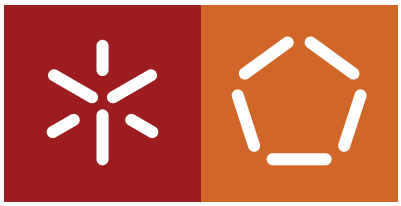
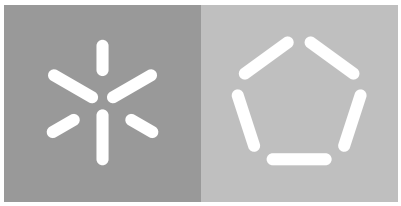


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Alexandre Ventosa da Silva

A fully configurable virtual laboratory of classical mechanics

November 2018



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Alexandre Ventosa da Silva

A fully configurable virtual laboratory of classical mechanics

Master dissertation
Master Degree in Computer Science

Dissertation supervised by
António Ramires Fernandes

November 2018

ABSTRACT

Nowadays many mathematical applications allow the user to introduce its own equations in the system and also observe through different possibilities the desired results. Regarding physics, an extended range of virtual laboratories allow the user to accomplish virtual physics experiments. These virtual laboratories consist in predefined scenarios where the user can change the value of the physics variables and then visualise the changes accomplished. Other virtual laboratories uses a physics engine allowing the user to create its own scenarios. However, the physical behaviour of the objects is hardcoded since it results strictly on the physics equations used internally by the physics engine.

This dissertation pretends to investigate how far and with what degree of scientific rigor it is possible to associate the idea of the user introducing its own equations with the idea of accomplishing virtual experiments of physics. As a proof of concept, this dissertation focus on a specific area of mechanics: the dynamic of rigid bodies. The result of this research is a virtual laboratory completely different relatively the others.

Our system has no knowledge about physics. Even the most general laws of physics such as the Newton's second law are not known by the system. To the system, any equation introduced is considered just as one more equation without any particular meaning associated to it. The same happens for any physics entity. For example, if the gravitational acceleration is introduced by the user, to the system it is just another attribute of the world.

Taking into account the dynamics of rigid bodies, an object can be identified as being, at any time, in one of three different states. These are: when a object is not in contact with any other, when an object collides with another object and they immediately separate, and when two objects remain in contact over time. The user must specify all the equations that drive each of these three states. Using its geometrical knowledge, the engine determines at any time in which state an object is. Also, the system provides all the relevant geometrical information. For instance, in a collision between two objects, the point and the two normals vectors of the collision are provided.

The graphical simulations reflects strictly on the equations introduced. Therefore, if the equations to solve a collision between two objects does not reflect the real underlying physics of the situation, it is possible that the objects simply ends-up penetrating each other. All the relevant numerical information about an experience can be processed through different forms. In fact, the user can request plots of variables, the graphical application of vectors on objects, and even the tracing of the variables at a specific event.

ACKNOWLEDGEMENT

I would first like to thank my dissertation advisor Professor Doctor António Ramires Fernandes of the School of Engineering at University of Minho. The door of Prof. Ramires's office was always open whenever I had a question about my research or the writing of the dissertation. I also consider that the weekly meetings we set ourselves were certainly a great help to the success of this research. He consistently allowed this dissertation to be my own work, but steered me in the right direction whenever he thought I needed it.

I also must express my very profound gratitude to my parents for providing me with unfailing support and continuous encouragement during those five years of study. This accomplishment would not have been possible without them. Thank you.

Alexandre

CONTENTS

LIST OF FIGURES

1

INTRODUCTION

The main theme of this dissertation is virtual physics simulation. While there are many readily available 2D/3D simulations, these are usually hand crafted for a specific simulation. Furthermore, the user is only allowed to play with the simulation's parameters, the underlying physics is most of the time hidden from the user.

We propose a different approach to build these simulations, which we could categorise as "do it yourself physics". In our proposal the user gets full control, not only of the parameters but also of the underlying physics, i.e., the equations themselves. In fact, before our system can simulate any physical phenomena the user has to introduce the equations that govern the specific phenomena.

In this chapter an initial approach to the problem is presented. The reasons justifying the need for this investigation are founded. An overview about the objectives we intend to achieve with this investigation is also presented.

1.1 DESCRIPTION OF THE PROBLEM

The lack of adequate and necessary laboratory material in many educational facilities, or allowing a student, in a home-work context, to learn physics through a virtual test environment, has led to the development of the most varied tools that reproduce through virtual laboratories different physics experiments.

The main objective of these virtual laboratories of Physics is to provide the student with new knowledge in the fields of Physics that he is studying. In fact, in physics classes, the focus given to the laboratory is central. Much more than a place at the back of the classroom, it is in the laboratory that physics students "do" physics. It is in this test environment that the students start out the typical activities of scientists - questioning themselves, applying procedures, collecting data, analysing data, answering questions from their colleagues and teachers, or thinking about new issues and possible scenarios to explore.

Currently, there are several software solutions that allow the student to recreate in a virtual environment certain Classical Physics experiments. These platforms offer a wide range of experiences covering the various areas of physics: hydrodynamics, hydrostatics,

kinematics, dynamics, electromagnetism, among others. This "supply" variety brings with it the disadvantage that, in one experiment, the user can only change a small set of parameters. It is also known that these experiments are generally defined based on a set of predefined scenarios, and it is not possible to change them. These characteristics allow to visualise the result of a certain test configuration and the variation, over time, of the values of the relevant physical variables which come within the context of the experiment. In short, the physics is built-in in these simulations, and the student is only allowed to set parameters. Most of the time the student does not even have access to the equations used in the simulation.

1.2 WORK MOTIVATION

The approach taken by these pedagogical tools does not allow the student to develop an in-depth knowledge and skills about "why" or "how" a particular physical phenomenon occurs. In fact, we believe that only by knowing the underlying Physics that sustains a particular physical phenomenon, is it possible to fully understand the behaviours that occur in an experience and apply the knowledge in other experiences or in real cases.

This research intends to create a tool that exploits a new paradigm in the teaching of Physics through virtual laboratories. In this sense, we propose a generic engine that has no knowledge of physics. It becomes the user responsibility to introduce all the mathematical formulas and equations that support a certain physical phenomenon that one wants to study. Even the most general laws of physics, such as, for example, Newton's second law, would not be known by the system a priori. This drastic approach may at first seem unnecessary and would complicate the work of potential users. However, as previously stated, we believe that only with a complete idea of the formulas and equations that underlie a physical occurrence, is it possible to properly extract and apply the concepts outside the restricted and closed context that characterise virtual experiences. As will be shown, to properly animate these equations the engine only needs to have built-in geometry knowledge. 3D simulations are provided by the engine based on the geometry it knows, and the equations the user inputs. As a proof of concept this dissertation focus on a specific subarea of classical mechanics: the dynamics of rigid bodies.

1.3 STRUCTURING OF THE DOCUMENT

In addition to the present chapter, this dissertation is structured in 5 chapters. A State of the art is presented in chapter 2. In chapter 3 our approach to the problem is presented. Chapter 4 covers how in practise an experience can be defined following the approach described in chapter 3. To demonstrate that the developed tool is sufficiently generic, in chapter 5,

we cover the steps about how the actual physics equations that drive the mechanics of rigid bodies can be specified in our tool. Using the graphical interface of the developed software, in chapter 6, we present various features of the tool related to how the numerical data inherent to a specific experience can be processed and visualised through various possibilities. Finally, a conclusion about all the work developed is presented in chapter 7.

2

STATE OF THE ART

Nowadays there are many programs dedicated to the teaching of physics that allow to recreate, in a virtual environment, experiments of physics such Central Connecticut State University, Neumann, Nunn, Duncan, Virtual Science Ltd, Xanthopoulos, Design Simulation Tecnologies, University of Colorado Boulder, among many others. The degree of freedom and configuration of the scenario and the bodies composing an experience, as well as the possibility of specifying a concrete response to certain events, such as a collision between two bodies, differs among solutions. In fact, all the applications analysed in the context of this dissertation implement the response to this type of events in a "hardcoded" way. It means that it is not possible for the user to specify the behaviour he would like to observe, or to really understand what physical foundations and mathematical equations are behind the observed physical phenomenon. The bodies or particles of an experiment possess a closed and predefined set of attributes. Each attribute corresponds to a certain physical variable which is necessary for the context of the experience. For example, in classical mechanics, one of these object attributes can be the mass, or the coefficient of restitution.

The possibility and also the way it is possible to reach conclusions about the data related to the events of the experience, for example, by providing plots of the physical variables, also differs greatly from one tool to another.

Given the wide range of software applications associated with this problem, it would not be possible to present in this chapter the characteristics and functionalities of each one individually. Thus, a set of the applications representing a relevant sample of the solutions is described below. In selecting the applications that would integrate this sample, care was taken to choose those whose functionalities and characteristics differ from the rest so that we present a broad idea of what is currently available and what functionalities can be extracted from this kind of tools. Also, in the case of solutions that offer similar functionality to the users, we choose the ones that appear to be more popular and used by a larger number of people.

In sections 2.1, 2.2, 2.3, 2.4 and 2.5, the main features and functionality of *Physion* Xanthopoulos, *VPLab* Nunn, *Interactive Physics* Design Simulation Tecnologies, *Phet Interactive*

Simulations University of Colorado Boulder and *My Physics Lab* Neumann are presented, respectively.

2.1 PHYSION

Physion is a computer program created by Dimitris Xanthopoulos in 2010 that allows the simulation of physics, in a two-dimensional way, associated with the mechanics of rigid bodies. This program can be used to easily create a wide range of interactive physics simulations and educational experiences. The application uses a physics engine called Box2D Catto to detect and resolve collisions between bodies present in a given scenario.

Using the tools provided by the application, the user can create several physical objects, such as circles, polygons, ropes, pulleys, among others. Each rigid body present in an experiment can be a static or dynamic body. A static body has a fixed initial position assigned and will not move under any circumstances during all the time of the experiment. A dynamic body is a free body not subjected to any restriction of movement, so forces acting on it can induce a change in its velocity. The application provides an attribute editor. It is thus possible to specify for each body its density, its position relative to the two dimensions of the scene, its restitution coefficient, the type of body (rigid or dynamic), its initial linear and angular velocity, among others. The user has to restrict himself to this finite series of predefined attributes because it is not possible for the user to declare new ones.

Through the window containing the scene properties, the user can specify the numerical value to be assigned to the gravitational acceleration in the two orthogonal axes or the frequency at which the scene is recalculated. The scene and its visualisation is configurable, therefore it is possible to change the background colour or the attributes related to the camera.

By default, a newly created scenario already contains three rectangles that form a static container. To static bodies is applied a brick texture to emphasise the idea that it is a fixed element of the scenario. Figure 1 represents a circle and a spring that connects the circle and the predefined left vertical rectangle. Accessing the spring attributes editor, it is possible to change the proper attributes of a spring, such as the constant k of the spring. The constant k corresponds to the elasticity of the spring. Physion uses Hooke's law to calculate the force exerted by a spring:

$$F = -k * x \tag{1}$$

The greater the displacement x in relation to the steady state of the spring and the greater constant k, the greater is the force exerted by the spring. Hooke's law is used internally by the system and the user can not modify the law or specify its own equation.

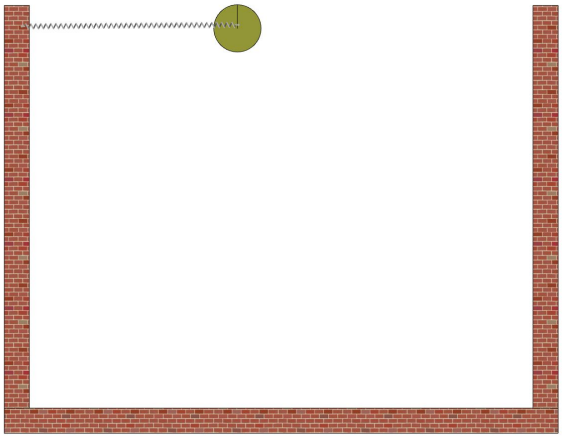


Figure 1.: A simple Physion scenario composed by the predefined container, a spring and also a sphere

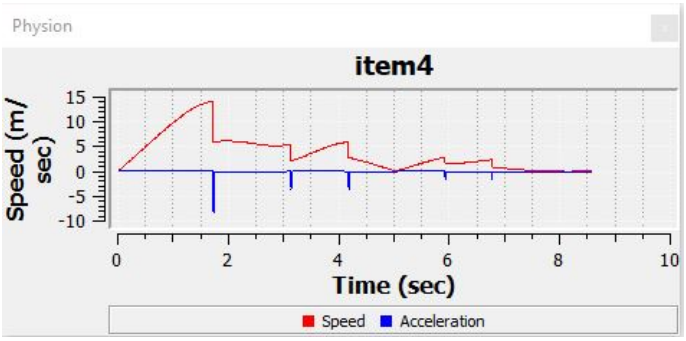


Figure 2.: An example of a plot generated by Physion relative to a attribute of an object that integrates an experience.

By changing the restitution coefficient of the circle, the spring constant, or the circle density, viewing the new graphical simulation associated with the new configuration allows us to understand quickly the effects induced by the changes made. For instance, for a density equal to 1, the circle does not touch the horizontal rectangle due to the force exerted by the spring, whereas for a density of 100, the spring can not counteract the gravitational force exerted on the circle, so it ends up quickly and inevitably colliding with the horizontal rectangle.

Physion allows the creation of plots. However, just to observe how the acceleration and speed of a particular body varies over time of experience. Figure 2 represents the plot obtained for the speed of a circle with a given initial position and radius, relative to the scenario previously described and for the concrete case of its density to be equal to 100.

One advantage of Physion is the ability for the user to enter their scripts written in javaScript. It is therefore possible for the user to specify through scripts some kind of behaviour that could not be possible to be specified by the conventional tools provided to the user. However, the script cannot be used to change the underlying physics of an

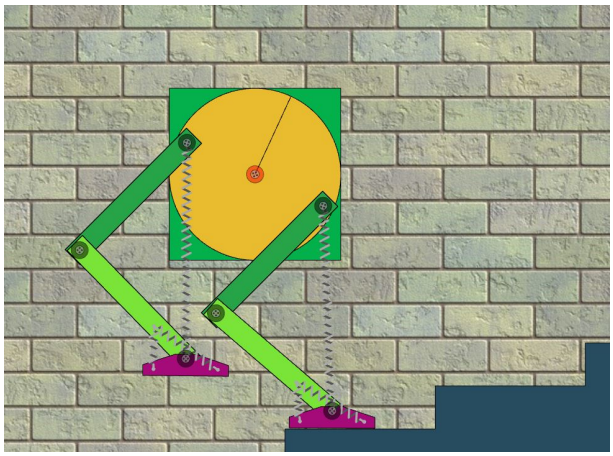


Figure 3.: Gadget created in Physion capable of climbing stairs

experience. All the physics equations are hardcoded in the physics engine used by the application. With that said, the degree of configurability remains limited and the scripts are mainly used for artistic purposes, such as for changing the colour of a spring according to its deformation at a given moment, or for specifying "invisible" forces, i.e., forces that act at distance, such as the gravitational force generated by a very large mass, and that acts in each of the bodies that is in its vicinity.

By combining conveniently the various primitives offered by the application it is possible to create interesting and relatively complex experiences. Figure 3 illustrates an experiment created with Physion where it is possible to observe a robot, composed of several springs and rectangles articulated together, that is able to climb the steps of a staircase.

2.2 VPLAB

Virtual Physical Laboratory (VPLab) is a suite of more than three hundred small interactive physics simulations. It is a paid tool and only available for Windows. However, it is possible to have access to a demonstration of the tool through seven executables, corresponding to seven different experiments. John Nunn began to develop this tool in the year 2000, and today is perhaps the most well-known and used tool for teaching of Physics in high schools around the globe. The tool is approved by the National Physical Laboratory, the UK's national weights and measures laboratory, based at Bushy Park in Teddington.

In all simulations provided by VPLab, a level of difficulty is assigned so one can see if the student's age corresponds to the expected age and its experience. "Basic" targets 12-16 year old's, while an "Advanced Level" experience is expected to be solved by a student between the ages of 17 and 19.

The three hundred and thirty-four experiments provided by the software are grouped into thirty-one different categories, corresponding to the different areas of research covered by Physics. In this sense, the tool makes it possible to perform and assimilate knowledge about astronomy, or from a completely different area such as Electromagnetism or Fluid Mechanics.

An experiment is represented by a predefined scenario. Figure 4 illustrates the corresponding graphical interface of an experiment included in the set of free samples. This experiment aims to present and explain the basic concepts of electricity. As it can be seen, the experiment is composed of a small electric circuit, consisting on a battery, an ammeter and a resistance. The user can learn about the theoretical concepts related to the experience or about the functionalities and the purpose intended for it. Figure 5 shows the two views presented to the user, the one on the left corresponds to the instructions, while the one on the right presents the theory.

By playing around with the two buttons associated with the battery voltage and the value of the resistance, the user sees interesting things happening. Thus, an increase in the voltage of the battery, causes an increase in the energy transported by each electric charge. Also, the charges begin to move faster. Relatively to the resistance, an increase in its value causes the opposite effect, that is, the charges move more slowly. In fact, there is a decrease in the value of the current (due to a greater dissipation of energy in the resistance). These phenomena are due to Ohm's law:

$$V = R * I \tag{2}$$

The graphical representation of the ammeter is an added value in the context of the experiment, since it allows to confirm the Ohm's Law by observing what the law predicts when changing the values of the voltage and the resistance.

All of the experiences provided by VPLab follow the same philosophy. It is possible to have access to a window explaining the theory behind the experiment, as well as another window explaining in detail what it is possible for the user to change (physical variables) and what it is expected to happen from the changes made. The degree of configuration by the user is limited and is essentially based on the change of the numerical value associated with a particular physical variable.

2.3 Interactive Physics

Interactive Physics is an application developed by Design Simulation Technologies that consists of a virtual environment in which it is possible for the user to set up their own test scenario through the tools provided by the program. There is a large community composed mostly of teachers who contribute to the creation of new experiences. Therefore, there is

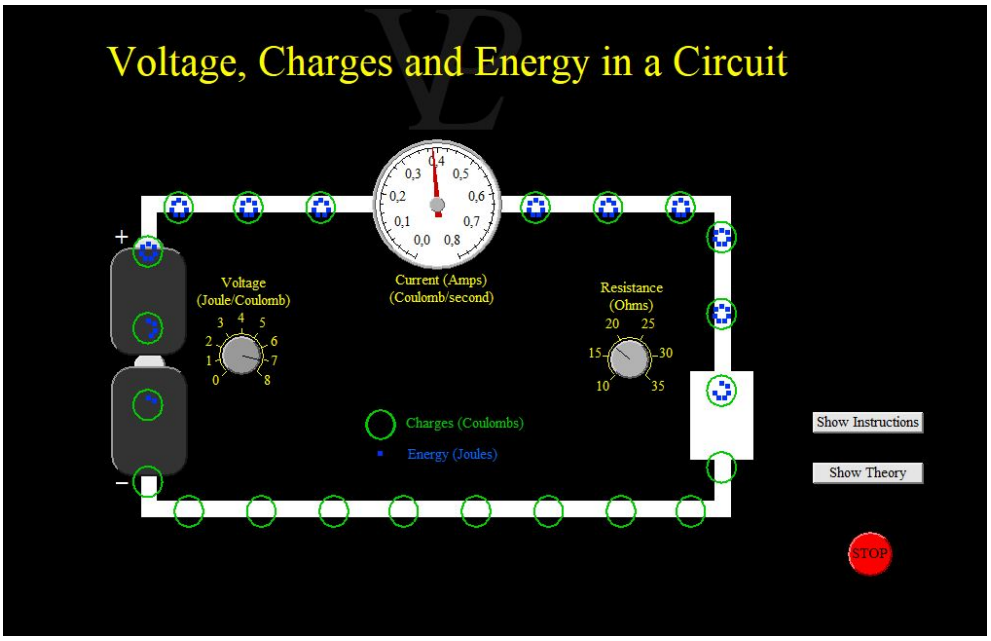


Figure 4.: Main window of experience "Explaining Electricity" that is part of the suite of experiences provided by VPLab

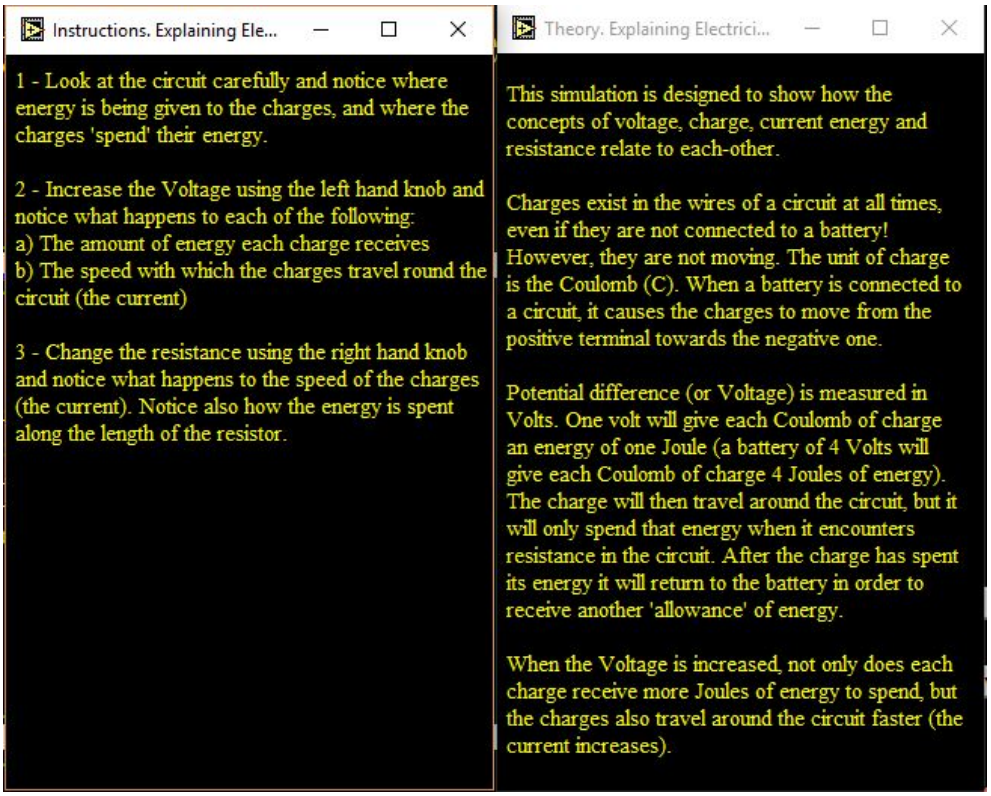


Figure 5.: From left to right, the physics theory and the instructions necessary to accomplish properly experience "Explaining Electricity"

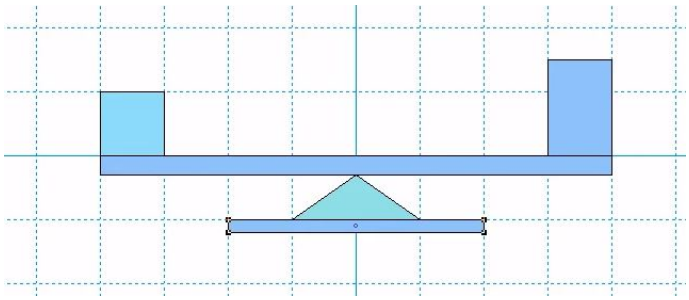


Figure 6.: A possible scenario in Interactive Physics composed by several rigid bodies to represent a scale and two weights

the possibility of anyone submitting their experience through the official website of the tool and, if approved, the experience will be available to the community. Using this tool it is possible to perform physics experiments that fall into the topics of collisions, electrostatics, gravitation, friction, kinematics, oscillations, projectiles, among others. The simulations are performed in a two dimensional environment.

The program allows the user to include a wide range of possibilities for the objects to integrate in the test environment. For instance, it is possible to create objects by drawing circles, blocks or any other type of polygons. It is also possible to create more complex objects such as ropes, springs, pulleys, among others. The application allows to simulate contact between bodies, as well as collisions or the frictional forces involved. Several families of forces are predefined. Setting their numerical value, allows the user to quickly observe their effect on the object. For example, the air resistance force induces a change on the velocity of the object. Interactive Physics allows the user to create plots. The possibilities to create different kind of plots are many. It is possible to visualise how the position, velocity and acceleration (linear or angular), as well as their momentum (linear or angular), vary over time. The application also allows the drawing of plots to observe the variation of the total force that acts in a body, as well as the torque, the gravitational force, electrostatic or the air resistance. The application is concerned with the user experience. It is possible to specify that a sound must be emitted for the instants in which two bodies collide. Another possibility of the application is to study the properties of the sound wave. For instance, the application allows the user to measure and listen to sounds affects to a certain volume and frequency.

