NOTE TO USERS

Université d'Ottawa • University of Ottawa

Université d'Ottawa · University of Ottawa

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

Saeid NOURIAN

AUTEUR DE LA THÈSE - AUTHOR OF THESIS

Master of Computer Science

GRADE - DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT - FACULTY, SCHOOL, DEPARTMENT

TITRE DE LA THÈSE - TITLE OF THE THESIS

xPheve
An Extensible Physics Engine for Virtual Enviroments

N. Georganas

DIRECTEUR DE LA THÈSE - THESIS SUPERVISOR

CO-DIRECTEUR DE LA THÈSE - THESIS CO-SUPERVISOR

EXAMINATEURS DE LA THÈSE - THESIS EXAMINERS

E. Petriu

M. Weiss

J.-M. De Koninck, Ph.D.

LE DOYEN DE LA FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

SIGNATURE

DEAN OF THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

xPheve

An Extensible Physics Engine for Virtual Environments

By
Saeid Nourian

A thesis submitted to the
School of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements for the degree of

Master of Science
In
Computer Science

Ottawa-Carleton Institute for Computer Science

School of Information Technology and Engineering

University of Ottawa

Your file Votre référence
ISBN: 0-494-01566-7
Our file Notre référence
ISBN: 0-494-01566-7

ABSTRACT

Almost any Virtual Reality application requires having a basic set of physical rules and properties such as gravity and collision elasticity. Traditionally, these physical laws were hard-coded within the application, adding an extensive amount of confusing codes that have little to do with the project requirements but more to do with satisfying the basic physics of the scene. Since every VR application would have its own implementation of physical behaviors, we face a problem that is commonly known as the "reinvention of the wheel"!

To avoid the redundancy problem and to adequately separate the application components from the low-level physics components, modern programmers use a generic library called *physics engine*. A physics engine contains within itself all the codes required for dynamic computation of physical behaviors and provides these services to other applications through its application interface (API).

There are several commercial physics engines available for programmers. Although they are computationally efficient, they lack a proper architecture. As a result, they face problems with extensibility and maintenance issues.

In parallel to the above improvement opportunity, there is another industrial consideration. Sun Microsystems introduced a new technology for constructing portable virtual worlds called Java3D. Since Java3D is new, there are no physics engines available for Java3D programmers.

This thesis proposes a new framework for developing virtual environments that obey physical laws. It introduces a new generation of physics engines that are fully customizable and extensible. This is due to a novel architecture that uses a container/plug-in pattern for implementing physical laws and attaching them to the physics engine. The physics engine is named xPheve (Extensible Physics Engine for Virtual Environments) and its current implemented is in Java and on top of Java3D libraries.

ACKNOWLEDGEMENT

This thesis would not be possible without help and support of Dr. Nicolas D. Georganas. I like to take this opportunity and express my gratitude for his invaluable guidance. I also would like to thank my parents for their moral and spiritual support and continuous encouragement. Finally I would like to thank Mojtaba Hosseini, Xiaojun Shen, Shervin Shirmohammady, Francois Malric and the rest of my fellow colleagues in DISCOVER lab.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1 - INTRODUCTION

1.1 Virtual Reality

The field of Virtual Reality has gained more popularity in the recent years due to the earlier accomplishments in the fields of Computer Graphics and Hardware Engineering. Expansion of high quality graphic cards and standardized 3D graphics libraries (such as OpenGL and DirectX) are the two factors that play a major role in achieving the current advancement in computer graphics and animations.

Virtual Reality is more than just complex graphics and dynamic animation. It is rather defined as a very interactive application whose ultimate goal is to provide humans with means to perceive, feel and interact with a computer generated reality.

Virtual Reality is a way for humans to visualize, manipulate and interact with computers and extremely complex data.[1]

Although it is not required that virtual reality worlds always correspond to real world behaviors, most Virtual Reality applications do contain highly interactive environments in which a great deal of real-world dynamics exists. A general goal in such applications is to provide the users with some of the real-world experiences, thus making the artificial environment appear more natural and realistic in their mind. This is precisely what the world "reality" emphasizes.

1.2 Applications of Virtual Reality

There are currently many modern applications that use virtual reality to serve traditional tasks in the fields of commerce, education, medical surgery, entertainment, training and others. With the continuous and rapid advancements in virtual reality technology, many more applications are yet to come. Virtual Reality is in fact a generic tool that provides

[1] ViewTec, the art of virtual reality: *http://www.viewtec.ch/techdiv/vr_info_e.html*

better visualization and easier control, thus can be applied to any application that requires a lot of human-machine interactions.

Let us consider online shopping as practiced today through text-based web pages. Although online shopping saves time and possibly money for its clients, it steals away the feeling and the certainty that was associated with traditional shopping. For example, the customers may no longer be able to inspect the items by picking them up and examining them closely. At best, this experience is replaced by a simple 2D snap-shot of the item. Needless to say, the joy of shopping experience is also greatly reduced in a text-based online shopping.

Now let us try to imagine enhancing the shopping experience of customers by introducing Virtual Reality to online shopping. As shown in Figure 1-1 [1], the perception and feeling of traditional shopping is greatly restored with the aid of Virtual Reality. At a same time, all the benefits of traditional text-base shopping are preserved as additional facilities (such as text-base searching).

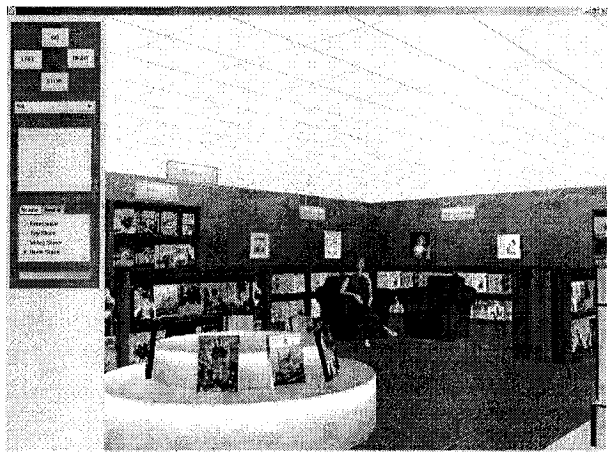


Figure 1-1: A virtual reality book store

Companies, Governments, and research laboratories all over the world have developed different VR applications in different categories. The above example of Virtual Shopping Mall was developed in the DISCOVER Research Lab of the University of Ottawa. This research laboratory is also responsible for the development of many other VR applications in the fields of multimedia, tele-medicine and tele-training ([2] and [3]).

Figure 1-2 demonstrates a snap-shot of VR training program for an ATM Switch Repair procedure [4].

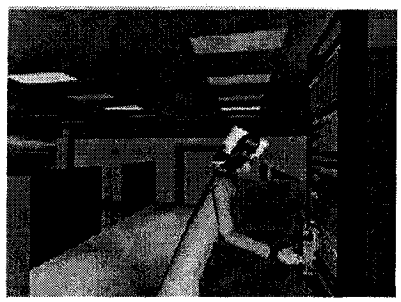


Figure 1-2: Remote training in a virtual environment

1.3 Physics Engine

Almost any virtual reality application needs to adopt certain physical behaviors in order to offer a real-world experience to the users. The human mind is shaped by living in a physical world, thus a virtual world that does not obey the same basic rules as a physical one is not pleasant to the human mind and its reality is quickly objected.

Rather than "re-inventing the wheel" and implementing physical laws in every virtual reality application, modern programmers use a general an application-independent library known as *physics engine*. A physics engine provides the VR applications with a variety of physical behaviors accessible through its application interface (API). The VR application can use the generic API to add some customized physical behaviors to their virtual worlds. Commercial physics engines gained popularity as gaming industries grew rapidly, especially during the late 1990s. HAVOK [8] and KARMA [9] are currently the two most popular physics engines used by game developers. These engines deal with the lower-level requirements such as collisions and gravity, enabling the developers to focus on higher-level issues such as general game scene and storyline.

Some physics engines have their own rendering mechanisms; some are independent of any specific rendering mechanism; others are built on top of well-known rendering

libraries such as OpenGL or DirectX and use their facilities for rendering requirements. The engines that have their own rendering mechanisms and geometry structures may achieve better performance because the rendering is optimized with the engine and vise versa. However, because these engines are specific to their own (usually non-standard) rendering facility, they cannot easily apply physics to scenes generated by other libraries.

The engines that are completely independent from the rendering mechanisms are easy to integrate with scenes generated with different libraries. However, because they are general, they cannot make use of the performance tune-up available in specific rendering libraries.

The engines that are constructed on top of the well-known rendering libraries have the advantage of using the power of the rendering library to optimize the performance. Also, because such standards are widely used, the problem of compatibility with other rendering tools is less critical. Such physics engines are in fact extending the current features of these libraries to add new features for physical behaviors. Both the rendering library and physics engine can be updated independently from each other as long as their API does not change dramatically.

Considering the above advantages, we decided to create a physics engine which will extend a well-known standardized 3D Graphics library. Although OpenGL has been the most popular standard library for developing interactive 3D Graphics, it is considered low-level and has its own disadvantages. The core implementation of OpenGL is procedure-based, as supposed to Object-Oriented, and thus it suffers from the usual extensibility and maintenance problems, as seen in most procedure-based programs. There is little or no structure in OpenGL scenes. There are also many issues that need to be dealt with when exporting OpenGL products from one platform to another.

In section 1.6, Java3D, which is a new 3D graphics technology introduced by Sun Microsystems, will be introduced along with a summary of its various benefits.

1.4 Existing Physics Engines

As it was mentioned in the previous section, HAVOK and KARMA are two commercial physics engines currently used in many VR applications especially for 3D gaming. Both HAVOK and KARMA are procedure-based. As a result, they lack the degree of extensibility and maintainability present in the object-oriented systems. The first step toward the design of a flexible architecture is by using the notion of inheritance and reusable components found in object-oriented languages. Both engines are primarily used for gaming; thus their design is optimized for fast rendering and interactions. Due to unavailability of technical details of the commercial products this thesis cannot elaborate much on the architectural design of HAVOK nor KARMA. It is however apparent from the API of these products that they are faced with usability and extensibility problems.

There has been some work toward implementing object-oriented based physics engine. One such work is an open source project called NOOPE (The Newtonian Object-Oriented Physics Engine). NOOPE has a simple architecture which consists of three main components: Physics, Entity and Law. Entities represent the physical objects in the scene. Each entity is associated with a list of properties. Law components consist of some codes that add forces to entities based on some computations. The Physics component maintains a list of entities and laws. Figure 1-3 shows the overall architecture of NOOPE.

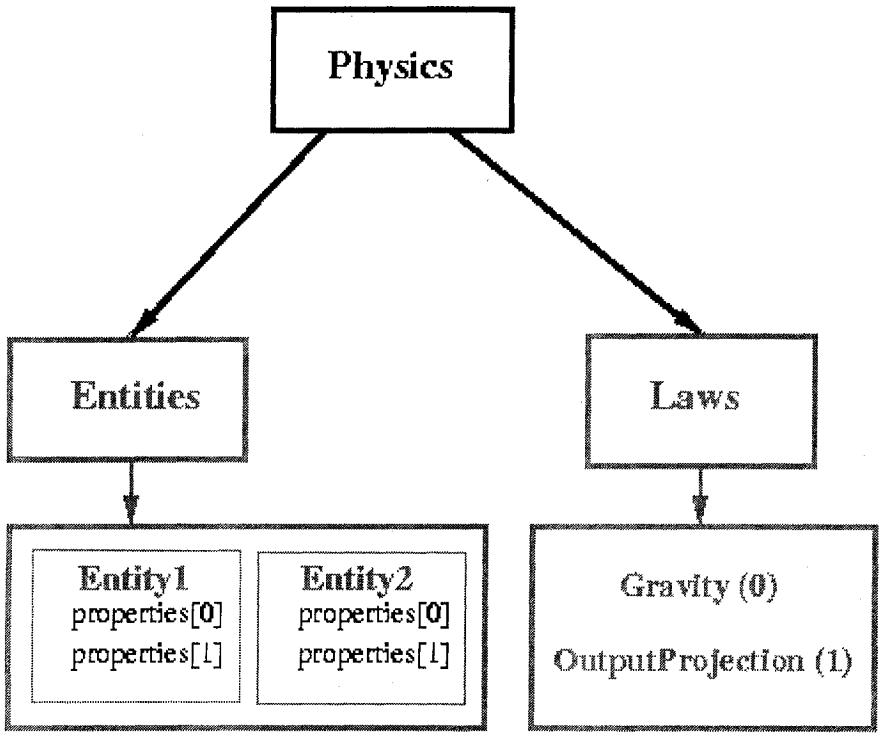


Figure 1-3: Architecture of NOOPE

Although NOOPE organizes Entities and Laws within an object-oriented structure, the flow of control is quite limited in this physics engine because at each update period all laws are activated even if a particular law needs to be activate only once or twice during the simulation. Needless to say that this unnecessary execution of laws causes considerable performance problems.

1.5 Java

The compatibility issues from platform to platform have been the major concern of application programmers for decades. Higher-level programming languages such as C and C++ are too much dependant on low-level system calls thus binding programs to specific platforms and making platform migration as difficult as reprogramming the entire application.

In 1995, Java arrived with a beautiful promise: platform independence. The language itself is high-level and object-oriented with a syntax inspired by C++. Underneath however, there is a middle layer responsible for providing system-call functions. The

middle layer is called Virtual Machine (VM), which is the medium between the operating system and Java applications. There are different implementations of VM for different platforms and so in order to be able to execute Java programs, it is required that the proper VM be installed on the machine. Once VM is properly placed in the system, any Java programs, no matter which platform they were originally developed in, can be executed without need to reprogram. Therefore, with the aid of VM and due to the fortunate fact that VM interface is standardized by Sun Microsystems, the dream of "write once, run anywhere" seems to have become true. Java also offers sophisticated yet easy to use network facilities. All these have converted Java to one of the most popular programming languages in the history of computer software.

It should be noted, that Java does allow direct system calls to specific operating systems, in which case the particular Java program loses its platform-independence and becomes what is known as a non-pure Java program. The benefits of VM are nevertheless there and whether an application gains from it or not depends on the developers.

1.6 Java3D

The Java 3D API is an extension to the Java Core API that consists of a hierarchy of Java classes which serve as the interface to a sophisticated three-dimensional graphics rendering. [5]

Java3D is a high-level API for generating scene graphs. It is built on top of a lower-level graphics generator (OpenGL or DirectX) and uses the lower-level API to communicate with graphics devices and take advantage of the 3D acceleration hardware. Thus Java3D provides facilities to develop scene graphs that are platform independent, yet have good rendering performance. Since its introduction in 1998, Java3D has gained noticeable popularity among application developers.

Java3D had three main goals when it was first introduced by Sun Microsystems:

- Provide a rich set of features for creating interesting 3D worlds, tempered by the need to avoid nonessential or obscure features. Features that could be layered on top of Java 3D were not included.

- Provide a high-level object-oriented programming paradigm that enables developers to deploy sophisticated applications and applets rapidly.

- Provide support for runtime loaders. This allows Java 3D to accommodate a wide variety of file formats, such as vendor-specific CAD formats, interchange formats, VRML 1.0, and VRML 2.0.

Unlike OpenGL, there is a clean high-level structure to Java3D scene graphs. It is based on a hierarchical structure which consists of a tree with many nodes. The leaves of the tree represent the geometry components of the scene. The nodes in middle and top layers represent the relations between the components and their position/orientation in the 3D world. This structure is implemented in an Object-Oriented form, with each node being represented by an object.

Java3D is an automated rendering library that optimizes the scene tree to get the best run-time performance. For this purpose, Java3D may decide to automatically combine some geometry components into one, or shrink the height of the scene tree by eliminating unnecessary nodes. Thus, Java3D programmers have less to worry about the structure of their scene graphs and its performance issues than those who use OpenGL.

The above factors make Java3D a suitable API for developing high-level scene graphs because the developer can focus more on the main functionality of their applications rather than low-level rendering issues.

1.7 Hierarchical Structure of Java3D Scenes

The 3-dimentional scenes in Java3D are structured in a tree hierarchy. There is a root node on the top. Every node may have one or more children, except the leaf nodes at the bottom of the tree. The basic rule is that each node may only have one parent (except the

root which has no parent). Figure 1-4 shows an example of a Java3D tree structure [6]. In addition to parent-child relationship between the nodes of a tree, some nodes may also have 'references' to other nodes.

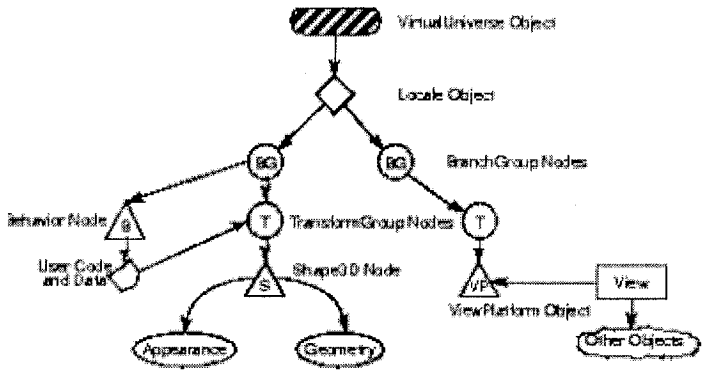


Figure 1-4: Hierarchical scene structure in Java3D

Every scene graph in Java3D starts with an instance of VirtualUniverse as a root node. This is the node that will contain the entire scene graph and any 3D object that needs to be displayed must be connected to VirtualUniverse. Although Java3D permits creation of more than one VirtualUniverse instance, the majority of applications require only one.

VirtualUniverse can have one or more Locale objects as its children nodes. Local objects define the origin of their attached branches. In this example, there is only one Local object with origin of (0,0,0).

Local objects can have one or more BranchGroup nodes as children. BranchGroups are nodes that group the parts of the scene graph in sub-graphs (also called *branch graph*). BranchGroups are especially important because they are the only nodes that can be dynamically added to the tree after the VitualUniverse becomes live (scene graphs that Java3D has started rendering are said to be *alive*). BranchGroups may have children of any type.

Translation and orientation of scene graphs in Java3D is specified within TransformGroup nodes. TransformGroups typically have reference to a Transform3D object that contains the transformation matrix. The entire sub-graph rooted at TransformGroup is affected by the manipulating the transformation matrix of TransformGroup.

BranchGroups and TranformGroups may have other BranchGroups and transformGroups as their children. However somewhere down the tree, the branch must end with a Leaf object. Leaves are nodes that cannot have any children. In Figure 1-4, leaves are shown in the form of triangles. Shape3D, Behavior and ViewPlatform are three examples are Leaf object as shown in the figure.

Shape3D nodes represent the geometrical 3D objects that we see in a scene graph. They each have reference to a Geometry object, which describes the geometry of shape, and an Appearance object, which describe the appearance attributes such as color and transparency.

Behaviors provide facilities needed for animations and interactions. Java3D has built-in mechanisms to trigger behaviors when certain condition(s) are met. Once triggered, behaviors may manipulate the attributes associated with other nodes as a response to the action that triggered the condition. In the example shown in Figure 1-4, the behavior manipulates the transformation of a TransformGroup.

The ViewPlatform can be thought of as the camera of our virtual world. Depending on the position and orientation of TransformGroup object(s) above ViewPlatform, a portion of scene graph is captured, clipped and rendered to the screen. ViewPlatform also has a reference to View (not shown in Figure 1-4) which contains the parameters needed for rendering the scene.

1.8 Java3D as the basis of Physics Engine

As it was discussed in previous sections, Java3D has many advantages over its preceding alternatives. Among its various important features, the top five are the ease of extensibility, platform-independence, object-oriented style, automated performance tune-up and built-in behavioral framework. Java3D lacks some of the benefits of its widely-used predecessors however. This is due to the fact that Java3D is a new technology thus less visible in the market of commercialized tools. For instance, to this date there has

been no attempt to develop a Java3D-based physics engine[2]. The lack of physics engine, as discussed before, will result in the problem of 'reinventing the wheel', and in today's competitive market is unacceptable to VR industries.

Developing a physics engine which is designed purely for Java3D is a major step toward boosting up the popularity of Java3D among VR and gaming industries. In addition to the necessity of having a physics engine, there are also some properties of Java3D that encourage the extension of core functionality to cover more advanced physical behaviors. This is due to the highly flexible and extensible Object-Oriented architecture that exists in Java3D. Transform-grouping and behavioral framework are two examples of Java3D features that can facilitate the development of a physics engine.

TransformGroups in Java3D, as defined in previous section, are objects that contain information about the location and orientation of the geometrical shapes in 3D space. In our physics engine, they can be used to store the location and orientation of physical objects in the physical scene. We can also extend TransformGroup to include additional information such as velocity and force.

There is an interesting resemblance between Java3D's Behaviors and the physical laws in a physics engine. A law can be represented by a Behavior in Java3D, because just like behaviors, laws are actions that are triggered when specifics events occur. Gravity law, for instance, applies gravity force to all objects when the physics engine is activated. Collision law, on the other hand, manipulates velocity vectors when a collision between two objects is detected. Therefore behaviors in Java3D can be extended to cover physical laws that fit nicely in the standard architecture of Java3D.

The well-designed hierarchical structure is yet another reason why Java3D is a good candidate to be the basis of a physics engine. As an example, consider a car as a whole in

[2] As explained before, the closest related work found was an open-source physics engine project called NOOPE (The Newtonian Object-oriented Physics Engine) in an effort to implement a generic physics engine in Java [7]. However no Java3D API is used in this project, and although it is possible to use the named engine with Java3D, in my opinion it requires a great deal of extra programming with a run-time overhead that will result in a performance decline.

a graph scene. Although the car consists of many smaller parts such as wheels and doors, in a simple VR application all these different parts have the same velocity and the same position relative to each other. Thus, a good approach would to group them together and assign one velocity and one position to the group, as a whole rather than to individual parts. This is precisely what the tree structure of Java3D can accomplish.

1.9 Problem Statement

Most available physics engines lack a proper architectural design and are faced with extensibility and maintainability problems. There is a need for a flexible physics engine that can automate the task of enforcing physical laws, yet be fully customizable.

