A Fast and Safe Framework to Prototyping Physical Worlds Using XNA and GPU

60360

Abstract

Computer games are becoming an important resource for many educational purposes. The increase of computational power and new paradigms of engine architectures not only reinforce this, but also make available new approaches to real time applications. It is also well known that interactive environments are strong mechanisms for a more efficient content comprehension and assimilation. One of these fields is physics, where experimentations and real time interactions are desirable. Although there are many software and computational environments focused on physics learning, few programming tools offer the facility for programming applications and experiments. This work presents a novel framework for physics programming that combines the facilities available in Microsoft XNA framework with C# and GPU acceleration. The present framework implements a complete set of functionalities for dynamic simulation of rigid bodies, encapsulating them completely with XNA.

Keywords:: Education for physics, XNA, CUDA, GPU aceleration

Author's Contact:

60360

1 Introduction

A considerable amount of the power increase in computer technologies is pushed by the video-games and interactive entertainment industry, which aims at more and more realistic virtual experiences for the players. Due this, games and rich environment simulations are applications that demand complex tasks to be developed. In fact, they are composed of various computationally intensive problems, such as artificial intelligence, physics simulation, scene database query, networking, audio, video, I/O, etc.

Beyond entertainment, it is well known that video-games can have many applications, such as education [Dede 1995; Papastergiou 2009; Gee 2004], training and health. An example of this is the American's Army game, created by the United States Army with the purpose of providing an experience in the US Army tasks, focused for civilians' usage [Strong 2002].

Computer games and simulation is proved to be extremely efficiently at all stages of education not only for reinforcing knowledge but also for archieving high-level cognitive, affective and psychomotor objectives [Papastergiou 2009]. With this in mind, Microsoft Robotics [Microsoft 2008], a commercial platform for developing robotic applications, was developed, targeting students and researches on robotics, given them a framework that is capable of prototype a simulation easily in a few period of time [Workman and Elzer 2009].

The increase in computational power is making it possible to apply more efficient real time physics calculations. Simulations that were impossible to be executed in real time are now becoming possible [Yeh et al. 2006] and a large number of libraries and frameworks for this purpose are becoming available. Beyond the direct application on game development, these tools are also becoming an appropriate mechanism for physics teaching and research by educators [Mayo 2007; Squire et al. 2004]. At this moment, most of these libraries can simulate rigid bodies, soft bodies and fluids, which act according to parameters that are set by the user in the simulation. Unfortunately, the usage of these libraries is not a trivial task and sometimes requires a lot of effort.

While some frameworks are purely code and allow programming almost any situation, some of these physics simulation frameworks are implemented using a visual interface, allowing an interactive feed-back of the application [Intel 2000]. The first situation requires highly specialized programming skill and the second brings many restrictions for the experiments and educational process. An alternative can be the direct usage of a commercial game [Price 2008; Price May 2008] to build an educational experiment, but the constraints are stronger than using the visual interface of the engine itself, as most commercial games were not mainly designed to focus on education. In this case, making a simple physics simulation could be hard or even impossible as most of the physical related attributes are hidden from the user due they irrelevance in the game.

Another option can be the use of a commercial game engine that hides most of the low level programming from the user like is the case of [Unity 2004]. In this case, experiments could be made easily from researches at cost of numerical precision, as most game engines needs to deal with many subsystems like sound, artificial intelligence and networking which are hardly ever necessary for experiments research.

For complex physics simulation, integration with some API, such as OpenGL, DirectX, and CUDA, is required in order to achieve satisfactory and interactive results. However, this approach would increase even more the complexity for the educators, due to tasks that require expertise in systems architecture and computer programming. In this case, an educator that has a good knowledge in physics would also need skills related to computer graphics and GPU programming, loosing focus on the main educational purpose.

Many of these physics libraries are based on C/C++, since they require a lot of optimizations and are very computationally intensive. These classes of languages are called unmanaged languages, and are much harder for a simple user. They present complicated housekeeping details and development difficulties that are not related to the intended application, which are time consuming and error prone tasks. Another issue related to this kind of languages is its compilation time that can be too long depending of the application's complexity and size. For these reasons, the use of script languages like Lua [Puc-Rio 1993] is an obligation to avoid application recompilation and increase productivity.