Figure 6 represents the configuration of a scenario, where several rectangles and a triangle were drawn. These objects are meant to represent a scale and two weights. If the simulation were to proceed immediately, these objects would all fall apart in disorder due to the action of the system pre-defined gravitational force. The pallet with the tools provided by the application allows, for example, by means of anchors and fixed points, that is possible to fix an object in the space, or to specify a point of rotation, respectively.

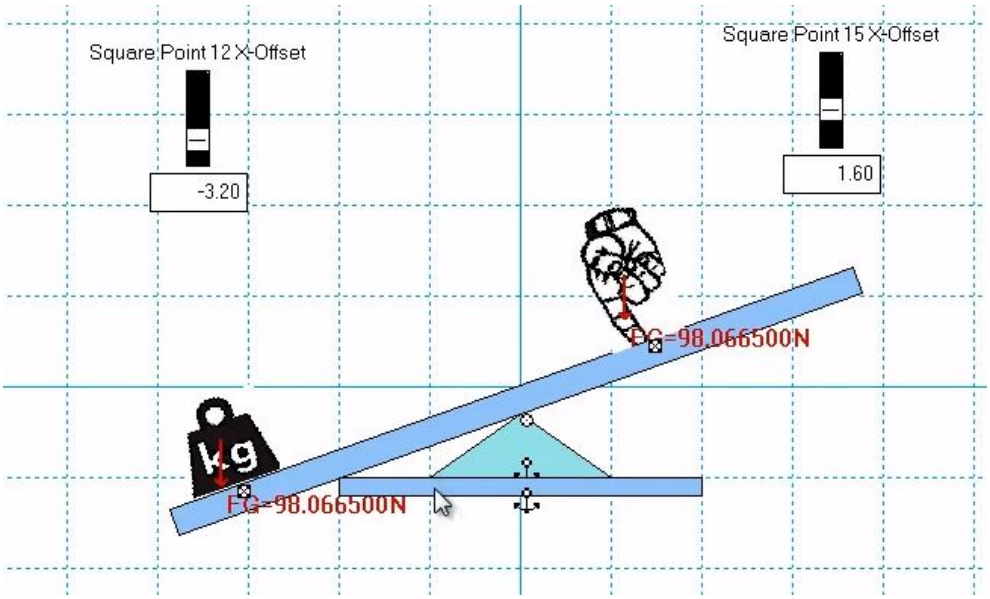


Figure 7.: Final configuration of the scenario described in figure 6. The anchors and rotation points required for a correct execution of the experiment were also specified

Interactive Physics also allows the design of vectors acting at some point in a body to represent the actual value of a physical entity. It is also possible to texture a given object to make the simulation visually more appealing.

After adding the necessary anchors and rotation points to represent the intended behaviour of a scale, the user is able to observe a functional scale. Figure 7 represents the result obtained, for an instant of time equal to 2 seconds. It is possible to observe that the two weights present in the scale were textured with pictures of a weight and a hand. Also, the two vectors were drawn in red. They are applied in each weight, representing the respective gravitational force, according to its direction and intensity. For the design of the forces, it was further specified that it was intended to visualise the norm of the vector, expressed in Newton in this case, since it is a force.

It is also possible to observe in figure 7 that two sliders were introduced in the experience. They allow to change the position of each weight placed in the scale. As intended, the side which the scale inclines depends on the chosen configuration of positions.

The physical phenomenon to be studied in this experiment is called torque. For the scale to be in equilibrium it is necessary that the total torque is 0, with the following equation being verified:

$$F_1 * d_1 = F_2 * d_2 \tag{3}$$

This means that the normal force acting on one side of the scale F_1 multiplied by the distance from its point of actuation to the centre d_1 (axis of rotation of the lever) is equal

to the normal force acting on the other side F_2 multiplied by the distance from its point of actuation to the centre d_2.

As we realised with the experience pictured in figure 7, the user can easily create a scale and place some weights on it and observe which side the scale tilts due to the torque. However, not even the notion of torque, and what are the entities involved in its calculation just as described in this section are mentioned in Interactive Physics.

2.4 PHET INTERACTIVE SIMULATIONS

Phet Interactive Simulations consists of a set of physics experiments, available free of charge and elaborated by the University of Colorado Boulder. At the Phet Interactive Simulations website the user can find experiences about the most varied topics of physics, biology, chemistry, among other scientific areas. Experiments run directly on the user's web browser.

Figure 8 shows a view of the window presented to the user for the "State of Matter: Basics" experiment. This experiment is included, along with fourteen other experiments, in the section of heat and thermodynamics. The experience intends to introduce the student to the basic concepts of pressure and temperature and how these vary in the context of a thermally insulated container with variable volume. The user is presented with an explanatory diagram regarding the theoretical concepts that are intended to be studied in this experiment. The user can interact with the scenario and observe the result the introduced variations induced. In that sense, through a pump, it is possible to increase the number of corpuscles present in the vessel. It is also possible to heat or cool the vessel, to change the nature of the molecule to be studied, or to increase or decrease the volume of the vessel. The pressure and temperature indicators allow visualising the numerical value associated with these two physical variables. This experience sensitises the student about the physical concepts in action through a qualitative analysis of the events. However, it is more complicated to extract more in-depth knowledge since it is not possible to rigorously quantify, for example, the energy supplied or withdrawn to the vessel by the heater and cooler. It is also not possible to accurately conclude the number of atoms or molecules present in the vessel, and the concrete result induced by the action of pumping on the variation of the number of particles.

In fact, the experiences of the Phet interactive simulations suite follow the characteristics of the experience described above. The goal of this type of software solution is to introduce, through appealing graphics and a modern interface, concepts related to physics present in a given context. The degree of customisation of the scenarios, as well as how the data inherent to the experience, can be scientifically processed, for example through graphics, is quite limited.

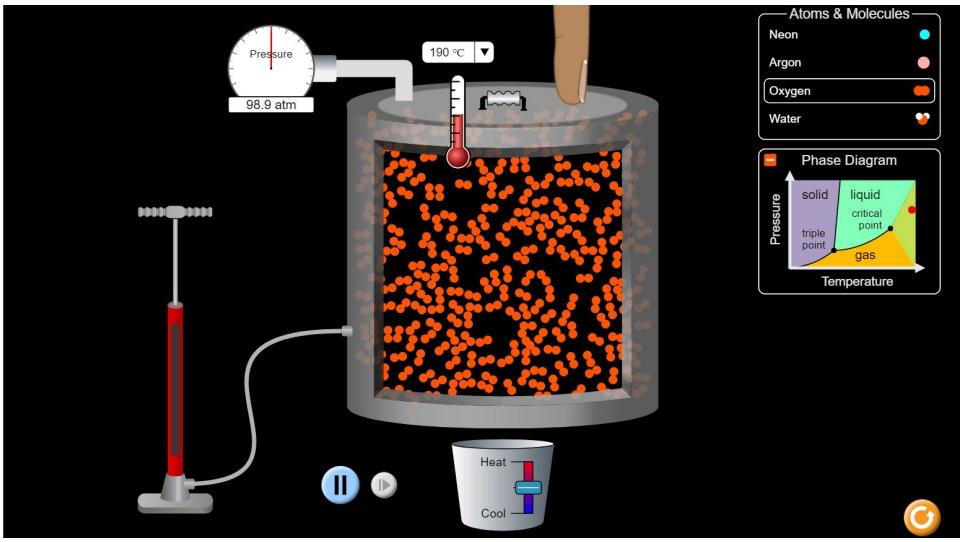


Figure 8.: "State of Matter: Basics" experience interface, included in the experiments provided by Phet Interactive Simulations

2.5 MY PHYSICS LAB

My Physics Lab is a website created by Erik Neumann in 2001 containing physics experiments. Over time, new experiences have been added by the author. Currently, the site makes available fifty-three experiments to be performed directly in the browser. All the experiments are access free. These experiments intend to present the concepts related to classical mechanics. Special emphasis is given to the mechanics of springs, pendulums, roller coasters and the dynamics of rigid bodies.

Figure 9 shows the "Single Pendulum" experiment environment. The experiment allows to change an advanced set of predefined parameters, such as the pendulum chord size, energy dissipation factor, pendulum mass, gravitational acceleration, among others. There are many possibilities for processing and presenting data. For instance, it is possible to request plots, relating any physical variables treated in the context of the experiment. For the concrete experiment of "Single Pendulum", it is then possible to relate, along with time, the following physical variables associated with the pendulum: angle, angular velocity, angular acceleration, kinetic energy, gravitational potential energy and total energy. The figure 10 illustrates the plot obtained for the variation of the kinetic energy of the pendulum as a function of time ranging from 0 to 14. The plot follows the simulation specifications for the numerical values assigned to the parameters shown in the figure 9.

The user has access to a theoretical context where the physics that underlies the movement described by the pendulum is explained, as well as the energies that come into play. It is also presented information to the user about how it is possible to solve numerically the differential equation associated with the pendulum movement. The various numerical

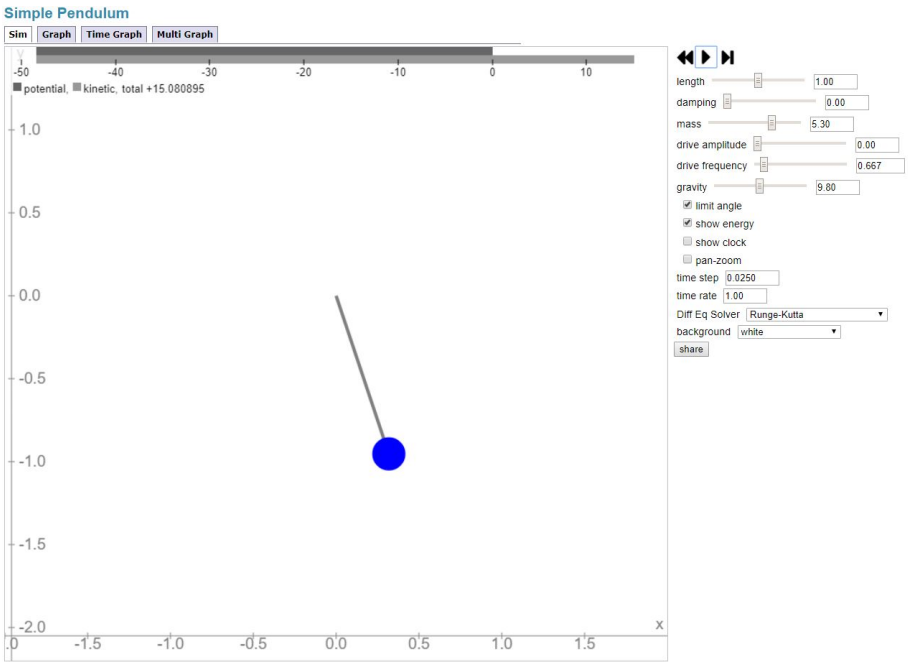


Figure 9.: Single Pendulum experience interface, included in the experiments provided by My Physics Lab

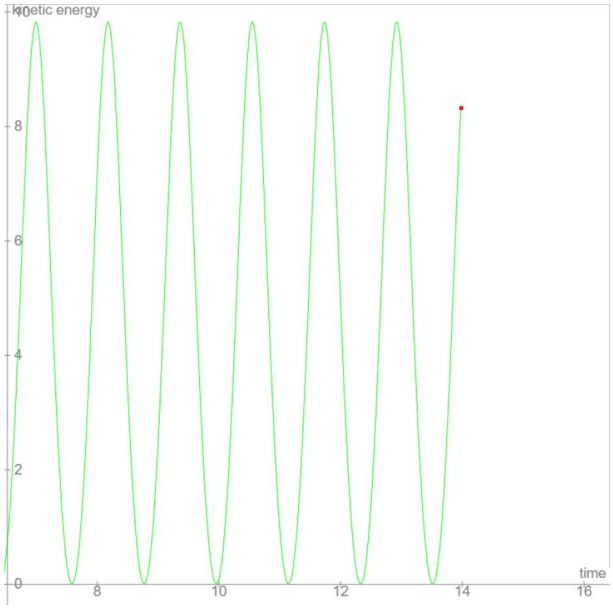


Figure 10.: Example of a plot generated by My Physics Lab

resolution hypotheses are explained. The user can select the desired solver as well as the integration step. The differential equation presented by My Physics Lab that translates the pendulum motion subject only to the action of the gravitational force is:

$$\frac{d^2\theta}{dt^2} + \frac{g}{L}sin(\theta) = 0 \tag{4}$$

where g is the acceleration of gravity, L the length of the wire and θ the angle described by the pendulum relatively to the vertical axis.

Most of the experiments provided by My Physics Lab follow the philosophy behind the Single Pendulum experience. Thus, regarding the possibilities of configuration, the user can vary the numerical value of a wide range of parameters, relate in a graph two physical variables of the experiment. Two other pedagogical peculiarities associated with this platform are the fact that one has access to the physical theory of the experience as well as information regarding the numerical resolution of the problem, starting from the physical equations presented in the section of the physical theory.

2.6 CONCLUSION

From the applications we covered in this chapter, we understand that two families of physics virtual laboratories are easily identified. On one side, applications such as 2.2 consist of predefined scenarios without the possibility of creating new objects to participate in the experiment. In this family of applications, the user has the freedom to change the value of some predefined physics entities associated to the experience and observe the induced results. On the other side, applications such as 2.1 allows the user to specify all the objects wanted to be part of the experiment. This is made possible by the existence of a physics engine. However, we saw that the physics equations used by the engines are hardcoded and the user has not the possibility to change or even visualise the equations. Therefore, any desired change to the underlying physics would only be possible to be done by the authors of the applications. From a pedagogical point of view, another disadvantage is the fact that since the student has no access to the equations used, it is difficult for him to associate the visualisation of a specific physics phenomena with its respective mathematical description.

3

THE VIRTUAFIZ PARADIGM

This chapter presents the paradigm and the philosophy behind the developed tool. Section 3.1 presents an overview of the system and what it intends to accomplish. In section 3.2 the reasons that led us to create a new and innovative concept in physics virtual laboratories are presented through theoretical examples. We name this concept Context and its formal definition is also presented in this section. Finally, in section 3.3 we explore some particularities of the engine behind VirtuaFiz. It is also in this section that a differentiation is made about which steps are user or system responsibility during the whole process of preparing and then performing a simulation.

3.1 OVERVIEW OF THE SYSTEM

As detailed in chapter 2, the physics virtual laboratories that exist today are essentially tools that allow to observe a certain physical phenomenon by adjusting the values of a predefined set of parameters "hardcoded" in the system.

While in VirtuaFiz the visualisation of physical experiments, and parameter adjustment, is also possible, the main emphasis is not on the visualisation of pre-built experiments, but rather to allow the creation of new experiments through the introduction of the equations that rule the experiment.

VirtuaFiz takes the idea of the specification of the physics by the user to an upper level. In fact, the system does not have any knowledge about physics. Every physical phenomena observed results strictly of the equations that the user specifies. Therefore, if some equation introduced in the system does not reflect the physical behaviour that would be observed in the real world, the observed result in VirtuaFiz will look unrealistic. This feature makes the major difference between previous solutions and VirtuaFiz. In fact, this approach can be seen as a new paradigm of specifying physics in virtual laboratories. Due to its nature, VirtuaFiz is especially geared towards the teaching of physics to the level teached in high school and also in some introductory subjects of classical mechanics in a college degree.

Regarding the test environment, the tool is quite similar in relation to the others. In fact, the system is able create graphical simulations with one or more objects evolving in a

world and then observe the physical interactions that happen between objects. From these interactions, changes in the trajectories and behaviour of the objects may occur. Generally in virtual laboratories, a three dimensional orthogonal referential is used to represent a specific position in the world. Also, the world space has no boundaries and is conceptually infinite. The world possess some attributes (time, for example). With respect to the objects, they possess attributes (position and mass, for example) and have a physical and geometrical type. The geometrical type of an object is its shape while its physical type can be *static* or *dynamic*. Contrary to a *dynamic* object, a *static* object will not see any of its attributes updated during a simulation. VirtuaFiz follows globally the same conventions described above to specify the world or an object. However, contrary to the others tools, all the attributes are defined by the user. For the system they have no particular meaning (contrary to the gravitational acceleration or the restitution coefficient attributes we can see in many labs, for example). The meaning of an attribute is the one the user intends to give him conceptually. Exceptions are the predefined world attribute *time* and the predefined geometric objects attributes *position*, *size* and *orientation*.

3.2 THE CONCEPT OF CONTEXT

As we stated in section 3.1, VirtuaFiz intends to follow a different approach about how a simulation is specified and what the system "knows" about physics. Since according to the new paradigm we identified, all the physics is specified by the user, to understand the necessity of the new concept we introduce in this section nothing better than look at two simple examples and remember the physics equations we all used in high school to describe the situations encountered.

Let's start with a simple example of a sphere, starting from a position p_0 at time t_0, and evolving in space where a constant gravitational field, with acceleration a, is acting. Consider also that the sphere has some initial linear velocity v_0. For a given moment t, the position p of the sphere can be known using the following kinematic equation:

$$p = p_0 + v_0(t - t_0) + \frac{1}{2}a(t - t_0)^2 \tag{5}$$

Figure 11 shows a possible parabola trajectory followed by the sphere during its free fall motion as described by equation 5. The initial velocity and gravitational acceleration vectors have the direction of the semi positive axis X and the negative semi negative Z, respectively. $t0$, $t1$, $t2$, $t3$ and $t4$ represented in figure 11 are four equidistant time moments. As we can see from the figure, the fall delta made by sphere between two successive moments increases continuously from one pair to the next. This is due to the square power associated with the expression $(t - t0)$ in equation 5.

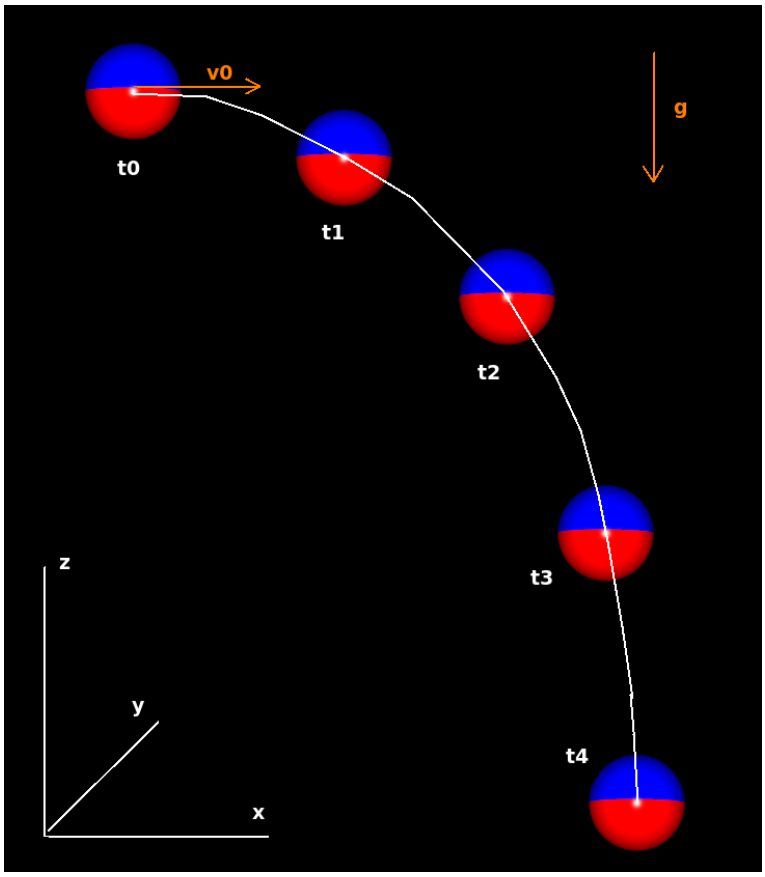


Figure 11.: An example of a parabola trajectory described by the sphere during its free fall motion according to equation 5

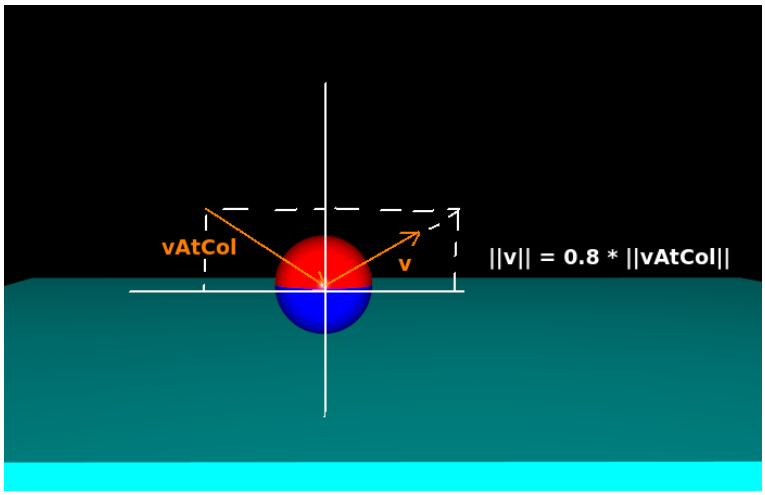


Figure 12.: The mathematical collision response calculation according to equation 6

Equation 5 describes the behaviour of a sphere in free fall. Now, let's assume that eventually the sphere, during its fall, collides with the ground, represented by a flat cuboid. Surely equation 5 above cannot be used to solve the collision response between the two objects. In fact, specific equations are needed to solve this particular problem. Let's just assume the cuboid is static, horizontally oriented, and large enough so that the sphere can only collides with its upper horizontal face. Also, let's consider at the moment of the collision, the sphere velocity is mathematically reflected but due to some loss of energy, it also reduced by a factor of 20%. The simple equation below can be used to calculate the velocity vector v of the sphere immediately after the collision, assuming axis Z is the vertical axis:

$$v = 0.8 * (vAtCol.x, vAtCol.y, -vAtCol.z) \tag{6}$$

where $vAtCol$ is the sphere velocity at the moment of the collision and the $.x$, $.y$ and $.z$ annotations represent the first, second and third components of a vector, respectively.

Figure 12 shows what happens to the sphere velocity when it collides with the cuboid according to all the conditions stated earlier.

This approach and all restrictions mentioned above make the simulation very limited about the behaviours that can observed and also physically unrealistic. In fact, the physics behind collision response are very complex involving many mathematical and physical concepts. The physics behind the collision response between two bodies independently of their shape or even their orientation in space at the moment of the collision are introduced in chapter 5. For now, let's just keep with the restrictions stated earlier. They keep the example simple and keep us from loosing the focus of what this section is about.

Through the two situations presented, we understand that an object in free fall and an object colliding with another one, introduces necessarily different scenarios, requiring dif-

ferent sets of equations for each particular scenario. In both situations described, the underlying physics are also different. For these reasons, VirtuaFiz introduces the notion of physics contexts used to differentiate into specific environments the different physical behaviours the user intends to observe.

Relatively to the specificities of a context in VirtuaFiz, a context is a specific environment where a given object evolves. At any time during a simulation, every object has only one specific context applied on it. Besides object and world attributes, context attributes can be defined within a context. One instance of these context attributes is made for every object involved in the context. The scope of these attributes is limited to the context. This means when an object leaves a specific context, the value of the attributes is lost and a later return to the same context will reset their values. On the other hand, an object attribute must be used when its meaning and value transcends a specific context application.

A context is composed by a set of functions introduced by the user. During a simulation, the attributes of each object, as well as the world and context attributes are constantly being updated using the functions of the context where the object evolves. The functions tells the physical behaviour to be observed in a particular situation. This is made possible by functions whose name is the name of an object, world or context attribute. The number of objects involved simultaneously in a specific context instance depends on the type of the context. A context have one of the three following types: free fall, collision and contact.

Free fall contexts must be defined to describe the physical behaviour wanted when an object is not in physical contact with any other object. Since in a free fall context an object does not interact with another and can maintain this behaviour for a continuous time, a free fall context instance is applied to a single object and is applied continuously during a sequence of frames.

Collision contexts must be defined to describe the physical behaviour wanted when two specifics objects collides. In the real world, when two objects collide, depending on its materials, the objects stay in contact during a more or less important time interval. However, in general this time interval tends to be relatively small so in practise we can consider it to be zero. Therefore, contrary to a free fall context, a collision context application is not continuous and occurs only one time within a frame. One could argue that this reasoning is not valid in the presence of a non elastic collision when the objects are in contact for a longer time. However, if the user builds a collision context using real physics, by defining, among others, the restitution coefficient and all the necessary physical equations, the behaviour of the objects after the collision will be just as it would be observed in real life.

