Generic Physics

A CONCEPTUAL INTERFACE FOR RIGID BODY PHYSICS ENGINES

Carlos Gomes
IST - Technical University of Lisbon
Av. Prof. Cavaco Silva
Tagus Park
2780-990 Porto Salvo, Portugal
carlos.gomes@tagus.ist.utl.pt

ABSTRACT

Starting in the XVII century with the scientific revolution, Physics revolutionized the world with great inventions and discoveries. Nowadays, with the ever increasing capabilities of computers, gamers became more demanding in terms of realism, which led to the latest addition to their virtual worlds – Physics.

Starting by being hardcoded on the application, physics processing evolved into software libraries called Physics Engines. They focus on implementing the physics effects often in close connection with the graphics processing. This way the developer is responsible for embedding it on its own application. This approach works well but limits the scalability of the applications. This happens because they were specifically developed using the implementation details of the chosen physics engine. Change to a different physics engine, specially a better one, becomes very difficult. It often means redeveloping the application from scratch.

A possible solution would be to have an intermediate layer capable of making the bridge between our applications and the physics engines in a generic way. As such in this paper we show a solution to the question: "Is it possible to define a generic physics abstraction, powerful enough to allow the developing of virtual worlds applications, independent of the solution underneath?"

To develop our solution we analyzed classical physics and the majority of physics engines currently on market. Focusing on Rigid Body Physics and Collision Detection this analysis allowed the extraction of concepts, their relationship and essential simulation functions to ultimately develop the conceptual model of our solution.

Finally this model was implemented using an object oriented programming into an interface system. For our case study we implemented the interface with two engines- Ogre3D for the graphics and ODE for the physics. Finally we used our implementation to develop a simple Virtual World Editor.

We used our Editor to demonstrate that our conceptual model resulted in a generic physics programming interface that allows the creation of physically enabled simulations. Although only demonstrated on the rigid body domain theoretically this approach can be applied to all physics domains.

Keywords

Physics, Physics Abstraction Layer, Implementation Independent Development.

1. Introduction

Since ancient times, people have been trying to understand the behavior of matter: the falling to the ground of unsupported objects, the different properties of materials, and so on. In the XVII century, after centuries of medieval obscurity, this curiosity triumphed finally, with the scientific revolution: physics stepped up to become one of the greatest fields of science, matched only by the ancient a priori science of mathematics.

Known physical effects that occurs in nature can be described (at least approximately) in a mathematical form – this is especially true for the most studied ones, those which directly humans can sense like gravity, collisions and movement. This property allows us to simulate them in computer programs. However, these calculations are complex and require a high processing power.

The processing power requirements lead to the appearance of two approaches to physics simulation [1]: **High Precision Physics** and **Real Time Physics.** In the first one the physical effects are calculated with great precision. Such approach has historically been applied by scientists with access to supercomputers in various fields of investigation, although recently the movie industry has started using it to create special effects (ILM [2]) as well as computer animated movies (e.g. Pixar's [3]). On the other hand, in Real Time Physics a rapid response of the system must be guaranteed in order to maintain a "real-time" sensation for the user. To achieve this, simplified models of reality that "appear" to be real are used. Real Time Physics is the focus of this work, especially on the video games domain.

Imagine a video game where you have nearly or completely realistic models, sounds and a human-like A.I. At first glance it seems as if it is a movie. Imagine however that whatever you do in this virtual world you always see the same hand-made animation over and over again and no matter what you try, all the objects in the environment remain forever static. After a while your initial sensation of a movie-like virtual world is gone, and so is the whole experience. As such it is imperative that if we want to create even more realistic worlds inside our video games we need to simulate the real world physical effects to the greatest extent possible for the player to truly immerse himself into the virtual world. With this purpose in mind, a specialized software component has been created: the physics engine.

There are several physics engines on the market and most of them are designed to be embedded in the application code. The common practice is to mix graphical and physical code all together for performance and historical reasons. Until recently

most of the physics calculations (e.g.: collision detection).were done mainly by the graphics component of the application. In earlier times, developers had to solve physics problems but these were just not important enough to justify being taken care of separately and were calculated on the graphics engine. The similarity in some calculations between graphics and physics also contributed. This common approach works very well and delivers the most performance tuned application possible but with associated costs in terms of scalability.

For instance, let's consider long term applications that aren't supposed to end in a "final boxed product", like many research programs. When development is finished, they use the latest technology of their day but as time passes, the same technology that once was cutting edge becomes obsolete. As components are usually hard coded and share a great dependency between them, the cost of changing them is so high, that developers often choose to rebuild the application from scratch, rather than updating its technology. To worsen the problem, physics engines implementations are quite different from each other.

If physics engines shared a common base this situation could be solved. This way the problem we are addressing in this paper can be resumed in the following question: "Is it possible to define a generic physics abstraction, powerful enough to allow the developing of virtual worlds applications, independent of the solution underneath?"

To answer the question we studied the main physical concepts that form the basis of every physics engine and how they relate and interact with each other. We thought that if we were able to clearly define the concepts relationship model we could produce a programming interface that translates that model into the computation world. This way the resulting interface system, being created based on a generic model, became possible to connect to every physics engine. Furthermore our interface system was totally independent of the other application components, especially from graphics.

As a result, choosing/changing a physics engine in applications developed with our interface system becomes easier as no application code must be changed. Developers just need to connect the chosen physics engine to our interface system. The same is true when upgrading our interface with new features as previous applications remain functional although not using the new functionality.

1. Physics Engines: State of The Art

In the beginning, even when video games were very simple, the physical effects were already present, as you can see in the classic Tetris where the pieces "fall" from the top. This "fall" is nothing less than a primitive simulation of gravity.

The years passed and more complex games made their appearance. If we take the example of Doom we see the introduction of several physical effects like gravity (when the character jumps or when he carries too much weight and slows the movement speed), friction (when stepping on different ground like sand) and even effects like velocity, acceleration and primitive collision detection (like hitting an wall).

These effects (gravity, friction, velocity, acceleration and collision detection) are called Newtonian Physical Effects [1] and make the basics of a physics simulation.

