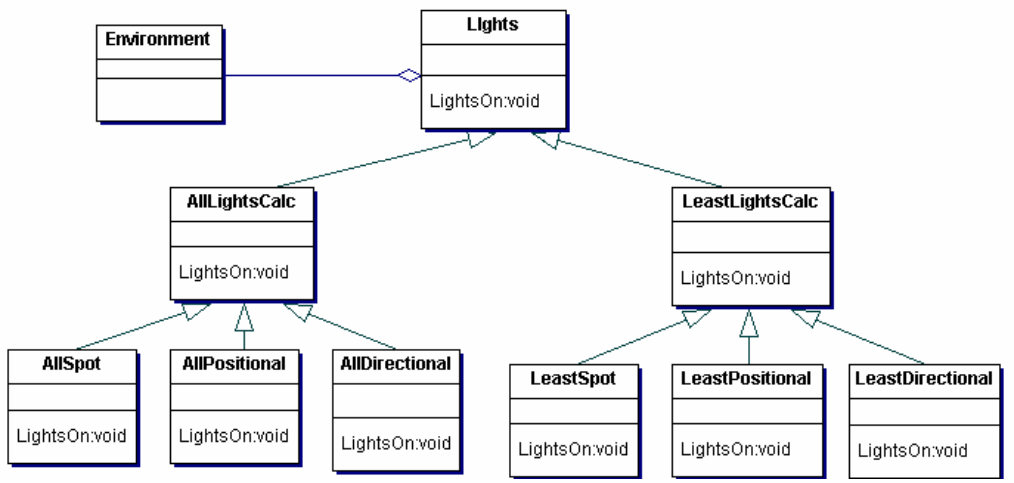


**Figure 5: Strategy pattern - Lighting**

The Strategy pattern is used to avoid problems of “needless” repetition of code between classes that provide almost the same functionality, with minor changes into one or more functions (algorithms). It is actually based not only on the inheritance, but on the composition of the objects as well. The pattern is quite similar to the State mentioned above as it can be easily observed. It results in defining families of related algorithms, giving a choice of implementation and providing an alternative to subclassing. One last benefit is the elimination of conditional statements by encapsulating the behaviour, when it varies, in separate Strategy classes. If the pattern had not been applied as shown in Figure 5, there would have been the need for implementing a conditional statement similar to the one referenced in Figure 4.

Finally, the Bridge pattern has been used so as to implement the mechanism of designing the objects and defining their appearance (texture, material). Most 3D packages provide the potential of colouring each vertex of an object either by applying to it a uniform color with predefined characteristics (material) or matching each vertex with a color extracted from a bitmap file (texture). Additionally, every object of a scene can either be a 3D Mesh (3D studio Max file, Maya file etc) or a primitive (sphere, cube). The Bridge pattern was applied in order to describe the above structure as shown in Figure 6.