Let's observe figure 13 to understand the necessity of having a third type of context in VirtuaFiz. In the real world, every time a cube collides with a ramp, a very strong force (much more than the gravitational force acting on the cube) acts during a small interval of time, on the cube. The result of a force acting during a certain amount of time is called

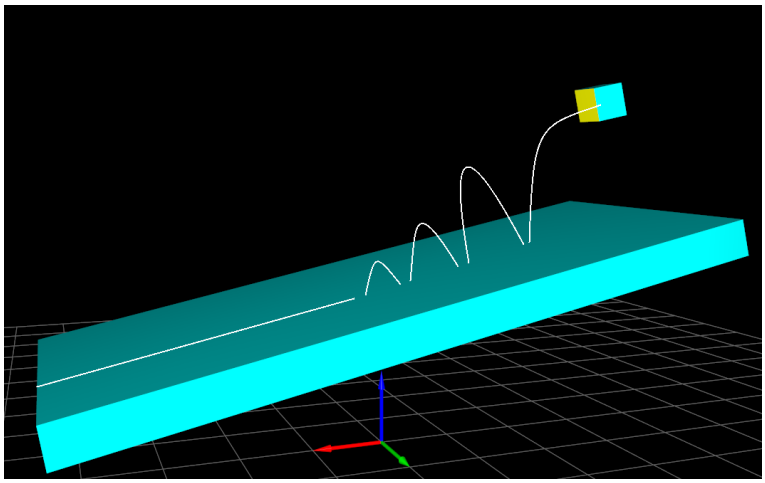


Figure 13.: A simulation of a cube colliding multiple times with a ramp until it ends up sliding on its surface

an impulse. The effect of the impulse on the cube is to generate a change in its velocity. Overtime, after multiples collisions and energy losses, the impulse acting on the cube is no longer effective to induce a change in the velocity of the cube that would fight the gravitational force and separate it from the ramp. In this case, regarding the collision normal axis, the normal reaction and gravitational forces ends up cancelling each other continuously over time, making the cube remaining in a prolonged contact with ramp.

In order to cope with the possibility that the user wishes to specify real physics in its contexts by following, for example, an impulse approach for collisions and a force approach for prolonged contacts, VirtuaFiz allows the definition of Contact Contexts. Therefore, when a collision context is no longer effective, the system automatically switches to a contact context. A collision context is no longer effective when after its application the objects are not able to separate or end up penetrating each other. This is what would happen in the simulation pictured in figure 13 if there was not a VirtuaFiz contact context defined for this simulation. Just as a free fall context, a contact context is applied continuously. Contrary to a free fall context instance and a collision context instance that always contain one and two objects, respectively, a contact context instance can contain an arbitrary number of objects. Figure 14 shows two different contact context instances that are simultaneously applied over time. In the figure, the instance pictured on the left contains 3 objects while the one on the right contains 4 objects.

3.3 THE ENGINE BEHIND VIRTUAFIZ

Before we explain some particularities of the system, it is important to detail the philosophy followed by VirtuaFiz. As previously stated the system has no knowledge about physics.

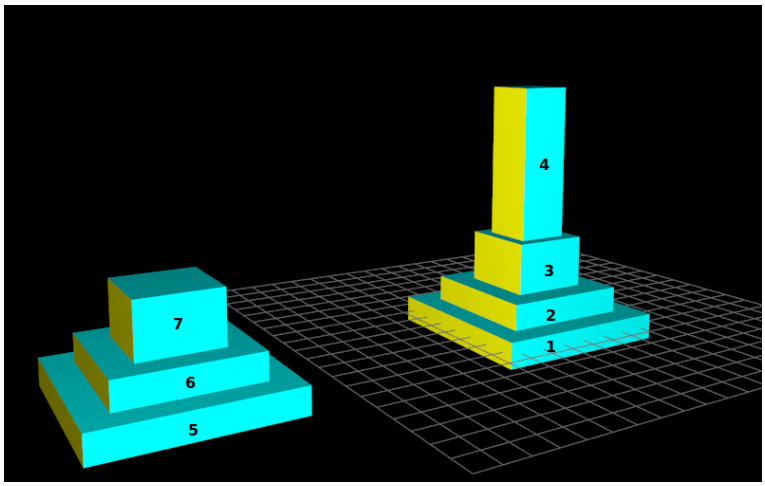


Figure 14.: A VirtuaFiz simulation containing 7 objects distributed in two different contact context instances

However, the engine has knowledge about the geometry involved in a simulation, and uses that knowledge to determine which context to apply at any particular time to each particular object.

For instance, the collision detection between any two objects involved in a simulation is done by the system and not the user. Therefore, the user can idealise the system as being a black box responsible to determine which context to apply on each object. Furthermore, the system is also responsible for updating the graphical simulation based on the attribute values computed in the contexts.

Also, if no equations are specified inside a context being applied on a specific object, the engine will have no idea how to update any of the attributes of the objects. Therefore, all the object attributes will remain constant when using this context. Graphically, this means the object remains stationary.

The engine also provides the user the required geometric attributes for each context in the form of predefined context attributes. For instance, considering a collision the engine provides two predefined context attributes that can be used in the equations of the context. These attributes are the contact point and the collision normal vector regarding each object involved in a collision. Notice that in situations where the geometry in collision is not a point but a portion of a plane or a line segment, the engine provides also only a point. In these cases, the point is the result of the interpolation of the extremities of the line segment or the points that delimit the geometrical figure.

Although collisions can occur between several objects simultaneously, VirtuaFiz handles collisions between pairs of objects. In order to make the whole collision handling process simpler to be expressed by the user, the engine applies a sequence of simpler collisions where in each step only two objects are involved. The solution encountered does not impair

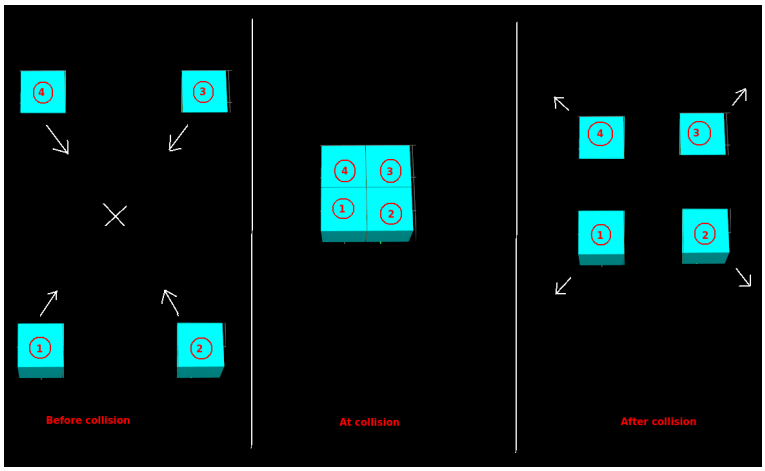


Figure 15.: Due to its specific initial position and initial velocity, each cuboid is in collision with the other three

the system ability to solve harder and more improbable outcomes of some scenarios. For example, in a scenario where three or more objects collide simultaneously, the system first solves one collision between two randomly chosen objects and then repeats the process if there are still valid collisions to solve. In fact, after solving one collision between a pair of objects it is possible that one object goes in a direction that just makes invalid simultaneous collisions thought need to be solved. Figure 15 shows this possibility occurring in a scenario involving four cuboids with the same size and mass.

Looking at the figure, we understand that, at the time of the collision, each cuboid collides with the other three. The engine must choose from 6 different pair of cuboids to resolve the first collision. If the system decides to resolve first the collision between cuboid 1 and 3 followed by the collision between cuboid 2 and 4, the collisions between cuboid 1 and 2 becomes invalid and will not be resolved since at that point both objects are heading back towards their initial position. The same happens for cuboids 3 and 4. Notice this specific situation only occurs if the physics equations introduced in the collision context by the user follows the underlying physics of collision.

For contact contexts, the amount of geometrical data provided by the engine is greater. For instance, contrary to a collision context, more than one contact normal and one contact point can be provided by the engine relatively to each object involved in the contact context instance. The number depends on each situation. Looking again at figure 14, we understand that for each object that does not make the extremities of the stacks, the engine provides two contact normals and two contact points. In fact, in this case, each object is in contact with one on top and one on bottom. Due to the multiple possibility of contact normals, to access the desired contact normal, the user must specified a direction. For example, considering axis Z as the vertical axis, specifying the direction $[0, 0, -1]$ would provide the normal of the contact between the object and the object immediately below. Given a direc-

tion specified by the user, the engine also provides a special context attribute for each object attribute. The value of this context attribute results on the summation of the object attribute regarding all the objects that form a chain of contact starting at the object and pointing in the direction specified. Notice that the object itself is not considered to be part of the chain when the summation is calculated. For instance, considering again the scenario pictured in figure 14 and also that all objects possess an object attribute called *my_attribute* equal to 1, the value of the special attribute regarding *my_attribute* and direction $[0, 0, 1]$ is 3 and 0 for object 1 and object 4, respectively. Finally, in contact contexts, for each direction specified by the user, the engine provides a context attribute that tells if the chain of objects in contact in the direction specified contains at least one static object. For example, considering that in figure 14 all the objects but object 1 are dynamic, the value of this special attribute for object 3 is *true* and $false$, regarding directions $[0, 0, -1]$ and $[0, 0, 1]$, respectively. As we will see in chapters 4 and 5 all these geometrical attributes provided by the engine will prove to be helpful when we build collision contexts and contact contexts.

One aspect not yet covered is how the system determines, when a geometrical collision between two objects occurs, if it is to apply to a collision context or a contact context. At the time of the collision, the collision context is always applied and all the necessary attributes are updated. After, the engine tests if the collision response was enough to separate the objects. This is done by applying the necessary free fall context to each object and see how they behave after a very small time interval. If they penetrate each other or simply do not separate that means the collision context was not effective enough, so we are in a presence of a contact behaviour. However, if the objects are separated, the contact context is not applied and the objects are kept in their respective free fall contexts.

The user can also specify a set of directives about which objects are allowed in each context. In a more general sense, it is also possible to specify the geometrical and physical type of the objects. That makes possible, for example, to define two free fall contexts each with different equations and specify that one applies to spheres (context 1) and the other one to cuboids (context 2). Then, when the engine defines that a specific sphere must start a free fall context application, the context 1 is selected from the list and start to be applied on the sphere.

This is particularly relevant when considering collisions. The collision between two spheres can be handled differently from a collision between a sphere and a cuboid, using, a simpler set of equations, for example.

At the moment of creating a new simulation, it is also possible to select specifically the contexts to participate in the simulation. In this sense, for example, let's say we have two different free fall contexts. One is specified to be applied on sphere (context 3) and the other to be applied on every object not taking into account its geometrical type (context 4). If the two contexts are include in the simulation, every time a sphere pretends to starts a

free fall motions, context 3 is applied. This happens because despite the fact that according to the restrictions imposed both contexts are valid to be applied on the sphere, the engine selects the most restricted to the situation. Naturally, in a new simulation involving the same sphere but with context 3 not selected to be part of the simulation, context 4 will be applied to the sphere every time a free fall motion is engaged.

4

THE VIRTUAFIZ SIMULATION SPECIFICATION

In this chapter, we explore the particularities needed to be taken consideration during the process of specifying a VirtuaFiz simulation. The main features of a simple programming language created specially for the VirtuaFiz environment is presented in section 4.1. In section 4.2 we cover all the necessary steps to build two very simple yet workable simulations.

4.1 THE VFLANG PROGRAMMING LANGUAGE

In many scientific and educational softwares it is important and advantageous to have a programming language behind that supports the entire user experience. These languages are developed with the functionalities and characteristics necessary to respond to the functionalities that these softwares propose to support. MatLab MathWorks Inc or Wolfram Mathematica Wolfram Research are two examples of software that, along with the associated software product, have led to the appearance of a proprietary language. The development of VirtuaFiz has led to the development of a supporting programming language.

We named VFLang the custom programming language that gives support to the functions defined inside the VirtuaFiz contexts. VFLang is a small interpreted programming language. It is a very simple language, with only the features that are considered relevant to respond to the family of problems that led to the need for its development. Its simplicity, however, is not a limiting factor in the elaboration of complex functions. In fact, the language has support for conditional structures following the if-then-else structure which can naturally be nested, with no limit of depth.

The possibility of using recursive functions is also an added value in the elaboration of contexts with high complexity.

In VFLang, the basic structural unit is the function. This is characterised by a name, which serves as an identifier for when it is intended to invoke the function. A function can receive zero or more arguments, and in the declaration of a function, each received argument is also characterised by a name. In a function without arguments, the left and right parenthesis after the name of the function can be omitted. One function can call another function only if it is also defined in the same context.

VFLang possesses a library with many useful geometrical functions. Moreover, all the functions are native. Since they are written in C, calling these functions in the contexts instead of calling regular functions written by the user, can greatly reduce the time needed to compute a simulation. So, their usage is highly encouraged. They are used multiple times in chapter 5 to facilitate the writing of the functions that enable us to abstract from the coding of many helpful geometrical transformation. The description of some of these functions can be found in appendix A.

Let's see an example of a function written in VFLang. First, let's consider the Newton universal gravitation law. Given the mass m_1 and m_2 of two different bodies and the distance between them r, the intensity of the gravitational force suffered by each body can be calculated as follow:

$$F_g = -\frac{Gm_1m_2}{r^2} \tag{7}$$

G is the universal gravitational constant. Its value is $6.67408 * 10^{-11} m^3 kg^{-1} s^{-2}$.

The following shows how the universal gravitation law can be written in VFLang:

```
gravitational_force(m1, m2, r) = -(6.67408 * 10^-11 * m1 * m2)
                                                * r^-2
```

Relatively to the programming paradigm adopted, the language can be considered as being hybrid as it joins characteristics of functional and imperative languages. Let's understand this particularity quickly by considering that the following function and the function *universal_gravitation_force* could be inside a context:

```
my_random_attribute = universal_gravitation_force(10, 15, 1)
```

Also, let's suppose a world attribute called *my_random_attribute* exists in the system. As we know from section 3.2, an attribute is updated by calling a function with the same name. Because function *my_random_attribute* has the same name of *my_random_attribute*, it can be seen as an imperative attribution. There is a modification of the system internal state after the function is called. In this case, attribute *my_random_attribute* would be updated by the result of $universal\_gravitation\_force(10, 15, 1)$. On the other hand, calls of function *universal_gravitation_force* do not modify the internal state of the system. It is a pure functional function without any side effect.

In order to maintain its simplicity and its use restricted for the family of problems we want to support, VFLang does not support the creation of new data types. There are four data types that allow operation on numbers and booleans. These are:

- Real number
- Matrix of real number

- Vector of real number
- Boolean

There is no limit to the size of a matrix or a vector in VFLang. The vectors are represented by a left bracket, then followed by the comma-separated vector components and finally by a right bracket. The matrices follow the same syntax, since a row of the matrix is represented as a vector. As we can see in the universal gravitation function, the data type of the its argument and the data type of its result are not specified. In fact, in VFLang, as in some other high-level languages, data types are not specified at the time of declaration; these are then determined during run time. This feature makes writing code more compact and high-level.

To keep the language simple, VFLang does not allow the programmer to create new operators. VFLang has the three classical logical operators that let you define arbitrary complex logical expressions:

- Logical conjunction (&)
- Logical disjunction (|)
- Logical negation (!)

To perform arithmetic operations on matrices, vectors and real numbers the following operators can be used:

- Unary or binary addition (+)
- Unary or binary disjunction (-)
- Division (/)
- Multiplication or external product (*)
- Internal product (.)
- Exponentiation (^)

For the same operator, the data type of the result and the nature of the operation to be performed depends on the data type of the operands involved. So for the function:

```
myFunction(x,y) = x * y
```

if the operands are two matrices, the result of the multiplication of the two matrices is returned. On the other hand, if the two operands are two vectors, the result of the external product between the two vectors is returned. For the case of two real numbers, the image of the function consists of the simple multiplication of the two numbers.

To compare the value of different operands the following operators can be used:

- Equal (==)
- Not equal (!=)
- Less or equal (<=)
- Higher or equal (>=)
- Less (<)
- Higher (>)

VFLang also has operator # that allows to access to a particular row of a matrix or a particular element of a vector. For example:

```
[[1,2,3,4],[5,6,7,8]] # 2 # 3
```

by applying two time the operator # we gain access to the third element of the second row of the matrix.

4.2 THE SPECIFICATION OF TWO SIMPLE SIMULATIONS

In section 3.2, we presented an example of a sphere falling and colliding with a cuboid to emphasise the fact that different physical behaviours are described using different mathematical equations. The context approach followed by VirtuaFiz was also explained in the same section. In this section, we use this same example as the scenario for our first VirtuaFiz specification. At the end of the section, we describe, through another example, how attributes from different objects evolving in the same context application can be referred in a function.

Notice that using the VirtuaFiz software, the user introduces all the specification about his simulation through the VirtuaFiz graphical interface. In this document, for convenience we will use instead a textual specification to represent all the necessary information about a simulation. Let's start our VirtuaFiz specification by describing the objects involved and setting some value to its attributes.

The sphere can be identified as follow:

```
object {
    name = my_sphere
    physical type = dynamic
    geometrical type = sphere
    attributes = {
        position = [0, 5, 0]
        orientation = [0, 0, 0]
```

```
        size = 1
    }
}
```

From the textual specification above, we just specified a new object called *my_sphere* whose geometrical type is *sphere* and physical type is *dynamic*. The three necessary predefined object attributes *position*, *orientation*, *size* are also specified and its initial values set. In VirtuaFiz, the predefined attribute *orientation* must be expressed using a three dimensional vector where the first, second and third coordinates refers to the X, Y, Z rotational Euler angles, respectively. The predefined attribute *position* also must be a three dimensional vector since a position must be specified according to the three dimensional orthogonal basis used by the engine.

Following the same textual notation, the cuboid can be specified as follow:

```
object {
    name = my_cuboid
    physical type = static
    geometrical type = cuboid
    attributes = {
        position = [0, 0, 10]
        orientation = [0, 0, 0]
        size = [50, 50, 1]
    }
}
```

As we can see, the sphere and cuboid size attributes use different types to represent its value. In fact, in VirtuaFiz the predefined type of the attribute size for a sphere is a single real that expresses its radius, while for the cuboid it requires a three dimensional vector containing the size of the object measured in the three orthogonal axis.

With the objects described, it is time to specify the contexts that we want to use in the simulation. Two contexts are easily identified: a free fall context and a collision context. Let's start with the free fall context. An empty free fall context can be specified as follow:

```
context {
    name = my_free_fall_context
    type = free fall
}
```

As we can see from the specification above, there is no function specified in the context. As was stated in section 3.3 when no function is specified no physics is applied. In this case the sphere would be stationary in space since none of its attributes are updated. However, this is not the physical behaviour we want to represent in this context. Let's remember the

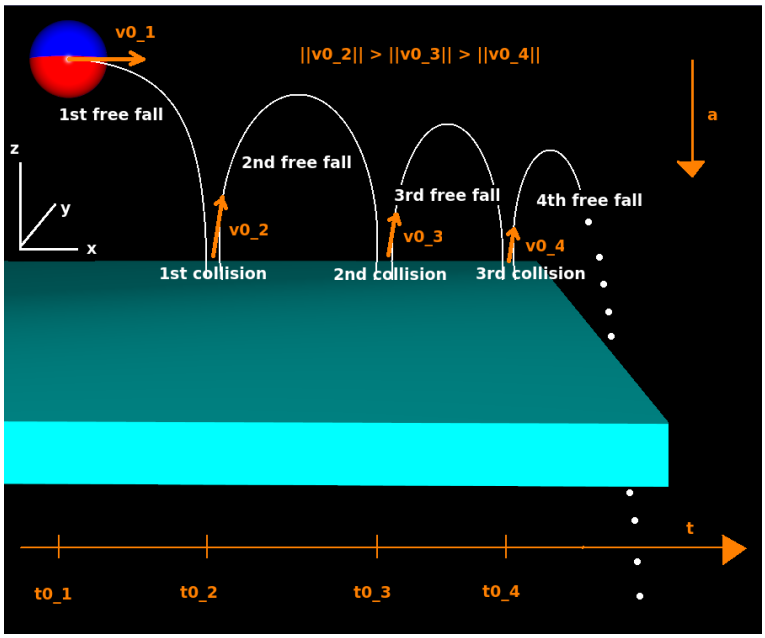


Figure 16.: A possible outcome of the simulation involving the sphere and the cuboid.

physics equation we came up in section 3.2 to calculate the position p of the sphere at time t:

$$p = p_0 + v_0(t - t_0) + \frac{1}{2}a(t - t_0)^2 \tag{8}$$

where p_0 is the initial position vector, v_0 is the initial velocity vector, a is the acceleration, and t_0 is the moment when the simulation starts.

Looking at the variables used in equation 8, it is easy to understand that variables t and a can be declared as two world attributes since they are related only to the world and we don't need multiples instances of them. Their specification can be done as follow:

```
world attributes {
    t = 0
    a = [0,0,-9.81]
}
```

The value of attribute a was chosen taking into account the intention to see the sphere colliding with the cuboid. In fact, different values for attribute a makes the object follow different trajectories. Figure 16 helps to understand how variables $pos0$, $v0$ and $t0$ has to be specified. The figure shows graphically the trajectory that the sphere can follow according to the initial values we give to the attributes. The different contexts application are also emphasised in the figure.

As we can observe in figure 16, it is possible that the sphere ends up colliding multiple times with the cuboid before falling to the infinite. However, equation 8 describes mathe-

matically only one free fall parabola. So, for each one of the four free fall context application identified in the figure, different values for the initial velocity (variable $v0$) and initial position (variable $pos0$) of the sphere are expected. A different value is also expected to the time when a new free fall context applications starts to be applied (variable $t0$). Since these variables are only needed inside a specific context application and can be discarded at the end of it, they can be defined as context attributes. So, our free fall context specification is updated to:

```
context {
    name = my_free_fall_context
    type = free fall
    functions = {
    }
    attributes = {t0, v0, pos0}
}
```

Also, we want the value of these three variables to be calculated only one time at the beginning of the free fall context application so they can retain their initial value during all the application. Recall that this is the opposite of what we want to happen with the sphere attribute *position* that is expected to be updated at every frame. In contexts that are applied continuously (free fall and contact contexts), each attribute is computed only one time during the context application if it is specified as a *compute first time only attribute*. This possibility proves to be helpful in many situations. For example, using a context attribute and specifying it as a *compute first time only attribute*, allows to have a time stamp about the moment the context starts to be applied on the object. What happens is that the function used to update its value will be called only one time (at the beginning) and access the value of the world time attribute. As intended, in successive frames during that context application, its value will not be updated.

With that said, the context can be specified as follow:

```
context {
    name = my_free_fall_context
    type = free fall
    functions = {
    }
    attributes = {t0, v0, pos0}
    compute_first_time_only_attributes = {t0, v0, pos0}
}
```

Finally, relatively to the free fall context, we just need to specify the value of these three variable that will be used across the context application. Just as a regular attribute, the attribute is updated by specifying functions inside the context. Since we want t_0 and pos_0

to have the value that t and *position* has at the beginning of the context, respectively, we can update the context by adding two functions:

```
context {
    name = my_free_fall_context
    type = free fall
    functions = {
        pos0 = position
        t0 = t
    }
    attributes = {t0, v0, pos0}
    compute_first_time_only_attributes = {t0, pos0}
}
```

If we want to know the initial velocity of the sphere at each free fall context application, it is a good idea to add a new attribute to the sphere that records its velocity at every moment of the simulation. Let's call this attribute simply v and consider it has vector (10, 0, 0) as initial value. Thus, the sphere object can be represented as follow:

```
object {
    name = my_sphere
    physical type = dynamic
    geometrical type = sphere
    attributes = {
        position = [0, 5, 0]
        orientation = [0, 0, 0]
        size = 1
        v = [10, 0, 0]
    }
}
```

Considering that the acceleration is constant, the following equation can be used to calculate at every moment t the velocity vector v:

$$v = v_0 + a(t - t_0) \tag{9}$$

Introducing the equation as a VFLang function in the context we get:

```
context {
    name = my_free_fall_context
    type = free fall
    functions = {
        pos0 = position
        t0 = t
```

```
            v = v0 + a*(t-t0)
        }
        attributes = {t0, v0, pos0}
        compute_first_time_only_attributes = {t0, pos0}
    }
```

Just as we did for $t0$ and $pos0$, attribute $v0$ must be specified as a *compute first time only attribute*. By adding another function to the context, attribute $v0$ is specified to hold during the whole context application the value of the velocity the sphere has at the beginning of the application.