In addition Java3D, a new Java-based library for creating 3D scene graphs, lacks a few critical tools needed in order to be acceptable industrial-wise. One such missing tool is a physics engine which is commonly used by VR and gaming industries for applying basic physical behaviors such as gravity and collision avoidance to their virtual worlds. This thesis contributes to a solution for both of the above problems.

1.10 Thesis Contributions

This thesis proposes to design and develop a framework for a Java3D physics engine that enforces physical laws upon the objects in a virtual scene. It is proposed that the physics engine extend the core Java3D libraries and follow the same structure and framework that govern the Java3D functionalities. This is to ensure that the physics engine can be easily integrated with any scene graph constructed by Java3D.

In addition to the above general goal, the following non-functional objectives are also pursued during development of the physics engine and become the contributions of this thesis:

- Object-Oriented Architecture: Object-oriented (OO) programming is known to be the most effective coding style which satisfies the main principles of proper

software development. Most existing physics engines are procedure-based; therefore developing a physics engine in OO style is a relatively new approach.

- Pure Java (therefore cross-platform): As discussed before, there are many advantages to Java in comparison with other programming languages. Java is Cross Platform for instance, because it runs on top of Java Virtual Machine.

- Use Java3D technology: As discussed before, there are many advantages to Java3D in comparison with other rendering libraries such as OpenGL. Java3D is a high-level API that hides the low-level details of rendering graphics. Its structure is very suitable for developing a physics engine.

- Easily maintainable: It must be easy to integrate the physics engine with existing VR application. It should also be easy to adjust the physics engine to corresponding to the static/dynamic changes in the scenes.

- Easily extensible: New physical laws and custom behaviors must be easy to define and implement in order to extend or modify the functionality of the physics engine.

- Acceptable performance: The rendering performance of Java3D must not decrease when the physics engine is activated. For this purpose, it is important that the CPU and memory usage of the physics engine is kept low at all times.

A physics engine is a library of complex computations and object interactions. If it is not designed well, the physics engine can easily lose performance, create ugly ad-hoc code or just be impossible to extend any further. The above requirements make it clear that the goal of this research project is not to have a fully-functional physics engine, but rather to have a well-designed and expandable one.

The thesis is organized as follows. In Chapter 2, we describe how to represent physical objects and laws in an object-oriented manner. Chapter 3 presents more details on the object-oriented architecture of the physics engine including a container/plug-in structure that is used for representing physical laws. Chapter 4 consists of detailed examples of

implementing laws using the container/plug-in structure. Chapter 5 starts with an overview of the industrial benefits that is accompanied with using this physics engine. It then explains some of the technical and business issues behind integration of this physics engine with VR applications. Finally, Chapter 5 presents our conclusions.

CHAPTER 2 - BASIC ELEMENTS

There are infinitely many physical elements embedded in the physical nature surrounding us. Some are as primitive as mass and velocity in rigid bodies; others can be as complex as the electrostatic charges in the sub-atomic particles. The primitive elements are more evident to the conscious human mind, thus those elements must exist in almost any type of Virtual Reality applications. This chapter will explain how the basic elements of physics can be incorporated in an object-oriented structure.

2.1 Position and Orientation

The most fundamental attributes of geometrical objects in a 3D world are their location (x, y, z) and orientation (d_x, d_y, d_z, angle). The physics engine is responsible for continuously updating the location and orientation of objects in order to correspond to the physical attributes that are affecting the object (e.g.,. net force).

Java3D already contains a data structure for maintaining the position and orientation of geometries within a scene. These data are stored in 4-by-4 matrices in order to maximize efficiency and rendering performance. Each 4-by-4 matrix is used to represent three pieces of information: position, orientation and scale (for this application we do not need the scale factor). The computations involved in finding the matrix values are complex, however, thanks to the Transform3D class included in the Java3D library, we do not need to worry about them. The API provided by Transform3D receives the position and orientation values separately and automatically generates the corresponding 4-by-4 matrix (Figure 2-1).

Transform3D Methods (partial list)

Transform3D objects represent geometric transformations such as rotation, translation, and scaling. Transform3D is one of the few classes not directly used in any scene graph. The transformations represented by a Transform3D object are used to create TransformGroup objects that are used in scene graphs.

`void rotX(double angle)`
Sets the value of this transform to a counter clockwise rotation about the x-axis. The angle is specified in radians.

`void rotY(double angle)`
Sets the value of this transform to a counter clockwise rotation about the y-axis. The angle is specified in radians.

`void rotZ(double angle)`
Sets the value of this transform to a counter clockwise rotation about the z-axis. The angle is specified in radians.

`void set(Vector3f translate)`
Sets the translational value of this matrix to the Vector3f parameter values, and sets the other components of the matrix as if this transform were an identity matrix.

Figure 2-1: Transform3D class in Java3D provides a simple API for generating complex transformation matrices.

2.2 Grouping Geometries

It makes sense to group geometry components together and treat them as a whole. For example, a car object may have a body component, and four wheel components. The body component itself may be consisted of smaller components such as doors and lights. We could treat each small component as a separate physical object; however that would make the system too complicated. Instead, it is common practice to group them together in a single physical object and assign the physical attributes to the whole object. For instance, we can assign one velocity to the car as a whole because in a moving car all the sub-components have the same velocity (Figure 2-2).

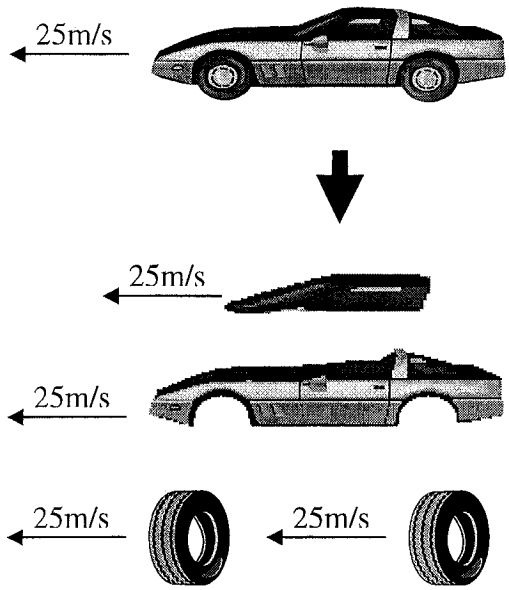


Figure 2-2: Most VR applications do not require treating every small component as a separate physical object. In a simple car, all parts move together, rotate together and behave as a whole.

In order to apply the above concept to the scene geometries we need a mechanism for grouping objects in such a way that their relative position and orientation stays the same while the position and orientation of the object as a whole is variable. Java3D's TransformGroup meets all these requirements. A TransformGroup is a node object that can have one or more children nodes. Each TransformGroup has its own position/orientation matrix (Tranform3D). When the position or orientation of a TransformGroup changes, all of the nodes underneath it are also translated, therefore the entire sub-tree translates and rotates together. Thus, we can use the TransformGroup to group a car into one whole component, as shown in Figure 2-3.

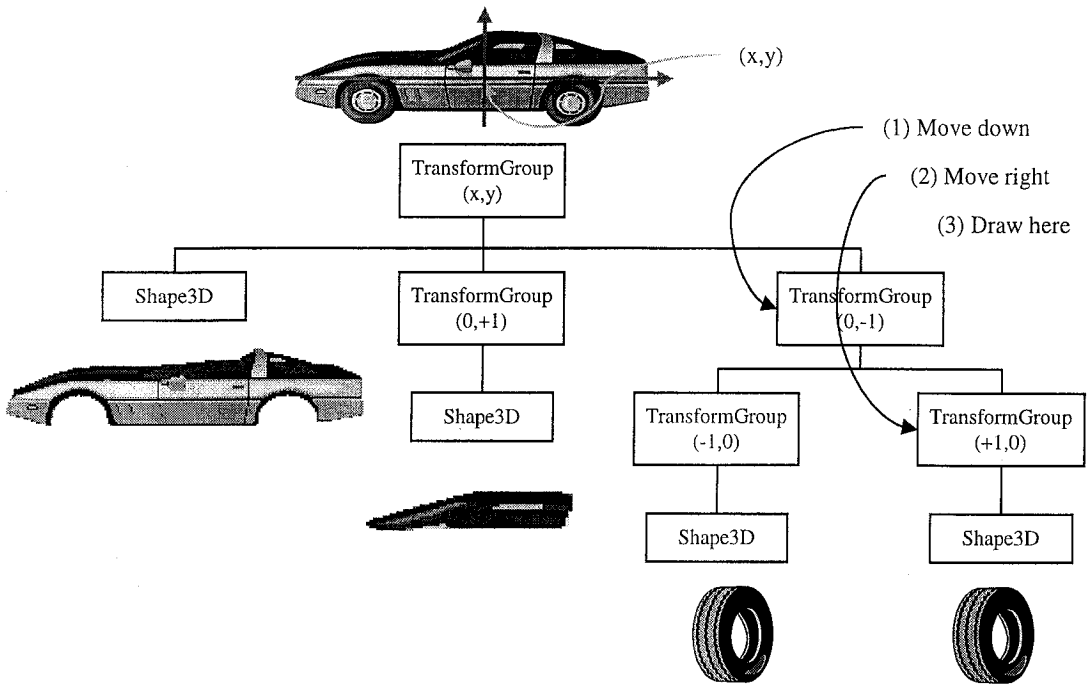


Figure 2-3: The grouping mechanism in Java3D facilitates the task of representing complex objects in a hierarchical structure.

2.3 Physical Object Representation

A physical object represents an entity consisting of one or more geometry components, all of which have the same physical behavior (they all move together and rotate together). In the real world, every molecule or atom is one physical object that constantly interacts with its surrounding environment. This representation is practically impossible, thus in a typical VR application complex objects such as a car or a person may be represented by one or a few physical objects in order to reduce the weight of computations.

As it was discussed before, TransformGroups in Java3D facilitate the task of grouping geometries together. Also every TransformGroup contains a Transform3D object for representing the position and orientation of the group as a whole. Therefore TransformGroup is a good candidate for representing a physical object. Figure 2-4 shows how TransformGroup can be extended to a Physical Object that represents a car.

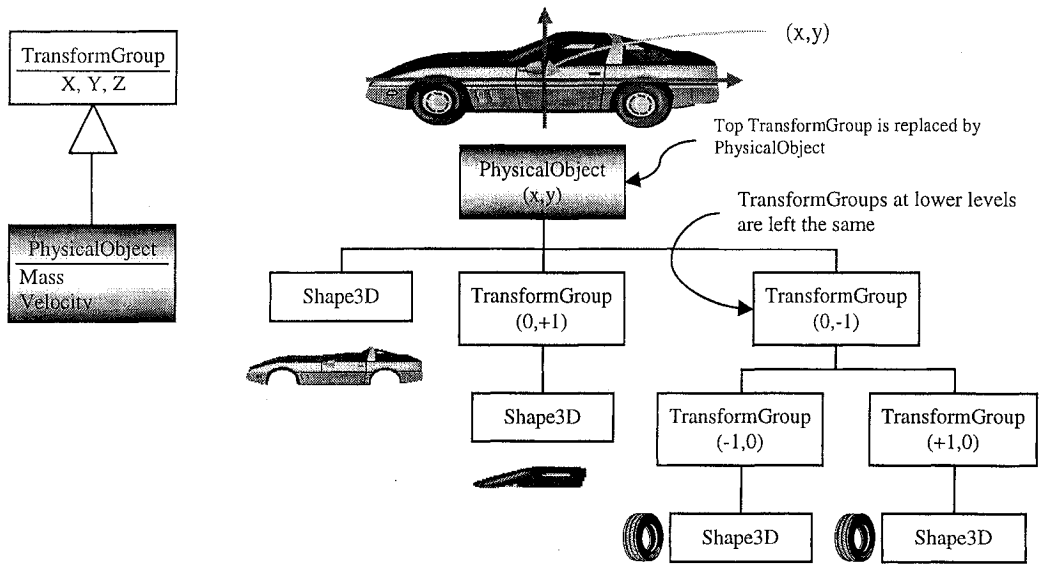


Figure 2-4: A group of objects can be represented as one physical object.

2.4 Mass, Force and Movement

The movement of objects in a physical environment depends on the various forces that are affecting them. Newtonian laws are used at this stage to decide which basic physical attributes must be included in the Physical Object specification. In particular, the following equations denote that *force* and mass are the basis of movement:

$F = m \cdot a$ where F = force, m = mass, a = acceleration

$a = F / m$

For a given mass and net force, the position of a physical object can be computed at every update period Δt, as shown below:

$a = F / m$

$v_{new} = v_{current} + a.\Delta t$ where $v_{current}$ is the current velocity and v_{new} is the new velocity

$v_{avg} = (v_{current}+v_{new}) / 2$ where v_{avg} is the average velocity during Δt

$p_{new} = p_{current} + v_{avg}.\Delta t$ where $p_{current}$ is the current position and p_{new} is the new position

One of the main responsibilities of a physics engine is to continuously update the position of the physical objects based on the forces that are affecting them. The value of Δt is

critical in simulating a realistic physical behavior. Δt should contain the number of milliseconds between *now* (current time $t_{current}$) and the *last update time* (last time when the position of object p was updated: $t_{lastUpdate}$). Figure 2-5 shows an example of a simple time line used for updating the position of physical objects.

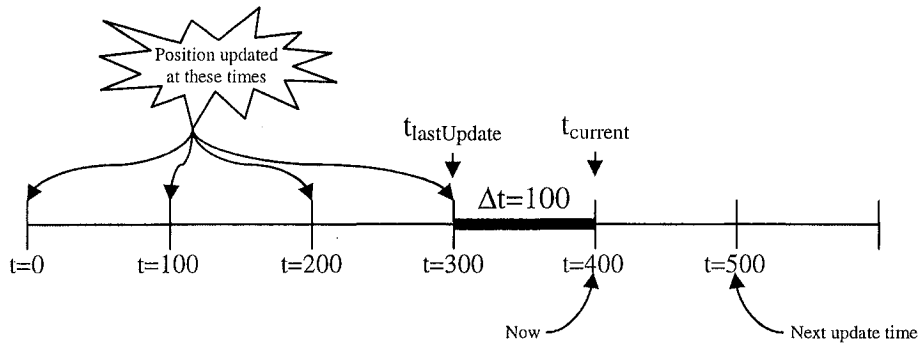


Figure 2-5: Update time-line with regular intervals.

The model in Figure 2-5 is unrealistic because in most operating systems it is impossible to run an update in fixed intervals, especially in Java where the Garbage Collector may suspend the thread execution while it is releasing the unused memory back to the operating system. Figure 2-6 shows a more realistic time-line diagram.

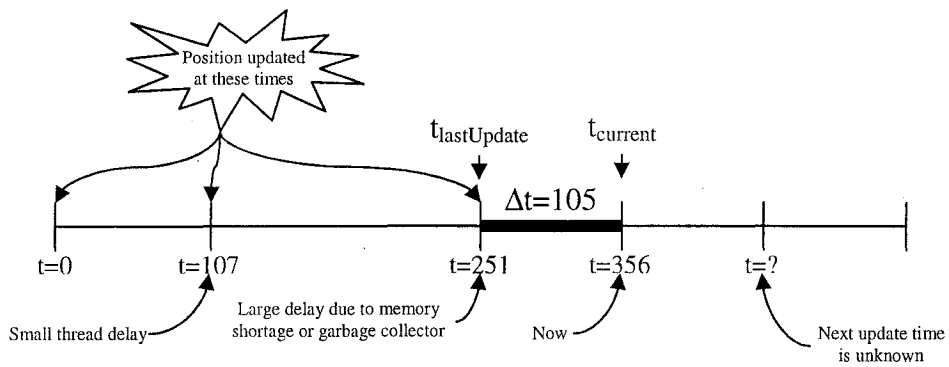


Figure 2-6: Update time-line with irregular intervals.

To account for these variations in update time, the following simple formula is used for calculating Δt:

$$\Delta t = t_{current} - t_{lastUpdate}$$

Therefore the physics engine must store not only the current time but also the time of last update for every physical object in the scene. Current time can be retrieved from the operating system. After an update, the current time is stored in a local variable of the updated physical object to be used as $t_{lastUpdate}$ in the next round.

2.5 Forces

At a given time several forces may be acting together on a single physical object. In order to calculate the acceleration, the net force is computed first. There are two important attributes associated with every force:

1. Direction and magnitude
2. Duration (in milliseconds)

The second attribute is required because different forces usually have different durations. A force applied as a result of a collision lasts a few milliseconds only whereas the gravity force is applied indefinitely (duration is infinite).

Since every force has its own duration time, we cannot simply add them together in order to compute the net force. Therefore every physical object must maintain a data structure to store a list of all forces that are currently being applied on it (Figure 2-7).

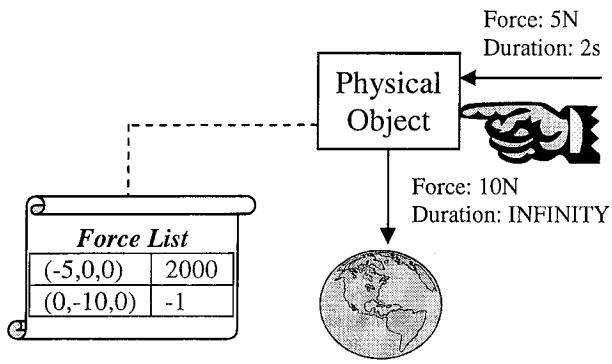


Figure 2-7: At any given time several forces with different magnitudes and directions may affect a given physical object.

Figure 2-7 shows the format of force data structure within the physical objects. To speed up the computations, the magnitude and direction are combined together in one Vector object. The unit for duration is in milliseconds and -1 is used to represent INFINITY.

Now that every physical object has its own list of affecting forces, the net force must be calculated at every update period. The basic rule is that at every update period a given force can affect the physical object for duration of Δt at most:

(1) if force.duration < Δt	forceEffect = force * force.duration; remove force
(2) if force.duration = Δt	forceEffect = force * Δt; remove force
(3) if force.duration > Δt	forceEffect = force * Δt; force.duration = force.duration - Δt

Conditions (1) and (2) multiply the force by MIN(force.duration, Δt) and then remove the force because after this update the force duration will be zero, thus no longer necessary. Condition (3) however multiplies the force by Δt even though the force duration exceeds Δt. This is because Δt denotes the present time (present update) and, if the force effect exceeds Δt, that will alter the order of physical events by collapsing future events into present time. In order to re-schedule the remaining effect of the force, the duration of force is updated (decreased by Δt) and the force is allowed to stay in the list until the next update period.

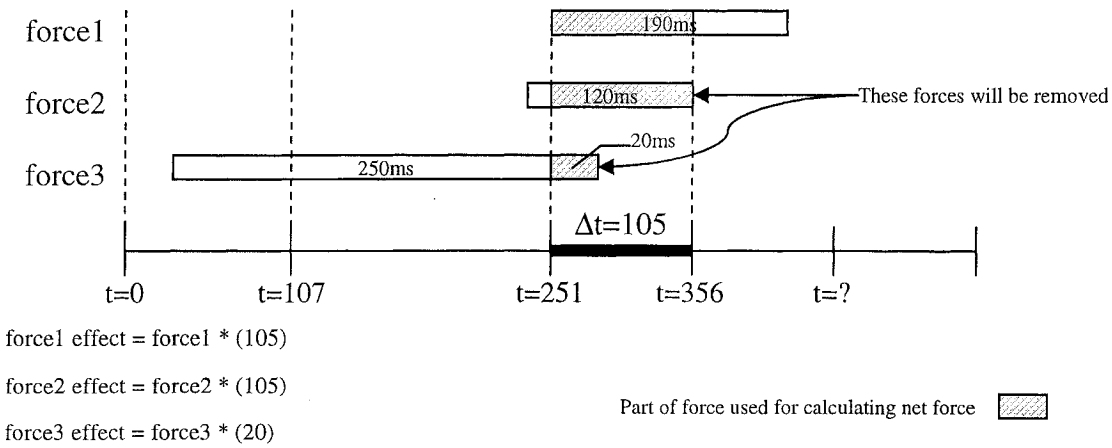


Figure 2-8: From time aspect, there are three categories of forces: (1) Forces that last for Δt and continue to exist; (2) Forces that last for Δt but end; (3) Forces that exist for less than Δt.

Since the calculated net force is the sum of the forces multiplied by their duration, the unit is in Newtons. Time (N.ms) instead of just Newtons (N). To account for that, the computation set in section 2.4 is rearranged as follows:

$a_t = F_t / m$	where F_t is net force multiplied by duration (F. Δt) and a_t is a.Δt
$v_{new} = v_{current} + a_t$	where $v_{current}$ is the current velocity and v_{new} is the new velocity
$v_{avg} = (v_{current}+v_{new}) / 2$	where v_{avg} is the average velocity during Δt
$p_{new} = p_{current} + v_{avg}.\Delta t$	where $p_{current}$ is the current position and p_{new} is the new position

In summary, the physics engine periodically updates the position of each physical object in the scene by calculating the net force (multiplied by Δt) and executing the above sequence of computations. Once the value of p_{new} is computed, a public method of TransformGroup is used to set the new position, thus physically moving the object to its new location in the scene.

2.5.1 Force ID

In many applications it is useful to be able to distinguish forces by assigning them a unique id. An example of a scenario where such feature is proved useful is a car racing application in which the driven car accelerates forward by applying a forward force. Since the duration of acceleration depends on the duration in which the up arrow key is being held down, the most convenient solution would be to apply a force with infinite duration when the arrow key is pressed and remove that force as soon as the key is released.

In order to remove a particular force, it must be uniquely identified. A simple yet efficient technique is to use a static counter to keep track of the next available unique id. The counter is initially set to 0 and is incremented every time a client requests a new unique id. The following pseudo-code demonstrates how this technique is conveniently used in a car racing application:

```
Initialization:
        PhysicalObject car ← new Car()
        forwardForceID ← Force.getNextAvailableID()

Method onKeyPressed:
        Force forwardForce ← new Force(forwardDirection, infinitDuration, forwardForceID)
        car.addForce(forwardForce)

Method onKeyReleased:
        car.removeForce(forwardForceID)
```

Figure 2-9: A Pseudo-code that demonstrates how force id can be used in an application.