In this paper we present a framework that gives educational software developers and educators with minima experience with programming an appropriate environment that facilitates the production of educational content, in particular on physics. The tools developed by this framework can also be used by research groups. This solutions intends to be a fast, easy, and safe way for exploring physics using computer graphics and interactivity. As it is an extension of the Microsoft XNA [Microsoft 2004] framework, it inherits its facilities. The proposed framework was developed with the following requirements:

- Memory safety, avoiding the user to worry about issues that are not related to the application itself;
- Fast interactivity and speed for showing the results to the user when executing the application;
- A fast physics end very efficient simulation that runs on GPU instead of CPU;
- Easy to use, even for non-programmers;
- Expansibility, allowing the development of other kinds of applications in the future;
- Modularity;
- Low coupling with other modules inside XNA framework.

By adopting XNA as the base for the application, an efficient memory safety, ease and minimal compilation time is achieved, since these are benefits allowed by the C# language. We present a discussion about the efficiency related to unmanaged languages, like C++.

Choosing the Microsoft XNA and C# language was important for handling different kinds of input devices, graphics, sound, and network tasks easily. Due to its rich documentation and good software architecture, our proposed tool may be easily extended for more complex environments related to different specific physical educational processes. We built the framework completely based on Nvidia CUDA [NVidia 2004]. CUDA is a computer architecture that uses a C like programming language. Using this, the GPU can be fully explored as a parallel processor, allowing a very fast and efficient approach.

The rest of the paper is organized as follows. Section 2 discusses related work, section 3 presents the framework architecture and its core components and section 4 explains the usage of these components. Section 5 presents the results and section 6 brings an example of developing a physical world simulation using the framework. Finally, section 7 concludes the paper.

2 Related Work

Many efforts had been made in order to facilitate the development of real time physics simulation by end users. One of the main objectives consists on hiding low level aspects to the developer, like file manipulation, direct low level programming, and graphics integration, among others. Following these criteria, Kačić-Alesić [Kačić-Alesić et al. 2003] proposed a plug-in that could do rigid and deformable body dynamics, particle dynamics, and hair and cloth simulation at a basic level. Although this work presents an elegant architecture, it still needs some kind of integration with the application, assuming that the user has a minimum knowledge of low level computer programming.

Shapiro [Shapiro et al. 2007] developed a toolkit that enables the users to create dynamic controllers for articulated characters under physics simulation, mixing pose animations with physically based motion. To prevent the user from using low level languages, this work provides script languages for interaction with the simulation. Although it is a good solution for end users and educational purposes, it is restricted only to articulated characters, not allowing more general physics simulation to be done.

In [Popović et al. 2000], the authors propose a simulator where the user can give the initial position, velocity, and final position of the simulated object. The simulator automatically generates physics based motion based on the input constraints. This simulator is easy to be used by end users but unfortunately is not capable to make different physics simulations.

PhysX [NVidia 2008] is a powerful physics engine that can be used with GPU acceleration, but requires knowledge of low level computer programming and requires graphics integration knowledge from the user for graphical feed-back. In some cases, a script language is also needed to avoid the simulation recompilation in case of parameters changing in the application.

In [Joselli et al. 2008], the author proposed an architecture of physics simulation engine with automatic process distribution between GPU and CPU with physics simulation done in CUDA. This achieves a better performance but still requires knowledge of low level programming by the user to be used.

VPython [Scherer 2000] is a framework that uses the Python programming language for creating 3D simulations and is used by researches in various scientific fields [Mayo 2007] including Physics research. Although easy to use and fast interactively to show the simulation results, a simulation that requires a high physics processing and/or deal with a large amount of data maybe not be able to run in real time due the fact that it uses the CPU for all the simulation processing.

Our framework has a simulation that runs on GPU instead of CPU to achieve better performance but do not require any type of low level computer programming to use. It provides the user a fast, ease, and safe environment for researches using most of the technology that is used in cutting edge games. At the same time, it is very easy to expand due its modularity, allowing the addition of new functionality to supply the user necessity without much effort.

3 The Framework Architecture

The proposed architecture was built with easy use and extensibility in mind, adopting a plug-in strategy [Birsan 2005]. In this case, anyone who needs a functionality that is not present in the framework can add it without the necessity of modifying or even recompiling the framework, by only attaching the new developed module to it. As its main target is educators, it needs to be easy to work with and understand as most of the users do not have background in computer programming. Thus, the framework needs to provide mechanisms that involve as few programming as possible, letting the educators focus on the physics simulations as much as possible. In Figure 1 a simple class diagram of the entire framework is shown. As it can be seen, the core subsystems that compose it are the services, the game screens, and the game object subsystem. Each of them is presented in more detail below.