With all that said, to reach the final version of the free fall context we just need to introduce a last function in the context responsible to update attribute *position*. This function is responsible to update correctly the position of the object at every frame according to equation 8:

```
context {
    name = my_free_fall_context
    type = free fall
    functions = {
        pos0 = position
        t0 = t
        v = v0 + a*(t-t0)
        v0 = v
        position = pos0 + v0*(t-t0) + 0.5*a*(t-t0)^2
    }
    attributes = {t0, v0, pos0}
    compute_first_time_only_attributes = {t0, pos0, v0}
}
```

Since the free fall context is only composed by these 5 functions, *position* is the only attribute of the object that will be updated at every frame. The orientation and the size of the object will remain with their values unchanged since there is no function responsible to update them.

Now we have reached the final version of the context, it is important to explore in detail what is a *compute first time only attribute*. In fact, a *compute first time only attribute* has an important particularity we have not explored yet. This is, internally a *compute first time only attribute* is computed before the others attributes and more importantly, inside the function responsible to update the attribute, all the references to an other attribute does not induce a function call. The actual value of the other attribute registered in the system at that moment is used instead.

With that said, we understand, for example, that if attribute $pos0$ was not specified as *compute first time only attribute* there would be a computational issue when the context is

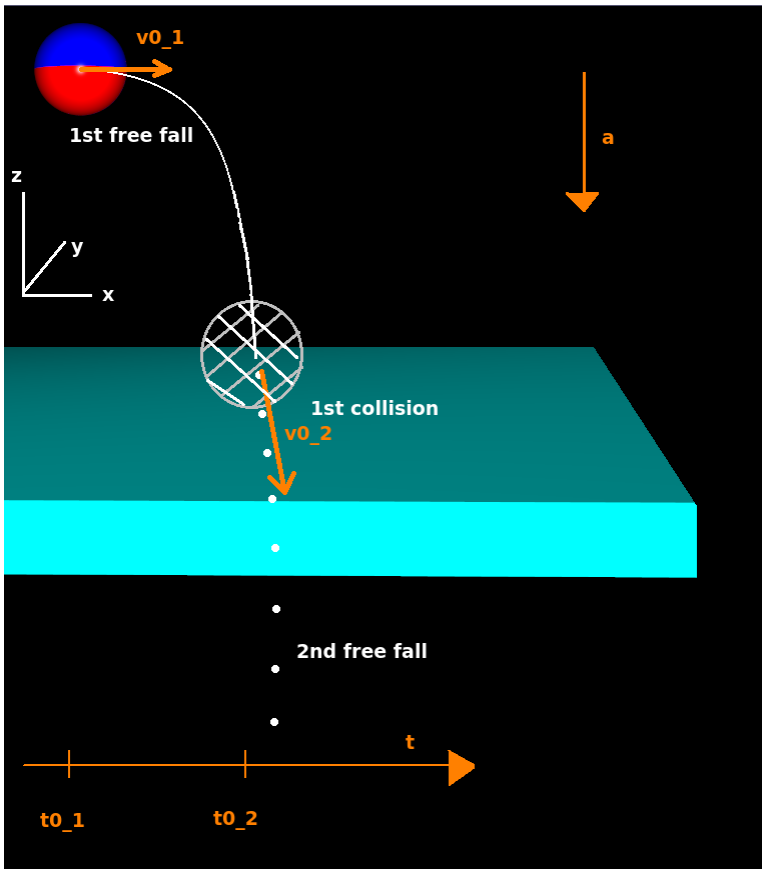


Figure 17.: A possible outcome of the simulation involving the sphere and the cuboid. The sphere penetrates the cuboid since the collision response specified is empty.

applied. In fact, the code of functions *pos*0 and *position* when executed by the VFLang virtual machine would have generated a stack overflow error due to a infinite loop. This would occur simply because function *position* refers to the context attribute *pos*0. Then function *pos*0 refers to the object attribute *position* which would have caused function *position* to be called again.

Relatively to the collision context, let's start by doing just what we did for the free fall context. An empty collision context can be specified as follow:

```
context {
    name = my_collision_context
    type = collision
}
```

Figure 17 shows what happen if the context specified is used as it is now in the simulation.

As we can see from the picture, the sphere ends up penetrating the cuboid. This result is what we can expect for the situation. With an empty collision context, none of the spheres

attributes are updated at the time of the collision. Therefore, the velocity vector of the sphere (given by attribute v) is left unchanged relatively to its value at the time of the collision and its value is then used as the value of $v0$ in the second free fall application. For this reason, the two free fall context applications can be seen as forming a continuous one. In fact, the two pieces of parabola that represent the trajectories of each free fall context application are two pieces of the same parabola. It is exactly the same parabola described by the sphere in a single free fall context application if the cuboid did not exist. Notice that in this case we consider that the contact context is empty or simply not defined in the simulation. Otherwise, the engine would apply the necessary contact context after considering that the collision context application is not effective enough to separate the objects (as happen in this case by having an empty collision context).

The penetration observed is not the behaviour we intend to observe in this simulation. Let's remember the equation we introduced in section 3.2 to solve naively the collision response:

$$v = 0.8 * (vAtCol.x, vAtCol.y, -vAtCol.z) \tag{10}$$

where $vAtCol$ is the sphere velocity at the moment of the collision and the $.x$, $.y$ and $.z$ annotations represent the first, second and third coordinates of a vector, respectively.

In order to calculate attribute v correctly, we need to specify the value of variable $vAtCol$. We want $vAtCol$ to hold the velocity of the sphere at the moment of the collision, just the moment before the collision response is solved. This can be achieved by specifying the variable as a context attribute and adding a new function to the context:

```
context {
    name = my_collision_context
    type = collision
    functions = {
        vAtCol = v
    }
    attributes = {vAtCol}
}
```

Analogously at what we did in the free fall context, $vAtCol$ must be defined as a *compute first time only attribute*. This directive forces the system when executing function $vAtCol$ to use the value of attribute v as it is in the system at that time saving us from stack overflow errors.

```
context {
    name = my_collision_context
    type = collision
    functions = {
```

```
        vAtCol = v
    }
    attributes = {vAtCol}
    compute_first_time_only_attributes = {vAtCol}
}
```

Finally, we just need to introduce a new function in the context that computes attribute v according to equation 10:

```
context {
    name = my_collision_context
    type = collision
    functions = {
        vAtCol = v
        v = 0.8 * [vAtCol # 1, vAtCol # 2, -vAtCol # 3]
    }
    attributes = {vAtCol}
    compute_first_time_only_attributes = {vAtCol}
}
```

v is the only object attribute we want to update at the time of the collision. In fact, since a collision context is always applied only one time within a frame, we can consider that it is not its responsibility to update the position of the sphere. This might be done after in the free fall context at very frame according to the new velocity calculated in the collision context. However, nothing prevents the user to instantly change the position of the sphere to any desired position in the collision context. Physically that would make no sense and this behaviour would reflect in a poor graphical simulation.

To fully specify the simulation we just need now to specify the simulation itself. Considering the simulation starts at $t = 0$ with a duration of 30 time units, we can specify the simulation as follow:

```
simulation {
    name = my_first_simulation
    initial time = 0
    duration = 30
    objects_involved = {my_sphere, my_cuboid}
    contexts_involved = {my_free_fall_context, my_collision_context}
}
```

So this is it. Our simulation is fully specified and can be entered as is in the VirtuaFiz environment. The result observed would be very similar to the one figure 16 shows.

As mentioned previously, the underlying physics of collision is much more complex than a simple reflection of the velocity vector with the collision normal vector. However, for a

less attentive viewer, graphically it can be very misleading in some situations. However, if the ground is inclined, the reflection calculated according to equation 10 will graphically look very unrealistic since the vertical axis (axis Z) is used as the collision normal. Without introducing more functions in the collision context, we can improve a bit the realism of collisions if we use instead the collision normal vector provided by the engine. The collision normal vector is internally defined as *collisionNormal*. The collision context can be improved as follow:

```
context {
    name = my_collision_context
    type = collision
    functions = {
        vAtCol = v
        v = 0.8 * reflectVector(vAtCol, collisionNormal)
    }
    attributes = {vAtCol}
    compute_first_time_only_attributes = {vAtCol}
}
```

reflectVector is a VFLang native function that given two vectors, reflects the first vector considering the second vector as the surface normal vector.

Using all the capacities provided by the VirtuaFiz environment to build arbitrary complex collision contexts requires that a last but important aspect about the way contexts works needs to be covered in this section. Since in our first simulation the collision context is always applied on a dynamic sphere and a static cuboid, all the attributes of sphere are updated by calling a function with the exact same name. However, for example, in a collision context where two dynamic objects are evolving we cannot follow this approach. In fact, in this case attributes of both objects need to be updated. But since two objects can have attributes with the same name, how does the system know which function is to be used to update each attribute? An easy solution to the problem would be to specify inside the context, a group of functions for each object. However, this solution lacks of flexibility about the possibilities of the physics that can be specified in these functions. In fact, it is possible that the user wants to write a function that involves simultaneously attributes of both objects. This possibility is very common as we will see in chapter 5 where we intend to build collision contexts using real physics supporting collision response.

VirtuaFiz solves the problem by allowing the user to specify aliases for each object attribute involved in the context. Let's see how they can be used by specifying a simple collision context involving this time two dynamic objects.

For this purpose, let's imagine a new simulation composed only by spheres. For this simulation we can use the same free fall context of the previous example. Let's consider that all the spheres have the same mass and there is no frictional forces acting on the spheres

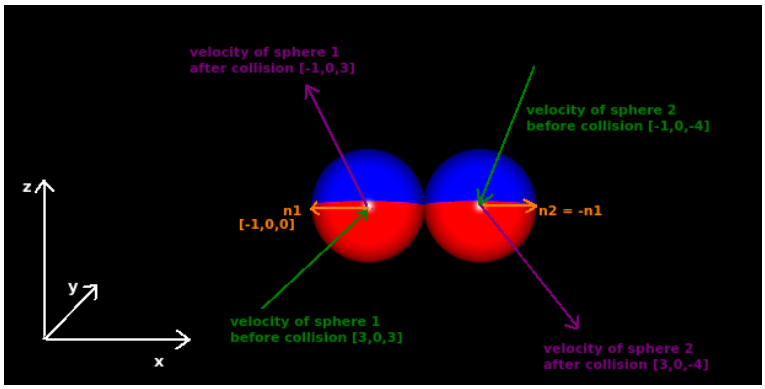


Figure 18.: A possible collision between two spheres and the respective response according to the environment described

surfaces during the collisions. Also, there are no energy losses during the collision (elastic collision). Considering the restrictions mentioned, the actual collision response between two of these spheres is actually relatively simple. The velocity of the first sphere after the collision $v1_{after}$ is given by:

$$v1_{after} = v1_{before} + ((v2_{before} - v1_{before}) \cdot n1)n1 \tag{11}$$

where $v1_{before}$ is the velocity of the first sphere before the collision, $v2_{before}$ is the velocity of the second sphere before the collision, and $n1$ is the normalized normal collision vector relatively to the first sphere.

Regarding the second sphere, its velocity after the collision $v2_{after}$ is given by the following equation:

$$v2_{after} = v2_{before} - ((v2_{before} - v1_{before}) \cdot n1)n1 \tag{12}$$

Figure 18 show a possible collision response between two of these spheres. All the relevant numerical information about the collision is detailed in the picture.

So this is the kind of reasoning we want to put in the functions of our collision context. From equations 11 and 12 we understand, that we will need five aliases: the collision normal vector of the first sphere, and also the velocity of each sphere before the collision to compute the velocity of each sphere after the collision. Also, following what we did for the previous collision context, we consider the velocity of each sphere at the time of the collision as a *context attribute* and a *compute first time only attribute*. You may remember from section 3.2 that since the context is applied simultaneously on two bodies, each has its own instance of the context attribute. Therefore, we just declare one attribute in the context specification. Our collision context can be updated as follow:

```
context {
    name = my_collision_context2
    type = collision
    attributes = {vAtCol}
    compute_first_time_only_attributes = {vAtCol}
    aliases = {
        use first_object_vAtCol as attribute vAtCol of object with index 0
        use second_object_vAtCol as attribute vAtCol of object with index 1
        use first_object_v as attribute v of object with index 0
        use second_object_v as attribute v of object with index 1
        use n1 as attribute collisionNormal of object with index 0
    }
    functions = {
        first_object_vAtCol = first_object_v
        second_object_vAtCol = second_object_v
    }

}
```

Relatively to the aliases declared, it is important to point out that index 0 or index 1 is just a way to refer to a specific object inside the context. They are not related with the object identity/name. In fact, depending in the order selected by the engine, in two separated collision context application with the same two objects involved, the same object can be referred the first time with index 0 and the second with index 1.

Aliases can also be particularly helpful to access the attributes of a specific object in contact contexts when the number of objects involved in the same context instance can be arbitrary high. In chapter 5 we will get back to them when we define new contexts.

Returning to the example, we just need to specify the value after the collision of attribute v of both objects attending to the physics mentioned earlier. This is done by defining two new functions in the context called $first\_object\_v$ and $second\_object\_v$. After introducing the two function in the context according to equations 11 and 12, we reached the final version of the context:

```
context {
    name = my_collision_context2
    type = collision
    attributes = {vAtCol}
    compute_first_time_only_attributes = {vAtCol}
    aliases = {
        use first_object_vAtCol as attribute vAtCol of object with index 0
        use second_object_vAtCol as attribute vAtCol of object with index 1
        use first_object_v as attribute v of object with index 0
        use second_object_v as attribute v of object with index 1
```

```
        use n1 as attribute collisionNormal of object with index 0
    }
    functions = {
        first_object_vAtCol = first_object_v
        second_object_vAtCol = second_object_v
        first_object_v = first_object_vAtCol +
                         ((second_object_vAtCol - first_object_vAtCol) . n1)*n1
        second_object_v = second_object_vAtCol -
                          ((second_object_vAtCol - first_object_vAtCol) . n1)*n1
    }
}
```

5

ADVANCED CONTEXTS WITH REAL PHYSICS

In section 4.2 we saw in practice through simple examples how a simulation can be specified. However, to not loose the scope of what the section is about, the collision context specified did not use real physics and the free fall context was left incomplete.

In this chapter, using the practical knowledge about VirtuaFiz acquired in chapter 4, we build new contexts that use real physics. The required physics concepts will be introduced gradually during this chapter.

As a final result of the content presented in this chapter, it is expected to have a free fall, collision, and contact contexts that translate the physical behaviours that would be expected in the real world. As a scenario to the simulation we intend to build, let's consider the simulation has an arbitrary number of *dynamic* and *static* objects, each identified by a specific *position*, *orientation* and *size*. The specification of two of these objects can be specified as follow:

```
object {
    name = my_cuboid1
    physical type = static
    geometrical type = cuboid
    attributes = {
        position = [0, 0, 0]
        orientation = [0, 0, 0]
        size = [30, 30, 1]
    }
}

object {
    name = my_cuboid2
    physical type = dynamic
    geometrical type = cuboid
    attributes = {
        position = [0, 0, 5]
        orientation = [0, 0.5, 0]
        size = [3, 2, 1]
```

```
        }
    }
```

The value of the only predefined world attributes t (time) can be initialized as follow:

```
    world attributes {
        t = 0
    }
```

In section 5.1 we go throw all the steps to reach a fully functional free fall context that uses real physics. In section 5.2 and 5.3 we follow the same approach for the collision and contact contexts, respectively.

5.1 THE FREE FALL CONTEXT

The final version of the free fall context presented in section 4.2 can be used as a starting point to the free context we intend to build in this section. The context is specified as follow:

```
context {
    name = my_complete_free_fall_context
    type = free fall
    functions = {
        pos0 = position
        t0 = t
        v = v0 + a*(t-t0)
        v0 = v
        position = pos0 + v0*(t-t0) + 0.5*a*(t-t0)^2
    }
    attributes = {t0, v0, pos0}
    compute_first_time_only_attributes = {t0, pos0, v0}
}
```

If the objects in VirtuaFiz were simple particles we could use this context as it is as our final version. In fact, particles by definition don't have an orientation and an angular velocity. However, when we deal with 3 dimensional objects, these two last concepts are important and make all the difference between a non-realistic simulation and one that follows the laws of physics. In physics, an angular velocity is defined by a rotation axis and by the angular change that occurs around the axis by each time unit. For convenience, we will follow the right hand rule convention to express in which way the angular change rate is considered and its value will be expressed in rad/s. To facilitate the calculation and to reduce errors induced by approximations in operations involving floating point numbers, we will use a more compact representation called scaled-axis that only needs three numbers

instead of four (three for the axis and one for the angle). In this representation, a specific angular velocity is represented as a vector that results of the multiplication between the normalized rotation axis and the angular change rate. We will use this representation to specify the angular velocity of the objects involved in the simulation. Let's call this object attribute *a_v*. After adding attribute *a_v* and also attribute *v* to represent the linear velocity, the specification of object *my_cuboid*2 looks as follow:

```
object {
    name = my_cuboid2
    physical type = dynamic
    geometrical type = cuboid
    attributes = {
        position = [0, 0, 5]
        orientation = [0, 0.5, 0]
        size = [3, 2, 1]
        v = [10, 0, 0]
        a_v = [0, 5, 0]
    }
}
```

In this case, according to the scaled-axis representation, object *my_cuboid*2 rotates around the positive semi-axis Y at a rate of five radians per second. Figure 19 shows what we can expect to the changes in position and orientation suffered by cuboid *my_cuboid*2 during the first free fall context application according to the initial values given to the world attributes and the object attribute.

Contrary to the linear velocity that is updated at every frame using the fourth function of the context, the angular velocity does not require an update since the gravitational force has no effect in the angular velocity of an object and we are not considering air resistance. In VirtuaFiz, the only physical matter is the rigid body involved in the experience. However, the user could easily simulate the air resistance by adding a function to the context that updates the angular velocity at each moment t by removing a small portion of its value. So attribute *a_v* will keep its initial value during the whole context application until the object eventually collides with another one.

Due to its angular velocity, the orientation of the object requires an update at every frame of the simulation. So, it is necessary to add a function responsible to update this attribute. Analogously to what was done with the initial position of the object, we can add a context attribute, responsible to hold the orientation of the object at the beginning of the context application. Instead of storing this attribute using the XYZ euler representation, in this example we will use quaternions. They will prove to be helpful, as we will see. The context can be updated as follow:

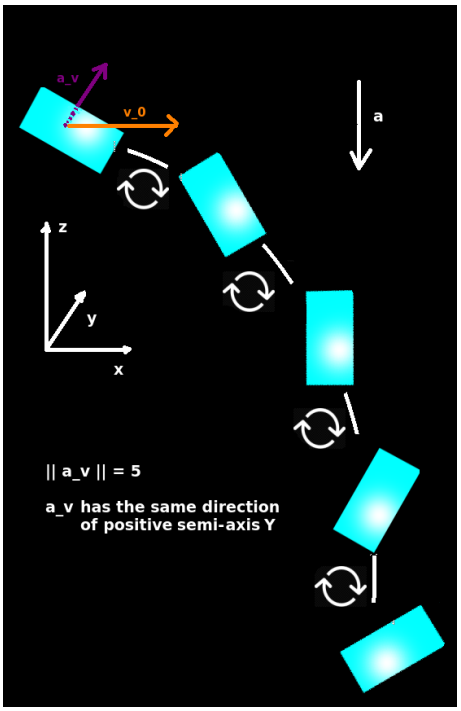


Figure 19.: Changes in position and orientation suffered by the cuboid during the first free fall context application

```
context {
    name = my_complete_free_fall_context
    type = free fall
    functions = {
        pos0 = position
        t0 = t
        v = v0 + a*(t-t0)
        v0 = v
        position = pos0 + v0*(t-t0) + 0.5*a*(t-t0)^2
        orientationQuaternion_i = eulerXYZToQuaternion(orientation)
    }
    attributes = {t0, v0, pos0, orientationQuaternion_i}
    compute_first_time_only_attributes = {t0, pos0, v0, orientationQuaternion_i}
}
```

As we can see from the context specification, VFLang native function *eulerXYZToQuaternion* is used to convert the orientation from a euler XYZ to a quaternion representation. Since we want *orientationQuaternion_i* to keep its initial value during the whole context application, it was specified as a *compute first time only attribute*.

The next step is to build a quaternion that represents the change of orientation that occurred between $t0$ and t. Using the scaled axis attribute a_v, the normalized rotation axis can be calculated as follow:

```
angular_velocity_axis = normalize(a_v)
```

and the angular change around *angular_velocity_axis* between t and $t0$ can be determined using the following function:

```
angular_velocity_angle = norm(a_v) * (t-t0)
```

VFLang native function *normalize* performs vector normalisation, and *norm* returns the length of a vector.

With the two values calculated (*angular_velocity_axis* and *angular_velocity_angle*) it is possible to calculate the quaternion that represents the change in orientation suffered by the object since $t0$. By calling VFLang native function *quaternionFromAxisAndAngle* with arguments *angular_velocity_axis* and *angular_velocity_angle* we get the desired quaternion *orientationQuaternion_update*:

```
orientationQuaternion_update =
        quaternionFromAxisAndAngle(angular_velocity_axis, angular_velocity_angle)
```

Finally, to get the orientation of the object at time t, we just need to multiply the quaternion that holds the orientation at $t0$ (*orientationQuaternion_i*) with the quaternion that holds the change in orientation between t and $t0$ (*orientationQuaternion_update*). Since the system is expecting the orientation attribute to be expressed using the XYZ euler representation, a final conversion is needed using the VFLang function *quaternionToEulerXYZ*. Thus, function *orientation* is expressed as follow:

```
orientation =
quaternionToEulerXYZ(
    multiplyQuaternions(orientationQuaternion_update, orientationQuaternion_i)
                    )
```

After introducing the four functions in the context, we reached the last version of our free fall context:

```
context {
    name = my_complete_free_fall_context
    type = free fall
    functions = {
        pos0 = position
        t0 = t
```

```
        v = v0 + a*(t-t0)
        v0 = v
        position = pos0 + v0*(t-t0) + 0.5*a*(t-t0)^2
        orientationQuaternion_i = eulerXYZToQuaternion(orientation)
        angular_velocity_axis = normalize(a_v)
        angular_velocity_angle = norm(a_v) * (t-t0)
        orientationQuaternion_update =
                quaternionFromAxisAndAngle(
                    angular_velocity_axis, angular_velocity_angle)
        orientation =
                quaternionToEulerXYZ(
                    multiplyQuaternions(
                        orientationQuaternion_update, orientationQuaternion_i)
                    )
                )
    }
    attributes = {t0, v0, pos0, orientationQuaternion_i}
    compute_first_time_only_attributes = {t0, pos0, v0, orientationQuaternion_i}
}
```

5.2 THE COLLISION CONTEXT

In the real world, when a collision occurs between two objects, the material from which they are made compresses slightly. Whether it is a rubber ball or a stone, the molecules near the point of collision are pushed together fractionally. As they compress they exert a force to try to return to their original shape. The force that resists the deformation causes the objects to stop colliding: their velocities are reduced until they are no longer moving together. At this point the objects are at their most compressed state. If there is a tendency to return to their original shape, then the force begins to accelerate them apart until they are no longer deformed. In fact, this tendency is dictated by the material of the objects. For example, a rubber ball has a higher tendency to return to its original shape than a hollow ball made of metal. At this point, all the objects that have at least a minimal tendency are typically moving apart. All this happens in the smallest fraction of a second, and for rigid bodies the compression distances are tiny fractions of a millimeter. In almost all cases the deformation cannot be seen with the naked eye. From our perspective we simply see the two objects collide and instantly bounce apart. In physics, the change in velocity suffered by each object involved in the collision is calculated by considering the change occurs instantly. The change in velocity is called an "impulse". The great advantage of this approach is it makes possible to calculate the change in velocity for the objects involved in the collision independently of the material involved. To make it work, each pair of objects involved

	Golf Ball	Tennis Ball	Baseball
Concrete	0.91969	0.81138	0.53230
Grass	0.38730	0.50827	0.42328
Dirt	0.20412	0.67082	0.29580
Wood	0.32914	0.68007	0.50415

Figure 20.: The restitution coefficient for some pair of materials

in the simulation needs to have the coefficient of restitution specified. The coefficient of restitution controls the speed at which the objects will separate after colliding. It depends on the materials of the two objects. Different pairs of material will have different coefficients. Some objects such as a tennis ball on a racket bounce apart. Other objects stick together when they collide (an egg and a floor tile, for example). If the coefficient is 1, then the objects will bounce apart with the same speed as when they were closing. If the coefficient is 0, then the objects will coalesce and travel together.