A client may use a single force id to identify and group multiple forces together. In this case, a call to removeForce(id) would remove all forces with that id.

2.6 Physical Law Representation

Simply put, Physical Law is a component that, upon activation, modifies the attributes of one or more physical objects. Gravity is a physical law that applies a downward force to every physical object in the scene with a magnitude such that the objects would have a downward acceleration of $9.8m/s^2$. This law must be activated in the beginning of the simulation and also every time a new object is added to the physics engine.

Collision is another example of a physical law that must be handled. A simple version of the collision law is a component that reverses the velocity every time an object collides with another. In this case, the law is activated whenever a collision occurs.

From the above examples, it can be observed that physical laws behave similar to an event-driven system where events activate the laws, and the laws enforce themselves upon activation (Figure 2-10). But also there is concurrency involved, as all laws must be obeyed at all times. In Java3D both concurrency and event-driven patterns are combined together in the form of *Behaviors*.

Behaviors in Java3D are classes whose instances are activated when a certain condition is triggered. The condition can be one that is triggered after a certain number of milliseconds have elapsed, or a certain key is pressed, or a collision is occurred. All these conditions are monitored by Java3D's internal thread thus the effect is concurrent. In

addition to the convenience of use, there are some optimizations that Java3D performs on Behaviors in order to improve performance.

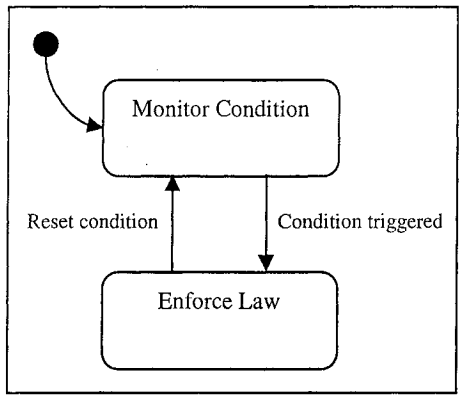


Figure 2-10: Behaviors in Java3D are built-in state machines with two simple states. Physical Laws can extend the functionalities of the behaviors as demonstrated in this state diagram.

Behaviors are therefore suitable representations for physical laws. In other words, physical laws can extend the functionalities of the behaviors to make them specific to physical behaviors. Figure 2-11 shows how the Behavior's abstract methods are implemented in PhysicalLaw class. Additionally, a new abstract method named enforce is introduced in the PhysicalLaw class. This method must be implemented in the subclasses of PhysicalLaw. GravityLaw, for instance, should include within its enforce method the code for applying a downward force to all physical objects in the scene.

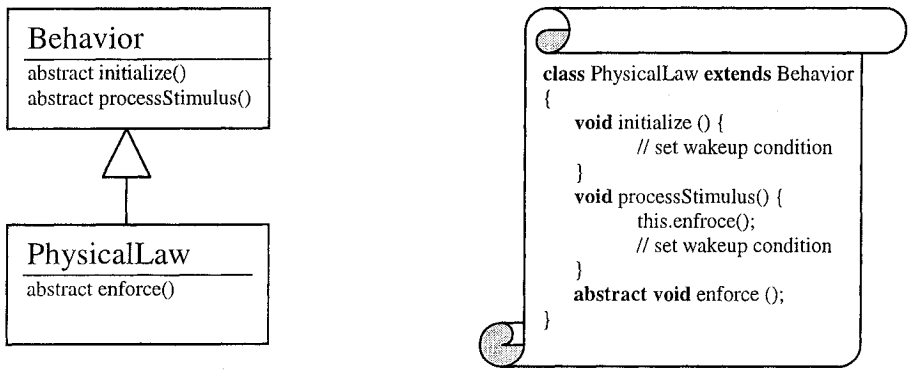


Figure 2-11: The codes for specific laws can be implemented in enforce() method. PhysicalLaw class assures that the enforce method is called whenever the given condition is triggered.

In chapter 3 the physical laws are further decomposed into two components for reasons of extensibility and to decouple the code for detecting physical events (such as collision) from the code for enforcing laws (such as calculating new velocity).

2.7 Physics Engine Representation

Although generally speaking the phrase physics engine refers to the entire system, from a technical point it is the name of the specific component responsible for the periodic update of physical objects in the scene. It is therefore another concurrent thread that runs in parallel with the physical laws.

The physics engine component periodically commands the physical objects to update their position (and orientation) in the scene. As it was mentioned before, physical objects can update their position according to the forces that are affecting them and Δt. The time of the physical world is managed by the physics engine component and is sent to the physical objects everytime an update is scheduled.

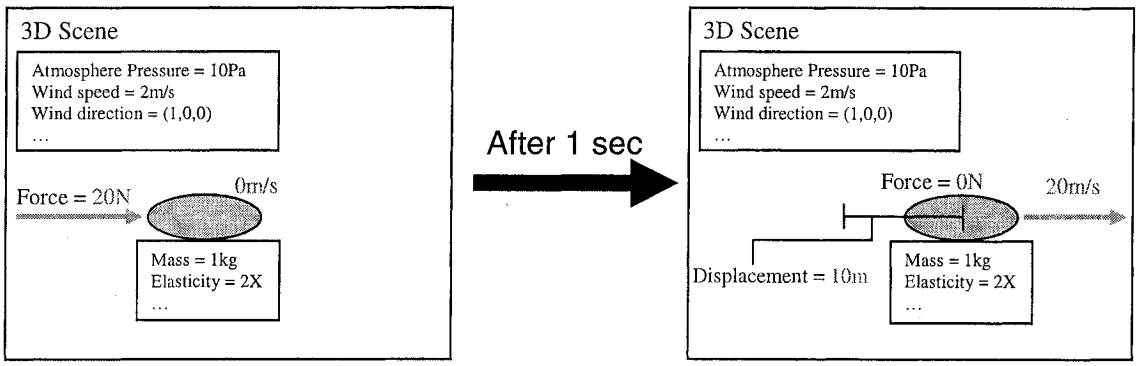


Figure 2-12: Effect of applying a 20N force for the duration of 1s

Figure 2-12 demonstrates a physical world in which the physics engine is set to update the position of objects every second. After the 1st second, the velocity of the object is increased from 0 to 20m/s and the object is translated corresponding to the average velocity at that duration (v_{avg}=10m/s, therefore p=10m).

An alternative for representing the engine is a Java thread. The advantages are that it can update the physical objects concurrently without putting too much load on the CPU.

Figure 2-13 shows how a physics engine that extends a Java thread can periodically update the physical objects in the scene. In chapter 3, we will discuss some of the disadvantages of using Thread representation and will propose a better model.

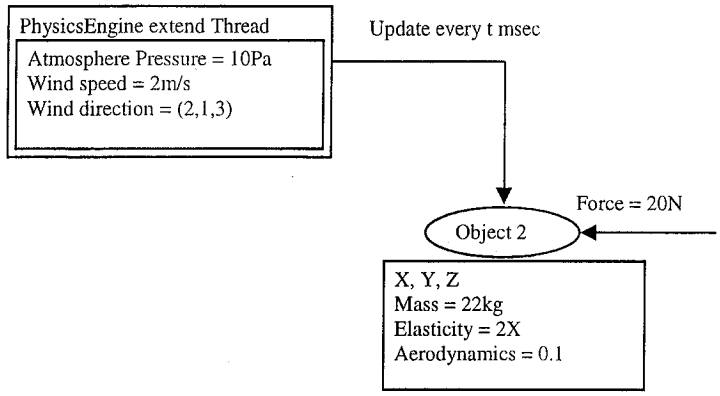


Figure 2-13: The PhysicsEngine component is the main controller responsible for continuous and periodic update of physical objects in the scene.

2.8 Control Flow Overview

This section explains how the different components of the physics engine interact with each other. Once the geometries of the virtual world are constructed the following simple steps can apply physical laws to the scene:

1. Construct one physics engine component.
2. Construct the physical laws that must be enforced.
3. For every geometrical object that must obey the physical laws, construct a Physical Object and add the geometry as its child.
4. Add the physical laws to the physics engine.
5. Add the physical objects to the physics engine.
6. Start the physics engine.

The order of some steps does not need to be exactly as above. Once the engine is started, a periodic and continuous update process will take place until the engine is either paused or terminated. At every update period the current system time is retrieved from the operating system and the update(t) method of each physical object is called. Then the engine goes to idle state until next update period arrives.

Since the calls to the update method are synchronous, control is handed to the PhysicalObject component every time an update is in progress. The update method computes and updates the translation of its corresponding physical object based on the computations explained in section 2.5. As a result, the geometry of the physical object will be rendered at its new location in the next frame. Control is then returned to the engine and the same procedure is repeated until the update method of all physical objects in the scene is called; after that, the engine goes to idle state until the next update period.

In parallel to the above execution sequence, one or more physical laws may get activated and perform some computations of their own. Most laws, such as the collision law, will modify the physical attributes of the physical objects. If this occurs, then the effects (such as change in velocity) will become apparent in the next update period, when the modified variable is used for calculating the new object translation.

Application clients are another type of entities that may manually modify the physical attributes by using the provided API. This is also done in parallel with the rest of the system. In a car racing for instance, pressing the up arrow key may be captured by the application, from which a forward force is applied to the physical object that represents a car.

The application can also use the API to dynamically add or remove physical objects or laws, or temporarily disable/enable the laws that are already added. The entire engine can be paused/resumed as well.

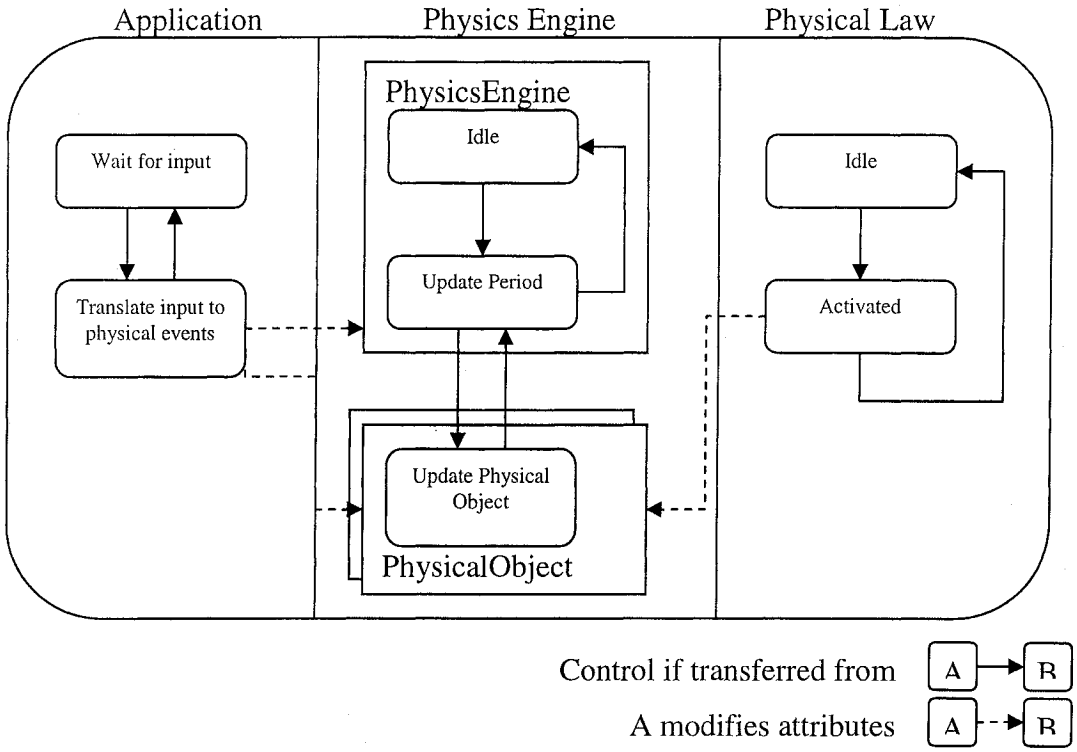


Figure 2-14: State diagram of events and control flow in the proposed physics engine.

The above state diagram represents a multi-threaded system. There are three threads in the diagram that are running in parallel. The PhysicsEngine thread must update the physical objects within regular intervals. It is therefore more critical than the others, since delays in the update process cause jitters which are quite visible to human eyes. The other two threads are less critical from the time point of view, as long as the delays are within certain limits. Take the collisions for example; if there is a small delay before the collision law is activated, the objects may move slightly inside each other, but the delay would not be long enough for the human eye to detect it (or it may seem that the objects deformed slightly and that is not unusual in real world).

It must be noted that this architecture assumes that all laws have same priority thus there are no measures taken to control the order in which laws are executed. For critical systems where some laws must be precedence over the others, additional steps must be taken to control the execution sequence of the physical laws. Also, this architecture allows execution of conflicting laws. In other words, it is possible that one law pushes a

physical object to right while another law is pulling it to left. It is the responsibility of the developers to make sure that no two conflicting laws act on a same object.

2.9 Summary

Three basic elements that are included in the physics engine:

- Physical Objects: representation of objects that are affected by physical laws.
- Physical Laws: components that contain mathematical equations for enforcing laws.
- Physics Engine: main controller that updates physical objects periodically.

It was discussed how some of the features included in Java3D can help in construction of the above components. In particular, TransformGroups are used to represent physical objects and Behaviors are used for activating physical laws.

The basis of movements in this physics engine is *force* and there is a data structure to store the list of all forces that are affecting the physical object at current time. The system needs to have a reliable technique for tracking time. The technique used in this physics engine considers the time variations within update periods and adjusts for that by applying a simple formula.

The system that governs the physics engine functionalities is based on a multi-threaded architecture. The PhysicsEngine component is the main thread that must update the physical objects within regular intervals. There is a thread that governs the physical laws and activates them when necessarily. The third component is the VR application that is using the physics engine to apply physical laws to its geometrical objects.

CHAPTER 3 - GENERAL ARCHITECTURE

The architectural design of a system is often based on not only the functional but also the non-functional requirements of that system. Most physics engine focus on the non-functional requirements that have visible effect on the end product. Performance and scalability are two examples of such requirements. In the design of our physics engine, the efforts were more focused toward other non-functional requirements such as usability, extensibility and modifiability in addition to performance and scalability. The need for an extensible physics engine is apparent from the fact that different VR applications have different physics requirements. Therefore, the goal here is to design an architecture that consists of a core structure with minimal functionality but one that can be easily extended on demand.

3.1 PhysicsEngine Component

Before going any further with designing an extensible generic architecture, we must first have a precise design of the heart of the system, the PhysicsEngine component. This section demonstrates some of the design alternatives, their pros and cons, and the details of the chosen design for this critical component.

3.1.1 Thread versus Behavior

As it was seen in chapter 2, the PhysicsEngine component is the main controller in the system and its primary function is to periodically update the physical objects in the scene. There are two design alternatives at this stage:

Alternative 1: The PhysicsEngine component updates the physical objects in the scene within specific time periods. For example, the physics engine can enforce an update every 20ms. This can be achieved with a Java Thread that has sleep duration of 50ms (Figure 3-1).

```
class PhysicsEngine extends Thread
{
   void run () {
        t = current time
        // update all physical objects
        for every object in the scene {
           object.update(t);
        }
        sleep(20);
   }
}
```

Figure 3-1: PhysicsEngine component can be represented with a Java Thread that runs every few milliseconds.

Alternative 2: The PhysicsEngine component updates the physical objects once every few frames. This can be achieved by using a Java3D Behavior with a special wakeup condition that triggers the behavior when the specified amount of frames is elapsed.

```
class PhysicsEngine extends Behavior
{
   private WakeupCondition frameCond = new WakeupOnElapsedFrames(1);

   void initialize () {
        // set wakeup condition to trigger this behavior in the next frame
        this.wakeupOn(frameCond);
   }
   void processStimulus() {
        t = current time
        // update all physical objects
        for every object in the scene {
           object.update(t);
        }
        // set wakeup condition to trigger this behavior in the next frame
        this.wakeupOn(frameCond);
    }
}
```

Figure 3-2: PhysicsEngine component can be represented with a Java3D Behavior that is triggered every few frames.

Tests have been performed to assess the performance effects of both alternatives. The following table summarizes the test results:

Test Setup	Few objects in the scene (1-15 objects)	Many objects in the scene (300 objects)
Test Results	• Both alternatives have good performance. • Physical laws such as collision behave correctly in both alternatives. • Alternative 2 consumes 100% of CPU cycles even if there is only one object in the scene. • Alternative 1 consumes little CPU cycles for few objects, but CPU usage increases as more objects are added.	• Both alternatives reveal jitters occasionally, but the jitters are slightly more apparent in alternative 1. • Alternative 1 is unable to enforce physical laws. Collisions are not detected. • Both alternatives consume 100% of CPU cycles.

The above test results reveal that the only advantage of alternative 1 is that it consumes less CPU resources initially; however, the CPU usage of both alternatives breaks even as the number of the objects in the scene grows. On the down side, the test revealed that Java3D's Behavior system is practically disabled when the Java Thread used in alternative 1 consumes 100% of CPU resources. This is due to the fact that Java3D Behaviors are designed to run in the background and make use of CPU resources only when they become available. Java Threads on the other hand have higher priority and can consume the entire CPU resources thus starving the Java3D Behaviors.

In summary, the use of Java3D Behavior (alternative 2) for updating physical objects results is a better solution since it has better performance and it does not disturb the execution of physical laws. Figure 3-3 demonstrates how the Behavior state diagram is simply transformed to represent the state machine of the PhysicsEngine component.

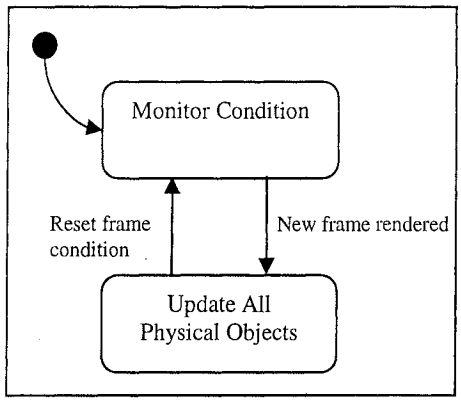


Figure 3-3: A simple state diagram of a PhysicsEngine component that is built based on Java3D Behaviors

3.1.2 Execution Control

The PhysicsEngine component must allow execution control by its client application. In other words, applications should be able to start, pause, resume or terminate the physics simulation at their convenient time, and the new execution state must be reflected over the system immediately.

Java3D Behaviors are capable of monitoring several conditions simultaneously. Therefore the above requirements can be achieved by adding new control conditions to the PhysicsEngine behavior, thus evolving it into the state diagram shown in Figure 3-4.

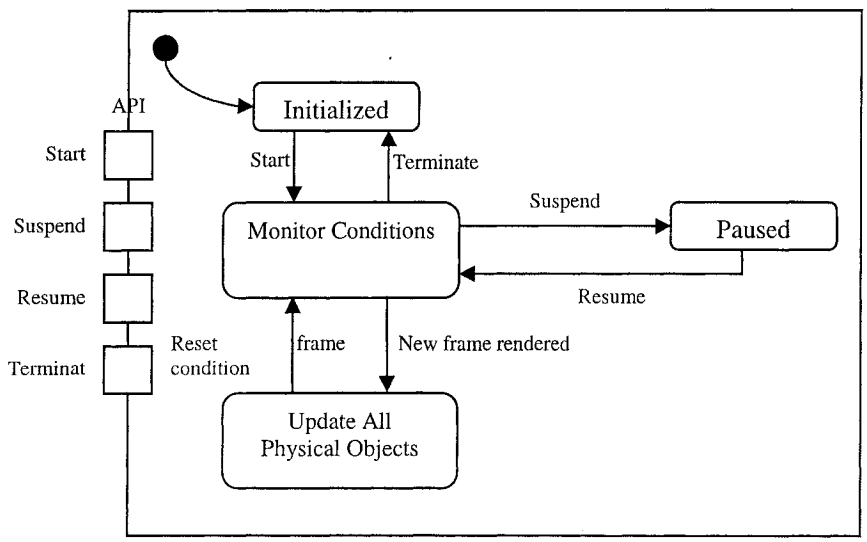


Figure 3-4: The complete state diagram of PhysicsEngine component

Upon construction, the component is in the *initialized* state. An application can start the physics simulation by issuing a *Start* command to the PhysicsEngine API. Technically, starting the physics engine involves capturing the current system time and storing it as the start time of the physics timer. When a physical object is scheduled to update itself, it needs to know the current time which is calculated by subtracting the *simulation start time* from the *current system time*:

$$t_{current} = t_{system} - t_{start}$$

An application may also suspend the physics engine for a period of time and then resume the engine. While suspended, the objects in the scene must behave as if the physics clock is stopped. This is achieved by storing the duration at which the engine is suspended and subtracting it from the current time. The improved equation for current time therefore is:

$$t_{current} = t_{system} - t_{start} - t_{paused}$$

Figure 3-5 shows how the above equation generates the desired pausing effect in the simulation of a moving ball.