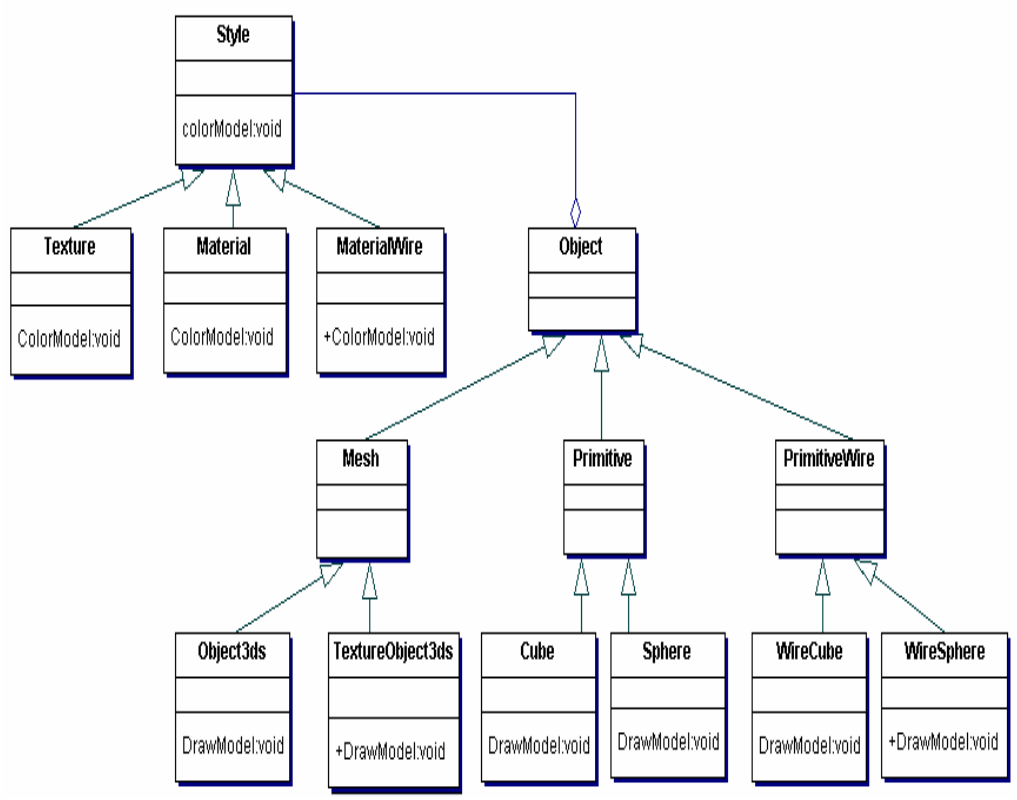


**Figure 6: Bridge pattern – 3D modeling**

The Bridge pattern is in contrast to the extensive use of inheritance, as this approach is not flexible enough. Consequently, it is employed in case you want to hide the implementation of an abstraction from clients, when changes should have no impact on clients or finally just to avoid a permanent binding between an abstraction and its implementation. So, when using the Bridge pattern, implementation and interface are decoupled, the extensibility is improved and implementation details are hidden from clients.

If the pattern has not been applied as described in Figure 6 the mechanism would have been implemented as in Figure 7, which is less extendable. More specifically, if later a demand for a loader of different 3D package appears (e.g. Maya loader for openGL) the Style class and the client that uses the instance of the Object class will remain the same. On the contrary, since the classes of Figure 7 are tightly coupled, the addition of that new class would further increase the already high complexity of the system.

