Lightweight Three-Dimensional Visualization For Hybrid Systems Simulation

Yingfu Zeng
Lightweight Three-Dimensional Visualization For Hybrid Systems Simulation

Master Thesis in Embedded and Intelligent Systems

School of Information Science, Computer and Electrical Engineering
Halmstad University
Box 823, S-301 18 Halmstad, Sweden

May 2012
Acknowledgements

First and foremost, I would like to thank my supervisor Professor Walid Taha for his continuous encouragement and support and excellent guidance throughout my study. I would also like to thank all my colleagues in the Effective Modeling Group. I really appreciate the help given to me by Adam Duracz, Jan Duracz, Mingkun Yang. Lastly, I would like to thank my parents for their continuous support. This report is dedicated to them.
Abstract

Simulation plays a key role in the computer-aided design of systems, primarily because it helps in catching design bugs at an early stage and in quickly ruling out many unreasonable options in the design space. There is a wealth of tools and languages that have been proposed over the years for the design and verification of hybrid systems and most of them provide 3D visualization utility for facilitating design process. However, their visualization systems often rely on some external configuration files written in naive 3D APIs like VRML or Java3D and complicated connections have to be established between the simulation data and the visualization. Numerous efforts have to be invested in visualization related modeling and certain level of understanding of the low level 3D libraries are needed. Thus it is not an easy job for non-professions to build a 3D visualization. In this thesis we explore the idea that in a hybrid system model, almost all the information need for visualization is already in place. This means that very few annotation should be enough to fully specify a useful 3D visualization. Based on this idea, we propose a lightweight 3D visualization framework which allows the user to expresses the three-dimensional object in a high level and abstract way and to take advantage of the data already in the simulation. We evaluate our approaching by building a prototype extension of modeling language called Acumen. The result of this thesis work is included in Acumen's official release version that can be download at www.acumen-language.org. To build a 3D visualization in Acumen, only very few annotations need to be added to the simulation model and are included inside one reserved variable. Non-professions like students who are studying Cyber-physical systems can easily use this tool to visualize their models in Acumen without any prerequisite knowledge regarding 3D visualization technology. We also developed a number of benchmarks to test the correctness and functionality of Acumen3D. Besides, as a larger case study into the expressivity of Acumen and its visualization efficiency, we developed a simplified model of a Ping Pong game. We gained experience about how to model complex hybrid system in Acumen. Moreover by comparing modeling such a hybrid system in Java3D and in Acumen3D, we find concrete evidence that modelling it in Acumen3D is easier in that of less code and less prerequisite knowledge and more supporting built in functions.
Contents

ACKNOWLEDGEMENTS ........................................................................................................3
ABSTRACT .................................................................................................................................4
CONTENTS ................................................................................................................................5

1 INTRODUCTION ......................................................................................................................1
1.1 PROBLEM STATEMENT .......................................................................................................1
1.2 RELATED WORK ................................................................................................................2
1.3 THESIS CONTRIBUTION ..................................................................................................2
1.4 OUTLINE OF THE THESIS .................................................................................................3

ACUMEN ....................................................................................................................................4
1.5 OBJECTS AND HYBRID LAWS ........................................................................................4
1.6 CLASS MAIN AND THE SIMULATOR PARAMETER ...........................................................5
1.7 FORMAL CLASS PARAMETERS, PLOT AND TRACE TABLE ..............................................5
1.8 CHILDREN AND LOCALITY REFERENCE IN ACUMEN .....................................................8

2 DESIGN ..................................................................................................................................9
2.1 CHOOSE THE SUITABLE 3D API .....................................................................................9
2.2 VISUALIZATION DATA FORMAT .....................................................................................12
2.3 3D VIEW AND USER INTERFACE ..................................................................................17

3 IMPLEMENTATION .............................................................................................................18
3.1 COLLECTING VISUALIZATION DATA ..............................................................................18
3.2 3D VISUALIZATION .........................................................................................................21
3.3 THE GRAPHICAL USER INTERFACE (GUI) ........................................................................24
3.4 BENCHMARKS AND RESULT ...........................................................................................25
3.4.1 Breaking ball ................................................................................................................25
3.4.2 Pendulum .....................................................................................................................26
3.4.3 Inverted pendulum .......................................................................................................28
3.4.4 Walking robot ...............................................................................................................29
3.4.5 Newton’s cradle ..........................................................................................................31
3.4.6 Three mass system ......................................................................................................33

4 CASE STUDY: A MODEL OF PING PONG ......................................................................36
4.1 INTRODUCE THE PING PONG MODEL ........................................................................36
4.2 MODELLING THE PING PONG GAME IN JAVA3D ..........................................................41
1 Introduction

Recently, there has been an increased interest in cyber-physical systems. These systems typically comprise a computational component, a physical component, and a communications or networking component. Because it is widely anticipated that the most important future advances in computing technology will be in the form of cyber-physical systems, a key challenge for the community is to develop the tools and techniques for accelerating innovation in this area. Simulation plays a key role in the computer-aided design of systems, primarily because it helps in identifying bugs at an early stage and in quickly ruling out many unreasonable options in the design space.