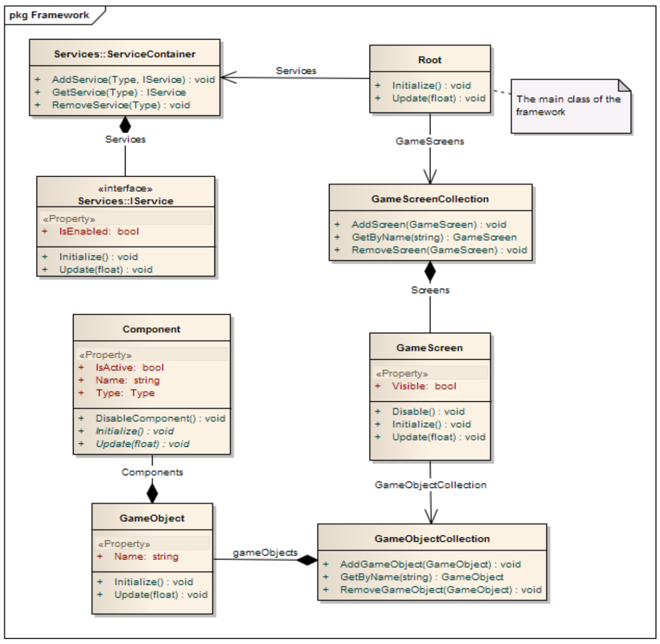

Figure 1: *The framework architecture*

3.1 Services

Services in the framework are modules that are available to be used entirely in all stages of the simulation. The basic concept is that services are unique and their tasks are well defined. Tasks that are strong candidates to be implemented as services are rendering, input event handling, and sound management. With this basic architecture of services, most of the references that a class needs to maintain are avoided, like a model that needs to use a render manager to be displayed.

Due to its nature, the framework only allows the instantiation of one service per type. In this case, if a service that was already previously added to the framework is added again, the framework will throw an exception.

As services are unique in the framework, problems of initialization can occur. One example of such problems is related to the physics service. Sometimes this class will need a service of rendering for debug purpose and probably will store a reference of this service inside of it. As services are created one after another, the physics service would not know if the rendering service is available at the moment of its initialization, requiring, at least, one boolean check to verify if the required classes have already been created, as it is possible to see in Listing 1. To avoid this problem, the services are initialized in two steps by the framework. In the first step, all services are created by the framework in a batch. In the second step, the framework calls the *Initialize* method of all services. At this moment, each service can initialize itself and asks for its services dependency in a safe way.

```
void Update()
{
  if (!renderingInitialized &&
    Framework.Instance.Services.GetService<
        Rendering>() != null)
    renderingInitialized = true;

  if (renderingInitialized)
    //render physics object
}
```

Listing 1: *Check for dependency service*

3.2 Game Screen

Game screens are used in the framework as a way to organize and manage a collection of scene elements. It can be thought as a collection of objects that are updated in batch by the engine, with some methods and properties to facilitate the organization and management of this collection in the framework.

Most physics libraries use some kind of container to organize all objects in the simulation that can interact with each other. In this case, objects that reside in a space have no influence over objects that resides in another space, making possible more than one simulation running in parallel. Using game screens it is possible to simulate the same behavior, as the framework runs them in parallel and objects that are in a game screen does not have influence over objects that are in another, although the same object can be added to more than one game screen.

A game screen is an object derived from abstract class *GameScreen*. In order to facilitate the user, the framework already provides the derived class *PhysicsSimulationScreen*, which initializes and starts all required services necessary to work with physics simulation. If desired, a new *GameScreen* based class could be created by the user with minimal effort.

3.3 Game Objects

A game object is the basic and only element that can be simulated in the framework. It is based on the concept of Game Object Component System [Bilas 2002]. In this system, the author proposed the use of aggregation instead of inheritance to define a new functionality in a game object. Thus, as an example, game objects that need to be rendered must implement a component that handles this task. Using this approach, some problems related to inheritance are avoided and new game objects can be easily implemented by the user, who only needs to define new components to handle the required specific task.

In order to be simulated, game objects need to be attached to a *GameScreen* object, which is responsible to update each game object in the right time during the simulation. In advance, to better manage game objects, they need to be unique in the game screen. To achieve this, each game object has an identifier, which is set by the user during its instantiation. This identifier is used to avoid the same game object to be added more than once in the same game screen, although it can be added in more than one game screen. A game object is an instantiation of the class *GameObject*.