Until recently all the effects were hard coded into the applications and could not be reused on other games.

In 1998, a small company named Havok [2] was created with the goal of creating a reusable physics library that could be integrated in every game that licenses it. This was the birth of the Physics Engine as we know it today.

"A physics engine is a computer program (software component) that using variables such as mass, velocity, friction and wind resistance can simulate and predict effects under different conditions that would approximate what happens in either real life or a fantasy world." [3]

With the appearance of the physics engine the gaming industry was relieved of the burden of creating their own physical effects and could rely on third party companies to focus their efforts in researching and developing their physic engines.

Today physics engines not only simulate Newtonian Physics but also support many new "high-tech" effects like cloth or fluids simulation.

1.1 Features of a State of the Art Physics Engine

The main features of a state of the art engine are:

- **Collision Detection** - The goal of collision detection is to report when a geometric contact has occurred or will occur between two or more physically enabled objects on the virtual world. Collision between objects can be detected using bounding spheres or boxes [4], and with the world by using ray-casting.
- **Rigid Body Physics** - In physics, a rigid body is an idealization of a solid body of finite size in which deformation is neglected.
- **Character and Vehicle Physics** - Until the advent of video games physics, character motion was performed exclusively through a series of animations created in some form of 3D modeling program. With the power of physics the recent approach to characters/vehicles physics consist in building a skeleton/structure of the character/vehicle and affect it of physical effects.
- **Soft Bodies** - Soft body dynamics focuses on accurate simulation of flexible objects allowing for easier creation of secondary motion effects of muscle, hair, cloth, etc. Soft bodies differ from rigid bodies in that the latter don't suffer deformation.
- **Fluid Physics** - Typical fluid flows vary from rising smoke, fire, clouds and mist to the flow of rivers and oceans [5].
- **Destroyable Objects** - Current state of the art physics engines support fully destroyable environments.. Until recently this was not possible due to the high complexity of calculating physics for the resulting objects (and its interactions).

1.2 Physics Engines Analysis

In our analysis we have studied the following physics engines solutions: Newton Dynamics, NVPhysics, Ageia, Havok, Euphoria, Renderware Physics, Unigine, Physics and Math Library, TrueAxis, Pseudo, ODE and Bullet.

To be fair while comparing the solutions we separate those between the commercial ones that are financed and developed with a commercial intuit (either now or for a future commercialization) and the ones fully open source that are developed by good will developers.

The next table summarizes the commercial solutions described in this paper. CS/Free stands for Closed Source but free for non commercial use and C stands for Commercial.

Table 1: Commercial Physics Engines

	Newton Dynamics	NVPhysics	AGEIA	Havok	Euphoria	Renderware Physics	UniGine	PhysicsAndMathLibrary	TrueAxis	Pseudo
Type of license (OS / CS-Free /C)	CS-Free	CS-Free	CS-Free	C	C	C	C	C	C	C
Hardware Acceleration Support			PhysX	GPU						
Collision Detection	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Rigid Body Dynamics	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Soft Body Dynamics			Y	Y	Y			Y		
Fluid Dynamics			Y	Y	Y		Y	Y		
Character Dynamics	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Vehicle Dynamics	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Destroyable Objects and Debris			Y	Y	Y					Y

From this chart and from the pros and cons of every engine we conclude that Havok and Ageia are currently the best Physics Engine solution available on the market. Euphoria theoretically is at the same level of the other two but there is no way to prove that as no nowadays game uses it and there are no benchmarks available.

Both engines support hardware acceleration but each one according to their personal view as discussed earlier (either using the GPU capabilities or introducing the PPU).

It is also interesting to note in the field of "all-in-one" solutions Unigine appears to be technically superior to RenderWare although this one has a vast portfolio of successful games using it.

Table 2: Open Source Physics Engines

	Open Dynamics Engine	Bullet
Type of license (OS / CS-Free /C)	OS	OS
Hardware Acceleration Support		
Collision Detection	Y	Y
Rigid Body Dynamics	Y	Y
Soft Body Dynamics		
Fluid Dynamics		
Character Dynamics	Y	Y
Vehicle Physics	Y	Y
Destroyable Objects and Debris		

Pseudo is a very good Physics Engine if the developers are only interested in vehicle realism as it is highly optimized for that field.

As for the fully open source engines Table 2 summarizes the results.

These solutions are well underdeveloped when comparing to the state of the art commercial ones, but provide a free solution to add basic physical effects to small scale, academic or low budget applications.

Although fairly basic in its features, these solutions provide the necessary basis to program the advanced physical effects, so someone interested may develop those while continuing to use the engine free of charge.

The great advantage of these solutions is the community support that provides a great help to newcomers while some of the commercial solutions simply fail to deliver decent client support while developing.

2. A Generic Physics System

Before continuing we must explain the scope of this work. From our previous analysis we identified six major physical concept dimensions: Collision Detection, Rigid Body Physics, Character & Vehicle Physics, Soft Body Physics, Fluids and Destroyable Objects.

Looking at the studied engines we state that the great majority only implements the first three dimensions and the open source solutions only support those same dimensions. Being this work developed in an academic context it was decided to give priority to open source solutions. That way and due to time constrains, this work focused on developing an abstraction of physical concepts related solely to Collision Detection and Rigid Body Physics.

2.1 Simulation

In a virtual world application the typical simulation cycle is made of four simple stages: User interaction, Physics Update, Graphics Update and finally Render.

The cycle starts by processing user input such as moving the camera, applying forces, selecting objects and creating/destroying objects. Following this, Physics are updated so the collision detection is processed, forces are applied to objects and new positional/rotational information is calculated for every entity. After Physics, Graphics is processed. In this stage pre-rendered animations are updated and all graphical information like shaders and shadows are calculated. Finally the entire World is rendered to screen via the active camera. This is a typical simulation cycle for a virtual world simulation.

Due to the nature of physics engine, implementations often mix the Physics Update/Graphics Update stages in one big Application Update stage. This is because physics processing is done a par with graphics processing.