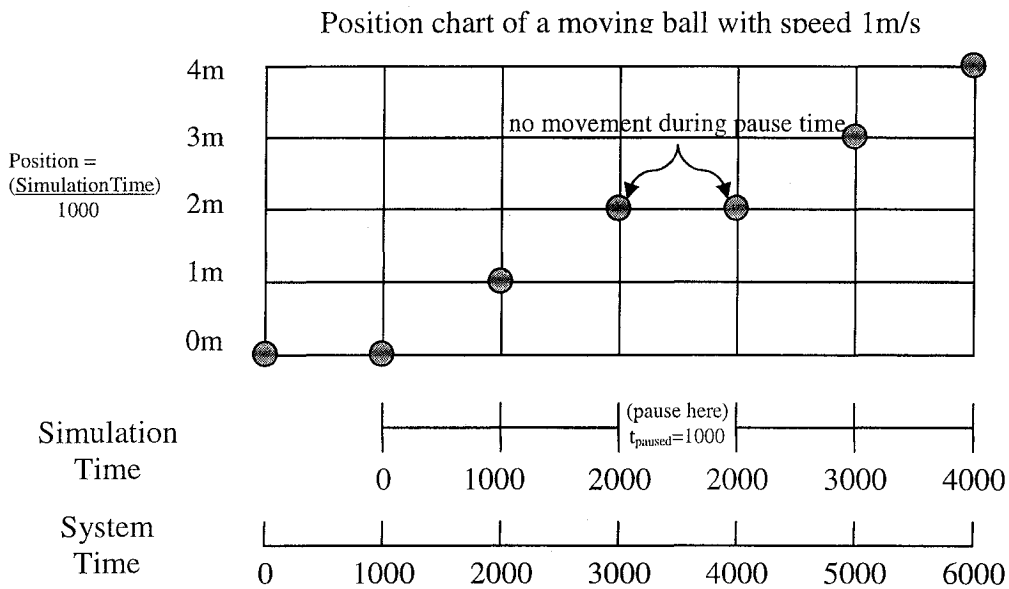


Figure 3-5: The simulation time is calculated by subtracting *start* time and *pause* duration from the system time. The updates on the physical objects are done based on the simulation time.

3.1.3 Wakeup Conditions

The state diagram in Figure 3-4 reveals that there are five triggering events that may cause transaction from one state to another. The most critical one is the condition that wakes up the behavior at every frame in order to update the physical objects in the scene. Java3D has a built-in condition called WakeupOnElapsedFrames that is specifically designed for this purpose. Figure 3-6 shows the partial API of this class.

WakeupOnElapsedFrames Constructor Summary

extends: WakeupCriterion

Class specifying a wakeup when a specific number of frames have elapsed.

`WakeupOnElapsedFrames(int frameCount)`
Constructs a new WakeupOnElapsedFrames criterion.

frameCount - the number of frames that Java 3D should draw before awakening this behavior object; a value of N means wakeup at the end of frame N, where the current frame is zero, a value of zero means wakeup at the end of the current frame.

`WakeupOnElapsedFrames(int frameCount, boolean passive)`
Constructs a WakeupOnElapsedFrames criterion with frameCount (see above) and passive parameters.

passive - flag indicating whether this behavior is passive; a non-passive behavior will cause the rendering system to run continuously, while a passive behavior will only run when some other event causes a frame to be run.

Figure 3-6: Partial API of WakeupOnElapsedFrames class

WakeupOnBehaviorPost Constructor Summary

extends: `WakeupCriterion`

Class that specifies a Behavior wakeup when a specific behavior object posts a specific event.

`WakeupOnBehaviorPost(Behavior behavior, int postId)`
Constructs a new WakeupOnBehaviorPost criterion.

Figure 3-7: Partial API of WakeupOnBehaviorPost class

For the remaining four conditions (start, suspend, resume, terminate) we use a Jav3D condition called WakeupOnBehaviorPost (Figure 3-7). This condition enables a class to wake itself up by making a call to the *post(id)* method of its super class (Behavior class). One alternative is to have an id assigned to each of the four conditions and set the behavior to monitor those conditions. However for efficiency purpose we use the

following technique in order to minimize the number of conditions that are monitored at any given time.

Initially the behavior is set up to use the WakeupOnBehaviorPost(1) condition. As soon as the start() method is called, the instance variable *isStarted* is set to true and the method post(1) is called. This will wake up the behavior which checks the status of isStarted flag. The new wakeup condition at this state is WakeupOnElapsedFrames(1) and with this the state machine completes its transaction to *Monitor Conditions* state (see Figure 3-4). From this point on, there will be many state transactions back and forth between *Monitor Conditions* and *Update All Physical Objects*. In addition to updating objects, there is a check for *isPaused* flag every time the behavior wakes up. If the flag is set then the wakeup condition is set to WakeupOnBehaviorPost(1) instead of WakeupOnElapsedFrames(1), thus the current state is transferred to *Paused*. The flag isPaused is set whenever a call to suspend() method is made. The execution can be resumed by calling the resume() method which sets the isPaused flag to false and makes a call to post(1) in order to make transition to *Monitor Conditions* state. Figure 3-8 is a better reflection of this new state machine.

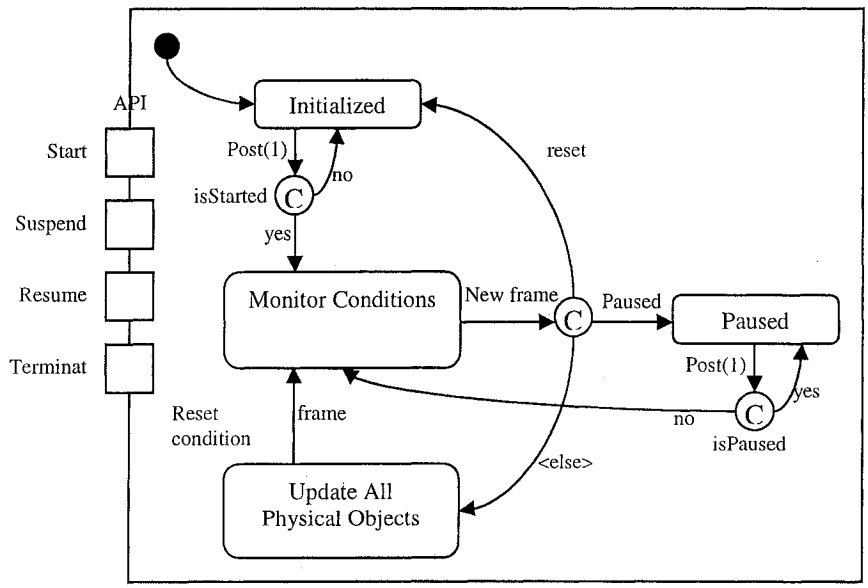


Figure 3-8: The improved version of state machine for PhysicsEngine component reduces the total number of events from 5 to 2 with the aid of flags. Also note that at every state only one condition is monitored (there is at most one out going arrow from every state).

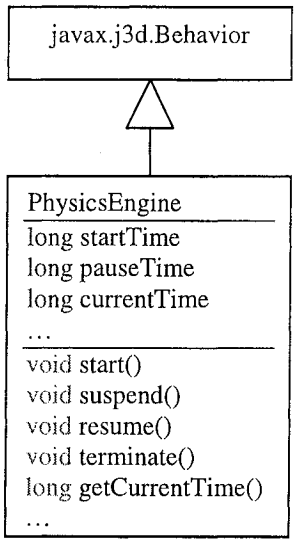


Figure 3-9: PhysicsEngine class diagram for execution control

3.2 Laws and Law Containers

The extensibility of any physics engine is directly proportional to the extensibility of its physical laws. Some applications for example, may need a precise Collision law that can detect exact collisions and then accurately compute the reflection path. Other applications may be more concerned with performance issues and prefer minimizing the computation by using a bounding box collision detection and respond with a simple reaction such as setting the velocities of the colliding objects to zero.

3.2.1 Container/Plug-in Architecture

It is common practice for application frameworks to adapt a container/plug-in model within their architecture [20]. This model can be applied to our physics engine in order to ease the integration of customized physical laws with the rest of the system. Simply put, we would like to be able to plug our customized laws in the physics engine without making any changes to physics engine code and watch the physics engine magically enforce the new laws upon the objects in the scene.

There are two aspects to every law: event and response. Events are conditions that cause a law to enforce some effects on some objects. For example, collision is an event that

causes the velocity of a bouncing ball to be reversed. For better flexibility we divide laws into two components, one for monitoring the events and one for generating response when the specified events occur. In our architecture (see Figure 3-10), the components in charge of monitoring events play the role of containers (LawContainer) and the components that contain algorithms for computing a response play the role of plug-ins (PhysicalLaw). The Law Containers themselves are plugged into the physics engine.

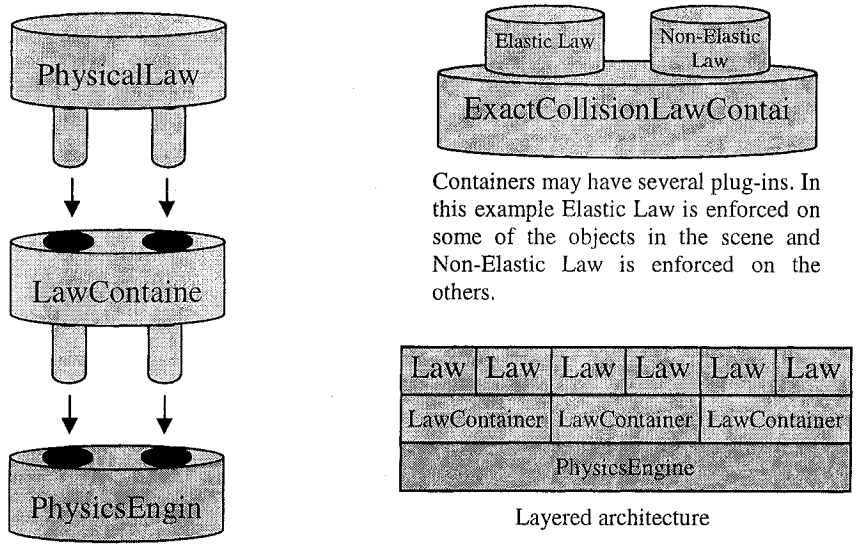


Figure 3-10: Container/plug-in architecture of the Physics Engine

In chapter 2 we used a simple behavior model to represent physical laws. In this chapter we are going to improve that initial design by applying the above container/plug-in structure.

3.2.2 LawContainer Component

3.2.2.1 Behavioral Representation

Law Containers are responsible for monitoring events that trigger law enforcement. As it was seen in chapter 2, the Java3D behavioral model is adequate for representing event-based systems. The table below lists some examples of how Behavior *conditions* can be used as events that trigger laws:

Law	Triggering Event(s)	Behavior Wakeup Condition	Explanation
Gravity	Every frame	WakeupOnElapsedFrames	Enforces gravity upon objects by adding a downward force at every update period.
	-Start of Physics Engine -New object added	WakeupOnBehaviorPost	Enforces gravity upon objects by adding a downward force of infinite duration every time the engine is restarted or when new objects are added.
Collision	Bounding-box collision	WakeupOnCollisionEntry	Enforces collision response upon two objects that are reportedly collided.
	Exact Collision	WakeupOnElapsedFrames	Enforces collision response upon two or more objects that are in collision. Collision is checked at every frame; Computationally heavy, but more precise result.

Java3D provides a set of useful wakeup conditions for behaviors that can be readily used in Law Containers in order to monitor some specific conditions. For example, WakeupOnAWTEvent condition can be used to trigger some laws upon pressing a key or moving the mouse. WakeupOnActivation is useful whenever a law must be enforced on an object as soon as it enters the visibility range. The following table summarized the 14 built-in wakeup conditions that Java3D currently offers.

Wakeup Criterion	Trigger
WakeupOnActivation	on first detection of a ViewPlatform's activation volume intersecting with this object's scheduling region.

WakeupOnAWTEvent	when a specific AWT event occurs
WakeupOnBehaviorPost	when a specific behavior object posts a specific event
WakeupOnCollisionEntry	on the first detection of the specified object colliding with any other object in the scene graph
WakeupOnCollisionExit	when the specified object no longer collides with any other object in the scene graph
WakeupOnCollisionMovement	when the specified object moves while in collision with any other object in the scene graph
WakeupOnDeactivation	when a ViewPlatform's activation volume no longer intersects with this object's scheduling region
WakeupOnElapsedFrames	when a specific number of frames have elapsed
WakeupOnElapsedTime	when a specific number of milliseconds have elapsed
WakeupOnSensorEntry	on first detection of any sensor intersecting the specified boundary
WakeupOnSensorExit	when a sensor previously intersecting the specified boundary no longer intersects the specified boundary
WakeupOnTransformChange	when the transform within a specified TransformGroup changes
WakeupOnViewPlatformEntry	on first detection of a ViewPlatform activation volume intersecting with the specified boundary
WakeupOnViewPlatformExit	when a View activation volume no longer intersects the specified boundary

3.2.2.2 Synchronous versus Non-synchronous Behaviors

As it was seen in the previous section, WakeupOnElapsedFrames(n) is a condition that wakes up the behavior after n frames are rendered from the time this condition was set. This condition is special in the way that it is the only synchronous behavior condition. When the specified number of frames are elapsed the behavior is waken up immediately.

All the other conditions are asynchronous; in other words, the Java3D behavior scheduler may wake up the behavior a few milliseconds after the condition is triggered. The advantage of this is that the CPU resources are managed properly and no behavior is starved. Synchronous behaviors on the other hand consume much of the CPU resources, therefore it is recommended to use them only for critical tasks.

In our physics engine, the task of updating the position/orientation of objects in the scene is critical; therefore WakeupOnElapsedFrames was used as the wakeup condition of the PhysicsEngine component. Some physical laws may be critical, others may not; this usually depends on the context and application. However in general, most laws are non-critical and can afford to take a few millisecond delays. So it may be worthwhile to use WakeupOnElapsedTime instead of WakeupOnElapsedFrames for say collision testing.

3.2.2.3 Container for Many Laws

Law Containers can have several laws plugged into them simultaneously. Typically all laws that share a common triggering event are plugged into a same Container. Each container maintains a list of the laws that are plugged into it; when the container detects that its wakeup condition is triggered, it loops through the list and enforces the laws one by one (Figure 3-11).

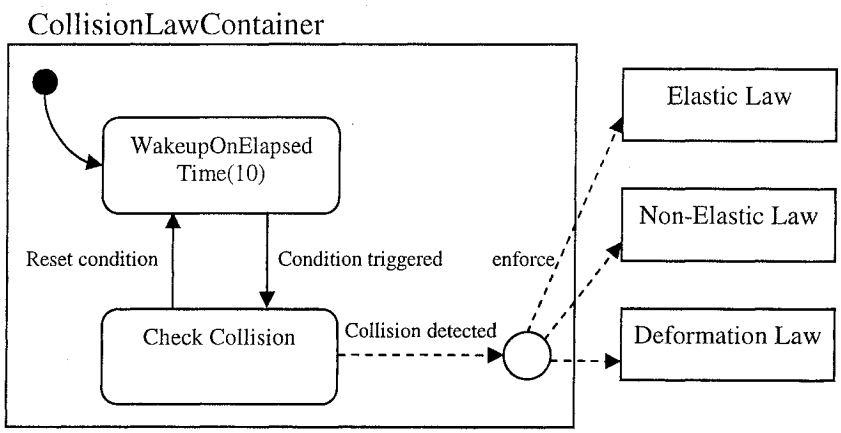


Figure 3-11: A Collision Law Container that checks for collisions every 10ms and enforces three variations of collision response when a collision is detected

An alternative to the structure of Figure 3-11 would be to construct three distinct Containers, each containing one of the three laws. However this approach is computationally inefficient because it would triple the load of computations for detecting collisions.

3.2.2.4 LawContainer Class

In our physics engine, the law containers are always defined as sub-classes of the LawContainer class, which extends the Behavior class and contains the methods that are common among all containers. The class diagram below demonstrates the elements of the LawContainer class. The instance variable *wakeupCondition* contains the wakeup condition of the behavior. The abstract method *enforceLaws* is automatically called whenever the wakeup is triggered. Law containers are notified through *lawChanged* method every time a new object is added to or removed from a law.

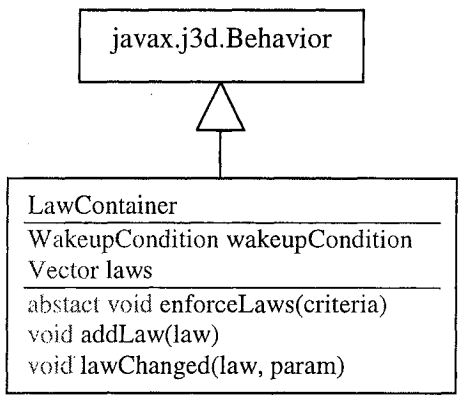


Figure 3-12: Class diagram of LawContainer

3.2.3 PhysicalLaw Component

This is the component responsible for enforcing laws after it is informed by its LawContainer that a certain event has occurred. Unlike LawContainers, PhysicalLaw components are stateless. They only contain mathematical, physical and geometrical algorithms that compute and update the attributes of the physical objects in the scene. The most common attributes that are modified by Physical Laws are the velocity and

force. Figure 3-13 demonstrates an application example of how laws within the physics engine must manipulate force to achieve expected physical behaviors. Note that the VR application does not need to worry about handling collisions.

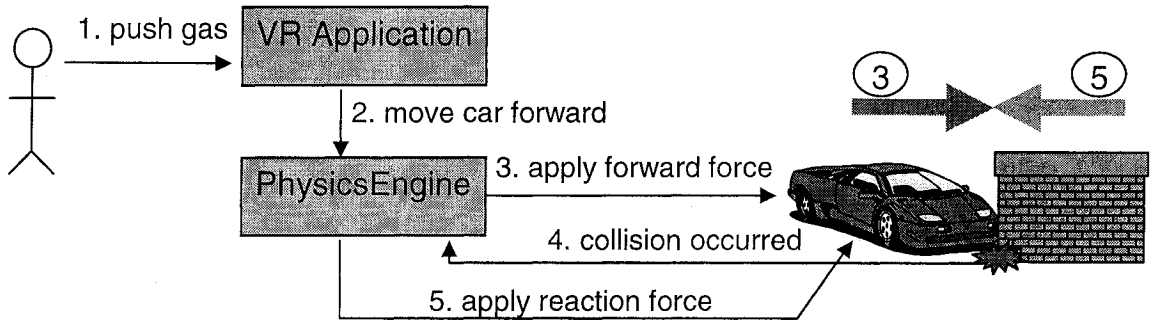


Figure 3-13: Force manipulation enables the collision law to prevent the car from penetrating the wall.

3.2.3.1 PhysicalLaw as a Plug-in

As it was mentioned earlier, Physical Laws are plug-ins that can be plugged into any LawContainer as long as it has the expected interface. Figure 3-14 demonstrates a high-level view of the interactions that occur between CollisionContainer and CollisionLaw.

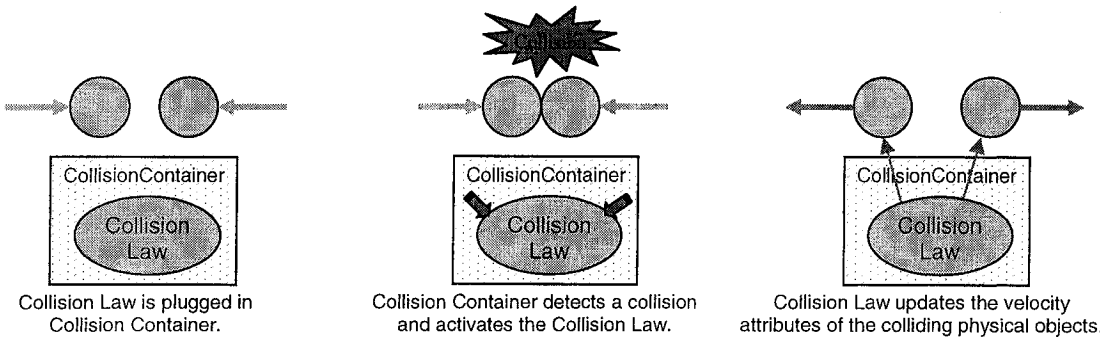


Figure 3-14: The steps of enforcing collision law

In our implementation of the physics engine, the expected interface simply refers to the public methods of Physical Law class. For example, CollisionLawContainer may expect the CollisionLaws to provide some methods for sending the list of colliding objects along with the normal of the collision surface. Java has a standard mechanism for defining interfaces.

3.2.3.2 Defining Interfaces in Java

It is a well-known pattern to used interfaces in order to define the communication channel between two or more components [19]. The key word *interface* in Java is used to define a set of empty methods (method headers) that must be implemented by other classes if those classes would like to be accessible through this interface. The following steps need to be taken in order to have a proper container/plug-in structure set up:

- Define an interface between Container and law (Figure 3-15);
- Define the container such that only laws that support that interface can be plugged into it (Figure 3-16);
- Create laws that must be plugged in the container by implementing the defined interface (Figure 3-17).

```
interface CollisionLaw
{
   // this method is called by the container of this law to report the colliding objects
   void setCollidingObjects (PhysicalObject obj1, PhysicalObject obj2);

   // this method is called by the container of this law to report the normal of the
   // collision surface
   void setNormal (Vector3f normal);
}
```

Figure 3-15: Sample code for defining an interface for CollisionLaw

```
class CollisionLawContainer extends LawContainer
{
   // laws that implement interface CollisionLaw can be added to this container with
   // this method
   void addLaw (CollisionLaw law)
   {
        // code to add the law
   }
}
```

Figure 3-16: Containers restrict the plug-in feature to those laws that implement the proper interface

```
class ElasticCollisionLaw implements CollisionLaw
{
   void setCollidingObjects (PhysicalObject obj1, PhysicalObject obj2) {
        // store colliding objects
   }
   void setNormal (Vector3f normal) {
        // sore collision normal
   }
   void enforceThisLaw ( ) {
        // code to enforce elastic response on obj1 and obj2 according to normal
   }
}
```

Figure 3-17: Any law that wants to be plugged in a container must implement the interface that is supported by that container.

3.2.3.3 Common Features of Physical Laws

There are some features and attributes that are common among all physical laws. For example, most physical laws require having a list of physical objects that must obey the law; Gravity law contains a list of objects that are affected by gravity, and collision law contains a list of objects that can collide together. The following table lists the common features that must be supported by all laws:

Feature	**Description**
Enable/Disable	It should be possible to enable/disable laws at runtime. When a law is disabled, it does not do anything when it is activated by its container.
Add/Remove Objects	Objects that are affected by the law can be added or removed at runtime.
Container Notification upon change	When a change, such as add/removal of an object, occurs sometimes the container needs to be informed about it. Each law must have a reference to its pointer and notify it whenever such changes occur.

To account for all these requirements without causing redundancies in the implementation of every law, the super class PhysicalLaw is introduced. The common methods and attributes exist in the PhysicalLaw class and all physical laws are required to subclass it.