Game objects, to be properly used, are composed of a collection of components, as discussed before. Components in game objects may need to use other components in order to work appropriately. This kind of dependency is solved by using a technique called dependency injection [Passos et al. 2008]. By using this strategy, components are "injected" automatically in compilation time into components that have this kind of dependency. This guarantees that all dependencies are satisfied inside a game object before the simulation starts.

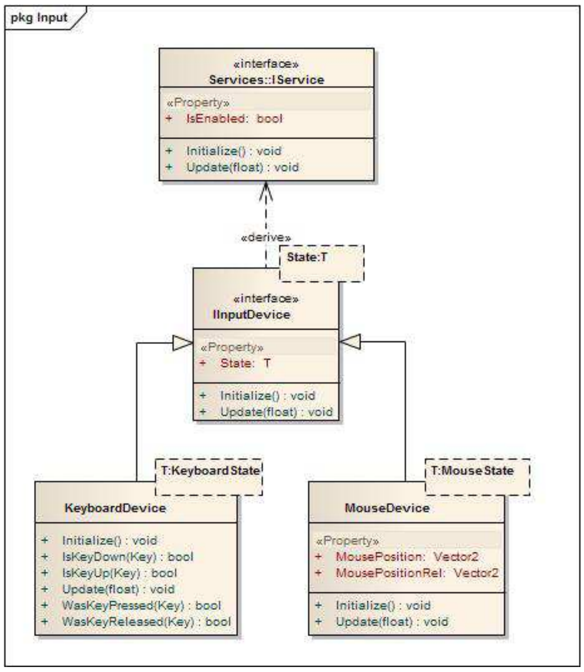

Figure 3: *The input device service*

Game objects and components that handle the physics simulation are implemented in the core of the framework. As an example, Listing 2 shows how a game object that represents a sphere-shaped rigid body can be used in the framework.

```
GameObject go = new GameObject("ball");
RigidBodySimulator simbody = new
    RigidBodySimulator();
simbody.AddShape(new Sphere(1,1,1));
go.Components.Add(simbody);
```

Listing 2: *Game object to simulate physics*

4 Built-in Services

As discussed before, services are an important mechanism to enable extensibility of the framework. Although they can be created by the user, the framework provides some basic services to allow it to run. All the services listed below are necessary for allowing an educator to define its physics simulation.

4.1 Input Service

Input services are responsible to deal with input devices. Currently, the framework has only support to deal with mouse and keyboard devices but others can be easily added by the user. In Figure 3, the diagram shows how it is implemented by the framework. As it can be seen, each device needs to implement the *IInputDevice* interface, which has a *State* member for querying about the state of the specified device as its own methods.

4.2 Graphics Service

Graphics tasks need to deal with different kinds of models that can be rendered differently from other kinds of models. One example of this is the shader effect that can vary from one model to another, requiring different parameters so that the model is properly rendered. As a consequence, each type of model could require the creation of a different class of render component in order to render the models of that type. Even more, *GameObject* is a sealed class, so it is not possible to simply extend this class to deal with custom parameters in the models.

In Figure 2, a sequence diagram is shown to better elucidate how a model is rendered in the framework. The adopted approach to work