In this section, we intend to build the most complete collision context, so our context needs to have in consideration the friction forces than occurs at the time of collision between two bodies in the real world. In physics, similarly to the coefficient of restitution, a coefficient is used to represent the friction between a pair of objects. The value of the coefficient varies between 0 (means no friction) to infinite (there is no force that can possibly overcome the friction). However, typically the values never goes beyond 10. For example, velcro on velcro has a friction coefficient of 6. Two tables containing the coefficient of restitution and the two different coefficients of friction for some pair of materials are pictured in figure 20 and 21, respectively. In this section, every time we refer to friction we are actually referring to the static friction coefficient. This is the kind of friction that has to be used considering that the objects, during a collision, are in contact during a very small amount of time. In the other hand, for extended contacts, for example, a cube going down a ramp, the dynamic friction coefficient needs to be used. Our common sense is sensitive to this type of situations. For example, when pulling a heavy piece of furniture, we feel it costs a lot more to pull it out (to overcome the static friction) than to let it move (to overcome dynamic friction).

For simplification purpose, let' consider all the objects involved in the simulation are made from the same material. We leave the users of VirtuaFiz with the possibility to improve this VirtuaFiz specification so every object can have a different material. A possible idea is to specify two matrices as two world attributes that are fulfilled with the coefficient of restitution and the friction coefficient of each pair of materials. Also, every object would

Materials	Static Friction	Dynamic Friction
Wooden crate on concrete	0.5	0.4
Wooden crate on ice	0.2	0.1
Glass on ice	0.1	0.03
Glass on glass	0.95	0.4
Metal on metal	0.6	0.4
Lubricated metal on metal	0.1	0.05
Rubber on concrete	1.0	0.8
Wet rubber on concrete	0.7	0.5
Performance tire on concrete	c.1.5	1.0
Velcro on velcro	6.0	4.0

Figure 21.: The static and dynamic friction coefficient for some pair of materials

have an attribute that indicates the type of material is made of. This object attribute would work as one of two indices needed to access to the desired element in the two matrices.

For now, the coefficient of restitution and the friction coefficient are defined as two world attribute:

```
world attributes {
    t = 0
    a = [0, 0, -9819.6]
    restitution_coefficient = 0.7
    friction_coefficient = 0.3
}
```

Figure 22 and 23 shows two different outcome of the same situation of a dynamic sphere colliding with a static cuboid. In figure 22 and 23 world attribute $friction\_coefficient$ is 0 and 0.3, respectively. In both situations, $restitution\_coefficient$ is 0.7. As we can observe from figure 23, the sphere, that has no angular velocity before the collision, ends up with some angular velocity after the collision due to the frictional impulse it suffered at the time of the collision. On the other hand, in figure 22, as expected, the sphere angular velocity remains with the same value. Since the collision is not perfectly elastic (it was specified as being less than one), the sphere never returns to the highest point of the previous free fall. In fact, after multiple collision with the cuboid, the sphere will end up resting continuously in its surface. Regarding the friction and the restitution forces, these are the physical phenomena we want our collision contexts to be prepared to face.

In this chapter, to specify our collision contexts we will follow the physics impulse approach for the friction and the restitution forces. A collision between a dynamic and a static

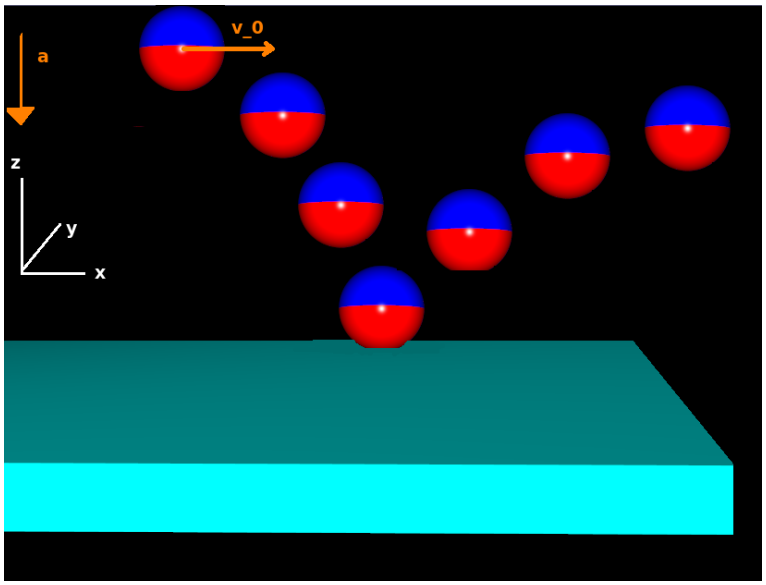


Figure 22.: Without a friction force acting on the sphere at the time of the collision, the angular velocity of the sphere remains unchanged

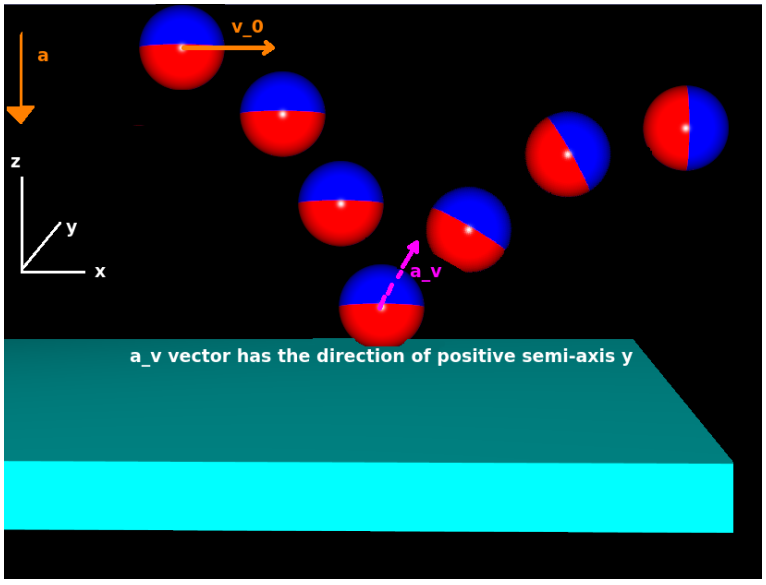


Figure 23.: The friction force acting on the sphere at the time of the collision, induces a change in the angular velocity of the sphere

object or two dynamic objects follows the same underlying physics. However, the equations used are different in some points since the static object by definition never change any of its attributes. In subsection 5.2.1 all the steps to reach a fully functional collision context between a static and a dynamic object will be covered. In the next subsection 5.2.2) the necessary changes relatively to the dynamic-static collision context will be presented to make a fully functional collision context between two dynamic objects. Notice it would have possible to create a unique collision context however, it would be necessary bigger and would be composed of many $if - then - else$ statements.

5.2.1 *Collision between a dynamic and a static object*

Before we introduce the functions needed in the collision context, let's understand the physics behind collisions. The physics concepts and equations behind the collision response presented in this chapter are based on the equations that can be found in Millington (2007). Understanding the following physics concepts will prove to be helpful to introduce the correct functions in the system. D'Alembert's principle states that any force acting on an object generates both linear and angular acceleration. The linear component a is given by Newton's second law where f and m represents the acting force and the mass of the object, respectively:

$$a = \frac{1}{m} f \tag{13}$$

The angular component is given by a physics concept called torque. Torque τ can be calculated using the following equation where d is the vector from the geometrical center of the object to the collision point:

$$\tau = d \times f \tag{14}$$

The torques generates an angular acceleration θ'' (second derivative of orientation θ) by:

$$\theta'' = I^{-1}\tau \tag{15}$$

where I is a square matrix with dimension 3 called inertia tensor. The inertia tensor tells how reactive an object is to an angular acceleration in a particular direction due to how its mass is distributed across its whole volume. The matrix is built having into consideration the dimension of the object in each axis and also its mass. The more mass is away from the center of mass, the more difficult (less reactive) the object is to suffer an angular acceleration. Figure skating is a great example to understand this concept. When a skater wants to rotates faster around its vertical axis, he bring his arms closer to his body. To the inertia tensor this

means reducing the size of the object in the two horizontal dimension. For a rigid sphere with constant density across its whole volume, inertia tensor I is given by:

$$I = \begin{bmatrix} \frac{2}{5}mr^2 & 0 & 0 \\ 0 & \frac{2}{5}mr^2 & 0 \\ 0 & 0 & \frac{2}{5}mr^2 \end{bmatrix} \tag{16}$$

where m and r are the mass and the radius of the sphere, respectively.

On the other hand, for a rigid cuboid with constant density across its whole volume, inertia tensor I is given by:

$$I = \begin{bmatrix} \frac{1}{12}m(d_y^2 + d_z^2) & 0 & 0 \\ 0 & \frac{1}{12}m(d_x^2 + d_z^2) & 0 \\ 0 & 0 & \frac{1}{12}m(d_x^2 + d_y^2) \end{bmatrix} \tag{17}$$

where m, d_x, d_y and d_z are the mass and the dimensions of the cuboid in axis X, Y and Z, respectively.

From equation 15 we understand that a higher value in torque originates a higher angular acceleration. Also, from the cross product in equation 14, we understand that the torque will be 0 if d and f makes an angle of 180 degrees. On the other hand, the torque will be at its highest value if the two vectors make an angle of 90 degrees. Figure 24 shows two dynamic cuboids without any initial angular velocity falling and ending up colliding with a static cuboid. Due to different orientations at the time of the collision, the torque suffered by the two objects is different. Regarding the left body, since the angle between the two vectors is 180 degrees, the torque is 0 so the body not suffers an angular acceleration. On the other hand, as we can see for the body on the right, an angle higher than 0 and less than 180 degrees makes the restitution force to have a torque effect on the body and induce an angular acceleration.

We said previously that we will solve the collision following the impulse approach. The same way the angular acceleration is calculated by the product of the inverse inertia tensor and the torque, the instantaneous angular change in velocity $\Delta\theta'$ is calculated by the product of the inverse inertia tensor and the impulsive torque u, which is the torque equivalent considering impulses instead of forces:

$$\Delta\theta' = I^{-1}u \tag{18}$$

The impulsive torque can be calculated following the same reasoning on how the torque is calculated in equation 14. However, we use the restitution impulse g instead of the restitution force f:

$$u = d \times g \tag{19}$$

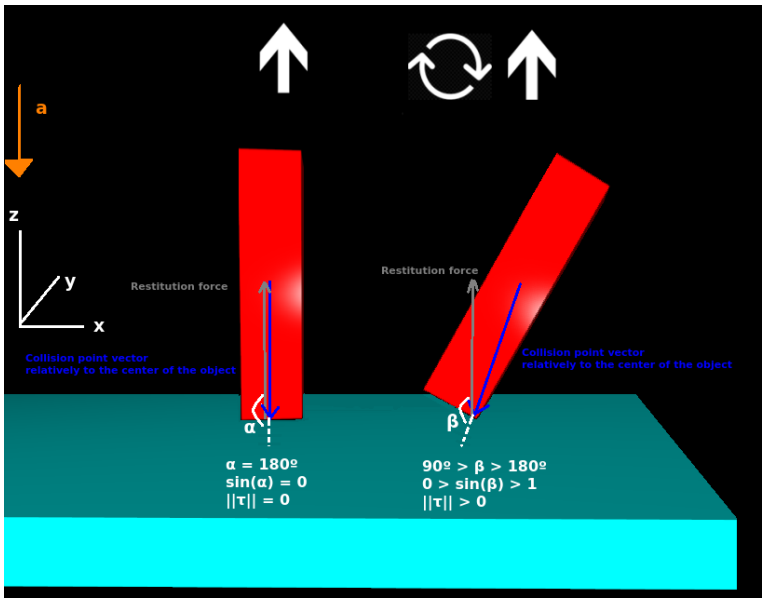


Figure 24.: The torque depends on the orientation of the objects at the time of the collision

To calculate the instant change in linear velocity Δv the impulse equivalent of equation 13 can be used:

$$\Delta v = m^{-1} g \tag{20}$$

At this point we have the necessary equations to calculate, given an impulse g, the change in linear and angular velocity suffered by the dynamic object. Before we proceed into the steps of how we can determine the impulse suffered by the object, we can update all the VirtuaFiz specification with all the physics theory we covered until now.

Regarding the objects attributes specification, two new attributes needs to be added. These are the inertia tensor and the mass. An update of the specification of object *my_cuboid2* using its last version presented in section 5.1 is shown bellow:

```
object {
    name = my_cuboid2
    physical type = dynamic
    geometrical type = cuboid
    attributes = {
        position = [0, 0, 5]
        orientation = [0, 0.5, 0]
        size = [3, 2, 1]
        v = [10, 0, 0]
        a_v = [0, 5, 0]
        mass = 15
        inertia_tensor = [ [1/12 * 15 * (2^2 + 1^2), 0, 0],
```

```
                        [0, 1/12 * 15 * (3^2 + 1^2), 0],
                        [0, 0, 1/12 * 15 * (3^2 + 2^2)] ]
    }
}
```

The inertia tensor matrix was constructed following equation 17.

Regarding the collision context specification, since in this subsection we are composing a collision context between a dynamic and static object, we don't need to specify aliases to access the attributes of the dynamic object. Also, surprisingly, as we will see until the end of this subsection, we don't need at any moment to access the attributes of the static object involved in the context. Therefore, we just refer to the attributes of the dynamic object using the names with which they were defined.

We can start by defining an empty collision context and restrict its applications to collisions involving necessarily a dynamic object and a static object.

```
context {
    name = my_collision_context_for_dynamic_static_objects
    type = collision
    applicability = {
        mandatory 1 dynamic object and 1 static object
    }
}
```

Also, let's define the linear velocity and the angular velocity at the time of the collision, as *context attribute* and *compute first time only attribute* and set its values accordingly:

```
context {
    name = my_collision_context_dynamic_static_objects
    type = collision
    attributes = {v_before, a_v_before}
    compute_first_time_only_attributes = {v_before, a_v_before}
    functions = {
        v_before = v
        a_v_before = a_v
    }
    applicability = {
        mandatory 1 dynamic object and 1 static object
    }
}
```

The linear velocity of the object after the collision is the vector addition between the linear velocity of the object before the collision and the desired change in linear velocity. Regarding the angular velocity, the reasoning is analogous. This is one of the great advantages of

using scaled-axis to represent the angular velocity or the change of angular velocity of a object. With that said, the context can be updated as follow:

```
context {
    name = my_collision_context_dynamic_static_objects
    type = collision
    attributes = {v_before, a_v_before}
    compute_first_time_only_attributes = {v_before, a_v_before}
    functions = {
        v_before = v
        a_v_before = a_v
        v = v_before + v_change
        a_v = a_v_before + a_v_change
    }
    applicability = {
        mandatory 1 dynamic object and 1 static object
    }
}
```

Using equations 18, 19 and 20, variables *v_change* and *a_v_change* can be calculated by defining some auxiliary functions. Following the same syntax of the collision normal, the position of the collision point is acceded using *collisionPoint*:

```
v_change = 1/mass * impulse
a_v_change = inverted_inertia_tensor_world_coord * impulsive_torque
impulsive_torque = center_to_col_point_vec * impulse
center_to_col_point_vec = collisionPoint - pos
```

When we introduced equation 18 we did not mention the fact that the inverse of the inertia tensor must be expressed in world coordinates. In collision, the orientation of the object is fundamental and has to be taken into consideration. So we need to transform the inertia tensor from object coordinates (as we previously defined it) to world coordinates. First, let's define a function that calculate the rotation matrix of the object:

```
orientation_matrix = subMatrix(eulerXYZToRotationMatrix(orientation), 3, 3)
```

Function *orientation_matrix* returns the rotation matrix of the object according to attribute *orientation*, which is represented using the XYZ Euler transformations. VFLang native function *eulerXYZToRotationMatrix* is used to calculated the desired transformation and *subMatrix* keeps the 3 by 3 upper matrix of the result. The result given by *orientation_matrix* is then used to calculate the inverse of the inertia tensor expressed in world coordinates as follow:

```
inverted_inertia_tensor_world_coord = orientation_matrix *
    squareMatrixInverse(inertia_tensor)*squareMatrixInverse(orientation_matrix)
```

Reading the transformations from right to left, the transformation made by $squareMatrix-Inverse(orientation\_matrix)$ allows us pass from world coordinates to object coordinates where the inverse of the inertia tensor expressed in object coordinates is calculated using VFLang native function *squareMatrixInverse*. Finally, to get the inverse of the inertia tensor expressed in world coordinates, the result is transformed by *orientation_matrix*.

At this point, the context can be expressed as follow:

```
context {
    name = my_collision_context_dynamic_static_objects
    type = collision
    attributes = {v_before, a_v_before}
    compute_first_time_only_attributes = {v_before, a_v_before}
    functions = {
        v_before = v
        a_v_before = a_v
        v = v_before + v_change
        a_v = a_v_before + a_v_change
        v_change = 1/mass * impulse
        a_v_change = inverted_inertia_tensor_world_coord * impulsive_torque
        impulsive_torque = center_to_col_point_vec * impulse
        center_to_col_point_vec = collisionPoint - pos
        orientation_matrix=subMatrix(eulerXYZToRotationMatrix(orientation),3,3)
        inverted_inertia_tensor_world_coord = orientation_matrix *
            squareMatrixInverse(inertia_tensor) *
            squareMatrixInverse(orientation_matrix)
    }
    applicability = {
        mandatory 1 dynamic object and 1 static object
    }
}
```

The effect of the impulse is to have the two points in collision (one belonging to each object) bounce apart. To calculate how the points bounce apart after the collision, we need first to calculate how the two points are approaching each other. How fast one particle travel relatively to another is called the relative velocity. The relative velocity of a particle relatively to another is simply the difference between the total velocity of the particle and the total velocity of the other particle. In this subsection, since in the context one object is static, the relative velocity of the dynamic point relatively to the static point is simply the total velocity of the dynamic point. The total velocity q' of a point in a object is given by the following equation:

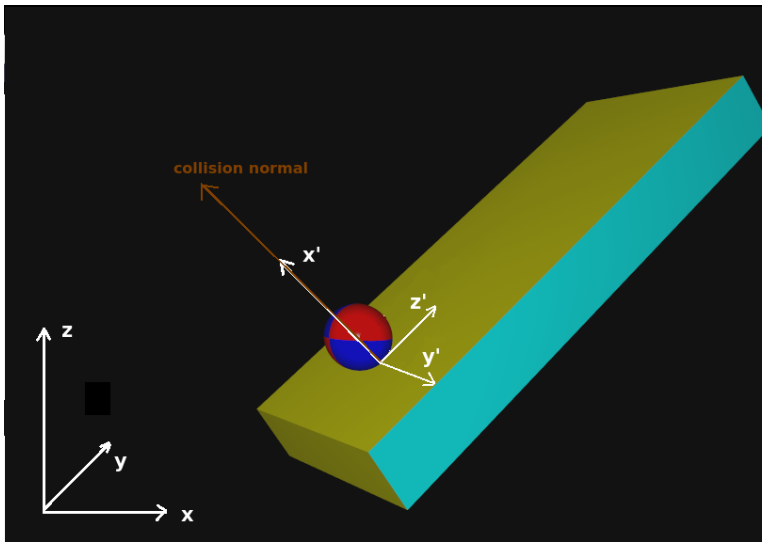


Figure 25.: The world system coordinates (left) and the collision system coordinates (right)

$$q' = \theta' \times d + v \tag{21}$$

where d is the vector from the center of the object to the point, θ' and v represents the angular and linear velocity of the object, respectively.

Until the end of this subsection, the relative velocity of the dynamic point relatively to the static point will be simply referred as the relative velocity between the two objects.

From the equation above we understand that the further away the point is from the center, the more total velocity the point has. Also, a higher angular velocity induces a higher total velocity. Our common sense is sensible to this kind of situation. For example, in the situation of two runners running on two different circular lanes, the runner on the outermost lane needs to run faster to stay at the level of the other runner. They run at the same angular velocity but different instantaneous linear velocity.

At this point, we intend to calculate the desired change in the relative velocity between the two objects. To simplify the calculation, we will switch many times between the world system coordinates (the one we worked until now) and the collision system coordinates. Figure 25 shows the collision system coordinates of a sphere in a collision with a cuboid. Axis x' has the direction of the sphere's contact normal, while y' and z' are two arbitrary axis that along with axis x' form an orthogonal basis. The origin of the referential is the point of the sphere in collision.

Regarding the scenario pictured in figure 25, if we were dealing with friction-less collisions, the impulse suffered by the sphere would only be in the direction of the collision normal (axis x'). However, with friction, the impulse vector can also have a non zero value in axis y' and z'. In fact, static friction kills the relative velocity in the y' and z' directions

(it acts to keep the velocity at zero in the contact plane). Therefore, components y' and z' of the desired change in the relative velocity are the symmetric of components y' and z' of the relative velocity at the time of the collision, respectively. To calculate the component x' of the desired change in the relative velocity Δr the following equation is used:

$$\Delta r_{x'} = -r_{x'} * (1 + c) \tag{22}$$

where c is the coefficient of restitution and $r_{x'}$ the component x' of the relative velocity at the time of the collision.

From the equation above we understand that a higher value in the restitution coefficient induces a higher change in the relative velocity in the x' direction. Also, for example, if the restitution coefficient is 0, the desired change in velocity in direction x' just cancels the relative velocity in direction x'. In this case, the objects will stick together after the collision.

Before we reach the final steps to calculate the impulse to apply on the dynamic object, let's update our collision context with everything new we covered from the last update. Let's start by defining a function that calculates the total velocity of the collision point of the dynamic object using equation 21:

```
total_velocity = v_before + (a_v_before * center_to_col_point_vec)
```

As we mentioned previously, the static object does not contribute to the relative velocity between the two bodies. So the relative velocity can be defined as follow:

```
relative_velocity = total_velocity
```

Function *relative_velocity_col_coord* returns the *relative_velocity* vector expressed into collision coordinates:

```
relative_velocity_col_coord = squareMatrixInverse(collision_basis)
                              * relative_velocity
```

where *collision_basis* is defined by:

```
collision_basis = arbitraryBasis(collisionNormal)
```

arbitraryBasis is a VFLang native function that given a three dimensional vector, returns a matrix containing this vector and two more that can form an orthogonal basis. The vector passed as argument is used as axis x of the newly created referential. Thus, the multiplication between this base and a vector expressed in this same base, returns the vector but expressed in world coordinates. Since in function *relative_velocity_col_coord* we want to get *relative_velocity* from world coordinates to collision coordinates, *relative_velocity* is transformed by the inverse of *collision_basis*.

The desired change in the relative velocity expressed in collision coordinates can be defined as follow:

```
desired_change_relative_velocity = [ desired_change_relative_velocity_x,
                                     -relative_velocity_col_coord # 2  ,
                                     -relative_velocity_col_coord # 3  ]
```

Using equation 22, function *desired_change_relative_velocity_x* is defined as follow:

```
desired_change_relative_velocity_x =
    -relative_velocity_col_coord # 1 * (1 + restitution_coefficient)
```

At this point, the context can be expressed as follow:

```
context {
    name = my_collision_context_dynamic_static_objects
    type = collision
    attributes = {v_before, a_v_before}
    compute_first_time_only_attributes = {v_before, a_v_before}
    functions = {
        v_before = v
        a_v_before = a_v
        v = v_before + v_change
        a_v = a_v_before + a_v_change
        v_change = 1/mass * impulse
        a_v_change = inverted_inertia_tensor_world_coord * impulsive_torque
        impulsive_torque = center_to_col_point_vec * impulse
        center_to_col_point_vec = collisionPoint - pos
        orientation_matrix=subMatrix(eulerXYZToRotationMatrix(orientation),3,3)
        inverted_inertia_tensor_world_coord = orientation_matrix *
            squareMatrixInverse(inertia_tensor) *
            squareMatrixInverse(orientation_matrix)
        total_velocity = v_before + (a_v_before*center_to_col_point_vec)
        relative_velocity = total_velocity
        relative_velocity_col_coord = squareMatrixInverse(collision_basis)
                                      * relative_velocity
        collision_basis = arbitraryBasis(collisionNormal)
        desired_change_relative_velocity=[ desired_change_relative_velocity_x,
                                           -relative_velocity_col_coord # 2  ,
                                           -relative_velocity_col_coord # 3  ]
        desired_change_relative_velocity_x =
            -relative_velocity_col_coord # 1 * (1 + restitution_coefficient)
    }
    applicability = {
        mandatory 1 dynamic object and 1 static object
    }
}
```

Using the desired change in relative velocity and knowing the change in relative velocity made by one unit of impulse, it is possible to calculate directly the necessary impulse. Since in the context there is only one dynamic object, the change in relative velocity between the two objects made by one unit of impulse is simply the change in total velocity of the dynamic point in collision made by one unit of impulse. From equation 21 we know that the total velocity of a point in a object is the sum of the linear velocity of the object and the instantaneous linear velocity of the point induced by the angular velocity of the object.