Analyzing the Physics Update stage we can see that it consists in several sub-stages: Collision Detection, Net Forces Calculation and Positional/Orientation Calculation. During an update, collisions are detected and motion is altered according to different parameters like friction and mass. Forces are also applied to the entities and motion is calculated using classical physics formulas[1]. Finally, the joint contribution of collisions and added forces result on new positional information for every entity. The order of events may be different from the specified but the final result of the physics update stage is new positional/rotational information for every entity on the world.

In light of this knowledge we can separate graphics from physics by making sure that each update remains independent from each other and passing the positional data through our abstraction. This way the simulation cycle of our system, displayed on Figure 1, adds a new stage that does the data transition between physics and graphics.

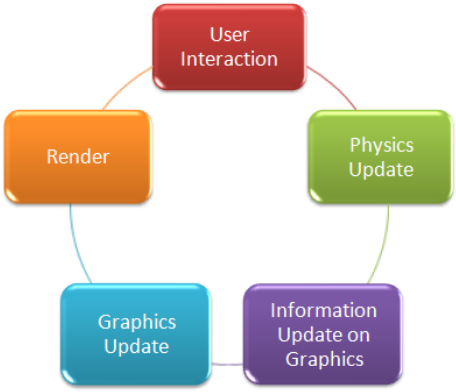

Figure 1 : Simulation Cycle in our System

To assure the correct separation of physics from graphics our abstraction is designed in a way that no graphical information is used in physics calculations. All information needed from physics processing to update graphics is passed through our abstraction as well.

2.2 Concepts

Since this work is focused on rigid body physics, it is imperative that we start our concept collection with Newtonian Physics[1]. These concepts are universal and therefore are common to every physics engine that supports this feature.

Besides physical concepts there are other concepts that we must include in our abstraction which we called system concepts. They are related to the way physics engines are structured and were discovered when we studied their internal architecture mainly from the open source ones.

2.2.1 Physical Concepts

Working with classical physics (Newtonian Physics) the domain we're working with is Motion.

Motion is a change in position of a body relative to another body or with respect to a frame of reference or coordinate system. Motion occurs along a definite path, the nature of which determines the character of the motion.[6]

To define the motion of an object we must also define the following Newtonian concepts:

- **Force** - *In physics, (Force is) a quantity that changes the motion, size, or shape of a body. Force is a vector quantity, having both magnitude and direction. The magnitude of a force is measured in units such as the pound, dyne , and Newton , depending upon the system of measurement being used.*[7]
- **Position/Orientation** - Motion implies a change in the position of the object. In reality it is not just a change in position that occurs. A change on the body orientation happens as well due to rotational effects. The position of the object consists in a three dimensional vector usually mapped in the Cartesian coordinate system. The Orientation is usually represented in physics by a rotation matrix.
- **Torque** - If we apply a force on a object in a position different of the center of mass, depending on the objects geometry, we have a resulting rotational movement. The responsible force is named Torque. It is a special kind of force that only changes the rotational movement of an object.
- **Inertia** - Newtonian Inertia is a property that measures the way an object resists the change to its state of motion.
- **Mass** - The concept Mass can be explained as the quantity of matter that a certain object possesses. The quantity of Mass in a rigid body is always constant because breaking objects are not considered.
- **Friction** - Friction is a force that goes against any movement by a body when in contact with other body. It is the force responsible for the common slowing down that we experience every day.

2.2.2 Physics Concepts

All the above defined concepts are present in every physics engine but come directly from Classical Physics. The following concepts share some connection to Physics but have an Engineering Nature. Some of them came directly from the analysis of the physics engines while others result from our previous knowledge on physics simulation. They encapsulate all the components needed to the correct functioning of the simulation.

- **Entity** - The entity is the basic building block of our abstraction. Every object of the virtual world is a specialized entity. It contains the physical properties, constrains, geometries and information about the object.
- **World** - The most generic concept in a physics engine is the World. The World is responsible for managing every component of the simulation (graphics, sound, artificial intelligence and input/output) and for the simulation cycle.
- **Stepper** - The Stepper is a special part of the World that is solely responsible for processing the physics simulation.
- **Collision Space** - A collision space is a group of collision geometries that belong to a logical group that we know will never collide with each other. This way the physics engine never checks for collisions between those geometries and furthermore can optimize the collision detection process.
- **Collision Geometry** - To simplify and accelerate the collision detection mechanism we use simple geometric forms that surround each object. When processing

collisions, these geometries replace the real object and because of their simpler design are faster to calculate.

2.3 Conceptual Model

With our concepts clearly defined we proceeded to define the relationship among them, the way they are connected and how they interact and influence the simulation.

Our conceptual model is divided in two parts. The first one is responsible for managing the simulation cycle and the simulation general settings. The second part defines the physics component of our Entity. Note that the graphics part is not present in our conceptual model as it was not the focus of our work.

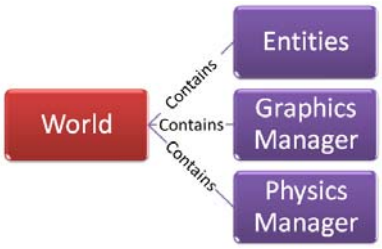

Figure 2: Conceptual Model: Simulation Control

The diagram presented on Figure 2 illustrates the simulation controller of our abstraction. The main concept is the World that controls the entire simulation cycle and contains all entities existing on the virtual world. Under the control of the World are also the physics and graphics managers. They are responsible for setting up the environment to correctly process both physics and graphics, accordingly. General settings are defined through the managers.

As for our Entity the diagram in Figure 3 illustrates how it is structured. The Entity is composed of two main blocks, one responsible for physics calculation and the other for graphics calculation. Once again the graphics block has not been included in our conceptual model because it is not our focus. The Physics block is rather complex and defines all the properties our entity must have to be able to calculate physics behavior.