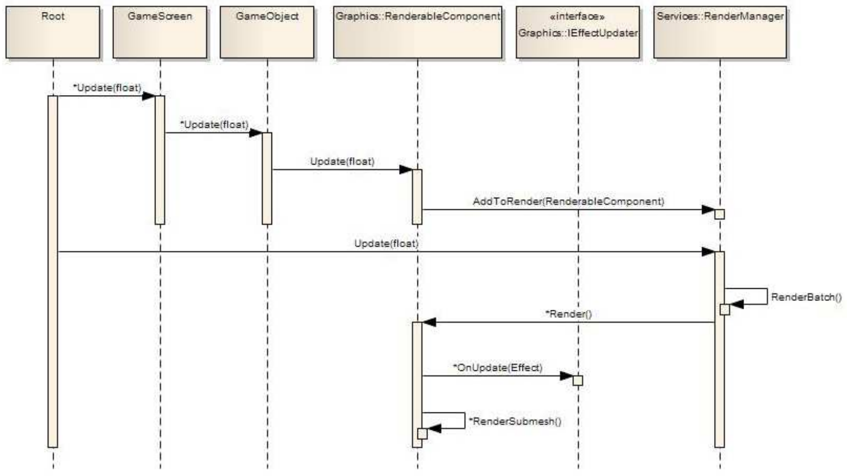

Figure 2: *The rendering sequence diagram*

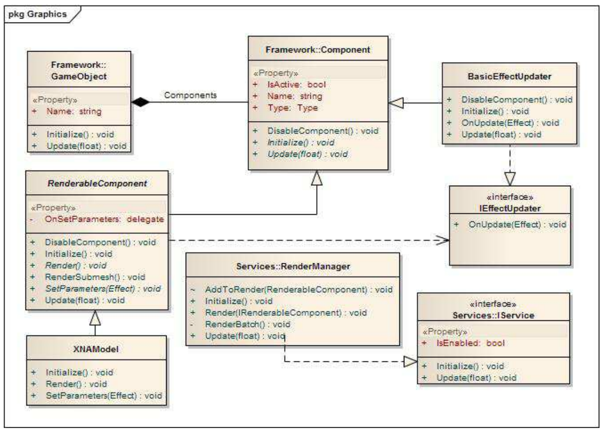

Figure 4: *The graphics service*

with graphics is shown in Figure 4. As an example, the *XNAModel* class is shown in the diagram. It is a wrapper for the built-in *Model* class from XNA, which is responsible for rendering geometry. In this case, the interface *IRenderable* needs to be realized in order to use the built-in *Model* class for rendering. This interface realization is done by the *XNAModel* class, which encapsulates the built-in *Model* class. With this approach, a *Content Pipeline* extension, which processes data types defined by users, can be easily created by using a custom model class that realizes *IRenderable* interface and maybe inherits from the built-in *Model* class. The use of this wrapper was necessary because this class does not implement the *IRenderable* interface, which is required to render a model.

The next step is the setting of custom shader parameters. In order to do that, the *RenderableComponent* also needs a class that implements the *IEffectUpdater* interface. In the diagram, *BasicEffectUpdater* is a class that implements this interface. It also must inherits from *Component* class because *RenderableComponent* will find it in its parents' component collection, due to the dependency injection method.

With this restriction, a class that implements *IEffectUpdater* interface also needs to inherit from *Component* class. As the components have a parent class, the *IEffectUpdater* will have access to all data from a game object, allowing custom parameters to be setup in a shader.

Finally, when it is time to render, the framework will invoke an internal method of the *RenderManager* service which will render all models that were added to its collection, making some optimization in this process.

4.3 Physics Service

This service is one of the most important in the framework, as it is responsible for the physical world simulation. This physics service was made using CUDA, which yields high performance physical world simulation in the framework, as physics tasks can be highly parallelized.

Following the overall framework architecture, for an object to be simulated using physics laws, a game object needs to be created and it must have a physics component, as can be shown in Figure 6. With this architecture, it is easy for the user to create physics objects in the framework, as the only requirement is setting a physic component to the game object to be simulated. To better elucidate the overall process inside the framework, the physics sequence diagram is shown in figure 5.

As an important feature, joints are also supported by the framework through the physics service which needs two game objects as a parameter. In this case, these game objects will react according to the joint constant added to them. At this time, the physics service only supports rigid bodies but in the future it will be possible to simulate fluids and soft bodies with less or no impact in the framework architecture.

5 Performance and Analysis

In this work, we evaluated the framework against jMonkeyEngine [jMonkeyEngine 2003], a 3D engine that uses Java for simulation programming. We choose jMonkeyEngine because Java is an easy to use and powerful language for developing real time simulations. Additionally, Java is very comparable to C# as both use a Virtual Machine (VM) for running applications. For this comparison, we used jMEPhysics [JMEPhysics], an interface that connects a variety of physics library to jMonkeyEngine, allowing the user to simple use it without worries about integrations with the render engine.

At first, we planned to evaluate the framework against VPython as it is widely used in academic for physics simulations [Salgado 2001]. Unfortunately VPython has not a built in physics module or does not use anyone available. In this case, the integration needs to be done by the user if physics simulations are necessary.

The tests were performed on an Intel Core 2 Duo 2.4Ghz CPU, 4GB of RAM equipped with a NVidia 9800 GTS GPU. Each instance of the test ran for 30 seconds and the average time to compute a frame was recorded for each one. To assure the results are consistent, each test was repeated 5 times.

A total of 10 different test instances were performed for each framework type, varying only the number of balls, ranging from 50 to 16000.