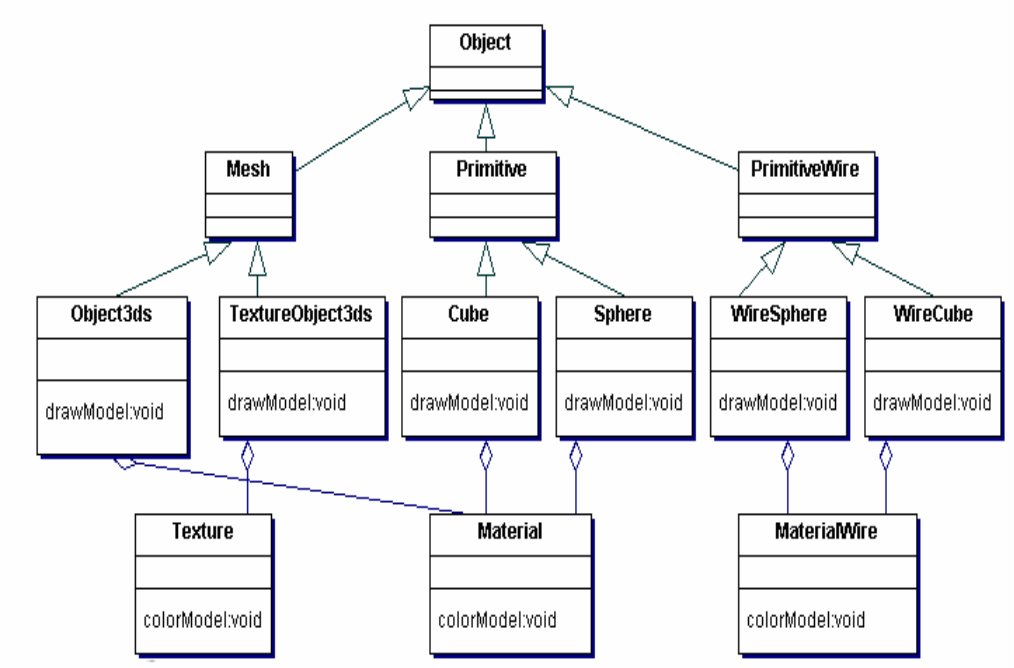


**Figure 7: 3D modeling no pattern implementation**

## *4. Future Work*

In order to investigate the complete spectrum of the effectiveness of patterns in game development, research need to be done on extracting instances of patterns in real games. Furthermore, it might prove useful to identify patterns that are more suitable for game design than others. Moreover, research could be performed in quantitatively evaluating the average benefits (employing well-known metrics) of every pattern in several games.

Additionally, it would be of great interest to further examine whether in large-scale industry games modules tend to be similar or vary according to the game genre. Finally, it is planned to complete the AynOmel 3D framework and examine its maintainability.

## *5. Conclusions*

This paper aimed at investigating the structure of a game engine. The main subtargets of the research was to introduce a module decomposition for the project, to introduce some basic tools that could accompany a game engine and finally examine the use of object-oriented design paterns in the developemnt of a game framework.

The paper approached the subject of design pattern application from two different perspectives, an empirical where a game framework was implementing using design patterns and a second one, more theoretical, where a qualitative analysis took place investigating how design patterns effect the maintainability of the aforementioned system. The results suggested that patterns are benefitial in game developement in the sense that "well-structured" game engines and frameworks can help developers create games with less cost and effort.

## *6. References*

[1] Alexander C., Ishikawa S., Silverstein M.(1997), *A Pattern Language – Town, Buildings, Construction,* Oxford University Press, New York

[2] Ampatzoglou A., Chatzigeorgiou A., *Evaluation of object-oriented design patterns in game development*, Information and Software Technology, to be published

[3] Argent L., Depper B, Fajardo R., Gjertson S. (2006) *Building a game development program*, IEEE Computer, Vol. 39, No. 6, pp. 52-60

[4] Bishop L., Eberly D., Whitted T., Finch M., Santz M. (1998), *Designing a PC game engine*, IEEE Computer Graphics and Applications, 46-53

[5] Bjork S., Lundgren S., Holopainen J. (2003), *Game Design Patterns*, Proceedings of Digital Games Research Conference 2003, Utrecht, The Netherlands

[6] Chidamber S.R., Kemerer C.F. (1994), *A Metrics Suite for Object Oriented Design*, IEEE Trans. Software Eng., vol. 20, no. 6, pp. 476-493

[7] Fullerton T. (2006), *Play-centric Games Education*, IEEE Computer, vol. 39, No. 6, pp. 36-42

[8] Gamma E., Helms R., Johnson R., Vlissides J. (1995), *Design Patterns: Elements of reusable Object-Oriented software*, Addison-Wesley Professional, Reading, MA

[9] Masuch M., Rueger M. (2005), *Challenges in Collaboration Game Design Developing Learning Environments for Creating Games*, Proceedings

of the 3rd International Conference on Creating, Connecting and Collaborating through Computing (C5'05), pp 67-74, Kyoto, Japan

[10] Nguyen D., Wong S.B. (2002), *Design patterns for games,* Proceedings of the 33rd SIGCSE Technical Symposium

[11] Rucker R. (2003), *Software engineering and computer games*, Addison Wesley, Essex, United Kingdom

[12] Schmidt D.C. (1995), *Experience Using Design Patterns to Develop Reusable Objects-Oriented Communication Software* ACM Special Issue on Object Oriented Experiences, vol 38. No 10

[13] Zyda M.s (2006), *Educating the Next Generation of Game Developers*, IEEE Computer, Vol. 39, No. 6,pp. 30-34