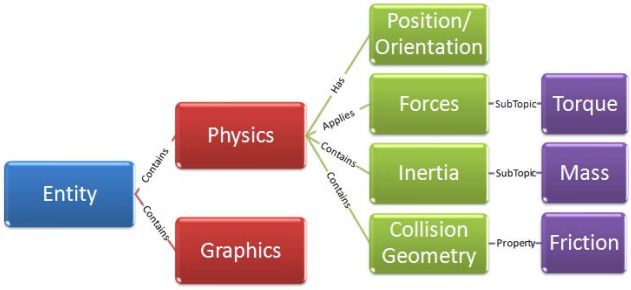

Figure 3 : Conceptual Model: Entity

The most basic information needed, consists on positional and rotational information. This information is updated in every simulation cycle in result of the other components. In order to allow interaction we apply forces of which torque is a special case. We also have the inertia concept, extremely necessary to calculate motion on which mass is included. Finally to make collision detection possible we have multiple Collision Geometries, with friction being a major parameter that influences the way objects react to collision.

With this conceptual model we provide the answer to our problem. We have developed a model based on the very foundations of classical physics. We used the same concepts that Isaac Newton discovered on the XVIII century and added the engineering concepts needed to perform a simulation on the computer. This way we have a generic model where every physics engine that supports rigid body physics can theoretically be mapped.

3. Implementation

Our conceptual model gives us the theoretical solution to our problem. To make it usable we need to transform the conceptual model into a programming interface.

From our conceptual model, the first step to translate it into a programming interface was to directly map the concepts into interfaces. As can be seen in Annex A, the majority of the interfaces correspond directly to the conceptual model. In spite of that, there are concepts that have not been converted into interfaces and others that appeared for the first time.

So in summary the major changes were:

Some concepts were merged into another interface:

- The Torque concept was included into the IForce Interface. We decided to merge them as Torque is a special type of Force and the methods used to control it don't justify an Interface on its own.
- The Mass Interface was included into the IInertia Interface. For simulation purposes the concept of Mass and Inertia is very close. Often they are treated as one. For this reason we decided to merge these concepts into one interface.
- The Friction concept was included into the ICollisionGeometry Interface. When a collision happens, the physics engines deal directly with the collision geometry. Having the friction properties directly accessible on the ICollisionGeometry interface provides an easier access to information and reduces information everhead.

Some new interfaces emerged. This resulted from deep analysis of physics engines.

- Interface IErrorHandling. To improve simulation stability physics engines provide error correction mechanism, thus the necessity of creating a new interface.
- Interface IAutoDisable. To maximize performance, physics engines have the capability to detect when an object is idle and stop processing it. Processing is resumed when the object suffers a collision is affected by a force. We decided to call this functionality auto disable and provide an Interface for this new concept.

Once the interface architecture was concluded we proceeded on to the interface's methods definition. We did it by using the knowledge acquired with our related work and by analyzing several physics engines API. We chose the methods carefully to make sure they were generic and essential to a rigid body physics simulation.

4. Case Study

4.1 Interface Concretization

Implementing our interface system required a myriad of technologies. Some were mandatory because they were part of this work initial description while others were chosen according to their features and suitability for an academic work.

From the start of this work we have decided that our interface system was to be implemented using the .Net Framework. The main reason behind this decision has to do with the research group where we developed this work. In the group there are many applications developed with the .Net Framework and this way we make it possible to integrate physics effects into already existing applications.

To make our interface system usable, we need to implement it using real engines. For graphics processing the engine used was Ogre3D [8]. Once again the main reason for this choice was the fact that at our group, existing applications use Ogre3D to process graphics. Furthermore there is another reason that makes this choice even better. Ogre3D is a free open source engine. This way we can alter the source code as necessary and learn from it.

For the physics processing we decided to use Open Dynamics Engine usually called ODE [9]. This choice was primary based on the results of our related work and the limitation imposed by the research group of using open source solutions. We decided to use ODE over Bullet due to the better documentation and the fact that we had an already implemented solution where ODE was integrated with Ogre3D. This integration was done by Tuan Kuranes and is free to use and modify[10]. This existing integration provided a looking window to start studying the way Ogre3D and ODE works.

Concretizing our interface system required three programming languages to be used and learnt. ODE interface is written in C and Ogre3D in C++. The already existing integration gave us a C++ interface to ODE so we can assume that both engines are written in C++.

Focusing on .Net Framework our interface system needed to be compatible with the CLR (Common Language Runtime)[11], the virtual machine of Microsoft .Net Framework. This brought about a problem because C++ is not directly supported on CLR as it has no automatic memory management. To solve this problem we used a new version of C++ called Managed C++ [12] to create a wrapper between the engines C++ code and the CLR.

Basically our interface was implemented using Managed C++ and connected with the engines native code when needed. The main disadvantage of having to use the wrapper is that further overhead of objects is created as we need to have managed objects and native objects running at the same time.

Compatibility with CLR means that every .Net Language may be used to develop using our interface. However we chose to develop using C#, once again because it is the *de facto* programming language at our research group.

Finally one last note on the integration between Ogre3D and ODE that we used. Besides providing a C++ interface for ODE, it also provided a conversion between Ogre3D and ODE data types. For instance, in Ogre3D rotations are represented as quaternions [13] while in ODE they are represented as rotation matrixes.

During development the integration was slightly changed by us in order to implement missing features or correct bugs.

4.1.1 Implementation Decisions

Implementing our interfaces consisted in developing the Managed C++ classes that provided the functionality of the interfaces methods. This was a process that ranged from directly calling an ODE function to more complex operations that required the saving of state information and calling several ODE functions. Besides that there some decisions we made while developing.

First of all we decided that we wanted a generic World and Entity class. The main advantage of this decision is that we may attach and detach managers to our World as we please. In our specific case of physics, in the limit, we can do something weird as to use a different engine for each step. All in runtime and by user command. But due to the intimate relation existing between the managers and the Entities components we must be careful when doing this. We must guarantee that the Entities components match the Manager type. If this doesn't happen the Entity isn't processed has the manager doesn't recognize a compatible physics component.

Besides changing managers in runtime we can turn them on or off by simply attaching and detaching them from the World. As the application state is stored in the World and in the Entity classes we can do any manager change without compromising the simulation.

The second important decision deals with performance. To increase it we collapsed several Interfaces into the same class. This is especially true for the Physics class that besides IPhysics implements the IAutoDisable, IErrorCorrection, IForce and the IVelocity interfaces. For the developer all remains equal because he works with the interfaces. From our point of view we get a performance gain by eliminating a great deal of overhead due to the creation of unnecessary class instances.