From the raw results, shown in Table 1, it is possible to see the performance of the XNA framework using GPU for physics over

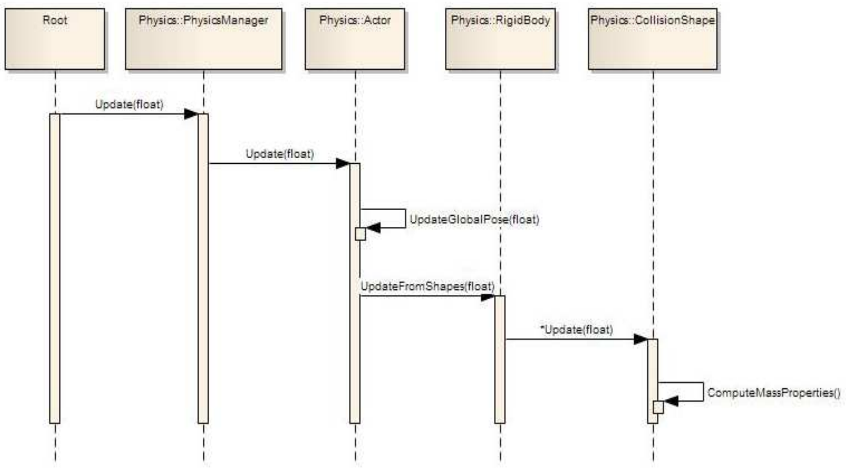

Figure 5: *The physics sequence diagram*

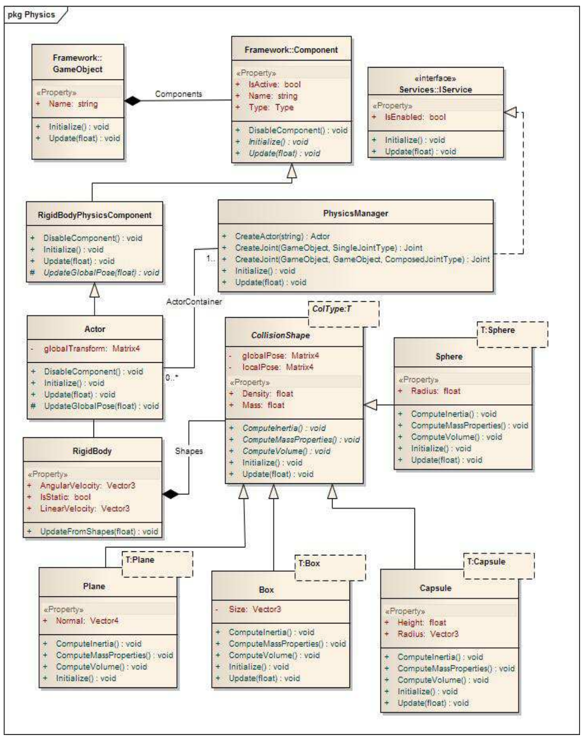

Figure 6: *The physics service architecture*

a CPU physics based. From this table, it is possible to see that physics simulations in JMonkey decreases in a exponentially manner whereas it is more linearly in XNA. Using GPU physics based, more accurate simulations can be made as more powerful is given for researches, as well.

6 Physical World Simulation

To show how easy people who does not have much knowledge about computer and graphics programming can work with the proposed framework, this section presents the basic steps necessary to create a physical world. In Listing 3, the basic steps to create a game object which will simulate physics are shown.

```
GameObject ball = new GameObject("ball");
RenderableComponent renderable = new
    RenderableComponent(
        new XnaModel(Framework.
            ContentManager.Load("BallModel")
            );
BasicEffectUpdater effectUpdater = new
    BasicEffectUpdater();
RigidBodySimulator sim = new
    RigidBodySimulator();
sim.Mass = 10.0f;
sim.AddShape(new Sphere(1,1,1));
ball.Components.Add(renderable);
ball.Components.Add(effectUpdater);
ball.Components.Add(sim);
physicsSimulationScreen.
    GameObjectCollection.Add(ball);
```

Listing 3: *Physics simulation setup*

Initially, to simulate the physical world, game objects need a component that handles this task. In this case, the built-in *RigidBodySimulator* component was used to this purpose, which, basically, updates the transformation matrix of a game object according to parameters that were set in it like mass and inertia, among others. This gives educators the freedom to simulate whatever physics condition is desired, requiring minimal effort as these parameters can be easily changed, even during the simulation. Another important parameter that needs to be set in rigid body simulation is the collision shape of the object to be simulated. In this example, collision was made by using a sphere shape but others are available, like box and capsule.