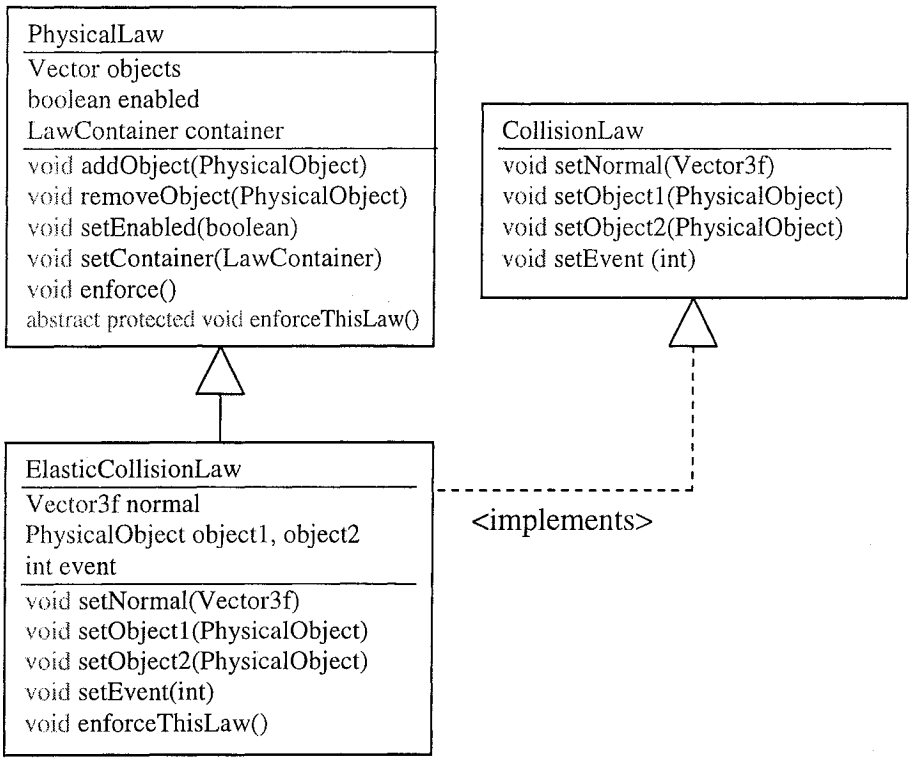


Figure 3-18: ElasticCollisionLaw inherits from PhysicalLaw to share the common features and implements CollisionLaw interface to be able to be plugged into the CollisionLawContainer.

Figure 3-18 shows how the concept of inheritance and interfaces can be combined for defining physical laws. Note that in the definition of PhysicalLaw there is a public method *enforce* and an abstract method *enforceThisLaw* which is implemented in its subclass. As it can be seen from the code segment in Figure 3-19, this is a simple mechanism to achieve two objectives:

1. Code segment in the enforce method prevents the law from being enforced if it is disabled (Figure 3-19).

2. Since *enforceThisLaw* is hidden to other components, the containers cannot call it directly. They are forced to call the *enforce* method, every time a law needs to be enforced.

```
abstract class PhysicalLaw {
   private boolean enabled;

   public void enforce( ) {
        if (enabled)
            enforceThisLaw( );
   }
   protected abstract void enforceThisLaw( );
}
```

Figure 3-19: The only way to enforce a law is by calling the enforce method of the superclass PhysicalLaw. This method in turn calls the enforceThisLaw method of the subclass which contains the actual implementation of the law.

3.2.4 Container/Plug-in Interactions

After laws are plugged into the containers and the containers are plugged into the physics engine, the interaction between components begins. The LawContainer is a Java3D Behavior; thus its state is controlled by Java3D Behavior Scheduler. If the wakeup condition (i.e. WakeupOnTimeElapsed) of a LawContainer is satisfied, then the Behavior Scheduler wakes up the container. The container will then check to see if any of the laws need to be activated (i.e. Collision checking). If so, the required data are sent to the law followed by a call to its *enforce* method which in turn starts a series of computations that will ultimately result in an update of the physical attributes in one or more physical objects. Figure 3-20 and Figure 3-21 demonstrate the sequence diagram of the container/plug-in interaction in the physics engine.

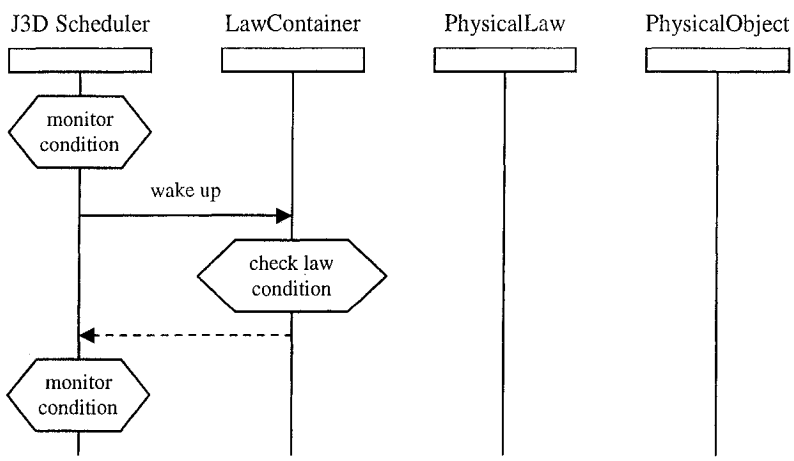


Figure 3-20: Sequence diagram of Container/Plug-in interactions when law condition is not satisfied yet

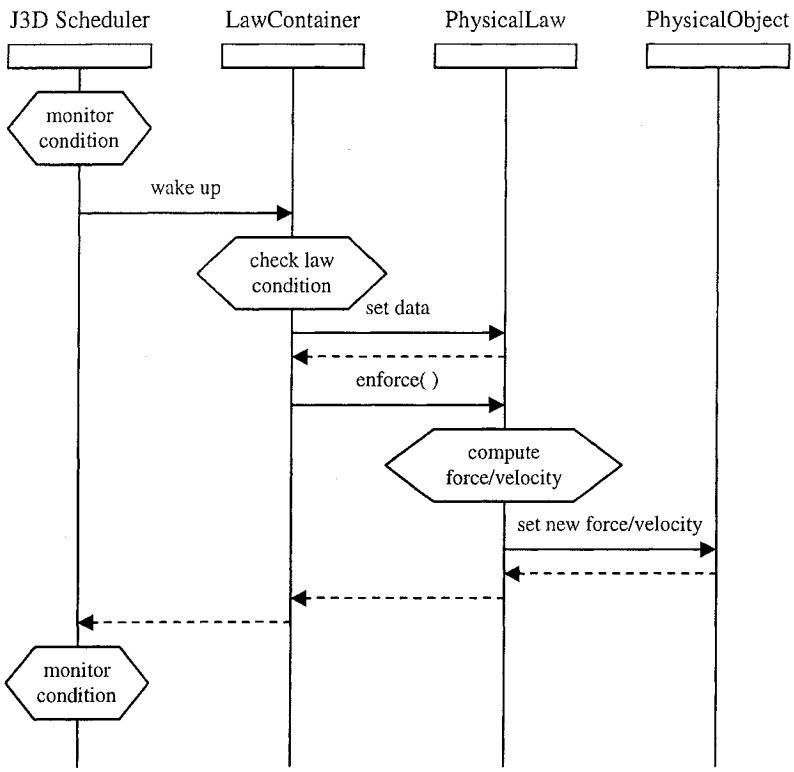


Figure 3-21: Sequence diagram of Container/Plug-in interaction when a law is activated

3.3 Physical Objects

Each physical object stores and maintains necessary data such as mass, force, current position, orientation and velocity. In addition to storing data, physical objects are directly bound to Java3D scene graph, therefore updates in position/orientation data will automatically update the actual position/orientation of the corresponding geometries in the 3D world.

As it was mentioned in chapter 2, physical objects provide a public *update(t)* method which is called by PhysicsEngine component periodically. Figure 3-22 demonstrates a class diagram that summarizes the features of the PhysicalObject component.

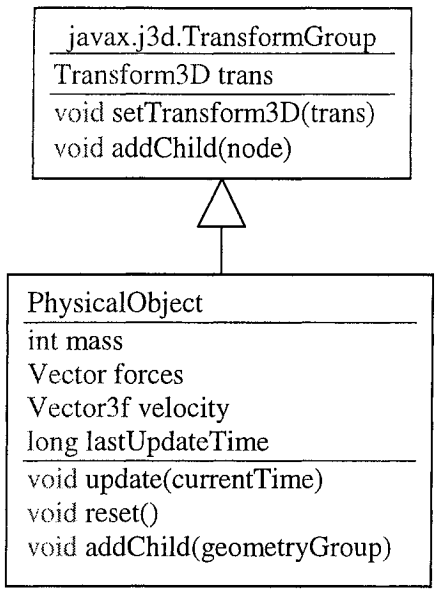


Figure 3-22: Class diagram of PhysicalObject

Applications that require additional features can extend PhysicalObject into subclasses and overload its *update* method or add extra public methods. For instance, an application may require that some of the objects in the scene be deformable. That is, if the force exerted on the object exceeds a certain limit the object geometry is deformed. The deformation itself may be further divided into several categories. In metallic objects, a force may result in bending the geometry, whereas same force may decompose the glass objects into several smaller physical objects.

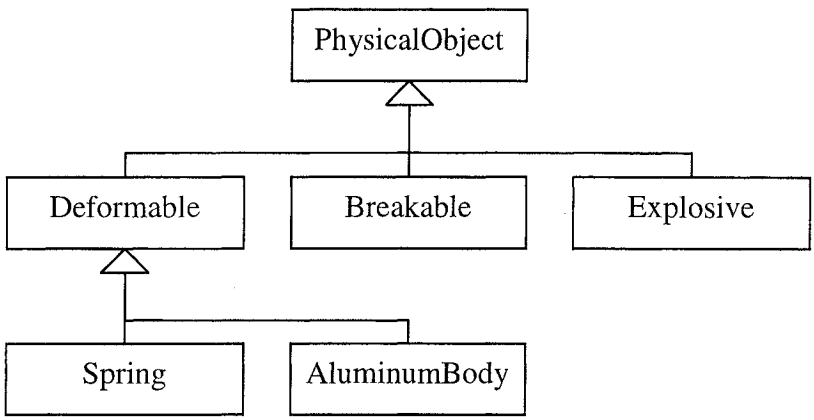


Figure 3-23: Customization of PhysicalObject

3.4 Object-Law Mapping

The architecture of our physics engine provides a simple mechanism for specifying which objects must obey which laws. An object representing a cloud of smoke does not need to obey collision law nor gravity; it does however need to be affected by the wind. Thus a mapping structure is required in order to customize the behavior of scene objects according to the nature of what they represent (see Figure 3-24 for examples of what types of physical objects may be affected by which laws of physics).

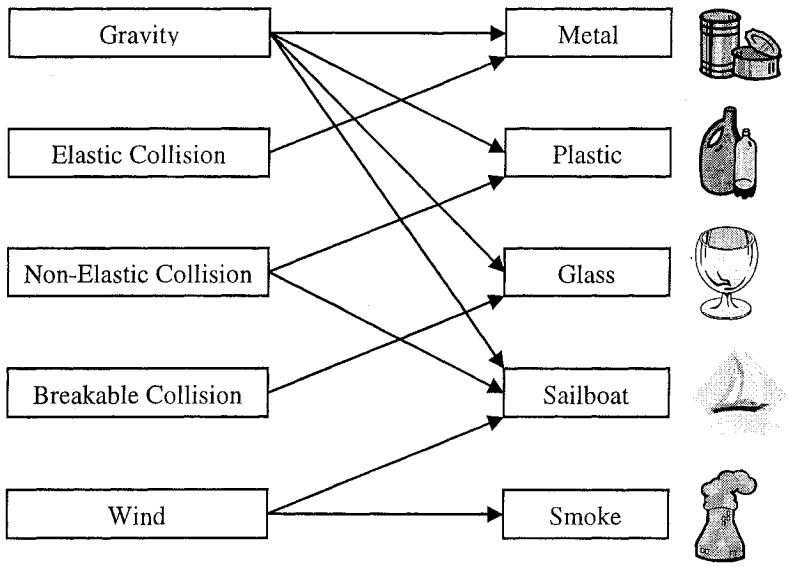


Figure 3-24: Different objects may be affected by different laws depending on the application context.

Typically every time an object is added to the physics engine, a list of its corresponding laws is sent as well. The physics engine would then map the objects to laws automatically (see the use of engine.addObject(...) method in Figure 3-25).

```
class MyClass {
   void main (String args[]) {
        PhysicsEngine engine = new PhysicsEngine( );

        GravityLaw gravity = new GravityLaw( );
        CollisionLaw collision = new CollisionLaw( );

        engine.addLaw(gravity, new GravityLawContainer( ));
        engine.addLaw(collision, new CollisionLawContainer( ));

        Vector collisionLawOnly = new Vector();
        collisionLawOnly.allLaws.add(collision);

        Vector allLaws = new Vector();
        allLaws.add(gravity);
        allLaws.add(collision);

        PhysicalObject balloon = new PhysicalObject( );
        balloon.addChild(balloonGeomtery( ));
        PhysicalObject car = new PhysicalObject( );
        car.addChild(carGeomtery( ));

        engine.addObject(balloon, collisionLawOnly);
        engine.addObject(car, allLaws);

        engine.start( );
   }
   Shape3D balloonGeomtery( ) {
        // code to create balloon geometry
   }
}
```

Figure 3-25: PhysicsEngine provides API for associating objects with a list of physical laws.

3.5 Summary

The architecture of the physics engine is engineered in such a way to maximize the extensibility and usability of the system while maintaining a good performance (see Appendix B for overall UML diagram). The PhysicsEngine component is optimized for performance and reflects object movements at each frame with frame of rate of over 30fps which is adequate for human eye.

The Physical laws and law containers are loosely connected to the physics engine through a container/plug-in mechanism. This has a major significance because the implementation of physical laws and containers frequently change due to upgrade or customization of the physical behavior of the VR applications. Physical objects contain the essential attributes necessary for most VR applications. They can be extended to cover additional attributes or modify the computations that are performed at each update.

Mapping between objects and laws is another significant feature that allows applications to easily customize the behavior of the individual objects in the virtual environments.

CHAPTER 4 - PHYSICAL LAWS

This chapter will discuss in detail the implementation of some of the key physical laws of the physics engine. As it was mentioned before, the architecture of our physics engine provides an effective approach for customizing the physical behaviors by allowing construction of new physical laws that can be easily plugged into the system. From the development point of view, this means that component developers can implement libraries of precise or efficient physical laws and application developers can selectively chose the laws that meet the specification of their projects.

4.1 Core Components versus Utility Components

Before going any further, the notion of core components must be cleared out in this context. Core components are those classes that form the base of our physics engine; thus they cannot be replaced and they are not to be modified under normal circumstances. The architectural design of the core components were discussed in chapter 3. This chapter is about the utility components that are constructed by extending (i.e. sub-classing) the core components.

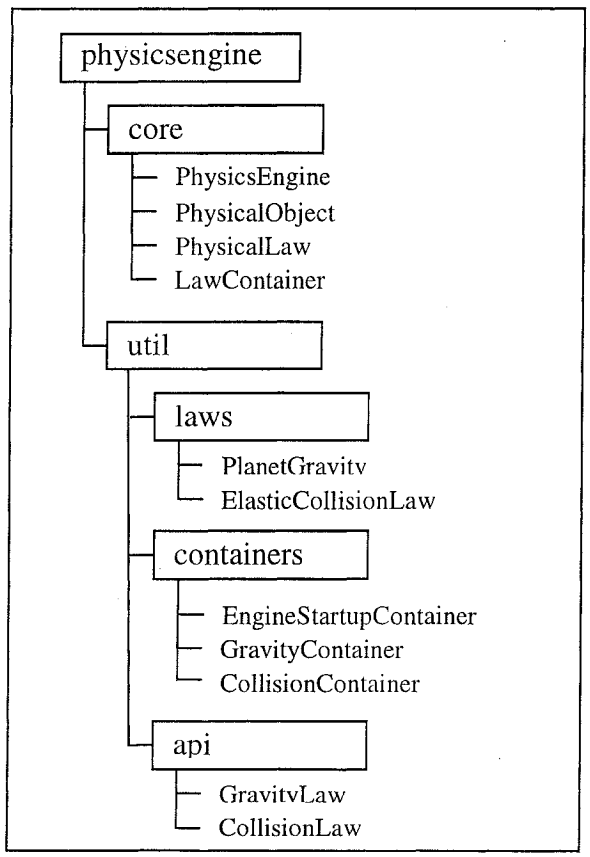


Figure 4-1: demonstrates the package hierarchy of our physics engine along with the public classes that are available under each package. Chapter 3 was more focused on the classes that are categorized under physicsengine.core package. This chapter will cover those that are listed under physicsengine.util.

4.2 An Implementation of Gravity Law

Gravity is perhaps one of the most fundamental laws of physics. The force of gravity is continuously affecting the objects on this planet. The effect of gravity varies from one context to another. The force of gravity of earth is larger than what exists in the moon for example. There is also a universal gravity law that holds for all planets and elements in space; but even the universal law of gravity is not sufficient for simulating the gravity of special phenomena such as black holes.

4.2.1 Gravity Law Interface

When designing an interface for a Law, the key idea is to identify the data that must be sent to the law every time it is activated. The interface must be generic and regardless of the implementation details. From a generic point of view, gravity laws compute and apply gravity forces to the scene objects. Since the scene objects are already accessible through the super-class *PhysicalLaw* and no additional data is required by gravity laws, the interface is empty (Figure 4-2).

```
package physicsengine.util.api;

interface GravityLaw {
}
```

Figure 4-2: Interface of Gravity Laws

Recall that all physical laws can be enforced by calling their *enforce* methods. The advantage of having an empty interface, as compared to not having an interface at all, is that it provides us a mechanism to assure that the law being plugged in a gravity container is indeed a *GravityLaw*. Otherwise, a client may plug a CollisionLaw [for example] by mistake and since the *CollisionLaw* also has an *enforce* method the compiler would not be able to catch the error.

4.2.2 Gravity Law

The implementation of the PlanetGravity law under physicsengine.util.laws package is a generic representation of the gravity on planets without considering the effects of surrounding planets (as compared to the universal law of gravity); it can be customized for different planets by assigning the appropriate gravity acceleration of that planet. For the case of earth, it applies a downward force to all physical objects in the scene in order to enforce a downward acceleration of 9.8 m/s^2.

Recall that every law must extend PhysicalLaw and implement an interface. The super-class PhysicalLaw provides several common functionalities to all its sub-classes. Implementing an interface is a way of assuring that the law and the container understand each other. PlanetGravity implements the interface GravityLaw that was designed in the previous section. Figure 4-3 demonstrates how PlanetGravity class inherits from PhysicalLaw and corresponds to GravityLaw.

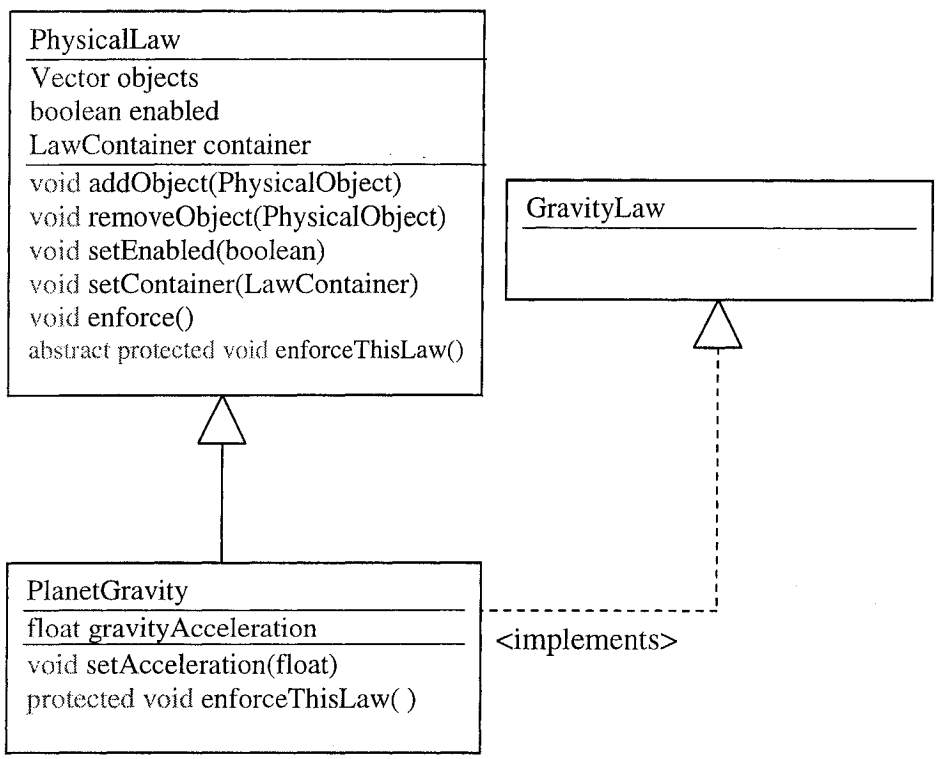


Figure 4-3: Class diagram of PlanetGravity

There are two design alternatives for implementing and enforcing gravity law:

Alternative 1: The gravity force is applied at every update period. This would require that the law be plugged in a container that calls the *enforce* method of the gravity law at every frame.

Alternative 2: The gravity force is applied only at the beginning of the simulation and also when new objects are dynamically added to the physics engine.

The following table summarizes the advantages and disadvantages of each alternative:

	Advantages	**Disadvantages**
Alternative 1	• Gravity force is added to physical objects at every update period. Therefore the physical objects are not required to store gravity force. • There is no need to be concerned about the physical objects that are dynamically added as they automatically receive gravity force in the next update period. • The object mass can dynamically change since a new force is computed at every update period based on the current mass of the object.	• It is computationally heavy, because it required execution of code at every update period.
Alternative 2	• Gravity force is only added once (with infinite duration); therefore it is computationally very efficient.	• A mechanism must be implemented to activate the law every time a new object is added. • A mechanism must be implemented to monitor the mass of objects and reactivate the law every time a change in an object mass is detected (possible but not implemented in this version).