The schema on Annex B provides a full overview of the implemented classes.

4.1.2 Event Handling System

Our event notifying system provides the developer with an easy to use way of knowing exactly what is going on the simulation. This way the developer gets all the information he needs without having to directly question each object repeatedly. Based on the event information the developer can react accordingly or simply inform the application user.

Our event system consists in the implementation of the observer design pattern. Figure 4 pictures the way the developer interacts with our event system. It is also how the Observer Pattern works.

Some interesting object provides events to the developer. In our example we are using an Entity providing three events. When someone feels the need to know when that event occurs, they register themselves to the event on the Entity. As simulation runs if the event happens, the Entity notifies every registered user that it happened and sends in additional useful information. With this information the user may react by simply consuming the data or changing the application status (or simply the Entity from where the event occurred).

In our implementation only Entities and the World possess events. This way global events like "detect all collisions" are detected on

the World and local events like “detect this Entity’s collisions” are detected on the Entity.

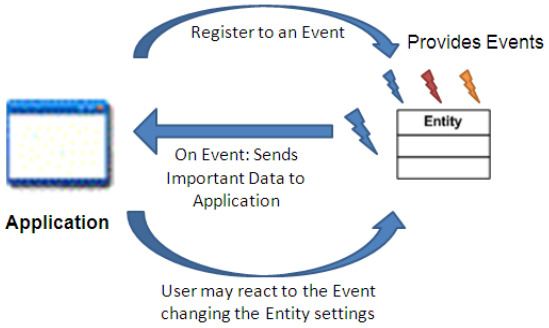

Figure 4: Application Screenshot. We created the World, Managers and defined the application cycle.

Currently we have implemented some events. The most important event is collision detection. When an object collides with other object the user is informed of the colliding entities and contact information like friction and bounciness.

Another useful event is to warn when an entity becomes inactive or becomes active. This has to do with the AutoDisable function described earlier. Finally we have another set of events to indicate when the entity is moving or rotating.

Our event system is designed to be expanded with easiness. We used a feature of the C# language called generics [14]. Generics enable the definition of algorithms and pattern implementations once, rather than separately for each type. This way we implemented generic event listeners so that to add a new event the developer just need to do three steps.

First of all define the callback function to be called when the event happens. Secondly, instantiate a new event listener using the new callback. Finally find in the code where the event may happen and trigger the notify action of the event listener when it happens.

4.2 Programming with concepts

Having our interface system implementation using Ogre3D and ODE we must test its capability to develop applications. Developing with our interface is really easy and fast. Following is a small example where we setup the simulation and create some entities.

The first step towards our goal is to define a World. In the example below we decided to add a Graphics and Physics Manager. We could add just one of them or even none and add them later. We also define the default collision space and the default stepper to use. In this case we chose a OdeSimpleSpace and a OdeQuickStepper, both being ODE-specific implementations for the generic concepts.

```
//Creation of a World
ION.Environment.World wrl = new World();

//Creation of the Managers
GraphicsManager gfx = new OgreGraphicsManager();
PhysicsManager phx=new OdePhysicsManager();

//Registering the Managers on the World
wrl.graphics = gfx;
wrl.physics = phx;

//Creation of the Collision Space
Space spc=new OdeSimpleSpace();
wrl.physics.baseSpace = spc;

//Determining which stepper to use
wrl.physics.activeStepper =
OdeQuickStepper.getInstance();
```

Now we have a World and two registered managers. All objects were created using default values. For instance the Physics Manager class automatically defined gravity as 9.8. Running this source code will result at nothing as we have not called the simulation cycle. The simulation cycle can be defined in its simple form, using the World or in its complex form using directly the managers:

Using the simplified form:

```
while (wrl.render())
       wrl.update();
```

It is as simple as it gets. We update everything and render the objects. The application’s stopping condition was defined to come from the render function.
The complex way that gives full control over the simulation is:

```
while (wrl.graphics.render())
{
   wrl.physics.step();
   wrl.synchronize();
   wrl.graphics.update();
}
```

Here we see the different phases of the simulation cycle divided. First we calculate positional data on the physics, then we synchronize physics with graphics using the world and finally we update the graphics component.
Right now we have an application that displays our world without anything more.

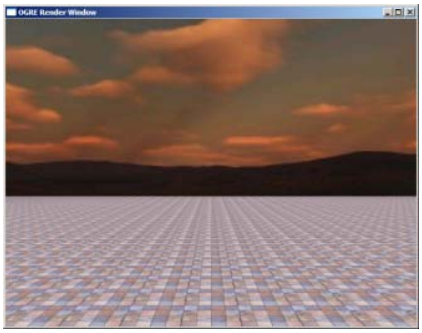

Figure 5: Application Screenshot. We created the World, Managers and defined the application cycle.

Right now we only created the World and the Managers for the physics and graphics components. To make things interesting we must add objects to the World and see them interact. To create an Entity is quite easy. In the following example we create an Entity and add a visual representation to it (no physics effects).

```
//Creation of the Entity
Entity ent = new Entity();

//Creation of the graphics component
ent.graphics = new Graphics();
```

```
//Creation of a Graphical Mesh (Graphical Geometry)
//and inclusion on the Graphics Component
Mesh mesh1 = new Mesh("ball.mesh");
            ent.graphics.meshes.Add(mesh1);

//Entity Registration on the World
wrl.entities.Add(ent);
```

As a side note on our Ogre3D/ODE implementation we created some graphical helper objects that weren't described on the previous chapter as graphics isn't our focus. Besides that we should only note that if we create an Entity and don't register it in the World, it doesn't exist in practice.

The final result of this simple code is pictured on

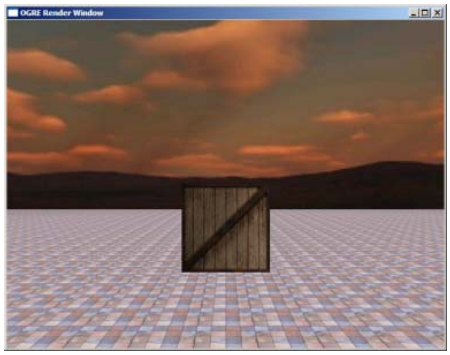

Figure 6: Example Application where we added an Entity with a graphics component using a Crate Mesh.