Simulating physics objects only requires the *RigidBodySimulator* component. As graphical output is important, a component needs to be added in this game object so that its rendering can be made possible. Here, it is used the built-in *RenderableComponent*. As every object to be rendered requires a shader and this needs parameters, *RenderableComponent* component needs a way to retrieve these parameters from the user. This is achieved by using a *BasicEffectUpdater* that is responsible to setup the shader with its parent game object transformation matrix, as can be seen in Listing 4. The benefit of using a graphics architecture like that is the reuse possi-

Table 1: *Comparison Tests*

# actors	time JMonkey	fps JMonkey	time XNA	fps XNA
50	1,67	598,41	4,25	235,29
200	7,12	140,26	4,93	202,84
500	26,45	37,80	6,46	154,80
1000	78,61	12,72	10,60	94,34
2000	144,92	6,90	18,43	54,26
3000	190,11	5,26	26,99	37,05
4000	202,02	4,95	34,33	29,13
8000	-	-	88,70	11,27
12000	-	-	148,20	6,75
16000	-	-	213,35	4,69

bility of this component by all *RenderableComponent* that requires the same parameters to work. Finally, a static game object to simulate a ground plane, not shown here for clarity, is added to the simulation.

With the steps above, the game object is ready to be simulated, only needing a *GameScreen* to do that. As this example simulates 2000 game objects with the same configuration, the *GameObject* class has a method called *Clone* that is responsible for duplicating a *GameObject*. This method was provided to facilitate the creation of more than one game object with the same components but with different parameters that can be setup in each of them. Finally, an image of this code running can be seen in Figure 6.

```
void Update(Effect eff)
{
  eff.Parameters["World"].SetValue(Parent.
      Transform);
  eff.Parameters["View"].SetValue(
        XNAEngine.Engine.Instance.Services.
              GetService<XNAEngine.Services.
              Camera>().View);
  eff.Parameters["Projection"].SetValue(
        XNAEngine.Engine.Instance.Services.
              GetService<XNAEngine.Services.
              Camera>().Projection);
}
```

Listing 4: *Shader parameters setup*

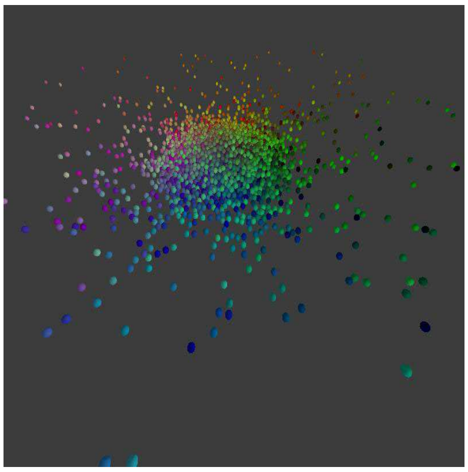

Figure 7: *The simulation of 2000 rigid bodies*

The code example shows how easy is the creation of an environment for physics research by anyone who does not have a background in computer programming or computer graphics. After setting the components desired in *GameObject*, this one will behave according to the parameters from which these components were configured. In reality, composition of behaviors is one of the key aspects that facilitate the use of the framework, as it has an architecture that only requires the composition of behaviors to create different kinds of game objects, which does not occur when using inheritance, as most functionally cannot be easily shared among other classes.

7 Conclusions

In this paper, a framework that enables real time physics applications with graphical feedback was presented. The main purpose of this framework is to help educators and researches in physics, without background in computer programming and graphics programming, to use technologies that are mainly used in games to simulate physical environments. This is achieved by using a solid architecture and a computer language that is easy to use and do not have collateral effects. Additionally, it uses one of the most promising approaches to simulate physics — the graphics processor unit and CUDA to parallelize the computations, releasing the CPU from these tasks.

Due to its ease to use and understand characteristics, the framework can be used in universities and schools to teach physics concepts. According to recent studies, theories that can be simulated and visualized using computer graphics are most easily assimilated by students than others that cannot [Johnson 2005]. With this in mind, physics educators can use the framework to present physics concepts that are possible to interact and, at the same time, see the results graphically.

The framework also has an architecture that allows it to be used in other fields other than education as, for example, games, just needing some additional services to be provided, like a sound manager.

The framework is a constant work in progress and we are now investigating how use the framework for simulating soft bodies and fluids to allow a wide range of physics simulation enabled in real time. We also plan to extend this to developing a visual editor to allow the simulation to be created visually, without requiring programming by the user of the framework. This could facilitate even more uses of the framework.