From the above table it is apparent that alternative 2 has a noticeable impact on improving performance. It has few disadvantages that put a little weight on the implementation side but the boost of performance certainly is worth it. Therefore alternative 2 is selected for implementing the gravity law.

Every law has access to a list of objects that must obey the law. Upon activation, the PlanetGravity component simulates the effect of gravity by looping through all physical objects in the list and adding a downward force to each of them. Recall from chapter 2 that an id can be optionally assigned to any Force (see Figure 4-4 for more details on the API of Force class). Every force that is applied with PlanetGravity component has an id that identifies the force as a gravity force.

The gravity is applied based on the mass of the objects in order to compute a force magnitude strong enough to cause the intended acceleration:

$$F = a/m$$

Where 'a' is a vector that corresponds to the acceleration of gravity force (acceleration of gravity is (0, -9.8, 0) on earth) and 'm' an integer that contains the mass of the object in kg. The unit for force 'F' is in Newtons. The code segment in Figure 4-5 demonstrates a simple implementation of the PlanetGravity law.

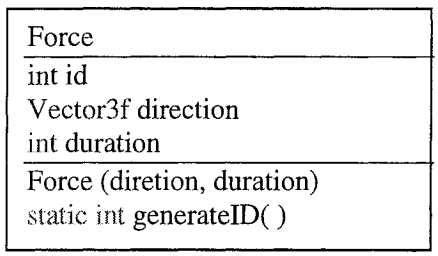


Figure 4-4: Class diagram of Force

```
package physicsengine.util.laws;

import physicsengine.core.*;

public class PlanetGravity extends PhysicalLaw {
    public static final int GRAVITY_FORCE_ID = Force.generateID();
    private float gravityAcceleration;

    ...

    protected void enforceThisLaw( ) {
        Enumeration objList = this.objects.elements();
        while (objList.hasMoreElements( ) ) {
            PhysicalObject obj = (PhysicalObject)objList.nextElement( );
            float gravityForce = obj.mass * gravityAcceleration;
            Vector3f force = new Vector3f(0f, -gravityForce, 0f);
            obj.addForce(new Force(force, -1, GRAVITY_FORCE_ID));
        }
    }

}
```

Figure 4-5: Code segment of PlanetGravity class that generates and applies the gravity force to all objects mapped to this law

Typically, the *enforceThisLaw( )* method of PlanetGravity is called only once in the beginning of simulation. It is called again when the simulation is restarted; but there are no steps taken to count for the new physical objects that are dynamically mapped to this law while the simulation is running. The code segment in Figure 4-6 demonstrates a simple solution which involves overloading the *addObject* method and applying gravity force to objects as soon as they are mapped to the law.

```
class PlanetGravity extends PhysicalLaw {
    ...

    // this method is called every time an object is added (mapped) to this law
    void addObject(PhysicalObject obj ) {
        // call the original superclass method to perform the usual operations
        super.addObject(obj);
        // computed and apply gravity force to the newly added object
        float gravityForce = obj.mass * gravityAcceleration;
        Vector3f force = new Vector3f(0f, -gravityForce, 0f);
        obj.addForce(new Force(force, -1, GRAVITY_FORCE_ID));
    }

}
```

Figure 4-6: PlanetGravity can apply gravity to physical objects that are dynamically added by overloading the addObject method.

In summary, PlanetGravity law performs its operations as follows:

1. Before the simulation is started, objects are mapped to the PlanetGravity law by calling its *addObject* method. PlanetGravity immediately computes and applies the gravity force to each object that is added.

2. When the simulation is started (or re-started), all forces in all physical objects are set to zero. Therefore the forces that PlanetGravity had applied prior to the start of physics engine are all deleted!

3. However, starting (or re-starting) the simulation would trigger the container of PlanetGravity (to be discussed in next section) which results in activating the PlanetGravity law. This will ultimately result in the execution of the *enforceThisLaw* method which re-applies the gravity force to all objects (see Figure 4-5).

4. If a new physical object is mapped to PlanetGravity while the simulation is still running, the corresponding gravity force is immediately calculated and applied to the object. The process is similar to step 1 except this time the force will have its physical effects on the object since the simulation has already started.

4.2.3 Gravity Container

As it was discussed in the previous section, the container of our gravity law must activate the law in the beginning of the simulation. For this, we need a Law Container with a condition that is triggered by the PhysicsEngine every time the engine is started (see Figure 4-7).

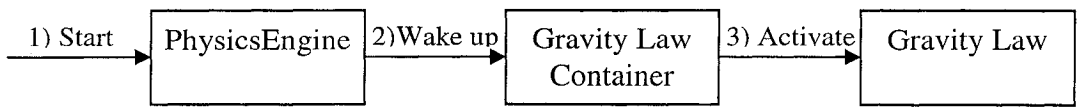


Figure 4-7: Upon Start command, PhysicsEngine wakes up the container of gravity law which then activates the gravity law.

To implement the above sequence of events, we start with PhysicsEngine component which is a sub-class of *Behavior* (see chapter 3). The PhysicsEngine is required to simply emit events that are later captured by the Java3D Behavior scheduler and ultimately result in waking up the container of our gravity law. For this task we use the *post(id)* method of the Behavior class(Figure 4-8). Recall that Behaviors can wake other Behaviors up by posting an id using *post(id)* method. Other behaviors that have WakeupOnBehaviorPost(wakeupID) as their wakeup condition are waken up if wakeupID is equal to id.

```
class PhysicsEngine extends Behavior
{
   static final int START_UP = 1;
   ...

   void start( ) {
      ...
      // wakeup containers that must be waken up upon START_UP
      this.post(START_UP);
   }

}
```

Figure 4-8: Use of post(id) method in PhysicsEngine for notifying the containers of simulation events

Since a container that activates one or more physical laws upon start of the physics engine is not specific to the gravity law, we name it EngineStartupContrainer (instead of GravityLawContainer). Figure 4-9 shows how the wakeup condition is set for EngineStartupContainer. Also note that the *enforceLaws* method does nothing more than simply calling the *enforce* method of each law; and since all laws have *enforce* method, this container can accept any law as its plug-in.

```
class EngineStartupContainer extends Behavior
{
   // The Constructor requires PhysicsEngine as its parameter
   EngineStartupContainer(PhysicsEngine engine) {
       // This container wakes up whenever engine posts START_UP
       this.condition = new WakeupOnBehaviorPost(engine, engine.START_UP)
    }

   // When this container wakes up, it loops throw all PhysicalLaws to call their
   // enforce method
   protected void enforceLaws(Enumeration criteria) {
       Enumeration enum = this.laws.elements();
       while (enum.hasMoreElements( ) )
          ((PhysicalLaw)(enum.nextElement( ))).enforce( );
    }
}
```

Figure 4-9: EngineStartupContainer simply calls the enforce method of all its physical laws. Any physical law can be plugged into this container.

EngineStartupContainer can be used as the container of our PlanetGravity law. Alternatively, one can create a sub-class of EngineStartupContainer in order to make it specific to gravity laws as shown in Figure 4-10. Note that every time a law is added, it is validated to make sure that it implements the physicsengine.util.api.GravityLaw interface.

```
class GravityContainer extends EngineStartupContainer
{
   // The Constructor
   GravityContainer(PhysicsEngine engine) {
       super(engine);
    }

   // addLaw is overloaded to make sure the plug-in has the expected interface
   void addLaw(PhysicalLaw law) {
       if (law instanceof physicsengine.utils.api.GravityLaw)
          super.addLaw(law);
       else
          throw new RuntimeException("law is not a GravityLaw");
    }

}
```

Figure 4-10: Extending EngineStartupContainer to make it specialized for Gravity Law

4.3 An Implementation of Collision Law

Unlike gravity, the implementation of the Collision Law is quite complex in both collision detection and collision response. In our architectural design, collision detection is handled by a container, and collision response is handled by a law that is activated by the container. The concept of collision detection by itself has been the topic of many research papers during the past decade, and researchers and game developers are still trying to find more effective and efficient algorithms for this task. This section will discuss how collision detection and response was implemented in our physics engine. Other more effective collision algorithms can be implemented and easily plugged into the system at anytime thanks to our flexible architecture.

4.3.1 Collision Law Interface

When a collision is detected by the container of a collision law, there are certain data that need to be transmitted to collision law before it can properly respond to the collision. The first and most obvious data is the list of colliding objects. A second set of data contains the collision information for each pair of colliding objects such as the point of intersection and collision normal. Figure 4-11 demonstrates the elements of CollisionLaw interface.

```
interface CollisionLaw {

    // A simple data structure to contain collision info
    public class CollisionInfo {
        Vector3f normal;
        Point3f intersection;
    }

    // Interface of collision law that receives collision info from its container
    void addPairOfCollidingObjects (Vector objs, CollisionInfo info);
}
```

Figure 4-11: Interface of CollisionLaw

Note than in addition to providing interface methods, CollisionLaw also provides a public data structure that must be used by containers for storing collision data. For every pair of

colliding objects that the container detects, a call to *addpairOfCollidingObjects* is made along with corresponding parameters (see Figure 4-12).

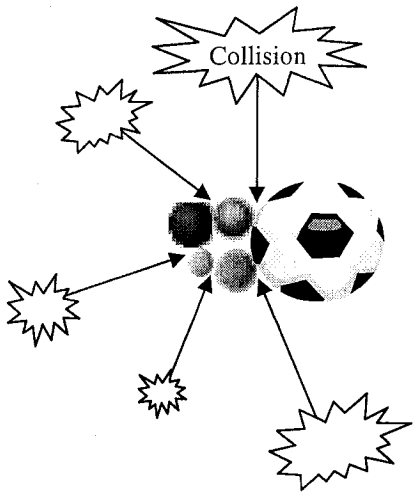


Figure 4-12: Pairs of colliding objects in a scene with five balls. For each pair, a CollisionInfo must be constructed and sent to Collision law.

4.3.2 Collision Law

Collision laws are laws that are responsible for generating response to collisions. There are many physical attributes such as elasticity and friction that can complicate collision response. The implementation of the Collision Law in utility package is deigned to respond to two types of collisions:

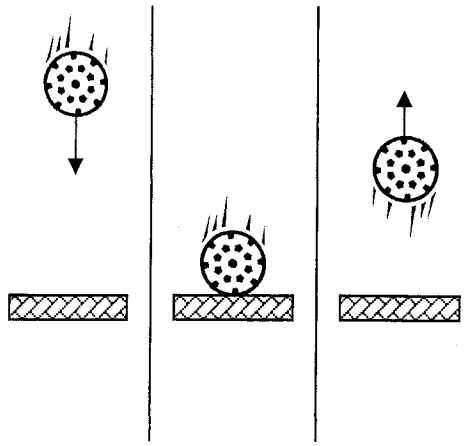
1. An object that collides with a steady object such as a floor. In this case, the steady object is not to be moved and the colliding object must bounce back.

2. Two objects collide together. In this case elastic collision is assumed; thus collision is responded according to the formula drawn from *conversation of momentum* concept.

4.3.2.1 Collision of One Object

If only one object is sent to our Collision Law, then it is assumed that the object has collided with a non-moving hard object (such as a floor or wall); therefore the goal of the computations at this stage would be to force the object to bounce back.

In a simplistic algorithm, one may assume that the moving object collides with non-moving object at a perpendicular angle (see right). If this is the case, the collision law can respond by simply reversing the velocity of the moving object:



object.velocity = -object.velocity

Most collisions however occur with an angle (θ), which is the angle between the velocity of the moving object with the normal of the collision surface. In such cases, symmetry of reflection is assumed; that is, the velocity of colliding object is rotated around the normal in addition to being reversed as shown in Figure 4-13.

Note that using the surface normal as a reference vector for computing θ enables us to calculate to the reflected velocity regardless of the orientation of the floor.

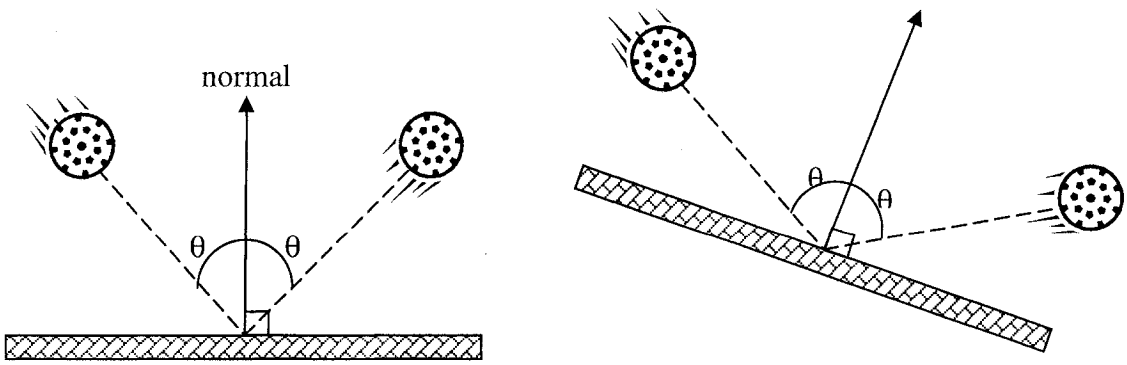


Figure 4-13: Collision of a moving object with a fixed floor

In order to compute the reflected velocity we decompose the incident velocity into two components:

- Surface Component: Portion of velocity vector parallel to the surface of collision.
- Normal Component: Portion of velocity vector parallel to the normal of collision surface.

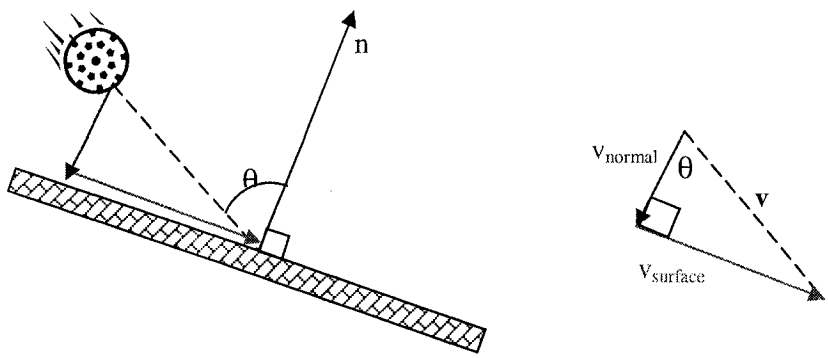


Figure 4-14: Components of Incident Velocity

As it can be seen in Figure 4-14, the components of the original velocity are perpendicular on each other and form a right angle triangle with the original velocity vector.

Computing Surface Velocity

There two unknowns at this stage: the direction of $v_{surface}$ and its magnitude. The following steps are taken for computing the direction:

1. Compute the cross product of velocity **v** with normal **n**. This will give us a vector perpendicular to both the normal and velocity.

$$n_{perpendicular} = v \times n$$

2. Compute the cross product of normal n with $n_{perpendicular}$. The result will be in direction of original velocity but parallel to the surface (Figure 4-14).

$$v_{surface} = n \times n_{perpendicular}$$

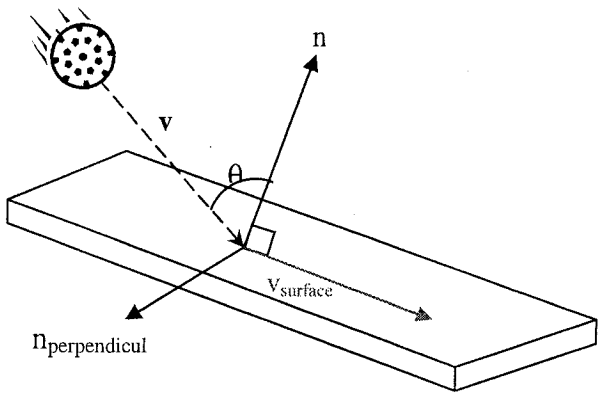


Figure 4-15: Direction of the Surface Velocity of velocity v with respect to v and normal n

Once the direction of surface component is found, the magnitude can be easily computed with the following steps:

1. Normalize the vector $v_{surface}$
2. Find the angle α between $v_{surface}$ and v (dot product can be used)
3. Computer the length (magnitude) of $v_{surface}$ by multiplying the length of v by $\cos(\alpha)$:

   magnitude = v.length() * $\cos(\alpha)$
4. Multiply the normalized $v_{surface}$ vector by its magnitude.

Computing the Normal Velocity:

The direction of Normal Velocity is the exact negation of the surface normal. The magnitude can be computed in a similar approach than the surface component. The following steps summarize these computations:

1. v_{normal} = n.negate( )
2. normalize v_{normal}
3. Find the angle α between v_{normal} and v (dot product can be used)

4. Computer the length (magnitude) of v_{normal} by multiplying the length of v by $\cos(\alpha)$:

   magnitude = v.length() * $\cos(\alpha)$

5. Multiply the normalized $v_{surface}$ vector by its magnitude.

Adding Surface Component and Normal Component

Once both components are computed, the reflection velocity can be found by adding the surface velocity to the negation of normal velocity as shown in Figure 4-16.

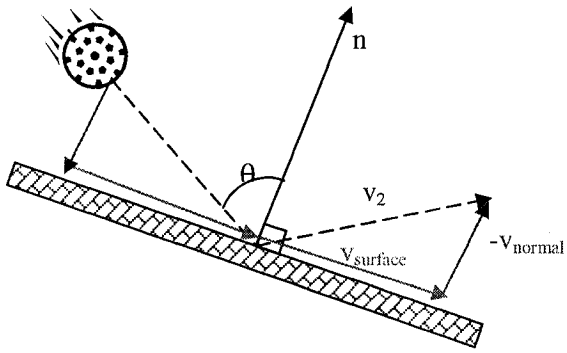


Figure 4-16: Final velocity $v_2 = v_{surface} - v_{normal}$

Decay of Vertical Velocity

For a more realistic effect, the magnitude of the normal velocity is usually set to be less than the original v_{normal}; otherwise a ball that is dropped on a floor will bounce up and down forever. In the real world, the bouncing (or elasticity) factor (between 0% and 100%) depends on the material on the colliding objects. In this implementation of Collision Law however, we simply use a global bouncing factor that is applied to all physical objects regardless of their composition. With addition of bouncing factor, the formula for computing the reflection velocity is as follows:

$$v_2 = v_{surface} - \gamma(v_{normal})$$

where γ is the bouncing factor with values between 0 and 1.

Note that in the above formula the surface velocity is not affected by the collision. In reality the surface velocity is slightly affected by the friction, however due to the insignificance of such effects we chose not to consider them in this implementation of the collision law.

4.3.2.2 Collision of Two Objects

For the case of collision between two moving objects the conservation law of momentum is used. The current implementation of Collision Law applies the formula for elastic collisions to computer the magnitude of the resulting velocities:

$$v_{1f} = \frac{m_1 - m_2}{(m_1 + m_2)} v_{1i} + \frac{2m_2}{(m_1 + m_2)} v_{1i}$$

The details of the solution that has let to the above formula are explained in Appendix A. Once the magnitudes of the two resulting velocities are found, then the direction is calculated with respect to the normal of the collision surface (see Figure 4-17). At this age, the original velocity v is decomposed into its surface and normal components using the same computation as show in previous section. The elastic formula above is only applied on the normal component, thus the surface component is not affected by the collision (it makes sense since the surface component is parallel to the surface).

Note that the magnitude computed with the above formula may result in a negative value, in which case the vertical component will end up heading toward a same direction as the original velocity v (see Figure 4-17). This is a valid reaction when the mass of the colliding object is bigger than the mass of the other object.

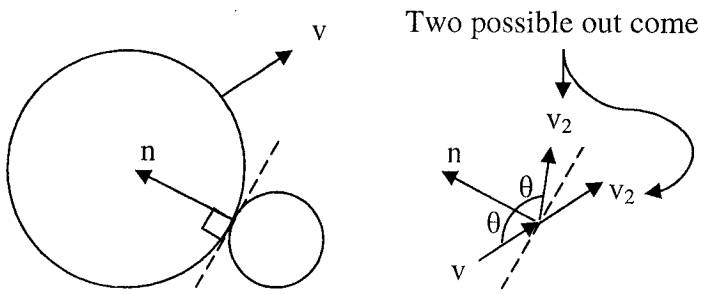


Figure 4-17: The direction of final velocity v_2 depends on the normal of collision surface

4.3.2.3 Collision Law Interface

In section 4.3.1 we specified an interface that must be supported by all laws because the containers will communicate with them through that interface. However the interface of a law is not limited to what is specified in its interface definition. Additional methods may be made available for additional customization, but these methods will only be used by the application directly and not by the containers.

As mentioned in the previous section, collision of one physical object with a fixed object such as a floor creates a bouncing effect which depends on the bouncing factor γ. Our implementation of Collision Law allows the applications to directly control the bouncing factor by providing an interface for setting its value. Figure 4-18 summarized the interface of this Collision Law.