As of now we only have a static object that doesn't move. To add Motion we will use our Physics component.

The necessary code to add physics effects to our Entity is as follows and really shows concept programming[15]:

```
//Creation of an Physics Component
ent.physics = new OdePhysics();

//Setting the Inertia Settings
//In  this case a Mass value of 1 is used and the Mass Distribution is of type Box
ent.physics.inertia                =                new OdeBoxInertia(1,mesh1.boundingBox);

//Setting a Collision Geometry
//In this case we use a Box Geometry and attach it to the main collision space defined in the beginning of the simulation - spc
CollisionGeometry          geom          =          new OdeBoxCollisionGeometry (mesh1.boundingBox,spc);

//Adding the Collision Geometry to the Entity

ent.physics.collisionGeometries.Add(geom);
```

As we see it is really easy to add physical behavior to our Entities. To sample test this we will position the crate a little bit up so that it will fall down on the floor under the gravity force. For that we change the following property:

```
ent.position = new Vector3D(0, 20, 0);
```

Now the resulting animation develops as pictured next:

Figure 7: Physics Enabled Object: The Crate object is moving due to the gravitational force.

This is the simpler example we can program using our Interface System. Note that the collision with the floor occurs because the Physics Manager creates an infinite plane simulating the floor as default.

4.3 World Editor

To demonstrate that our implemented interface is suited to be used to develop virtual world applications we decided to develop a small world editor. This editor was developed using solely our interface for the physics effects. Once again for the graphics processing we used a small implementation that provided basic graphics support. To develop the interface system of our editor we used Windows Forms to be able to develop a Windows-like interface.

4.3.1 Using our Editor

The overall user interface of our Editor is presented in Figure 8.

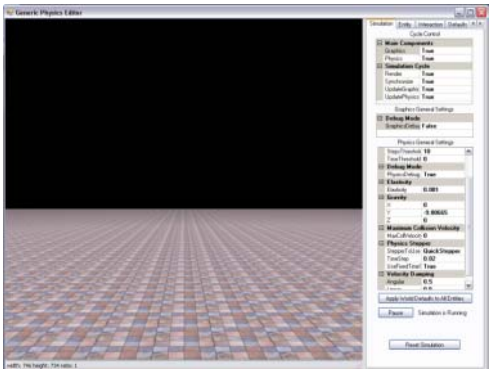

Figure 8 : Main view of the World Editor

It has two main parts:

- World Viewer – This is where the simulation is visualized in real time. It is totally interactive using the mouse and we can select objects and navigate on the world using the mouse buttons and the mouse wheel.
- World Controller – this is the menu of the editor. Here we can control in real time our simulation, create destroy entities and interact with them.

Tabs are divided as:

- Simulation: In this tab we can control the simulation cycle and manage global graphics and physics settings like gravity and the stepper type.
- Entity: In this tab we control every aspect of the creation/destruction of an Entity. It has the following sub-tabs:
    - Entity: In this tab we can create/destroy an Entity. We can also add or remove the graphics or physics

component. Entity settings for both components are managed here.

  - Collision Geometries: In this tab we can add or remove collision geometries from our Entity. We can also change its location within the entity to produce a complex collision set.
  - Event in this tab we can choose which event of this Entity to be monitored.

- Interaction: In this tab we provide the controls to interact with our Entities using for example forces or velocities.
- Defaults: In this tab we define the default values that configure our new created Entities.
- Helpers: In this tab we give access to some extra content on our editor like navigation speed configuration, skyboxes and camera change.
- Events: in this tab we can choose which global events to monitor. Additionally we can see information in runtime about triggered events through a textbox.

5. Conclusion

The main objective of this work was to give an answer to the question: "Is it possible to define a generic physics abstraction, powerful enough to allow the developing of virtual worlds applications, independent of the solution underneath?"

Our idea consisted in developing a model based on main physics concepts both from classical physics and an analysis of the functioning system of the physics engines. To develop applications using our solution we would build an interface system based on our conceptual model.

We started by analyzing the majority of current physics engines currently on the market. From this analysis we got a general view of the best solutions in the physics engines area. In conjunction with some study of classical physics we learned how a physics simulation works and extracted the main physical concepts needed to perform it. We focused our work solely on rigid body physics and collision detection due to time constrains.

From our conceptual model we developed an interface system using an object programming language. Finally as a case study to use as testing ground we implemented the interface with two engines (Ogre3D[8] for graphics and ODE[9] for physics) and used that implementation to develop a small world editor featuring physical behavior.

We believe that the objectives set out for this project have been achieved with relative success. We conceived a physics interface system based on the generic physical concepts and showed how they could be implemented using two greatly popular engines. On top of that, we showed that the interface alone can be used to develop complex applications fully featuring physical behavior. We also showed an application where physical settings can all be changed in runtime using exclusively our interface.

However it does not provide universal answers. The possibility of creating such a generic abstraction was demonstrated solely in the domain of rigid body physics and collision detection. Besides that, and due to time constraints, the interface system of other physics engine, needed to show in practice the possibility of change of different engines, was not tested. Theoretically these two conditions are satisfied by our interface. In the first case we should use the same approach of discovering the main concepts behind the physics domain in focus and then add their relationship to the conceptual model and finally to the interface. As for the second issue, our editor relies solely on our interface so if we implement it on other physics engine, changing it is completely transparent to the application. Having implemented the interface with ODE and being able to run a flexible rigid body simulation in runtime using our editor, gives us a very good indication that our model has captured the main concepts and can be mapped on other engines.

Besides answering our main question, our interface also has a great advantage – programming simplicity. In our case study we showed how simple and intuitive it has become to program a physically-enabled application. Basically we permitted the use of the concept programming paradigm [16] where the rule is: "*Your code should reflect the concepts in your application"* [15]. It simplifies application development because the programmer can articulate his thought in terms of concepts without caring about implementation details. Possibly a physics scientist with some programming skills should use our interface with little effort.