If we were dealing with friction-less collisions the change in the total velocity of the dynamic point made by one unit of impulse would be calculated easily since we know exactly how the unit of impulse vector looks like in this situation. It's a unit vector having the direction of the collision normal (axis x in collision coordinates). So in this case, considering equation 20 the change in linear velocity would be a vector with length m^{-1} pointing in the direction of the impulse (axis x in contact coordinates). To get the instantaneous linear velocity of the point induced by the angular velocity we would use equation 19 to get the the impulsive torque generated by one unit of impulse. Using this value and equation 18, we would know the change in the angular velocity of the object made by one unit of impulse. The change in the rotation-induced linear velocity would be obtained, as shown in equation, 21 by the cross product between the change in the angular velocity obtained and the vector from the center of the object to the point in collision. That's how we would have proceed to get the change in relative velocity made by one unit of impulse if we knew the direction of the unit of impulse vector.

However, in collisions with friction, we saw previously that the desired change in velocity can have components in the three axis of the collision referential. For that reason, the impulse vector can also have components in the three axis. Since we don't know the direction of the unit of impulse vector, all the equations mentioned above cannot be used directly to determine the change in relative velocity made by one unit of impulse.

Rather than use the contact normal, we need to use all three directions of the collision referential. But if the contact normal is replaced by a matrix (the collision basis), how do we perform the cross products that appear in equations 19 and 21? It turns out that we can create a matrix form of the vector product. The cross product of two vectors a and b

$$a * b \tag{23}$$

is equivalent to the matrix-vector multiplication

$$M_a * b \tag{24}$$

where M_a is called the skew-symmetric matrix of vector a.

To get the impulsive torque made by one unit of impulse considering the three directions of the collision basis instead of only the collision normal direction g shown in equation 19, the cross product

$$u = d * g \tag{25}$$

gives place to the matrix-matrix multiplication

$$M_u = M_d * M_g \tag{26}$$

where M_d is the skew-symmetric matrix of vector d (the vector from the the center of the object to the point in collision) and M_g is the basis matrix of the collision referential.

M_u is also a matrix. It represents how a unit of impulse with any possible direction, may affect the norm and the direction of the impulsive torque regarding the three directions of the collision referential.

The change in the angular velocity originated by the impulsive torque can be calculated following the same reasoning of equation 18. The matrix-vector multiplication

$$\Delta\theta' = I^{-1} * u \tag{27}$$

gives place to the matrix-matrix multiplication

$$M_{\Delta\theta'} = I^{-1} * M_u \tag{28}$$

Finally we adapt the first portion of equation 21, to get, using $M_{\Delta\theta'}$, the rotation-induced instantaneous linear velocity change q'_{rot} of the point in collision. The vector cross product

$$\Delta q'_{rot} = \Delta\theta' * d \tag{29}$$

gives place to the matrix multiplication

$$M_{\Delta q'_{rot}} = -M_{\Delta\theta'} * M_d \tag{30}$$

In the equation, the expression $M_{\Delta\theta'} * M_d$ is the reverse cross product since the skew-symmetric matrix of the vector appears on the right side of the multiplication instead of the left side (see equation 24). Just as for the regular cross product between two vectors, multiplying the result of the reverse cross product by minus one, gives the desired cross product.

As we said previously, we need to combine $M_{\Delta q'_{rot}}$ with the change in linear velocity to get the total change in relative velocity made by one unit of impulse. Equation 20 tells us that the change in linear velocity has the direction of the impulse and has length m^{-1}. Since

we don't know the direction of the impulse, we express the change in linear velocity made by one unit of impulse according to the three possible axis of the collision basis:

$$\begin{bmatrix} m^{-1} & 0 & 0 \\ 0 & m^{-1} & 0 \\ 0 & 0 & m^{-1} \end{bmatrix} \tag{31}$$

Notice that the change in linear velocity above is expressed in collision coordinates.

Now we know how to calculate matrix $M_{\Delta r\_g}$ that gives the change in relative velocity made by a generic unit of impulse and also vector Δr that represents the desired change in relative velocity, the desired impulse vector g is simply given by the following matrix-vector multiplication:

$$g = M^{-1}_{\Delta r\_g} * \Delta r \tag{32}$$

Let's update our collision context with the necessary functions to calculate the desired impulse vector. Since we need two times (equations 30 and 26) the skew-symmetric matrix of the vector that goes from the center of the objects to the point in collision we can start by defining a function that return the desired skew-symmetric matrix:

```
center_to_col_point_skew_sym = skewSymmetricMatrix(center_to_col_point_vec)
```

where *skewSymmetricMatrix* is a VFLang native function that returns the skew-symmetric matrix of the vector passed as argument.

Using equation 26, the impulsive torque originated from one unit of impulse is calculated using the following function:

```
impulsive_torque_unit_impulse = center_to_col_point_skew_sym * collision_basis
```

Combining equations 28 and 30, the following function returns the rotation-induced instantaneous linear velocity caused by one unit of impulse:

```
rot_induced_i_l_v_by_unit_impulse =
    - (inverted_inertia_tensor_world_coord * impulsive_torque_unit_impulse)
    * center_to_col_point_skew_sym
```

linear_velocity_by_unit_impulse returns the change in linear velocity (expressed in contact coordinates) caused by one unit of impulse according to matrix 31

```
linear_velocity_by_unit_impulse = identityMatrix(3) * (mass ^ -1)
```

where *identityMatrix* is a native function that returns a square identity matrix with dimension 3.

rot_induced_i_l_v_by_unit_impulse and *linear_velocity_by_unit_impulse* are combined to get the relative velocity change caused by one unit of impulse:

```
vec_change_unit_impulse = linear_velocity_by_unit_impulse
    + squareMatrixInverse(collision_basis) * rot_induced_i_l_v_by_unit_impulse
```

In the function above, to get *vec_change_unit_impulse* expressed in collision coordinates, we had to transform first *rot_induced_i_l_v_by_unit_impulse* from world to collision coordinates using the inverse of the collision basis.

Finally, using equation 32, the desired impulse (expressed in collision coordinates) can be calculated by the following function:

```
desired_impulse_col_coord = squareMatrixInverse(vec_change_unit_impulse)
                          * desired_change_relative_velocity
```

With all these new functions, the context specification looks as follow:

```
context {
    name = my_collision_context_dynamic_static_objects
    type = collision
    attributes = {v_before, a_v_before}
    compute_first_time_only_attributes = {v_before, a_v_before}
    functions = {
        v_before = v
        a_v_before = a_v
        v = v_before + v_change
        a_v = a_v_before + a_v_change
        v_change = 1/mass * impulse
        a_v_change = inverted_inertia_tensor_world_coord * impulsive_torque
        impulsive_torque = center_to_col_point_vec * impulse
        center_to_col_point_vec = collisionPoint - pos
        orientation_matrix = subMatrix(eulerXYZToRotationMatrix(orientation),3,3)
        inverted_inertia_tensor_world_coord = orientation_matrix *
            squareMatrixInverse(inertia_tensor) *
            squareMatrixInverse(orientation_matrix)
        total_velocity = v_before + (a_v_before*center_to_col_point_vec)
        relative_velocity = total_velocity
        relative_velocity_col_coord = squareMatrixInverse(collision_basis)
                                    * relative_velocity
        collision_basis = arbitraryBasis(collisionNormal)
        desired_change_relative_velocity=[ desired_change_relative_velocity_x,
                                           -relative_velocity_col_coord # 2  ,
                                           -relative_velocity_col_coord # 3  ]
        desired_change_relative_velocity_x =
            -relative_velocity_col_coord # 1 * (1 + restitution_coefficient)
        center_to_col_point_skew_sym=skewSymmetricMatrix(center_to_col_point_vec)
        impulsive_torque_unit_impulse=center_to_col_point_skew_sym*collision_basis
```

```
        rot_induced_i_l_v_by_unit_impulse =
            - (inverted_inertia_tensor_world_coord*impulsive_torque_unit_impulse)
            * center_to_col_point_skew_sym
        linear_velocity_by_unit_impulse = identityMatrix(3) * (mass ^ -1)
        vec_change_unit_impulse = linear_velocity_by_unit_impulse
        + squareMatrixInverse(collision_basis)*rot_induced_i_l_v_by_unit_impulse
        desired_impulse_col_coord = squareMatrixInverse(vec_change_unit_impulse)
                                    * desired_change_relative_velocity
    }
    applicability = {
        mandatory 1 dynamic object and 1 static object
    }
}
```

That would be all we need in our collision context if there was not a last aspect related to the friction needed to be approached. When we calculated the desired change in relative velocity we defined that the friction would kill all the relative velocity in the collision plane. However, this is not always true. For example, if the relative velocity in that plane is relatively high and the friction coefficient is relatively low, the frictional forces acting during the time of the collision are not able to completely kill all the velocity in that plane. Having in mind finding a solution for this particularity, let's start by seeing how the norm of the static friction force f_{static} is calculated:

$$|f_{static}| \leq \mu_{static}|r| \tag{33}$$

where μ_{static} is the static friction coefficient and r is the reaction force in the direction of the collision normal.

From the inequation above, we understand that the norm of the static friction is directly proportional to the normal reaction force. Also, the inequality remind us that the static friction is an adaptive force. This force counters, with the same norm, up to a maximum value, any force exercised in the object in the opposite direction. This result also applies on impulses. Up to a maximum value, the higher the impulse is in the collision normal, the higher the impulse can be in the collision plane (friction).

To check if the impulse in the contact plane (friction impulse) exceeds the maximum possible value, we can start by defining a function that computes the norm of the projection in the collision plane of the calculated impulse *desired_impulse_col_coord*

```
norm_friction_imp=(desired_impulse_col_coord#2 + desired_impulse_col_coord#3)^0.5
```

To check, according to equation 33, if the norm of the friction impulse exceed the maximum possible value, the following function is defined:

```
final_impulse_col_coord =
    if norm_friction_imp <= desired_impulse_col_coord # 1 * friction_coefficient
        then desired_impulse_col_coord
        else modified_impulse_col_coord
```

In the function above, we used a if-then-else statement to specify that if the norm of the friction impulse is less or equal than its maximum possible value then we use the value we previously calculated without any modification. Otherwise, we need to modify the impulse so the inequality is respected.

How *modified_impulse_col_coord* must be defined will be explained just after we define a function that transform *final_impulse_col_coord* from collision to world coordinates. At the beginning of this subsection, we defined several functions that calculate the new linear and angular velocity of the dynamic object according to *impulse* that would be calculated later. It is expected that *impulse* is expressed in world coordinates. Thus, we can define *impulse* as follow:

```
impulse = collision_basis * final_impulse_col_coord
```

Regarding vector *modified_impulse_col_coord* we define its three component separately using three auxiliary functions:

```
modified_impulse_col_coord = [modified_impulse_col_coord_x,
                              modified_impulse_col_coord_y,
                              modified_impulse_col_coord_z]
```

We saw previously that the norm of the impulse in the collision plane (friction) is directly proportional to the friction coefficient and the norm of the impulse in the collision normal direction. In this case what we intend to do is to downscale the impulse vector in the collision plane so its norm is equal to its maximum possible value. The direction of the impulse vector in the collision plane remains unchanged.

The direction of the impulse regarding axis Y and Z of the collision referential can be calculated using the following two functions:

```
impulse_col_coord_y_dir = desired_impulse_col_coord # 2 / norm_friction_imp
impulse_col_coord_z_dir = desired_impulse_col_coord # 3 / norm_friction_imp
```

Components Y and Z of the modified impulse can be specified as follow:

```
modified_impulse_col_coord_y=(modified_impulse_col_coord_x*friction_coefficient)
                             * impulse_col_coord_y_dir
modified_impulse_col_coord_z=(modified_impulse_col_coord_x*friction_coefficient)
                             * impulse_col_coord_z_dir
```

At this stage, we just need to know how to define *modified_impulse_col_coord_x*. To calculate *modified_impulse_col_coord_x* we need first to calculate the change in relative velocity in axis X made by one unit of impulse in axis X, *modified_vec_change_unit_impulse_x*. Having this value, *modified_impulse_col_coord_x* can simply be defined by:

```
modified_impulse_col_coord_x =
    desired_change_relative_velocity_x / modified_vec_change_unit_impulse_x
```

Previously, we determined matrix *vec_change_unit_impulse* that gives the change in relative velocity matrix according to a generic unit of impulse. We will use this matrix again to calculate *modified_vec_change_unit_impulse_x*. The first, second and third column represent the change in relative velocity made by a unit of impulse with direction of axis X, Y and Z (in collision coordinates), respectively. Thus, for example, the first, second and third element of the third column indicates, for a unit of impulse with direction of axis Z, the change in relative velocity according to axis X, Y, Z, respectively.

With that said, we understand that

```
vec_change_unit_impulse # 1 # 1
```

gives us the change in relative velocity in axis X caused by one unit of impulse in axis X.

However, the real change in relative velocity in axis X is not only this value. This is one of the three values we need to sum together to get the change in relative velocity in axis X caused by one unit of impulse in axis X. In fact, from what we covered about friction impulse, impulses in the direction of the collision normal (axis X) makes possible the existence of an friction impulse in the collision plane (axis Y and Z). Considering one unit of impulse in axis X, the impulse in the collision plane has a maximum norm equal to the restitution coefficient. This impulse has component in axis Y and Z. Depending on the first element of the second and third column of the matrix, the two components of the planar impulse can potentially induces a change in the relative velocity in axis X.

Putting this reasoning into functions, *modified_vec_change_unit_impulse_x* can be defined as follow:

```
modified_vec_change_unit_impulse_x = vec_change_unit_impulse#1#1
    + vec_change_unit_impulse#1#2*friction_coefficient*impulse_col_coord_y_dir
    + vec_change_unit_impulse#1#3*friction_coefficient*impulse_col_coord_z_dir
```

The final version of the specification of our collision context between a dynamic object and static object looks as follow:

```
context {
    name = my_collision_context_dynamic_static_objects
    type = collision
```

```
    attributes = {v_before, a_v_before}
    compute_first_time_only_attributes = {v_before, a_v_before}
    functions = {

v_before = v
a_v_before = a_v
v = v_before + v_change
a_v = a_v_before + a_v_change
v_change = 1/mass * impulse
a_v_change = inverted_inertia_tensor_world_coord * impulsive_torque
impulsive_torque = center_to_col_point_vec * impulse
center_to_col_point_vec = collisionPoint - pos
orientation_matrix=subMatrix(eulerXYZToRotationMatrix(orientation),3,3)
inverted_inertia_tensor_world_coord = orientation_matrix *
    squareMatrixInverse(inertia_tensor) *
    squareMatrixInverse(orientation_matrix)
total_velocity = v_before + (a_v_before*center_to_col_point_vec)
relative_velocity = total_velocity
relative_velocity_col_coord = squareMatrixInverse(collision_basis)
                              * relative_velocity
collision_basis = arbitraryBasis(collisionNormal)
desired_change_relative_velocity=[ desired_change_relative_velocity_x,
                                  -relative_velocity_col_coord # 2  ,
                                  -relative_velocity_col_coord # 3  ]
desired_change_relative_velocity_x =
    -relative_velocity_col_coord # 1 * (1 + restitution_coefficient)
center_to_col_point_skew_sym=skewSymmetricMatrix(center_to_col_point_vec)
impulsive_torque_unit_impulse=center_to_col_point_skew_sym*collision_basis
rot_induced_i_l_v_by_unit_impulse =
    - (inverted_inertia_tensor_world_coord * impulsive_torque_unit_impulse)
    * center_to_col_point_skew_sym
linear_velocity_by_unit_impulse = identityMatrix(3) * (mass ^ -1)
vec_change_unit_impulse = linear_velocity_by_unit_impulse
    +squareMatrixInverse(collision_basis)*rot_induced_i_l_v_by_unit_impulse
desired_impulse_col_coord = squareMatrixInverse(vec_change_unit_impulse)
                            * desired_change_relative_velocity
norm_friction_imp =
    (desired_impulse_col_coord#2 + desired_impulse_col_coord#3)^0.5
final_impulse_col_coord =
    if norm_friction_imp<=desired_impulse_col_coord#1*friction_coefficient
    then desired_impulse_col_coord
    else modified_impulse_col_coord
impulse = collision_basis * final_impulse_col_coord
```

```
modified_impulse_col_coord = [modified_impulse_col_coord_x,
                              modified_impulse_col_coord_y,
                              modified_impulse_col_coord_z]
impulse_col_coord_y_dir = desired_impulse_col_coord # 2 / norm_friction_imp
impulse_col_coord_z_dir = desired_impulse_col_coord # 3 / norm_friction_imp
modified_impulse_col_coord_y=(modified_impulse_col_coord_x*friction_coefficient)
                            * impulse_col_coord_y_dir
modified_impulse_col_coord_z=(modified_impulse_col_coord_x*friction_coefficient)
                            * impulse_col_coord_z_dir
modified_impulse_col_coord_x =
    desired_change_relative_velocity_x / modified_vec_change_unit_impulse_x
modified_vec_change_unit_impulse_x = vec_change_unit_impulse#1#1
    + vec_change_unit_impulse#1#2*friction_coefficient*impulse_col_coord_y_dir
    + vec_change_unit_impulse#1#3*friction_coefficient*impulse_col_coord_z_dir

    }
    applicability = {
        mandatory 1 dynamic object and 1 static object
    }
}
```

5.2.2 *Collision between two dynamic objects*

Before we go in detail about the differences needed to be taken into consideration in the functions, between a dynamic-static and dynamic-dynamic collision context, let's remember each step we went throw in subsection 5.2.1 to build our collision collision:

1. Define the functions that calculate the new linear and angular velocity suffered by the object given the impulse
2. Define the function that returns the desired change in relative velocity, Δr
3. Determine matrix $M_{\Delta r\_g}$ that translates the change in relative velocity made by one unit of impulse according to the three potential directions of the collision referential.
4. Determine the desired impulse according to $M_{\Delta r\_g}$ and Δr
5. Check if the friction impulse (components Y and Z of the desired impulse, in collision coordinates) exceeds its maximum possible. If yes, determine the necessary modified impulse.

Regarding step 1, for the collision context between a dynamic and a static object we end up with the following functions:

```
v_before = v
a_v_before = a_v
v = v_before + v_change
a_v = a_v_before + a_v_change
v_change = 1/mass * impulse
a_v_change = inverted_inertia_tensor_world_coord * impulsive_torque
impulsive_torque = center_to_col_point_vec * impulse
center_to_col_point_vec = collisionPoint - pos
orientation_matrix = subMatrix(eulerXYZToRotationMatrix(orientation), 3, 3)
inverted_inertia_tensor_world_coord = orientation_matrix *
    squareMatrixInverse(inertia_tensor) *
    squareMatrixInverse(orientation_matrix)
```

Newton Third Law states that any force induces another force with the same magnitude but with opposite direction. This result also applies on impulses. In a collision between two dynamic objects, both objects suffers an impulse with the same norm but opposite direction. Therefore, in the new functions, *impulse* is the impulse suffered by the first object and $-impulse$ is the impulse suffered by the second object:

```
obj1_v_before = obj1_v
obj1_a_v_before = obj1_a_v
obj1_v = obj1_v_before + obj1_v_change
obj1_a_v = obj1_a_v_before + obj1_a_v_change
obj1_v_change = 1/obj1_mass * impulse
obj1_a_v_change = obj1_inverted_inertia_tensor_world_coord*obj1_impulsive_torque
obj1_impulsive_torque = obj1_center_to_col_point_vec * impulse
obj1_center_to_col_point_vec = collisionPoint - obj1_pos
obj1_orientation_matrix=subMatrix(eulerXYZToRotationMatrix(obj1_orientation),3,3)
obj1_inverted_inertia_tensor_world_coord = obj1_orientation_matrix *
    squareMatrixInverse(obj1_inertia_tensor) *
    squareMatrixInverse(obj1_orientation_matrix)

obj2_v_before = obj2_v
obj2_a_v_before = obj2_a_v
obj2_v = obj2_v_before + obj2_v_change
obj2_a_v = obj2_a_v_before + obj2_a_v_change
obj2_v_change = 1/obj2_mass * -impulse
obj2_a_v_change = obj2_inverted_inertia_tensor_world_coord*obj2_impulsive_torque
obj2_impulsive_torque = obj2_center_to_col_point_vec * -impulse
obj2_center_to_col_point_vec = collisionPoint - obj2_pos
obj2_orientation_matrix=subMatrix(eulerXYZToRotationMatrix(obj2_orientation),3,3)
obj2_inverted_inertia_tensor_world_coord = obj2_orientation_matrix *
    squareMatrixInverse(obj2_inertia_tensor) *
    squareMatrixInverse(obj2_orientation_matrix)
```

Remember that all the aliases we use in the functions must be defined in the context specification. We will specified them all at once in the final version of this context specification.