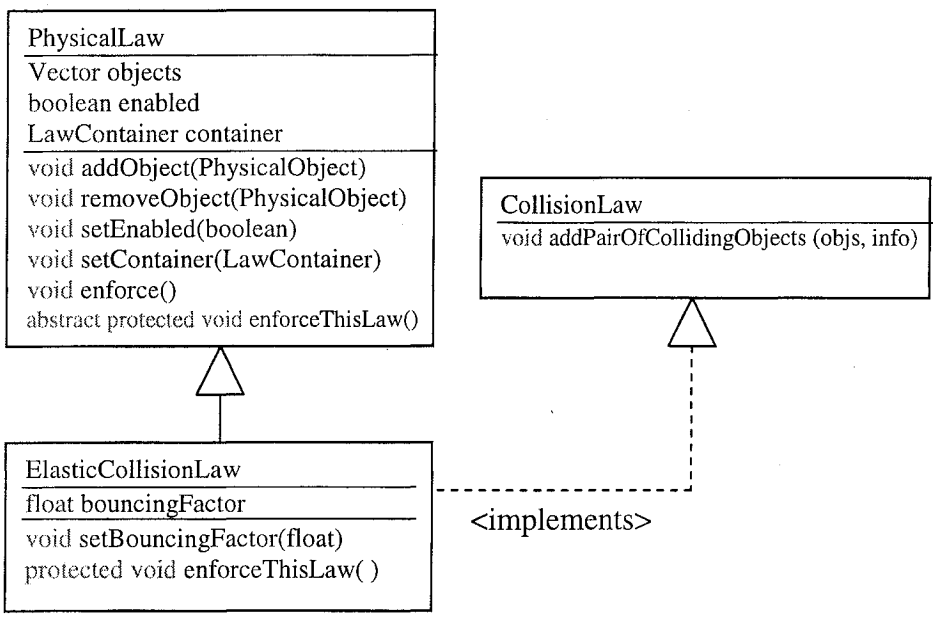


Figure 4-18: Class diagram of ElasticCollisionLaw

4.3.3 Collision Container

The CollisionLawContainer in util package uses Java3D build-in collision detection. Java offers two types of collision detection:

- Collision detection based on bouncing box
- Collision detection based on the exact geometry of the objects

Unfortunately our tests revealed that the geometry based collision detection does not work properly and misses collisions frequently. Therefore, we use Java3D's bounding box collision detection for a rough estimate of the collisions and then we apply our own algorithm to determine more precisely whether the reported objects are indeed in collision or not.

For bounding box collision detection we use the following wakeup conditions:

- WakeupOnCollisionEntry
- WakeupOnCollisionMovement
- WakeupOnCollisionExit

The constructor of each of the above conditions takes two parameters:

- A node that should trigger a collision event if any of its children collide with another object in the scene.
- The collision test technique. In our case, we pass USE_BOUNDS as the parameter to denote that bounding box must be used for collision test.

For every physical object (obj) that must be tested for collision, these conditions are constructed and added to the condition set of CollisionLawContainer:

```
new WakeupOnCollisionEntry(obj, USE_BOUNDS);

new WakeupOnCollisionMovement(obj, USE_BOUNDS);

new WakeupOnCollisionExit(obj, USE_BOUNDS);
```

As a result CollisionLawContainer, unlike GravityContainer, is required to listen for multiple conditions at a same time. For this reason, we use WakeupOr(array) condition which listens to an array of condition and wake the behavior up as soon one of the conditions are triggered.

```
class CollisionLawContainer extends LawContainer {
    WakeupCriterion[] conds;
    WakeupCondition wakeupCondition = new wakeupOr(conds);

    private void addCollidableObject(PhysicalObject obj) {
        WakeupCriterion[] newConds = new WakeupCriterion[conds.length + 3];

        // copy previous conditions to new array
        for (int ii=0; ii<conds.length; ii++)
            newConds[ii] = conds[ii];

        // add collision conditions for new object
        int i = conds.length;
        newConds[i++] = new WakeupOnCollisionEntry(obj, WakeupOnCollisionEntry.USE_BOUNDS);
        newConds[i++] = new WakeupOnCollisionExit(obj, WakeupOnCollisionExit.USE_BOUNDS);
        newConds[i++] = new WakeupOnCollisionMovement(obj, WakeupOnCollisionMovement.USE_BOUNDS);

        conds = newConds;
        this.wakeupCondition = new WakeupOr(conds);
    }
    ...
}
```

Figure 4-19: Code segment of a private function that constructs new collision conditions and appends them to the condition array every time an object is added.

Every time a new law is added to this container or a new object is added to a law that is contained by this container, the private method shown in Figure 4-19 is called to update the condition array. Once the conditions are properly initialized and the physics engine is started, Java3D Behavior Scheduler begins checking for collisions. When the bounds of any pair of objects intersect, the Scheduler triggers the collision condition of that object; this will wake our CollisionContainer up.

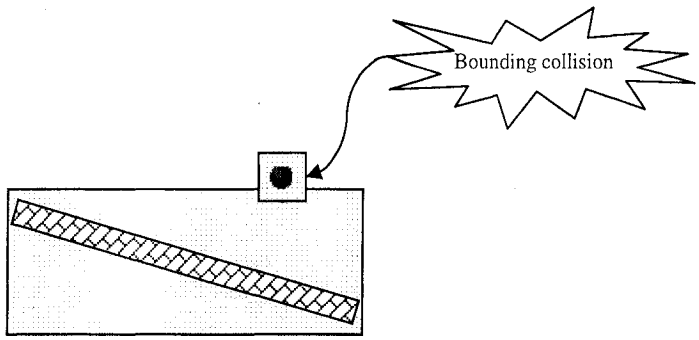


Figure 4-20: Problem with bounding boxes is that their collisions usually occur too early before the actual geometries get close enough.

When the collision container is woken up, we know that there is a possibility of a collision, but we need to perform additional computations to verify it (see Figure 4-23). The key to verification at this stage is using what is called *picking* in Java3D. Picking is a tool that will help us determine what lays ahead of a physical object and how far away (Figure 4-21).

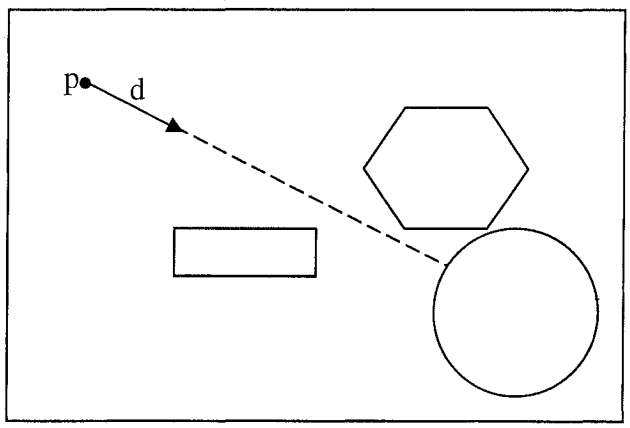


Figure 4-21: Given a starting point p and a direction d, Java3D picking can find the object that lays directly ahead.

The following steps are used to estimate if two objects are in collision:

1. Find the relative velocity of the first object with the second object:

$$V_{relative} = V_{obj1} - V_{obj2}$$

2. From the center of the first object start a pick in the direction of relative velocity. Where the pick tool hits its first surface, it should be on the surface of the 1st object.

3. From the surface of the first object start another pick toward a same direction but limit the distance to a small value τ where τ is the calculated based on the relative velocity of the two objects. If this pick finds any object then we can assume that our two objects are close enough to each other to enforce a collision response (Figure 4-22).

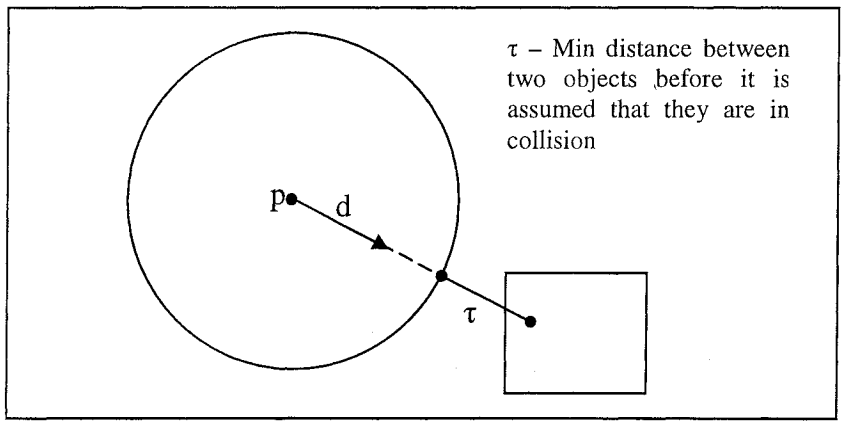


Figure 4-22: Two consequence picks are used to determine if two objects collide or not

The values of τ should be at least equal to the magnitude of the relative velocity multiplied by the update period Δt. This is because at each update period the distance between the two objects is varied by that much. Since it is possible that the behavior scheduler may not wake our container up until one more update periods are passed, it is recommended to use a value two or three times as much as the minimum.

The above steps provide a mechanism for collision avoidance rather than collision detection. This is good but if for any reason a collision could not be avoided, the above mechanism will fail to detect it. In order to include collision detection in addition to collision avoidance we adjust the pick tool to cover a distance range of $-\tau$ to τ.

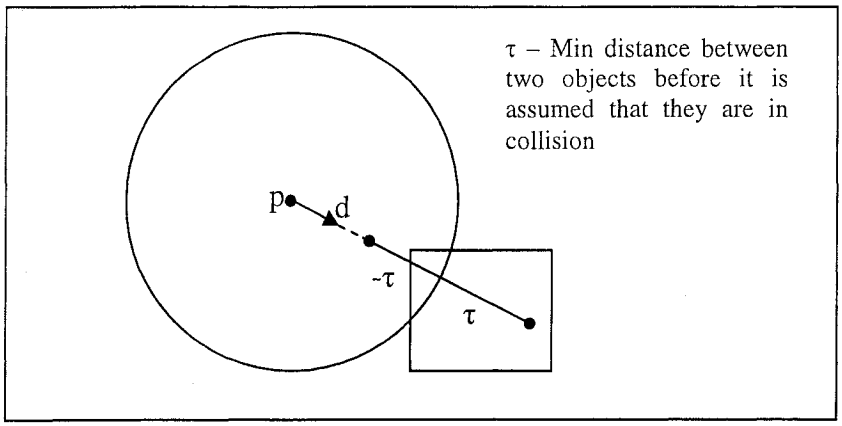


Figure 4-23: Adjustment of collision avoidance mechanism to cover post collision situations as well. Note that if we would only check for range of 0 to +τ then the collision between sphere and box would not be detected.

Once a collision is detected with the above computations, the collision normal is computed and is sent to collision law along with the list of colliding objects. Unfortunately this collision detection is limited to pairs of objects only; that is, the collision with a third object is not reported if two objects are already in collision. This is due to a lack of functionality in Java3D specification. For a more intuitive collision detection feature it is recommended to implement a fast collision detection algorithm independent of Java3D.

4.4 Summary

This chapter demonstrated how physical laws can be implemented within the extensible architecture of our physics engine as discussed in chapter 3. In particular, we saw the implementation of the gravity and elastic collision. For each law we defined an interface, a container and a law implementation. With the aid of the interface specification we are able to modify or replace the implementation of either container or law without changing the other. The vision is that in the future, the application developers can selectively chose their preferred laws or containers from an endless set of third party libraries and simply plug them into their systems without ever needing to see their codes.

CHAPTER 5 - APPLICATIONS

The content of this chapter is focused on both business and technical aspects of developing VR applications with the aid of physics engines. First we will review the needs of VR industries as seen from our point of view; then we shall discover the potential benefits of using an extensible physics engine such as ours in VR applications. The last section is dedicated to the showcase of our demo tool which allows a quick construction of physical scenes for testing purposes.

5.1 Industry needs

The answer to the question "what do industries need?" is a dynamic one. In other words, the needs of industries are constantly changing. The VR industries are no exception. Up until less than a decade ago, software applications were limited by the machine power. The slow CPUs and Graphic Cards resulted in one slim answer to the above question: "Performance".

In the recent years, the power of personal computers has increased exponentially. This has allowed the size of software programs to grow rapidly. Typical software projects now consist of hundreds of thousands lines of code and this has given birth to new software engineering challenges, such as usability and maintainability. No longer is performance the number one need of industries and with the current acceleration in hardware enhancement, it will not be surprising to see in the near future that the performance issues are nothing but a memory of "good old days".

To account for the new needs, major companies are introducing modern software frameworks and development kits, tools that advertise ease of maintainability and usability. Such change of priority is especially visible in the fields of web development, database and enterprise applications. Fields that deal with computer graphics, such as those in gaming and virtual reality are relatively behind as they still value performance.

Although it is reasonable to try to maintain a good performance in computer applications, especially those as demanding as Multimedia and VR applications, it should be noted that one does not need to sacrifice performance to win usability. A well-structured architecture may add a small overhead to the computational load but in the big picture that overhead is not as noticeable. On the other hand, a flexible physics engine such as ours can be customized for different applications with different priorities and requirements; this can ultimately result in a dramatic improvement in the performance. Not all applications need a precise collision detection algorithm and not all applications care to have friction or aerodynamic effects.

Traditionally, the developers of physics engines would estimate the needs of industries and release a development kid that matches the most common needs. The problem with this approach is that the functionalities offered by these physics engine are too little for some applications and too much for the others.

5.2 Embedding Physics in Virtual Reality Applications

It is quite easy to embed our physics engine in existing or new Virtual Reality applications[3]. The first step is to determine the specific needs of the VR application that uses the physics engine. The following steps summarize a typical approach for this:

1. Decide which geometrical objects need to be treated as one physical object. Typically, all geometrical shapes that move and rotate together are grouped as one physical object.

2. For each physical object gather information about the physical attributes such as mass and type of material.

[3] Our current implementation of this physics engine is in Java and on top of Java3D, thus it can be used in Java applications only. However the contents of this thesis are mostly focused on the architecture rather than a specific implementation. Bottom line is that any implementation of this architecture allows an easy integration with existing applications (whether in Java or C++ or some other object oriented programming language)

3. Decide which laws must exist in the application. Also determine the required precision level (i.e. is a collision law based on bounding box is sufficient? or exact collision detection is a must?)

4. Decide which physical objects must obey which laws.

Once the requirements of the targeted VR application are determined then the physics engine can be easily integrated and configured:

1. Add the physics engine library files to the VR application.

2. Construct an instance of the PhysicsEngine component.

3. Construct instances of the PhysicalObject components. For each physical object set its corresponding geometry shape(s) and configure its physical attributes.

4. Construct instance of PhysicaLaw and LawContainer components that meet the functional and non-functional requirements of the targeted application. Configure the attributes of the laws. Plug the laws into the containers and the containers into the physics engine.

5. Categorize the physical objects based on the laws that they must obey. For each category make a list that contains the laws.

6. Add the physical objects to the physics engine while mapping each of them to one of the law lists created in step 5.

7. Start the physics engine.

```
class VRApplication
{
   void main (String args[])
   {
        PhysicsEngine engine = new PhysicsEngine( );          (2)

        GravityLaw gravity = new GravityLaw( );
        CollisionLaw collision = new CollisionLaw( );          (4)
        engine.addLaw(gravity, new GravityLawContainer( ));
        engine.addLaw(collision, new CollisionLawContainer( ));

        Vector collisionLawOnly = new Vector();
        collisionLawOnly.allLaws.add(collision);               (5)

        Vector allLaws = new Vector();
        allLaws.add(gravity);                                  (5)
        allLaws.add(collision);

        PhysicalObject balloon = new PhysicalObject( );
        balloon.addChild(balloonGeomtery( ));                  (3)

        PhysicalObject car = new PhysicalObject( );
        car.addChild(carGeomtery( ));                          (3)

        engine.addObject(balloon, collisionLawOnly);
        engine.addObject(car, allLaws);                        (6)

        engine.start( );                                       (7)
   }
   Shape3D balloonGeomtery( )
   {
        // code to create balloon geometry
   }
}
```

Figure 5-1: Code segment that demonstrates the 7 steps that are involved in the integration of the physics engine with a VR application

Once the physics engine is started, the laws are enforced on the physical objects. Objects that must obey gravity will drop down and objects that must obey collision law will bounce up when they collide with the floor. The VR application can control and communicate with the physics engine by:

- Applying force to the physics objects whenever needed (i.e. when a car is supposed to move forward).
- Adding/removing physical objects dynamically.
- Adding/removing laws dynamically.
- Modifying the attributes of physical objects and laws dynamically.

- Pausing/Resuming the physics engine.

5.3 Libraries of Laws

As this is the birth of the first extensible physics engine, the lack of available law plug-ins is apparent. The current implementation of physics engine has only one implementation of gravity law and one implementation of collision law. So for now, the developers of VR applications need to implement the required physical laws themselves. However the vision is that as such approach gains popularity, there will be an endless variety of laws available for developers to use[4].

Each available law would be accompanied by a documentation that explains not only the component API but also the technical details (such as time or space complexity) of the algorithm(s) that was used, along with a list of its benefits [and drawbacks] in comparison with other similar laws. An application developer who is searching for a specific law that meets the specific requirement of his or her project would look at these documents and decided whether the particular law is suitable or not.

There are infinitely many different laws that exist for simulating different behaviors. Only the collision law by itself can behave in many different ways. For instance, objects may bounce back, break down, deform or even explode upon collision. These are all examples of some of the common events that occur in many games and simulations; thus it is apparent that making them available to the public within reusable components can save a great deal of time and money for the VR industries.

[4] Whether a law is available free of charge or with a cost depends on the company that has created it. Either way, the approach is accompanied with a whole new marketing opportunity for both application developers and component developers.

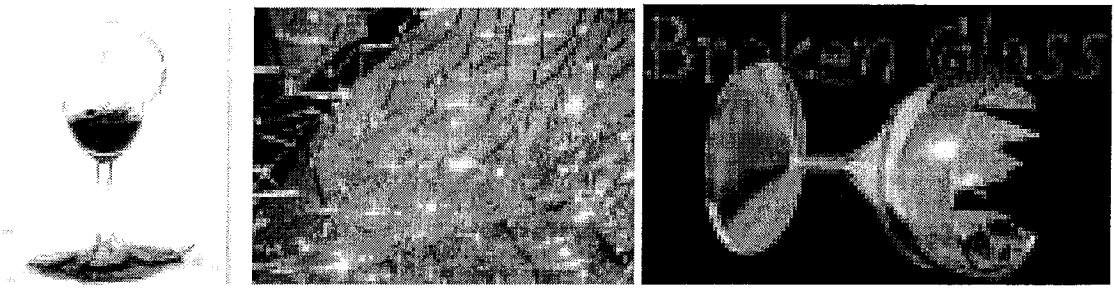
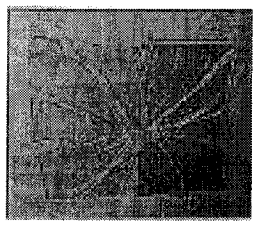


Figure 5-2: There could be infinitely many different algorithms for computing the break down of a glass. The glass of the window shield of a car breaks down differently than a wine glass or a pair of eye glasses. Also the glass of a window shield that is dropped on the floor breaks down differently than one that is hit by a bullet.



5.4 A Tool for Constructing and Testing Physical Scenes

We have developed a tool that demonstrates the ease and power of using our physics engine. The primary role of this tool is to help application developers to easily configure and test the features and laws that are readily available to them through the physics engine (and/or third part law libraries). The tool can also be used by the component developers who want to test their laws.

Simply put, this tool is a GUI (Graphical User Interface) that is built on top on the physics engine and uses the available API to set/get the parameters for configuring the engine and constructing physical objects and laws. The current version allows to:

- Star/Reset and Pause/Resume the Physics engine (see Figure 5-7);
- Construct Physical Objects with basic geometries (sphere and cubes);
- Map the objects to the existing laws (see Figure 5-4);
- Dynamically change the physical attributes such as mass and elasticity and non-physical attributes such as position and orientation (see Figure 5-5);
- Dynamically enable/disable the laws (see Figure 5-9);

- Dynamically add forces to each physical object (see Figure 5-5);
- Quickly rearrange and reconfigure the scene by using the shortcut buttons (see Figure 5-9 and Figure 5-10).

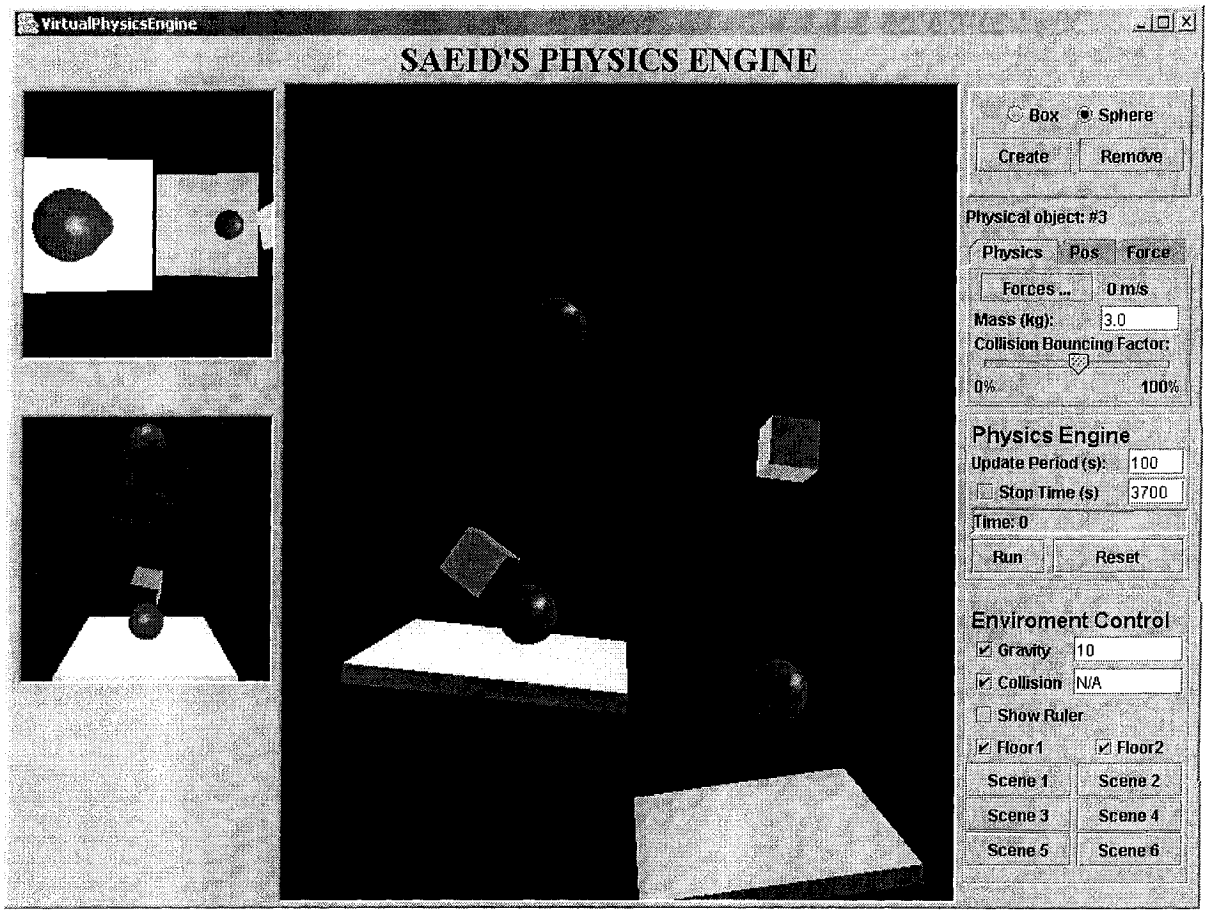


Figure 5-3: Main windows of the physics engine tool used for constructing and testing physical scenes

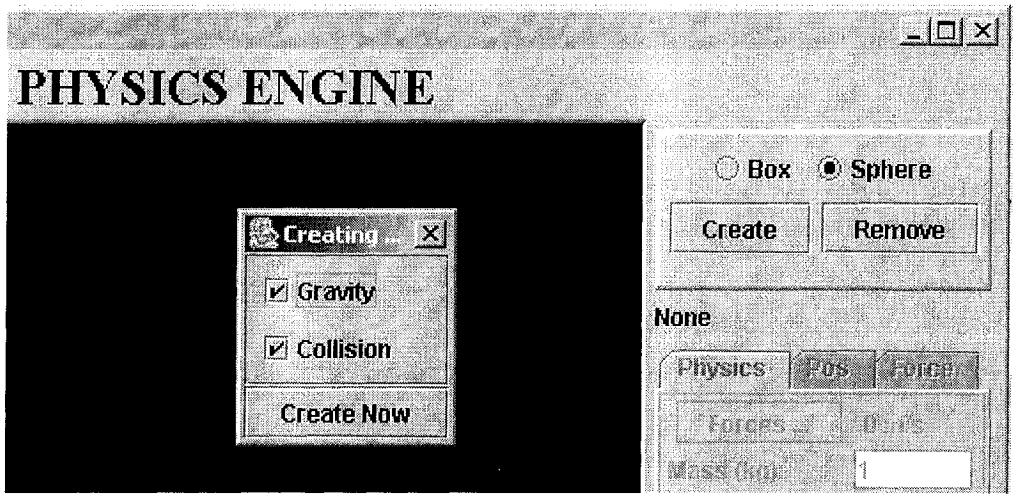


Figure 5-4: Objects are mapped to available laws before they are constructed.