As a global conclusion it can be said that the greatest importance of this project lies not in what it has accomplished per se, but what it has started. It was the first step in what can be a project with great potential of expansion and many, many upgrading lines. This was our purpose from day one, and we always had in mind that this was to be the "first stone" in a great structure. In fact, we'd like to leave some suggestions for future works, written on the next section of this chapter.

5.1 Future Work

As said earlier this work has a great potential for improvements. First of all we suggest that the interface be implemented with a state of the art physics engine like Havok [17]. Besides showing that the interface supports that engine we would have a connection with a state of the art engine that brings the next recommendation. We suggest that the conceptual model be extended with more features, in the limit all available in a state of the art engine. That's where the connection with Havok[17] would be very positive, as testing ground.

Leaving the physics domain we would suggest that the graphics component be further developed. As stated earlier, our work provides a minimal functionality in graphics without a structured architecture. It would be great that the same approach we used in this work was applied on the graphics domain. Finally we would recommend the upgrade of our editor with all the new features and an easier to use interface.

6. Bibliography

1. **Crowell, B.** *Newtonian Physics.* s.l. : Light and Matte, 1998.

2. Havok. [Online] [Cited: 09 24, 2007.] http::/www.havok.com.

3. Definition of Physics Engine. *Wikipedia.* [Online] [Cited: 09 24, 2007.] http://en.wikipedia.org/wiki/Physics_engine.

4. **Bade, A. et al.** Oriented Convex Polyhedra for Collision Detection in 3D Computer Animation. 2006, ACM Press.

5. **Bridson, R., Fedwik, C. and R.,Muller-Fischer, M.** Fluids Simulation. 2006, Vol. ACM Press, SIGGRAPH 2006 course notes.

6. Motion Definition. *Encyclopedia, Britannica Concise.* [Online] [Cited: 09 24, 2007.] http://www.britannica.com/ebc/article-9372696.

7. *The Columbia Encyclopedia.* 2007. Sixth Edition.

8. *Ogre3D.* [Online] [Cited: 09 24, 2007.] http://www.ogre3d.org/.

9. *Open Dynamics Engine.* [Online] [Cited: 09 24, 2007.] http://www.ode.org/.

10. OgreOde. *Tuan Kuranes Web Site.* [Online] [Cited: 09 24, 2007.] http://tuan.kuranes.free.fr/Ogre.html#OgreOde.

11. **Erik Meijer, John Gough.** Technical Overview of the Common Language Runtime. 2001, Elsevier.

12. **Eckel, Bruce.** *Thinking in C++*. s.l. : Prentice Hall, 2000.

13. **Erik D., Martin K., Martin L.** Quaternions, Interpolation and Animation. 1998, University of Copenhagen.

14. **Michaelis, Mark.** *Essential C# 2.0.* s.l. : Addison-Wesley, 2006.

15. Concept Programming. *Sourceforge.* [Online] [Cited: 09 24, 2007.] http://mozart-dev.sourceforge.net/cp.html.

16. **Savinov, A.** An Approach to Programming Based on Concepts. 2007, Technical Report RT0005, Institute of Mathematics and Computer Science.

17. *Havok.* [Online] [Cited: 09 24, 2007.] http://www.havok.com/.

18. Types of Physics Engines. *Wikipedia.* [Online] [Cited: 9 24, 2007.] http://en.wikipedia.org/wiki/Physics_engine.

19. Industrial Light and Magic. [Online] [Cited: 09 24, 2007.] http://www.ilm.com/.

20. Pixar Animation Studios. [Online] [Cited: 09 24, 2007.] http://www.pixar.com/.

21. Ageia Advanced Gaming Physics. *Ageia.* [Online] [Cited: 09 24, 2007.] http://www.ageia.com/pdf/wp_advanced_gaming_physics.pdf.

22. **Low, C.** Games and Physics: Design Issues. 2000.

23. **Ojo A., Estevez E.** Object-Oriented Analysis and Design with UML. 2005, e-Macao Report 19.

24. Newton's second law: law of acceleration. *Newton's Laws of Motion.* [Online] [Cited: 09 24, 2007.] http://newtons-laws-of-motion.com/second.html.

25. Newton's first law: law of inertia. *Newton's Laws of Motion.* [Online] [Cited: 09 24, 2007.] http://newtons-laws-of-motion.com/first.html.