References

BILAS, S. 2002. A data-driven game object system. Game Developer Conference.

BIRSAN, D. 2005. On plug-ins and extensible architectures. *Queue 3*, 2, 40–46.

DEDE, C. 1995. The evolution of constructivist learning environments: Immersion in distributed, virtual worlds. *Educational Technology 35*, 46–52.

GEE, J. P. 2004. *What Video Games Have to Teach Us About Learning and Literacy*. Palgrave Macmillan.

INTEL, 2000. Havok. `http://www.havok.com/`.

JMEPHYSICS. Jmephysics: interface between jme and physics engines. `https://jmephysics.dev.java.net`.

JMONKEYENGINE, 2003. Jmonkeyengine 2.0 on line documentation. `http://www.jmonkeyengine.com`.

JOHNSON, W. L. 2005. Lessons learned from games for education. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Educators program*, ACM, New York, NY, USA, 31.

JOSELLI, M., CLUA, E., MONTENEGRO, A., CONCI, A., AND PAGLIOSA, P. 2008. A new physics engine with automatic process distribution between cpu-gpu. In *Sandbox '08: Proceedings of the 2008 ACM SIGGRAPH symposium on Video games*, ACM, New York, NY, USA, 149–156.

KAČIĆ-ALESIĆ, Z., NORDENSTAM, M., AND BULLOCK, D. 2003. A practical dynamics system. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 7–16.

MAYO, M. J. 2007. Games for science and engineering education. *Commun. ACM 50*, 7, 30–35.

MICROSOFT, 2004. Creators club. `http://creators.xna.com/en-US/`.

MICROSOFT, 2008. Microsoft robotics. `http://msdn.microsoft.com/en-us/robotics/default.aspx`.

NVIDIA, 2004. Cuda zone. `http://www.nvidia.com/object/cuda_home.html#`.

NVIDIA, 2008. Physx. `http://www.nvidia.com/object/physx_new.html`.

PAPASTERGIOU, M. 2009. Digital game-based learning in high school computer science education: Impact on educational effectiveness and student motivation. *Comput. Educ. 52*, 1, 1–12.

PASSOS, E. B., SOUSA, J. W. S., NASCIMENTO, G., AND CLUA, E. W. G. 2008. Fast and safe prototyping of game objects with dependency injection. In *VII Brazilian Symposium on Computer Games and Digital Entertainment*, Sociedade Brasileira de Computao - SBC, 64–69.

POPOVIĆ, J., SEITZ, S. M., ERDMANN, M., POPOVIĆ, Z., AND WITKIN, A. 2000. Interactive manipulation of rigid body simulations. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 209–217.

PRICE, C. B. 2008. The usability of a commercial game physics engine to develop physics educational materials: An investigation. *Simul. Gaming 39*, 3, 319–337.

PRICE, C. B. May 2008. Learning physics with the unreal tournament engine. *Physics Education 43*, 291–296(6).

PUC-RIO, 1993. The programming language lua. `http://www.lua.org/`.

SALGADO, R., 2001. Vpython applications for teaching physics. `http://www.phy.syr.edu/~salgado/software/vpython/`.

SCHERER, D., 2000. Vpython. `http://vpython.org/`.

SHAPIRO, A., CHU, D., ALLEN, B., AND FALOUTSOS, P. 2007. A dynamic controller toolkit. In *Sandbox '07: Proceedings of the 2007 ACM SIGGRAPH symposium on Video games*, ACM, New York, NY, USA, 15–20.

SQUIRE, K., BARNETT, M., GRANT, J. M., AND HIGGINBOTHAM, T. 2004. Electromagnetism supercharged!: learning physics with digital simulation games. In *ICLS '04: Proceedings of the 6th international conference on Learning sciences*, International Society of the Learning Sciences, 513–520.

STRONG, A., 2002. America's army official website. `http://www.americasarmy.com/`.

UNITY, 2004. Unity3d. `http://www.unity3d.com`.

WORKMAN, K., AND ELZER, S. 2009. Utilizing microsoft robotics studio in undergraduate robotics. *J. Comput. Small Coll. 24*, 3, 65–71.

YEH, T. Y., FALOUTSOS, P., AND REINMAN, G. 2006. Enabling real-time physics simulation in future interactive entertainment. In *Sandbox '06: Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*, ACM, New York, NY, USA, 71–81.