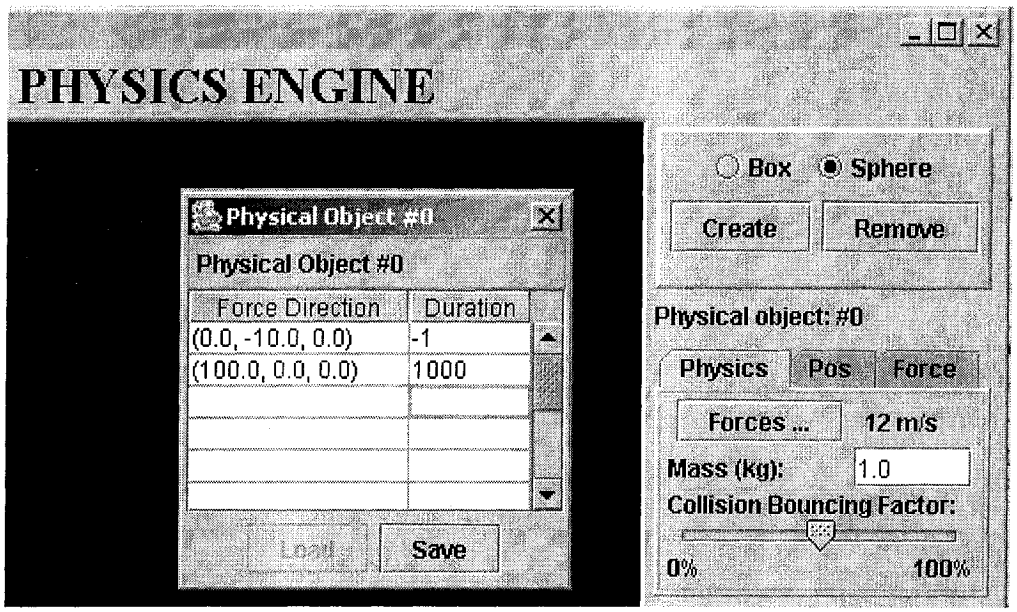


Figure 5-5: The physical attributes of objects such as mass and elasticity can be set in the "Physics" panel. The users can also see a list of current forces acting on a specific object by clicking on that object and pressing the "Forces" button. In this screen shot, the first force with duration -1 is the downward force that is applied by gravity. The second force with duration 1000ms is dynamically applied to the object by simply typing it down. This configuration can be saved and loaded again whenever required.

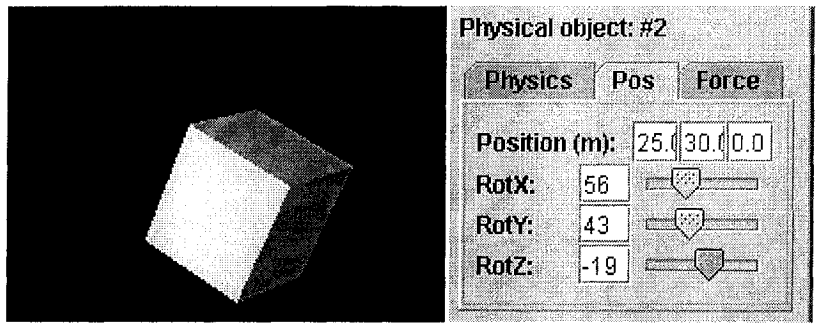


Figure 5-6: For better visualization and test controls, objects can be placed at any position with any orientation. For a precise position and orientation, the values can be entered in the text boxes of "Pos" frame. Alternatively objects can be moved and rotated with the mouse.

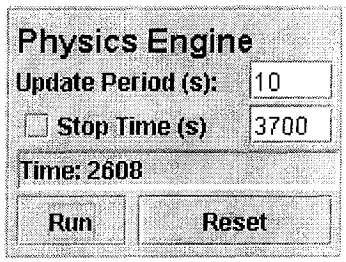


Figure 5-7: The physics engine can be configured and controlled with this control panel. The Reset button is used to set the timer back to zero. Run is used to start/resume the simulation. For precise measurements, the tester may want to suspend the engine after a specific duration. In this case, "Stop Time" must be checked and the stop time must be entered in the appropriate text box.

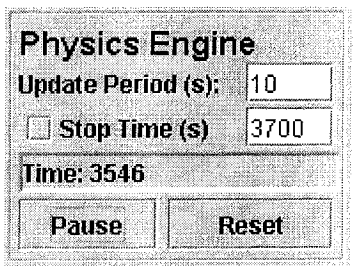


Figure 5-8: After the "Run" button is clicked, the simulation starts and the caption of the "Run" button is changed to "Pause". Clicking on "Pause" button suspends the simulation until run is clicked again.

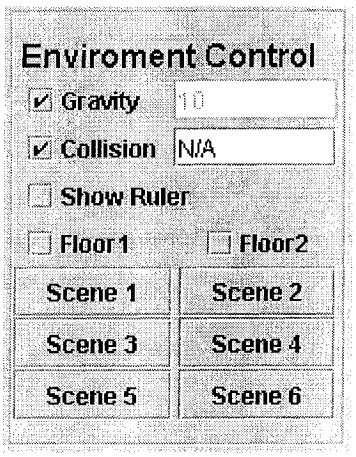


Figure 5-9: Physical laws can be dynamically enabled/disabled or modified through this panel. Also, for testing purposes, a distance ruler and up to two fixed floors can be added to the scene. The button "Scene1" through "Scene6" are a set of preconfigured scenes used for a quick switch of configurations and rearrangement of objects.

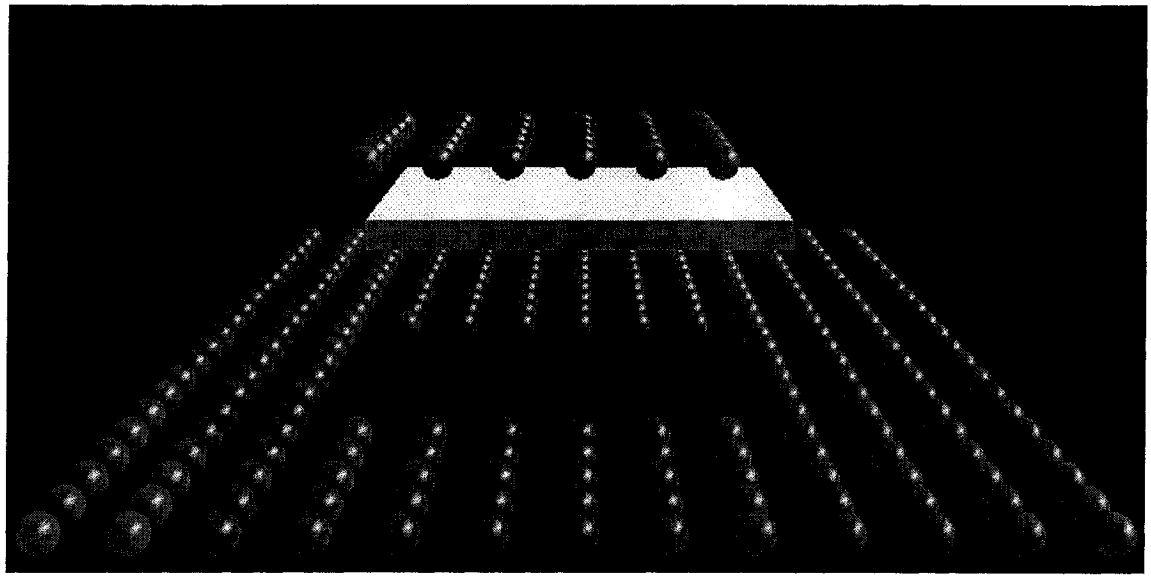


Figure 5-10: A scenario with many physical objects for testing the performance.

5.5 Summary

The flexible structure of our physics engine allows it to be configured for different applications with different needs. In addition, the development process of VR applications can be dramatically sped up by reusing the laws that are developed by others. The growth of a physics engine with this design is accompanied by a growth in the variety of reusable laws and containers that ultimately server two purposes:

- More variety of special effects in VR application such as geometrical deformation, explosion, weather effects and many more.
- Ease of balance between performance and precision by having many implementations of a same law with different algorithms.

Based on the current implementation of this physics engine, we have developed a tool that wraps a convenient graphical interface over the physics engine API in order to allow a quick and easy visualization of the physical behaviors. This handy tool can be used by both application developers and law component developers.

CHAPTER 6 - CONCLUSION

The use of physics engines in the development of virtual environments has noticeably improved the quality of modern VR applications and to some degree has eased up the efforts involved in developing them. However the components of current physics engines are tightly coupled with each other, making it impossible to reuse or extend their features.

This thesis introduced a new approach that addresses the existing problems with the current physics engines. In chapter 2, we learned that the physical objects in a virtual environment can be represented by the instances of a Class that uses instance variables to hold physical attributes such as position and force. Furthermore, physical laws can be represented by the instance of Classes that are event-driven and contain mathematical equations for enforcing laws. By using an object-oriented architecture, we succeeded in decoupling the components to a degree at which communication is done only through a well-defined interface.

In chapter 3, we described how to decompose laws further down into two components: Container component, that is event-driven and in charge of detecting situations that must enforce a law, and Plug-in component, that consists of the mathematical computations that must be executed as a response to the triggering event. With the aid of inheritance and a container/plug-in design pattern, it is made possible to extend and customize the behavior of the physics engine indefinitely. In addition, laws and containers can now be distributed within independent components that can be conveniently plugged into the physics engine without needing to access the source code of the physics engine.

Chapter 4 demonstrated a detailed example of implementing two fundamental laws of physics using the container/plug-in architecture.

In chapter 5, we described the potential benefits of an extensible and reusable physics engine for industries. We then discussed the technical and business issues involved in integrating the physics engine with VR applications. The proposed physics engine is implemented within a library called xPheve (Extensible Physics Engine for Virtual

Environments). In addition, there is a tool that wraps a GUI interface on top of the xPheve API and provides a convenient mean of constructing an arbitrary scene and applying physical laws to it. Existing VR applications can easily integrate xPheve and benefit from its various features. Maintaining such applications is a simple task because there is a clear separation between the components in charge of enforcing physical laws and those responsible for the high-level functionality of the application. Furthermore, when new laws become available, they can be plugged into the engine without need of recoding the application itself.

For future work, this architecture can be further extended in order to cover the special needs of the critical systems in which execution of laws must occur in a predefined order. Also, additional laws need to be implemented in order to further verify the extensibility of the architecture and make the required adjustments to correspond with the uncounted difficulties. More detailed experiments must be made, in order to measure the performance of the system in a precise manner.

APPENDIX A – Elastic Collisions

[16]

$$\begin{aligned} \vec{p}_{1i} + \vec{p}_{2i} &= \vec{p}_{1f} + \vec{p}_{2f} \\ m_1 v_{1i} + m_2 v_{2i} &= m_1 v_{1f} + m_2 v_{2f} \\ E_{k1i} + E_{k2i} &= E_{k1f} + E_{k2f} \end{aligned} \tag{206}$$

$$\frac{1}{2} m_1 v_{1i}^2 + \frac{1}{2} m_2 v_{2i}^2 = \frac{1}{2} m_1 v_{1f}^2 + \frac{1}{2} m_2 v_{2f}^2 \tag{207}$$

are the two relations that follow from momentum and energy conservation (in the one, e.g. - x - direction). Assuming we know m_1, m_2, v_{1i} and v_{2i}, can we find v_{1f} and v_{2f} ?

Yes. There are three ways to proceed. One is to directly solve these two equations simultaneously. This involves solving an annoying quadratic and we will avoid it. The second is to note that we can do the following rearrangement of the energy and momentum equations:

$$\begin{aligned} m_1 v_{1i}^2 - m_1 v_{1f}^2 &= m_2 v_{2f}^2 - m_2 v_{2i}^2 \\ m_1(v_{1i} - v_{1f})(v_{1i} + v_{1f}) &= m_2(v_{2f} - v_{2i})(v_{2i} + v_{2f}) \end{aligned} \tag{208}$$

(from energy conservation) and

$$m_1(v_{1i} - v_{2f}) = m_2(v_{2f} - v_{2i}) \tag{209}$$

(from momentum conservation). When we divide the first of these by the second, we get:

$$\begin{aligned} (v_{1i} + v_{1f}) &= (v_{2i} + v_{2f}) \\ (v_{2f} - v_{1f}) &= -(v_{2i} - v_{1i}) \end{aligned} \tag{210}$$

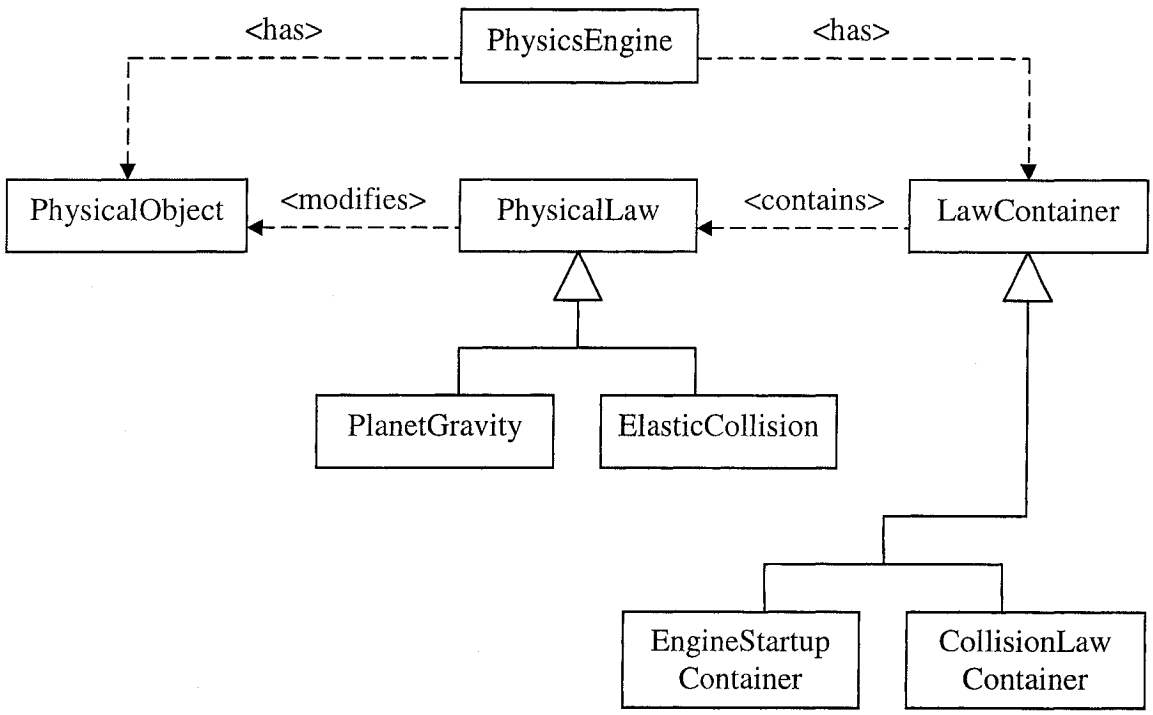
or **the relative speed of approach before a collision equals the relative speed of recession after a collision**.

Although it isn't obvious, this equation is independent from the momentum conservation equation and can be used with it to solve for v_{1f} and v_{2f}, e.g. -

$$\begin{aligned} v_{2f} &= v_{1f} - (v_{2i} - v_{1i}) \\ m_1 v_{1i} + m_2 v_{2i} &= m_1 v_{1f} + m_2(v_{1f} - (v_{2i} - v_{1i})) \\ (m_1 + m_2) v_{1f} &= 2m_2 v_{2i} + (m_1 - m_2) v_{1i} \\ v_{1f} &= \frac{m_1 - m_2}{(m_1 + m_2)} v_{1i} + \frac{2m_2}{(m_1 + m_2)} v_{1i} \end{aligned} \tag{211}$$

(back substitute for v_{2f}).

APPENDIX B – xPheve UML Diagram



REFERENCES

[1]: X. Shen, T.Radakrishnan and N.D. Georganas "vCOM: Electronic Commerce in a Collaborative Virtual World" , Electronic Commerce Research and Applications J. (Publisher: Elsevier Science)), Vo.1, No.3-4, Aut.-Winter2002 , pp. 281-300

[2]: X.Zhong, P.Liu, N.D.Georganas and P.Boulanger, "Designing a Vision Based Collaborative Augmented Reality Application for Industrial Training", it-Information Technology, Vol. 45, No. 1 (2003), pp.7-18

[3]: P. Boulanger , N.D. Geoganas, X.Zhong and P. Liu, "A Real-Time Augmented Reality System for Industrial Tele-Training ", Proc. Videometrics VII, IS&T/SPIE 15th annual Symp. on Electronic Imaging, Santa Clara, California, Jan. 2003, pp1-2

[4]: M.Hosseini, F.Malric and N.D.Georganas, "A Haptic Virtual Environment for Industrial Training", Proc. HAVE'2002 - IEEE Workshop on Haptic Virtual Environments and their Applications, Ottawa, ON, Canada, Nov. 2002, pp.37-41.

[5]: Introduction to Java3D, *by Jack S. Gundrum*, (as of Jan 2004): http://viz.aset.psu.edu/jack/java3d/web2001.html

[6]: Java 3D API Specification, *Sun Microsystems*, (as of Jan 2004): http://java.sun.com/products/java-media/3D/forDevelopers/j3dguide/Intro.doc.html

[7]: The Newtonian Object-oriented Physics Engine – NOOPE, *by Paul Evans, James Kermode, Miklós Reiter* (as of Jan 2004): http://www.srcf.ucam.org/nope/

[8]: Havok Game Dynamics SDK, (as of Jan 2004): http://oldsite.havok.com/products/game_dynamics/index.html

[9]: Karma, MathEngine Toolkit, (as of Jan 2004): http://www.mathengine.com/_mathengine_corp/_products/toolkits.html

[10]: Simulating Physics, CMLab, (as of Jan 2004): http://www.cm-labs.com/products/vortex/algorithms.php

[11]: A Framework for Physically Based Modelling in Virtual Environments, *PhD thesis by Mashhuda Glencross,* Department of Computer Science, University of Manchester, 2000, pp.32-69

[12]: Modelling Natural Meteorological Phenomena, *paper by Mashhuda Khote,* Department of Computer Science, University of Manchester, 1993.

[13]: Physically Based Modeling: Principles and Practice, Andrew Witkin and David Baraff, (as of Jan 2004): http://www-2.cs.cmu.edu/afs/cs.cmu.edu/user/baraff/www/sigcourse/index.html

[14]: Kelvin Chung and Wenping Wang, "Quick Collision Detection of Polytopes in Virtual Environments", ACM Symposium on Virtual Reality Software and Technology 1996, University of Hong Kong, Hong Kong, pp. 18-32

[15]: Computer Graphics, Scientific Visualization and Virtual Reality, 2003 ViewTec AG, "The perception of natural and synthetic images", http://www.viewtec.ch/techdiv/vr_info_e.html

[16]: Elastic Collisions, Conservation of Momentum, Collision Response computation, Robert G. Brown 2002-08-26, http://www.phy.duke.edu/~rgb/Class/phy41/node40.html

[17] Component Interface Pattern, *paper by Ricardo Pereira e Silva*, pp.10-11, (as of Jan 2004): http://time-tripper.com/uipatterns/

[18] Component Interface Pattern, *paper by Ricardo Pereira e Silva*, pp.10-11, (as of Jan 2004): http://time-tripper.com/uipatterns/

[19] The J2EE 1.4 Tutorial, *Sun Microsystems Inc*, Chapter 18-Enterprise Beans, (as of Jan 2004):
http://java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html