Regarding step 2, we have the following functions in the dynamic-static collision context:

```
total_velocity = v_before + (a_v_before*center_to_col_point_vec)
relative_velocity = total_velocity
relative_velocity_col_coord = squareMatrixInverse(collision_basis)
                            * relative_velocity
collision_basis = arbitraryBasis(collisionNormal)
desired_change_relative_velocity = [ desired_change_relative_velocity_x,
                                     -relative_velocity_col_coord # 2  ,
                                     -relative_velocity_col_coord # 3  ]
desired_change_relative_velocity_x =
    -relative_velocity_col_coord # 1 * (1 + restitution_coefficient)
```

For collisions between two dynamic objects, the relative velocity of the first object in relation to the second, is the difference between the total velocity of the first object and the total velocity of the second object. So for step 2, we have the following set of functions:

```
obj1_total_velocity = obj1_v_before+(obj1_a_v_before*obj1_center_to_col_point_vec)
obj2_total_velocity = obj2_v_before+(obj2_a_v_before*obj2_center_to_col_point_vec)
relative_velocity = obj1_total_velocity - obj2_total_velocity
relative_velocity_col_coord = squareMatrixInverse(collision_basis)
                            * relative_velocity
collision_basis = arbitraryBasis(obj1_collisionNormal)
desired_change_relative_velocity = [ desired_change_relative_velocity_x,
                                     -relative_velocity_col_coord # 2  ,
                                     -relative_velocity_col_coord # 3  ]
desired_change_relative_velocity_x =
    -relative_velocity_col_coord # 1 * (1 + restitution_coefficient)
```

In the dynamic-static collision context, for step 3 we used the following functions:

```
center_to_col_point_skew_sym = skewSymmetricMatrix(center_to_col_point_vec)
impulsive_torque_unit_impulse=center_to_col_point_skew_sym*collision_basis
rot_induced_i_l_v_by_unit_impulse =
    - (inverted_inertia_tensor_world_coord * impulsive_torque_unit_impulse)
    * center_to_col_point_skew_sym
linear_velocity_by_unit_impulse = identityMatrix(3) * (mass ^ -1)
vec_change_unit_impulse = linear_velocity_by_unit_impulse
    + squareMatrixInverse(collision_basis) * rot_induced_i_l_v_by_unit_impulse
```

In subsection 5.2.1 we considered the change in the relative velocity as the change in total velocity of the only dynamic object. In a collision between two dynamic objects, we sum

together the change in total velocity of each object made a generic unit of impulse to get the change in relative velocity made a generic unit of impulse. So the necessary functions for step 3 are the following:

```
obj1_center_to_col_point_skew_sym=skewSymmetricMatrix(obj1_center_to_col_point_vec)
obj1_impulsive_torque_unit_impulse=obj1_center_to_col_point_skew_sym*collision_basis
obj1_rot_induced_i_l_v_by_unit_impulse =
  - (obj1_inverted_inertia_tensor_world_coord * obj1_impulsive_torque_unit_impulse)
  * obj1_center_to_col_point_skew_sym
obj1_linear_velocity_by_unit_impulse = identityMatrix(3) * (obj1_mass ^ -1)

obj2_center_to_col_point_skew_sym=skewSymmetricMatrix(obj2_center_to_col_point_vec)
obj2_impulsive_torque_unit_impulse=obj2_center_to_col_point_skew_sym*collision_basis
obj2_rot_induced_i_l_v_by_unit_impulse =
    - (obj2_inverted_inertia_tensor_world_coord*obj2_impulsive_torque_unit_impulse)
    * obj2_center_to_col_point_skew_sym
obj2_linear_velocity_by_unit_impulse = identityMatrix(3) * (obj2_mass ^ -1)

vec_change_unit_impulse = obj1_linear_velocity_by_unit_impulse
    + obj2_linear_velocity_by_unit_impulse
    + squareMatrixInverse(collision_basis) * obj1_rot_induced_i_l_v_by_unit_impulse
    + squareMatrixInverse(collision_basis) * obj2_rot_induced_i_l_v_by_unit_impulse
```

For step 4 we simply use the same function we defined in the dynamic-static context:

```
desired_impulse_col_coord = squareMatrixInverse(vec_change_unit_impulse)
                          * desired_change_relative_velocity
```

The same applies for step 5. The same functions are used:

```
norm_friction_imp = (desired_impulse_col_coord#2 + desired_impulse_col_coord#3)^0.5
final_impulse_col_coord =
    if norm_friction_imp <= desired_impulse_col_coord # 1 * friction_coefficient
    then desired_impulse_col_coord
    else modified_impulse_col_coord
impulse = collision_basis * final_impulse_col_coord
modified_impulse_col_coord = [modified_impulse_col_coord_x,
                              modified_impulse_col_coord_y,
                              modified_impulse_col_coord_z]
impulse_col_coord_y_dir = desired_impulse_col_coord # 2 / norm_friction_imp
impulse_col_coord_z_dir = desired_impulse_col_coord # 3 / norm_friction_imp
modified_impulse_col_coord_y = (modified_impulse_col_coord_x*friction_coefficient)
                             * impulse_col_coord_y_dir
modified_impulse_col_coord_z = (modified_impulse_col_coord_x*friction_coefficient)
```

```
                                * impulse_col_coord_z_dir
modified_impulse_col_coord_x =
    desired_change_relative_velocity_x / modified_vec_change_unit_impulse_x
modified_vec_change_unit_impulse_x = vec_change_unit_impulse#1#1
    + vec_change_unit_impulse#1#2 * friction_coefficient * impulse_col_coord_y_dir
    + vec_change_unit_impulse#1#3 * friction_coefficient * impulse_col_coord_z_dir
```

Putting all the functions in the specification and defining the necessary aliases, the VirtuaFiz specification of our collision context between two dynamic objects looks as follows:

```
context {
    name = my_collision_context_two_dynamic_objects
    type = collision
    attributes = {v_before, a_v_before}
    compute_first_time_only_attributes = {v_before, a_v_before}
    functions = {

obj1_v_before = obj1_v
obj1_a_v_before = obj1_a_v
obj1_v = obj1_v_before + obj1_v_change
obj1_a_v = obj1_a_v_before + obj1_a_v_change
obj1_v_change = 1/obj1_mass * impulse
obj1_a_v_change = obj1_inverted_inertia_tensor_world_coord * obj1_impulsive_torque
obj1_impulsive_torque = obj1_center_to_col_point_vec * impulse
obj1_center_to_col_point_vec = collisionPoint - obj1_pos
obj1_orientation_matrix=subMatrix(eulerXYZToRotationMatrix(obj1_orientation),3,3)
obj1_inverted_inertia_tensor_world_coord = obj1_orientation_matrix *
    squareMatrixInverse(obj1_inertia_tensor) *
    squareMatrixInverse(obj1_orientation_matrix)
obj2_v_before = obj2_v
obj2_a_v_before = obj2_a_v
obj2_v = obj2_v_before + obj2_v_change
obj2_a_v = obj2_a_v_before + obj2_a_v_change
obj2_v_change = 1/obj2_mass * -impulse
obj2_a_v_change = obj2_inverted_inertia_tensor_world_coord * obj2_impulsive_torque
obj2_impulsive_torque = obj2_center_to_col_point_vec * -impulse
obj2_center_to_col_point_vec = collisionPoint - obj2_pos
obj2_orientation_matrix=subMatrix(eulerXYZToRotationMatrix(obj2_orientation),3,3)
obj2_inverted_inertia_tensor_world_coord = obj2_orientation_matrix *
    squareMatrixInverse(obj2_inertia_tensor) *
    squareMatrixInverse(obj2_orientation_matrix)

obj1_total_velocity = obj1_v_before+(obj1_a_v_before*obj1_center_to_col_point_vec)
obj2_total_velocity = obj2_v_before+(obj2_a_v_before*obj2_center_to_col_point_vec)
```

```
relative_velocity = obj1_total_velocity - obj2_total_velocity
relative_velocity_col_coord = squareMatrixInverse(collision_basis)
                            * relative_velocity
collision_basis = arbitraryBasis(obj1_collisionNormal)
desired_change_relative_velocity = [ desired_change_relative_velocity_x,
                                     -relative_velocity_col_coord # 2  ,
                                     -relative_velocity_col_coord # 3  ]
desired_change_relative_velocity_x =
    -relative_velocity_col_coord # 1 * (1 + restitution_coefficient)

obj1_center_to_col_point_skew_sym=skewSymmetricMatrix(obj1_center_to_col_point_vec)
obj1_impulsive_torque_unit_impulse = obj1_center_to_col_point_skew_sym
                                     * collision_basis
obj1_rot_induced_i_l_v_by_unit_impulse =
    - (obj1_inverted_inertia_tensor_world_coord*obj1_impulsive_torque_unit_impulse)
    * obj1_center_to_col_point_skew_sym
obj1_linear_velocity_by_unit_impulse = identityMatrix(3) * (obj1_mass ^ -1)
obj2_center_to_col_point_skew_sym=skewSymmetricMatrix(obj2_center_to_col_point_vec)
obj2_impulsive_torque_unit_impulse = obj2_center_to_col_point_skew_sym
                                     * collision_basis
obj2_rot_induced_i_l_v_by_unit_impulse =
    - (obj2_inverted_inertia_tensor_world_coord*obj2_impulsive_torque_unit_impulse)
    * obj2_center_to_col_point_skew_sym
obj2_linear_velocity_by_unit_impulse = identityMatrix(3) * (obj2_mass ^ -1)
vec_change_unit_impulse = obj1_linear_velocity_by_unit_impulse
    + obj2_linear_velocity_by_unit_impulse
    + squareMatrixInverse(collision_basis)*obj1_rot_induced_i_l_v_by_unit_impulse
    + squareMatrixInverse(collision_basis)*obj2_rot_induced_i_l_v_by_unit_impulse

desired_impulse_col_coord = squareMatrixInverse(vec_change_unit_impulse)
                          * desired_change_relative_velocity

    }
    applicability = {
        mandatory 2 dynamic objects
    }
    aliases = {

    use obj1_collisionNormal as attribute collisionNormal of object with index 0
    use obj1_mass as attribute mass of object with index 0
    use obj2_mass as attribute mass of object with index 1
    use obj1_pos as attribute pos of object with index 0
    use obj2_pos as attribute pos of object with index 1
```

```
        use obj1_orientation as attribute orientation of object with index 0
        use obj2_orientation as attribute orientation of object with index 1
        use obj1_inertia_tensor as attribute inertia_tensor of object with index 0
        use obj2_inertia_tensor as attribute inertia_tensor of object with index 1
        use obj1_v as attribute v of object with index 0
        use obj2_v as attribute v of object with index 1
        use obj1_v_before as attribute v_before of object with index 0
        use obj2_v_before as attribute v_before of object with index 1
        use obj1_a_v as attribute a_v of object with index 0
        use obj2_a_v as attribute a_v of object with index 1
        use obj1_a_v_before as attribute a_v_before of object with index 0
        use obj2_a_v_before as attribute a_v_before of object with index 1

        }
    }
```

5.3 THE CONTACT CONTEXT

As we mentioned in section 3.2, contrary to a collision context, a contact context is continuously applied. This is due to the fact that it is intended that the physics equations introduced by the user translates mathematically the inherent concept of contact as teached in physics courses. This is: an equilibrium of forces in a particular direction during a certain amount of time. However, in practise, this concept is not used by the vast majority of the physics engine, such as the well known Catto and Coumans. These engines follow an impulse-based approach for both collisions and contacts. In fact, for these engines the contact is resolved with micro-impulses guided by the same equations we used in 5.2. At every frame, these micro-impulses are sequentially applied on the necessary pair of objects until there are no left objects in penetration. The fact is the majority of these engines follows this approach because the physics of impulses is much more simpler than the physics of forces. A more complete explanation about this topic can be found in Millington (2007). However, since this approach does not translates the real notion of contact, it has also its disadvantages. In fact, in some situations, it is known that the objects can appear to vibrate. Due to the high mathematical complexity of the physics for generic contacts considering a force-based approach, it would not be conceivable to present these concepts here since our application targets students of high school or college frequenting introductory courses of dynamic of rigid bodies. In fact, we consider that even the generic physics of collision between a simple pair of object following an impulse approach presented in 5.2 already requires the student to possess an advanced mastery of mathematics and physics.

In this section we build a contact context that follows the laws of physics but can only be applied in some specific scenarios. The scenarios supported have mainly the same complex-

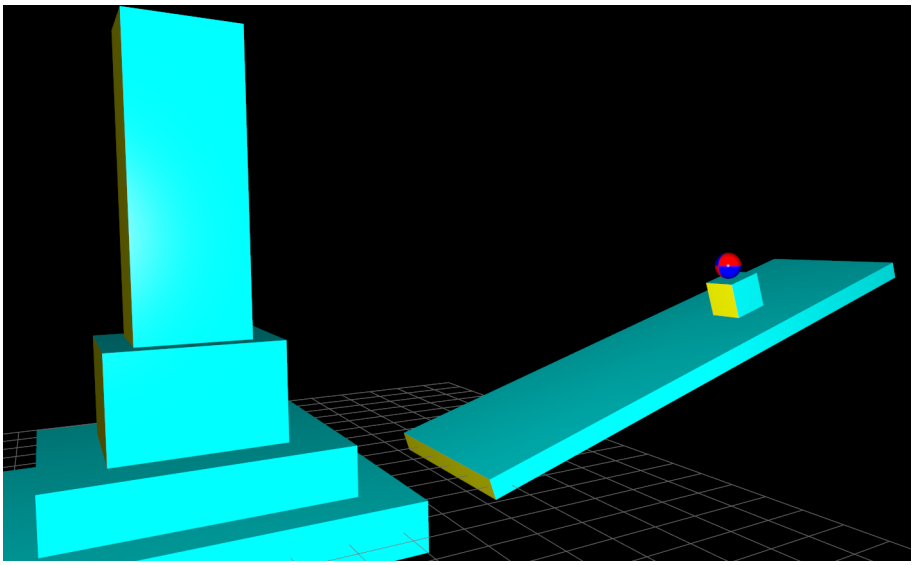


Figure 26.: A scenario composed by two different contact context instances whose complexity is supported by the contact context built in this section

ity of the ones the student faces in class, such as, one or more cuboids stacked going down a ramp (represented by a flat inclined cuboid) or a pyramid of cuboids in contact. Figure 26 shows a possible scenario supported by the contact context we build in this section. In this case, the engine considers two different contact context instances: one instance composed by the ramp, the cuboid and the sphere and one instance composed by the cuboids of the pyramid.

Looking at figure 27 gives us an idea of the equilibrium of forces that exist during a prolonged contact. The forces involved are the gravitational force and the normal reaction force. Since we are considering the gravitational acceleration to have the direction of the vertical axis Z, it will be helpful, as we will see, to request the engine to compute the geometrical predefined context attributes we mentioned in section 3.3 relatively to directions $[0,0,1]$ and $[0,0,-1]$. With that said the first version of our contact context looks as follow:

```
context {
    name = my_contact_context
    type = contact
    geometric_data_directions = {[0,0,1], [0,0,-1]}
}
```

The parallelogram law tells us that the resulting force is the vector sum of the forces involved. Therefore, we can start by defining the following VFLang function:

```
Fr = Fg + Nr
```

By Newton's second law, the gravitational force is simply the product between the total mass and the gravitational acceleration a we defined previously as a world attribute:

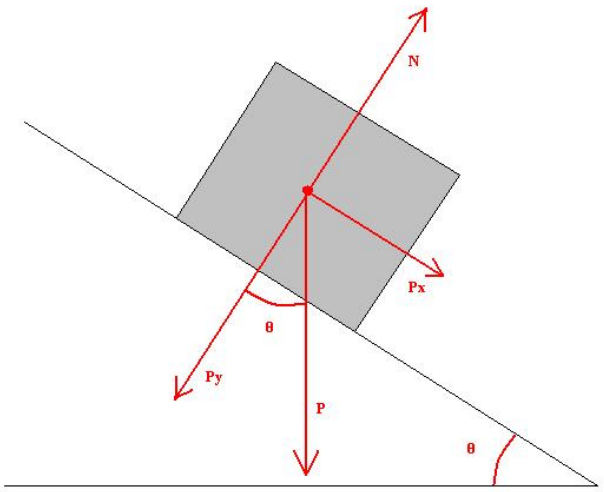


Figure 27.: The forces and its components involved in a contact between two objects

```
Fg = total_mass * a
```

where *total_mass* is given by:

```
total_mass = mass + massSumDirVec1
```

*massSumDirVec*1 is one of these geometrical predefined context attributes computed by the engine. Remember that each object involved in the context has its own instance. The family of this attribute is given by the following syntax: $[objectAttributeName]SumDirVec-[directionNumber]$. Remember from section 3.3 that this family of predefined context attributes results on the summation of the given attribute considering all the objects that form a chain of contact starting at the object and considering also the direction selected by the user. The system assigns an index to a direction depending on its position in field *geometric_data_directions* defined in the context specification. Therefore, *massSumDirVec*1 is the summation of the object attribute *mass* in the direction $[0, 0, 1]$ and *massSumDirVec*2 is the summation of the object attribute *mass* in the direction $[0, 0, -1]$.

To define the normal reaction force one important detail needs to be taken into consideration. In fact, since the engine has not any notion of physics, to the engine a contact context instance does not involve necessarily a static object. Internally, the condition the engine checks is simply if the group of objects stick or not all together over time. However, we know, that in physics the notion of contact, involves necessarily a static object. With that said, we must check using another predefined geometrical context attribute if a path of objects in contact, starting at the object, leads to a static object. For this purpose, we use the predefined context attribute *hasPathToStaticObjectDirVec*2. Looking back at figure 26 we can understand the importance of using *hasPathToStaticObjectDirVec*2 considering the pyramid. In fact, if the base of the pyramid is a static object we want an effective reaction

normal force to be transmitted to each object of the pyramid to counter each gravitational force. On the other hand, if the base of the pyramid is also a dynamic object, we simply want to observe the pyramid falling as a whole due to the solo action of the gravitational acceleration.

As shown in figure 27 the normal reaction force depends on the contact normal vector. As we mentioned in section 3.3, given a direction, the engine provides the contact normal vectors between the objects and the others that are in direct contact with it. In our case, considering the pyramid, each object in the middle is in contact with one on top and one on bellow. However, to calculate the normal reaction force we just need the normal vector of the contact between the object and the one bellow. Therefore, to access the desired contact normal vector direction $[0, 0, -1]$ must be used. Since in theory, considering a particular direction, an object can be in direct contact with one or more, the predefined context attribute *collisionNormalsDirVec*2 provides all the possible vectors in the form of a matrix. In our case, we just select the first vector that is represented as the first row of the matrix. With all that said the normal reaction force Nr can be defined as follow:

```
Nr = if hasPathToStaticObjectDirVec2 then Nr_norm * normal_vec else [0,0,0]
Nr_norm = Fg . -normal_vec
normal_vec = collisionNormalsDirVec2 # 1
```

By Newton's second law the resulting acceleration can be calculated as follow:

```
a_final = Fr/total_mass
```

The same kinematic equations we used to build our free fall context can be used to describe the motion of the object when subjected to a constant acceleration in any particular direction. Adding the necessary VFLang functions and joining the functions we define in this section, we reach the final version of our contact context:

```
context {
    name = my_contact_context
    type = contact
    geometric_data_directions = {[0,0,1], [0,0,-1]}
    functions = {
        pos0 = position
        t0 = t
        v0 = v
        Fr = Fg + Nr
        Fg = total_mass * a
        total_mass = mass + massSumDirVec1
        Nr = if hasPathToStaticObjectDirVec2
             then Nr_norm * normal_vec else [0,0,0]
```

```
        Nr_norm = Fg . -normal_vec
        normal_vec = collisionNormalsDirVec2 # 1
        a_final = Fr/total_mass
        position = pos0 + v0*(t-t0) + 0.5*a_final*(t-t0)^2
        v = v0 + a_final*(t-t0)
    }
    attributes = {t0, v0, pos0}
    compute_first_time_only_attributes = {t0, v0, pos0}
}
```

6

THE VIRTUAFIZ ENVIRONMENT

In this chapter we present some of the interesting pedagogical features that integrate the VirtuaFiz environment. As we will see this features are visualised through the VirtuaFiz interactive graphical interface. In section 6.1 we present the notion of a VirtuaFiz event. How it is possible to trace all the variables introduced in the system at the time of a specific VirtuaFiz event is also demonstrate in this section. In section 6.2 it is shown how, in VirtuaFiz, the trajectories of the objects and the attribute that are vectors can be visualised. VirtuaFiz allows the generation of plots of any variable introduced by the user. How the plots can be visualised through the VirtuaFiz interface is shown in section 6.3. Finally, in section 6.4, a brief description of the VFLang interactive interpreter is presented.

6.1 TRACING OF VARIABLES

During the computation of a simulation, VirtuaFiz identifies all the relevant events. An event occurs at a specific moment of time and has one of the three following types:

- The beginning of a continuous free fall context application
- The beginning of a continuous contact context application
- A single collision context application

Using the graphical interface, the user have access to a list of all the events that occurs in a specific simulation. By selecting a specific event, the user have access to the tracing of all the world attribute and also all the object and context attributes associated to the objects that participate in the event. For instance, considering an event of type *a single collision context application*, three tables are displayed. One contains the values of the world attributes before and after the event occurs. Also, a table for each of the two objects involved in the event are displayed containing the values of the objects and contexts attributes before and after the event.

The information provided by VirtuaFiz regarding a specific *a single collision context application* involving a cuboid and a sphere is shown in figure 28. This event is from a

simulation created using the contexts defined in chapter 5. In the tables of figure 28 we can see that no world attribute is not modified by the occurrence of the event. The same happens for some object attributes such as the position or the size of the object. This is what we would expect since in the collision context we defined in section 5.2 the only attributes modified by the context are the object attributes related to the linear and angular velocity of the objects involved.

As we can see from figure 28, before selecting the desired event from the list, the user has the possibility to filter the events that appears on the list. This can be done by selecting the type of the event, the interval of time when the event must occur or also the objects that must participate in the event.

6.2 Visualisation of trajectories and vectors

The user can visualise the trajectory followed by each object since the beginning of the simulation until the actual time. Using the interface, the user can set this option on or off at any moment. Using the contexts defined in 5, figure 29 shows a capture of the graphical simulation presented by VirtuaFiz at a specific time. As we can see from figure 29 the display of the trajectories are a great help to understand what happens in a simulation. Without them, by looking at the same figure, it would not be possible to understand where each object come from and the collisions that occurs between the object and the others.

VirtuaFiz allows that all the attributes that are tridimensional vectors can graphically be represented at the time of a specific event. In order to differentiate graphically the vector representing the value of the attribute before the event from the vector representing the value of attribute after the event, the user can choose a colour to each vector. By default, a vector representing a world attribute is applied on the origin. However, if desired, the user can specify any point he would apply to see the vector applied on. Considering an object attribute, three options are proposed regarding the point of application of the vectors. The vectors can be applied on the center of the object, the collision point (if the event is *a single collision context application*) or a custom point introduced by the user. In most cases, the vectors are applied on the object center but in some specific cases, such as the position vector, the vector must be applied on a custom point so its visualisation can makes sense. In this case, the custom point would be the origin of the referential.

Considering the same scenario pictured in figure 29 and considering also the event triggered by the second collision suffered by the small sphere, figure 30 shows a capture of the VirtuaFiz window.

As we can see from figure 30, some objects are painted using a grey scale while others are represented using coloured line segments. This allows to quickly differentiate graphically the objects that are involved in the event from the ones that are not. The objects involved

Figure 28.: The interface displayed by VirtuaFiz containing all the relevant information about a specific event

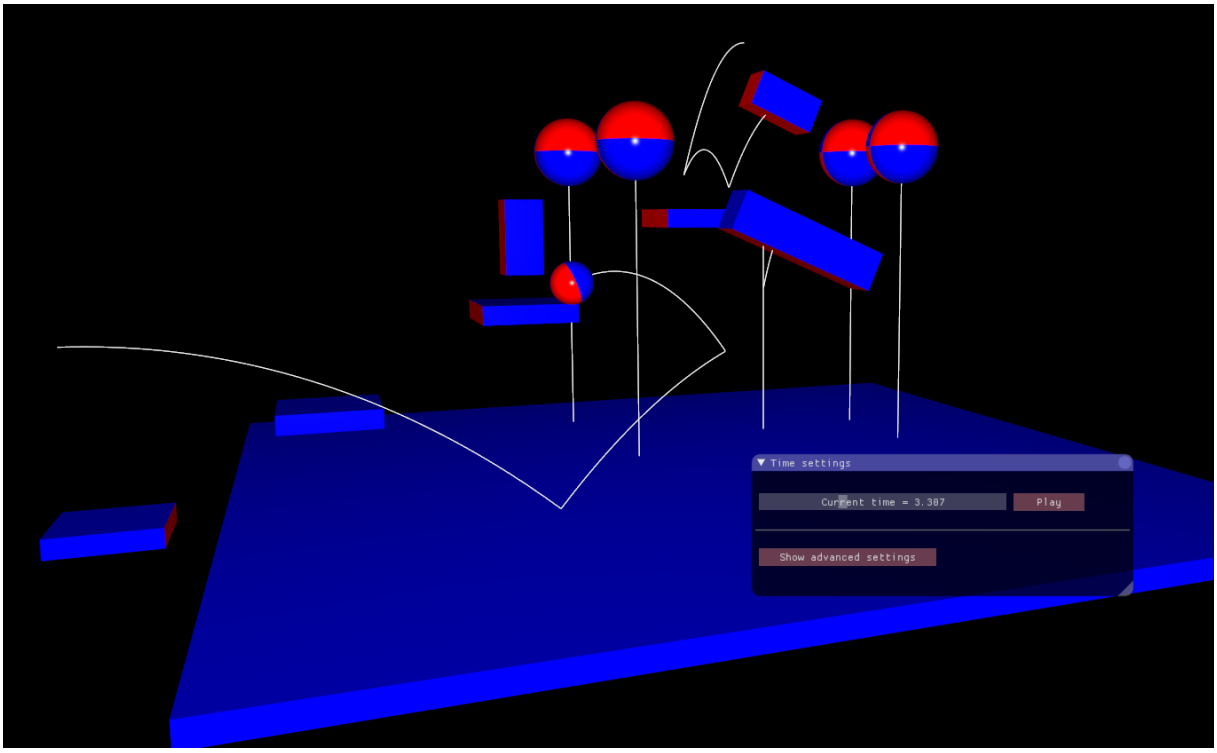


Figure 29.: A possibility of trajectories followed by the objects in a simulation

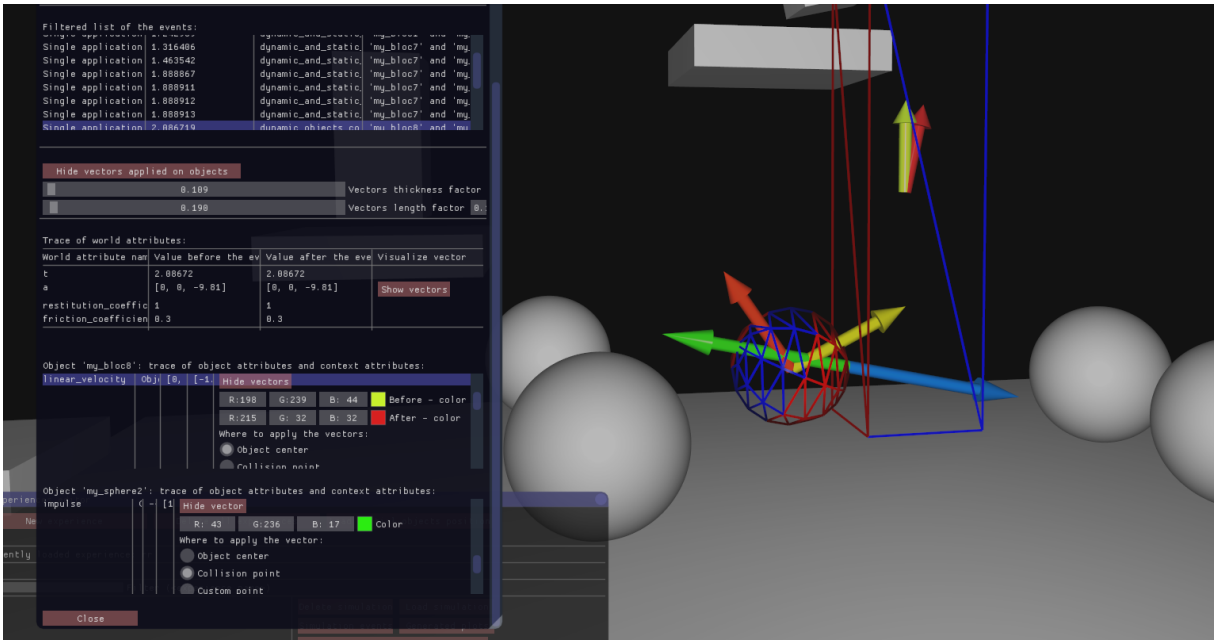


Figure 30.: The visualisation of vectors applied on objects at the time of an event

in the event are only represented using coloured line segments when the user request to visualise vectors. That was the case in the figure. Otherwise, they are represented just as in figure 29. This wire effect allows to visualise better the point where the vector is applied. In figure 30, we can see two sliders that allow to control the thickness and the relative length of all the vectors. This is fundamental considering that the system has no idea about the scale used conceptually by the user. In this simulation, to make the graphical visualisation of the vectors more appealing and adjusted to the scale used, all the vectors have only 19% of their original size.