7. Annex A

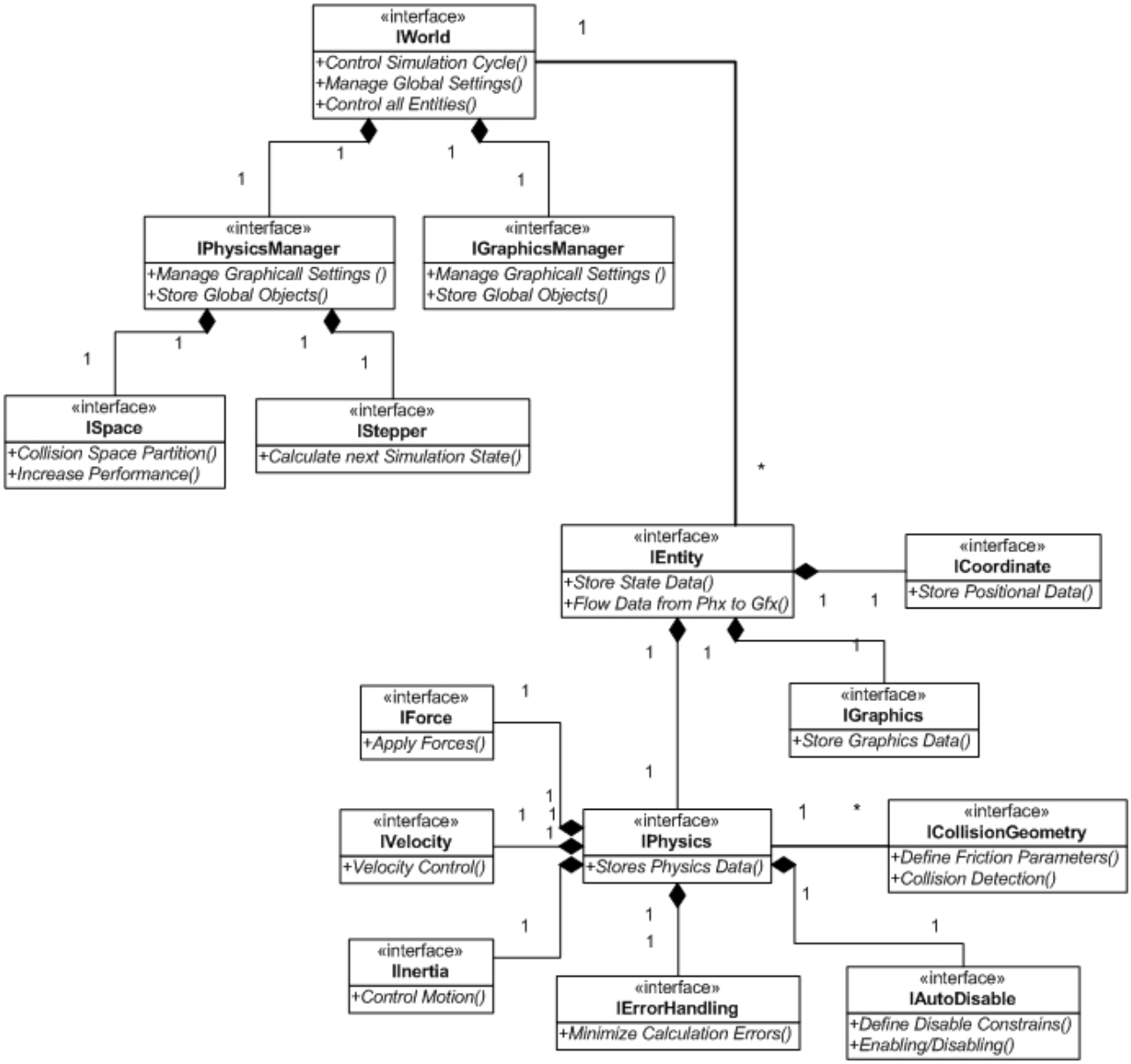

Figure 9: Final Interface Architecture

8. Annex B

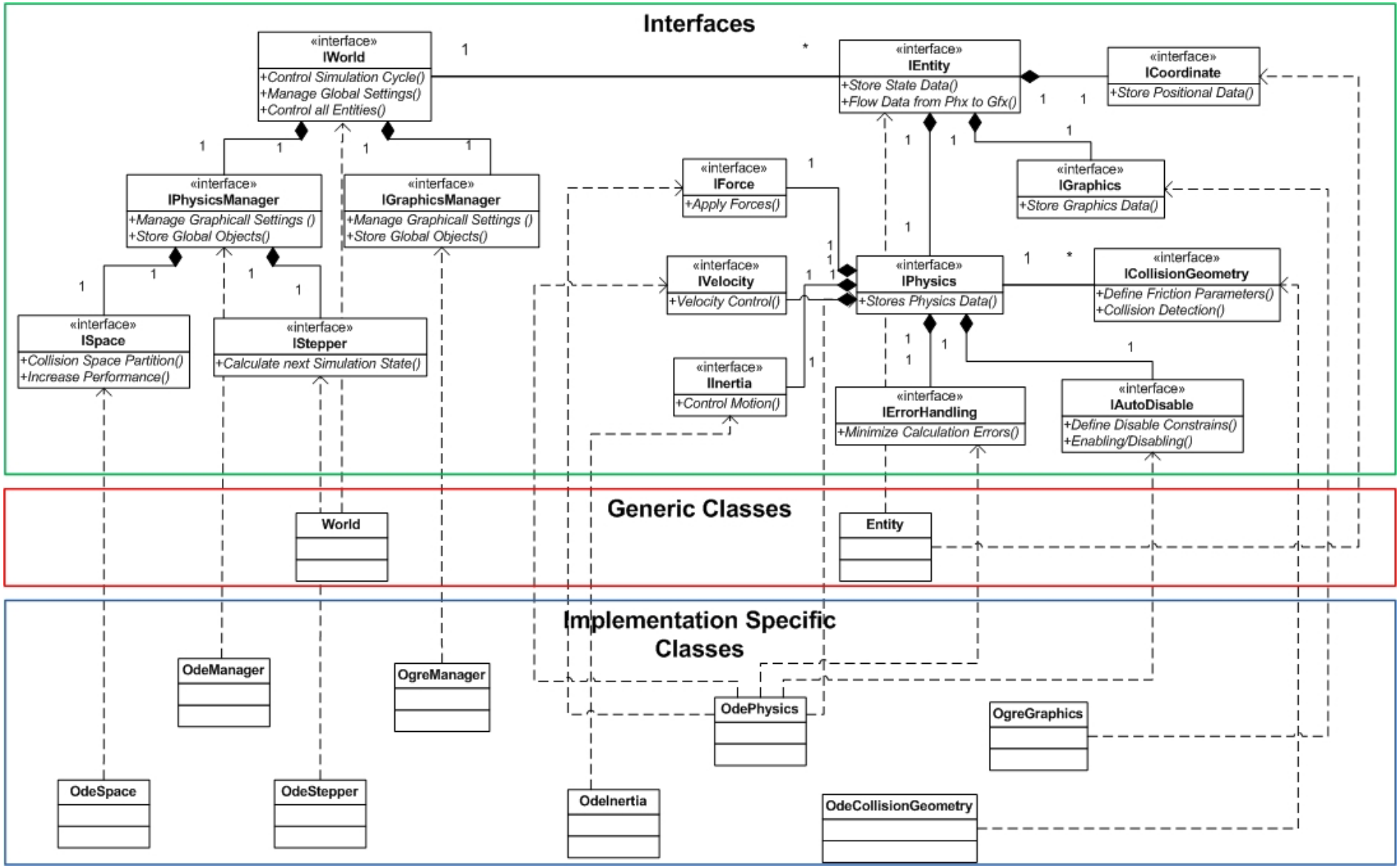

Figure 10: Ogre3D/Ode Implementation Schematic