In the graphical interface, it is specified that the vectors representing the value, before and after the event, of object attribute *linear_velocity* must be drawn in yellow and red, respectively. Also, as it is supposed to be, the vectors are applied on the center of each respective object. Considering context attribute *impulse*, we see in the interface that only one vector can be drawn. This is what always happen considering context attributes. In fact, by definition, the value of a context attribute is unknown before the context is applied on the object. Due to the physical meaning associated to the impulse vector, it was requested, as intended, that *impulse* must be applied on the collision point. In the contexts presented in chapter 5, we specified that the impulses applied on the two objects involved in the collision context instance should obey the impulse version of the Newton's third law. This means, in a collision between two objects, that an impulse applied applied on an object, induces necessarily the application of an impulse with the same norm but opposite direction on the second object. This is precisely what we can observe in the figure through the green and blue vectors that represents, respectively, the impulse suffered by the sphere and the impulse suffered by the cuboid.

6.3 VISUALISATION OF PLOTS

VirtuaFiz allows the user to request the generation of the plot of any variable entered in the system. The plot of a variable shows how the value of a variable changes during the interval of time associated to the simulation. The possibility of how a variable is plotted depends on its type. In fact, for a vector, the system asks the user how the variable should be plotted. The options are to plot the norm of the vector or to plot separately each component of the vector. For a boolean attribute, its value is converted to a real. *true* is represented by 1 while $false$ is represented by 0. Notice that an attribute that is a matrix cannot be plotted.

Figures 31 and 32 shows a capture of two different simulations where for each we can see the configuration of the objects at a given time as well a plot for a given attribute present in the simulation. Both simulations use the contexts defined in chapter 5.

The simulation pictured in 31 contains two objects: a dynamic sphere without initial velocity that bounces a flat static cuboid. Figure 31 shows the plot of the third component

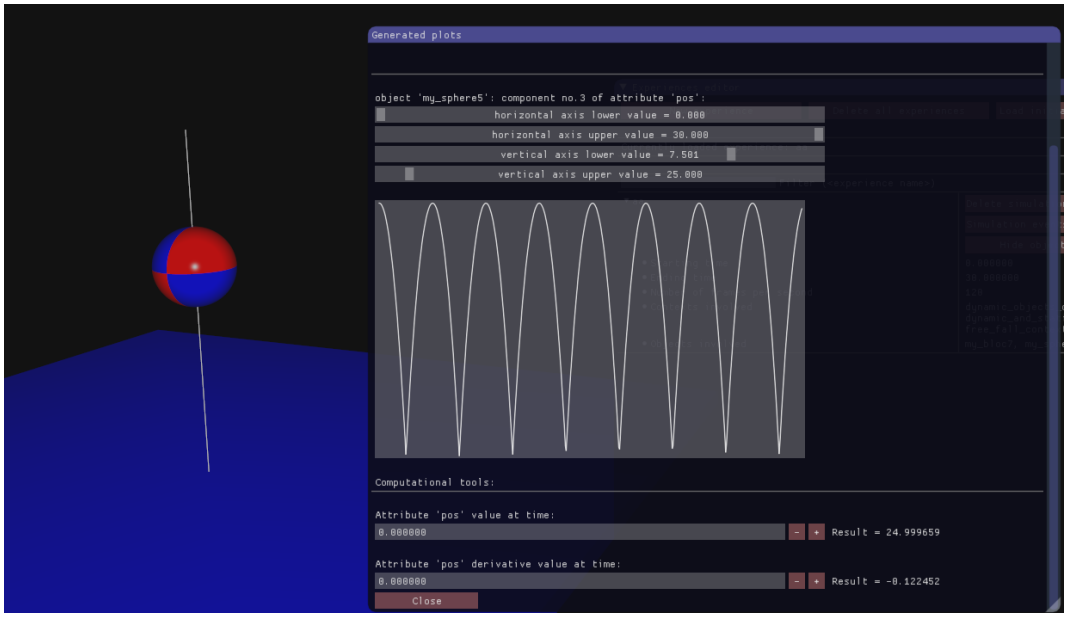


Figure 31.: A plot showing coordinate z of a dynamic sphere changing over time in a simulation composed by the sphere and a static cuboid

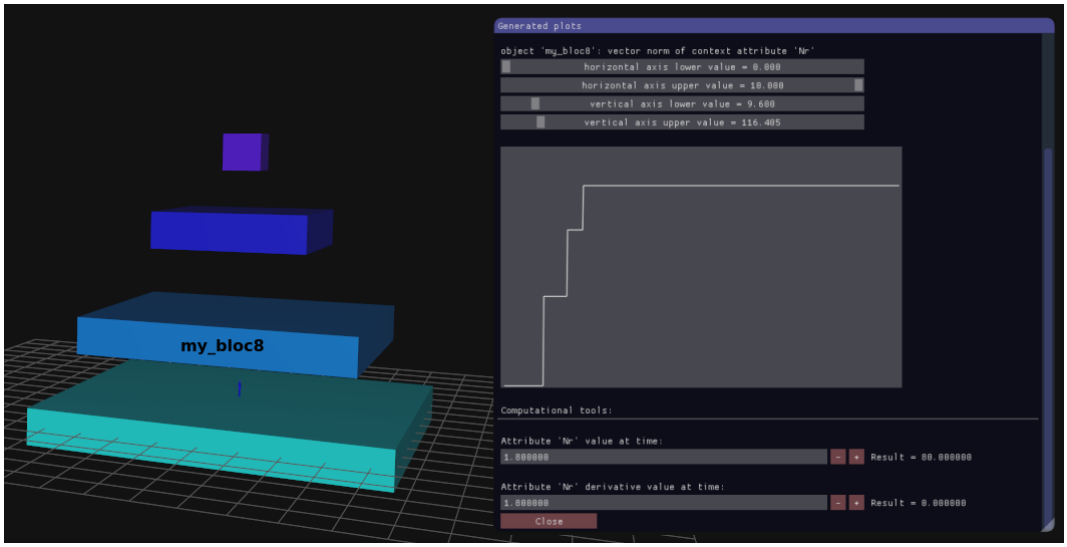


Figure 32.: A plot showing the variation over time of the normal reaction force's norm of an object in a simulation composed by 3 dynamic and 1 static cuboids

(coordinate z) of the position vector of the sphere. The observed parabolas in the plot are what we would expect from the kinematic equation introduced for the position in the free fall context defined in section 5.1. Also, looking at the plot, we can deduce that the restitution coefficient in this simulation is 1 since the sphere, after each collision, reaches again its original position. As we can see from the figure, the user can ask for the specific value of the attribute at a specific moment of time. It is also possible to ask for the derivative at that time.

Relatively to the simulation pictured in figure 32, it consists on 3 dynamic cuboids and 1 static cuboid (the one in the grid). In this simulation, the restitution coefficient was deliberately set to 0, so after the collision, each cuboid sticks with the other. The plot shown in the figure is relative to the norm of the context attribute Nr (the normal reaction force defined in the contact context) for object my_bloc8. The curve represented in the plot is also what we would expect in this case. Before my_bloc8 collides with the static cuboid, the normal reaction force the static cuboid transmits to my_bloc8 does not exits. After, the collision, the normal reaction force is due only by my_bloc8's own mass and the gravitational acceleration. Then, when each of the two others cuboids collides with my_bloc8, the normal reaction force exercised by the static cuboid augments in proportion to the mass of the two others cuboids. Looking at the change in the normal reaction force value over time, we can deduce that, in this case, each cuboid has a greater mass than the one above.

6.4 THE VFLANG INTERPRETER

VirtuaFiz can also be used as an auxiliary tool when solving physics exercises on "paper". In fact, VirtuaFiz provides an interpreter for the VFLang programming language. Using the interpreter, the user can define its own functions. To compute the result of a particular valid VFLang expression, operator $ must be place before the expression. Using the interpreter the user must understand that the concepts of VirtuaFiz context and attribute does not exist in that environment: everything is a function. The functions declared using the interpreter cannot be accessed in the contexts. The same happens in the opposite direction.

Figure 33 shows how the interpreter can be used to calculate, using the equations presented in 4.2, the collision response between two spheres. Considering the intended pedagogical component, VirtuaFiz has an advantage over many other interpreted languages. In fact, in the same session, the user can redefine the same function the number of times intended. This allows to test many configurations. For example, considering the example shown in figure 33, after the last command introduced:

```
$second_object_v
```

we can redefine, for example, function $first\_object\_vAtCol$ to return a different vector:

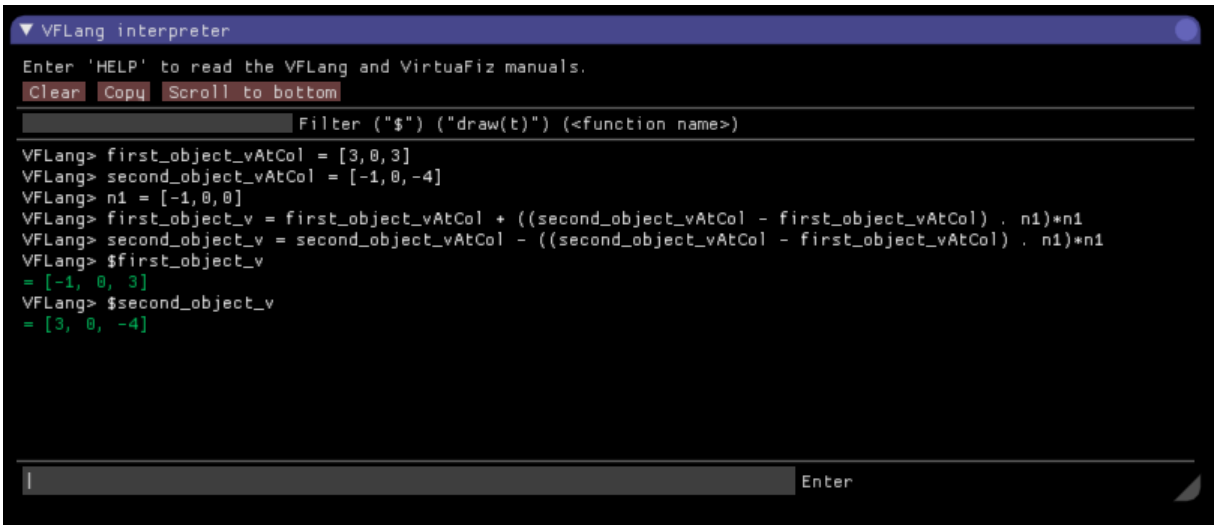


Figure 33.: The calculation of the collision response between two spheres using the VFLang interpreter

```
$first_object_v_AtCol = [6,0,6]
```

and then after introducing the command:

```
$second_object_v
```

we see a different result from the one shown in figure 33. Notice that after redefining a function it is not necessary to introduce (recompile) again all the functions that depend on the function redefined. In this case, we did not reintroduce *first_object_v* and *second_object_v* to calculate properly the new velocity after the collision of the second object. In fact, the VFLang virtual machine dynamically calls the proper function in runtime.

To allow the user to define and manipulate more complex functions that use a greater amount of data, sequences are supported using the interpreter. The elements of the sequence must be one of the types presented in 4.1. The following program shows how a sequence of real numbers can be the sorted in VFLang following the quick sort algorithm strategy:

```
quickSort(seq) = if seq=={} then {}
                 else quickSort(lesser(tail(seq), head(seq)))
                      ++ {head(seq)}
                      ++ quickSort(greater(tail(seq), head(seq)))

lesser(seq, num) =  if seq=={} then {}
                    else if head(seq)<num then
                            {head(seq)} ++ lesser(tail(seq), num)
                         else lesser(tail(seq), num)
```

```
greater(seq, num) =  if seq=={} then {}
                     else if head(seq)>=num then
                             {head(seq)} ++ greater(tail(seq), num)
                          else greater(tail(seq), num)
```

As we can understand form the code above, a sequence is constructed using curly brackets. Operator $++$ concatenates two sequences. *head* and *tail* are two VFlang natives functions. *head* returns the first element of the sequence while *tail* returns a sequence without the first element.

The program presented above could be in a file called $my\_quick\_sort.vfl$ and it would be simply imported into the system by introducing the following command in the interpreter:

```
import "my_quick_sort.vfl"
```

Analogously, the functions can be exported to a file using the command:

```
export "my_file.vfl"
```

The VFLang interpreter reports to the user all the parsing and runtime errors that might occur with all the relevant information. For example, the attempt to introduce in the interpreter:

```
my_func(x,y) = ()x+y
```

gives the following message:

```
:-( Interpreter [1:17] Parsing Error. In function 'my_func': parse error on input ')'.
```

After having introduced properly my_func, the evaluation of an expression given by:

```
$my_func(true,50)
```

throws a runtime error and the following message is displayed:

```
:-( Interpreter [2:17] Runtime Error. In function 'my_func': function '+' called with
invalid/incompatible argument(s).
```

7

CONCLUSION

The motivation that led us to carry out this research is the fact we realised at that time no physics virtual laboratories that leaves all the mathematical declaration of the laws of physics to the user exists. Having in mind making possible in practise what we consider to be a new teaching paradigm, we developed a virtual laboratory called VirtuaFiz. As a proof of concept, we focus this research, as well VirtuaFiz, on a particular subject of Classical Mechanics: the dynamic rigid bodies. Our approach is based on the fact that, considering the dynamic of rigid bodies, the objects involved in an experience are in a particular state at any moment. Physically, each state consists on different physics equations. Throughout this document, following our approach, we demonstrate all the specificities needed to be taken into consideration to make possible the declaration, through equations, of the underlying physics of each intended behaviour desired to observed. Our application can be seen as a black box responsible to determine which physics equations introduced by the user are necessary at a given moment of time. This is made possible by the fact that the engine behind our application has a strong knowledge in geometry. Beyond the visualisation of the experiences driven by the user physical equations, VirtuaFiz allows the processing of all the numerical information in different forms, such as plots, visualisation of vectors applied on objects or even the tracing of the value of the involved variables at the time of a specific event. We believe that these additional features and the fact that all the physical behaviours visualised in an experience depend solely on the user equations, make VirtuaFiz a complete tool and specially oriented toward the teaching of physics.

7.1 FUTURE WORK

By looking at all the developed work and the observed results, we consider that all the objectives we set ourselves at the beginning of this dissertation were reached. However, as it could not be, there is always the possibility for improvements and exploration. Considering the time necessary to compute an experience it is suitable but not as low as we would like to be when we compare VirtuaFiz with well known physics engines. In fact, although, care was taken when developing VFLang, there is still the possibility for a lot of optimisations

in the byte-code generated by the compiler and also, how and in which other the byte-code is executed by the virtual machine. Out-of-order execution and byte-code instructions reordering can be definitely two ways to explore.

Relatively to the objects in VirtuaFiz, the support for objects different than spheres and cuboids can be easily accomplish. Most of the work would pass to introduce this possibility in the collision detector developed. Also, the generic contexts using real physics we presented can be used as they are for any kind of object. To improve VirtuaFiz, it can also be interesting to develop the necessary mechanisms that would make possible that, for example, two specifics objects acts as one (as they would be glued together). This would allow for example, to define complex objects using, for example, only cuboids. Following this approach, it would be necessary to define the inertia tensor of the complex object combining somehow the inertia tensor of each simple object that is part of the complex object.

Relatively to the collisions in VirtuaFiz, it would definitely interesting to investigate how it is possible to remove the simplification about the collisions being instantaneous and the objects being always rigid. Therefore, the user would introduce the necessary physics equations to resolve the deformation of the objects. Then, the user would visualise, the deformations occurring on the objects during the whole interval of time associated to the collision.

We also consider that more effort can be done to improve the user experience. Test with users would definitely give us the right feedback about the points that need to be improved relatively to the interface and also to study if the use of VirtuaFiz can improve a student success in Physics.

Of course, there is also always the possibility to investigate, how, what we consider to be a new teaching paradigm in physics, would be implemented in others areas of physics, such as electromagnetism and hydrodynamic. In fact, it is not guaranteed that the context approach we presented and that works well in this case, can be implemented or would be the best solution in others areas of physics.

Appendices

A

SOME USEFUL VFLANG NATIVE FUNCTIONS

In this appendix, some useful VFLang native functions are presented. Notice that the library of native functions has more functions than those presented here. Among the functions not presented, the library contains 6 trigonometrical functions *inverseTangent*, *inverseSine*, *inverseCosine*, *sine*, *cosine*, *tangent* that can be used to perform all kind of trigonometrical calculations. Also, to get the number of dimension, normalize or calculate the norm of a vector with any dimension, native functions *vectorDimension*, *normalize* and *norm* can be used, respectively.

VFLangNative.arbitraryBasis

Build an orthogonal basis

Arguments

⟨ axisX ⟩

AXISX The 3-dimensional vector used as axis X of the orthogonal basis to be created

Returns

The 3-by-3 matrix representing the orthogonal basis built

VFLangNative.skewSymmetricMatrix

Build the skew symmetric matrix of a vector

Arguments

⟨ vec ⟩

VEC The 3-dimensional vector for which the skew symmetric matrix is calculated

Returns

The 3-by-3 skew symmetric matrix of the vector

VFLangNative.multiplyQuaternions

Calculate the product of two quaternions

Arguments

⟨ qua1 , qua2 ⟩

QUA1 The 4-dimensional vector representing the quaternion on the left side of the product

QUA2 The 4-dimensional vector representing the quaternion on the right side of the product

Returns

The 4-dimensional vector representing a quaternion given by the product of the two quaternions

VFLangNative.quaternionToEulerXYZ

Convert an orientation from a quaternion to a Euler XYZ representation

Arguments

⟨ qua ⟩

QUA The 4-dimensional vector representing the orientation expressed in the form of a quaternion

Returns

The 3-dimensional vector representing the orientation expressed in an Euler XYZ representation

VFLangNative.quaternionToRotationMatrix

Convert an orientation from a quaternion to a rotation matrix

Arguments

⟨ qua ⟩

QUA The 4-dimensional vector representing the orientation expressed in the form of a quaternion

Returns

The 4-by-4 rotation matrix representing the orientation

VFLangNative.eulerXYZToRotationMatrix

Convert an orientation from a euler XYZ representation to a rotation matrix

Arguments

⟨ eulerXYZ ⟩

EULERXYZ The 3-dimensional vector representing the orientation expressed in an Euler XYZ representation

Returns

The 4-by-4 rotation matrix representing the orientation

VFLangNative.eulerXYZToQuaternion

Convert an orientation from a euler XYZ representation to a quaternion

Arguments

⟨ eulerXYZ ⟩

EULERXYZ The 3-dimensional vector representing the orientation expressed in an Euler XYZ representation

Returns

The 4-dimensional vector representing the orientation expressed in the form of a quaternion

VFLangNative.rotationMatrixToQuaternion

Convert an orientation from a rotation matrix to a quaternion

Arguments

⟨ rotMat ⟩

ROTMAT The 4-by-4 rotation matrix representing the orientation

Returns

The 4-dimensional vector representing the orientation expressed in the form of a quaternion

VFLangNative.rotationMatrixToEulerXYZ

Convert an orientation from a rotation matrix to an Euler XYZ representation

Arguments

⟨ rotMat ⟩

ROTMAT The 4-by-4 rotation matrix representing the orientation

Returns

The 3-dimensional vector representing the orientation expressed in an Euler XYZ representation

VFLangNative.rotateVector

Rotate a 3-dimensional vector

Arguments

⟨ vec, axis, angle ⟩

VEC The 3-dimensional vector to be rotated

AXIS The rotation axis

ANGLE The angle of rotation expressed in radians

Returns

The 3-dimensional vector rotated accordingly

VFLangNative.reflectVector

Reflect a 3-dimensional vector

Arguments

⟨ vec, surfaceNormal ⟩

VEC The 3-dimensional vector to be reflected

SURFACENORMAL The 3-dimensional vector representing the normal vector of the surface in which the vector is reflected

Returns

The 3-dimensional vector reflected accordingly

VFLangNative.squareMatrixInverse

Compute the inverse of a square matrix

Arguments

⟨ mat ⟩

MAT The square matrix to be inverted

Returns

The square matrix inverted

VFLangNative.matrixTranspose

Compute the transpose of a matrix with any dimensions

Arguments

⟨ mat ⟩

MAT The matrix to be transposed

Returns

The matrix transposed

VFLangNative.subMatrix

Build a submatrix of a matrix

Arguments

⟨ mat, numRows, numCols ⟩

MAT The matrix to be reduced in its dimensions

NUMROWS The first n rows to keep for the new matrix to be constructed

NUMCOLS The first n columns to keep for the new matrix to be constructed

Returns

The submatrix built accordingly

VFLangNative.matrixDimension

Returns the number of rows and the number of columns of a matrix

Arguments

⟨ mat ⟩

MAT The matrix whose dimensions is wanted to be known

Returns

A two dimensional vector containing the number of rows and the number of columns of the matrix

BIBLIOGRAPHY

Erin Catto. Box2D, A 2D Physics Engine for Games. http://box2d.org/. Accessed: 2017-09-30.

Central Connecticut State University. Virtual Physics Labs. http://www.physics.ccsu.edu/LEMAIRE/genphys/virtual_physics_labs.htm. Accessed: 2017-09-30.

Erwin Coumans. Bullet Real-TIme Physics Simulation. https://pybullet.org/wordpress/. Accessed: 2017-10-20.

Design Simulation Tecnologies. Interactive Physics. http://www.design-simulation.com/IP/index.php. Accessed: 2018-11-12.

Chuck Duncan. KET Virtual Physics Labs. https://virtuallabs.ket.org/physics/. Accessed: 2017-09-30.

MathWorks Inc. MATLAB MAtrix LABoratory. https://www.mathworks.com/products/matlab.html. Accessed: 2018-01-11.

Ian Millington. *Game physics engine development*. Morgan Kaufmann Publishers, 1 edition, 2007. ISBN 9780123694713.

Erik Neumann. My Physics Lab – Physics Simulation with Java. https://www.myphysicslab.com. Accessed: 2017-09-30.

John Nunn. VP Lab - The Virtual Physical Laboratory. http://vplab.ndo.co.uk. Accessed: 2017-09-30.

University of Colorado Boulder. Phet Interactive Simulations. https://phet.colorado.edu/en/simulations/category/physics. Accessed: 2018-11-12.

Virtual Science Ltd. Virtual Science Experiments. http://www.virtual-science.co.uk/virtualscienceexperiments.html. Accessed: 2017-09-30.

Wolfram Research. Wolfram Mathematica. https://www.wolfram.com/mathematica. Accessed: 2018-01-11.

Dimitris Xanthopoulos. Physion. http://www.physion.net. Accessed: 2018-01-4.