Some work has been done in the area of visualization and animation in connection with hybrid system design applications, both simple 2D animations and more advanced 3D animations. However, much effort has to be invested in the model building and in the non-standard integration into the simulation environment. There is a large potential for using animations for better understanding the performance and correctness of a system, especially for non experts where a simple 3D visualization often will give a more immediate impression than numbers or 2D plots. Our aim has been to make an easy to use yet powerful concept and tool for integrating 3D animations into simulation but with a modest added design effort. This would make it possible to build animation model as a standard part of simulation models, enhancing the understandability of their results.

1.1 Problem Statement

As the scale and complexity of the model increases, the resulting information from the simulation will very likely be vast and indigestible. Extracting the most critical problem from a large amount of information can require a lot of effort and time. 3D-visualization on the other hand, has the merits of being more straightforward and easy to understand, which can greatly facilitate the design of novel hybrid systems. However, most tools and languages for designing hybrid systems rely on external configuration files written in naive 3D APIs like VRML, CAD and Java3D to build the virtual scene for the 3D visualization. Thus it is not easy to work with for non-professionals. Moreover, they often separate visualization system from the simulation or design models so that complicated signal connections has to be established in between. A lot of effort has to be invested in the visualization related modeling and a certain level of understanding of the low level 3D libraries are needed. In this thesis work, the main problem is:

How to integrate 3D-visualization with simulation environment in a way that the user can easily build the 3D animation with very little additional modeling efforts and without a prerequisite knowledge of the low level 3D graphic libraries.

While we were building the extension functionality of Acumen [1,2,3], we found out that the examples which have been made so far in Acumen are all small models. It is important to explore the possibility of modeling more complex hybrid systems in Acumen. At the same time, we need to evaluate our approach by comparing to other methods. The second problem is:

How do we model larger systems in Acumen, and what is the advantages and disadvantages of using Acumen to visualize complex hybrid systems comparing to other languages like Java3D.
1.2 Related Work

The design of hybrid systems is no exception and the most used and popular tools are indeed simulation based. In this domain, there are strong industrial offerings that are widely used: first and foremost, the Simulink/Stateflow toolset that has become the de facto standard in industry for system design capture and analysis [4]. There are also some freely available advanced tools developed in academia that are getting attention from the hybrid system community. HyVisual developed at U.C. Berkeley is one of them.

Simulink's 3D visualization [5] system relies on a central configuration VRML [12] file defining the scene and the input channels for a specific visualization task. The VRML file contains the complete kinematic and degrees of freedom for scene manipulations as well as the CAD Data [6]. In the corresponding Simulink model, the VR Block provides the inputs for the desired degrees of freedom in the scene. This leads to increased configuration efforts if the model has to be changed, because both the VRML configuration and the signals have to be adapted. It also requires the developer to have a solid knowledge of the 3D graphic technology, in this case, the VRML libraries. One strength of our implementation is its complete integration of the visualization into simulation, there is no need for external scene configuration, thus preserving the developer from deepening the underlying 3D libraries. And there is no need for complicated signal connections, the visualization data is inside the simulation model using a very simple syntax representation and is automatically collected to generate the animation.

HyVisual is a hybrid systems modeling framework providing a block diagram visual syntax for specifying continuous dynamics and a bubble-and-arc syntax for specifying modal behaviours [7]. It is based on Ptolemy II and written in Java. Similar to Simulink's 3D visualization system, HyVisual separates visualization from the simulation. It uses a subsystem called GR (graphical) scene to arrange geometry and transform actors in a directed-acyclic-graph to represent the location and orientation of objects in a natural world scene, then the GR scene is converted into a Java3D [8] representation for rendering. Other CT (continuous time) or DT (discrete time) subsystems were used to model the physical characteristics of the system and connect with the GR graph to build the 3D visualization [9]. Acumen3D differs from HyVisual in integrating visualization into simulation and uses high level abstractions like strings to represent the graphical scene instead of naive Java3D codes.

1.3 Thesis contribution

We extended the Acumen language with support for 3D visualization and interaction during the execution of the simulation. To build a 3D visualization in Acumen, only very few annotations need to be added to the simulation model and are included inside one reserved variable. Non-professionals like students who are studying Cyber-physical systems can easily use this tool to visualize their models in Acumen without any prerequisite knowledge regarding 3D visualization technology.

We also conducted a case study of a simplified model of a Ping Pong game to gain experience about modeling complex hybrid systems in Acumen and to test the correctness and effectiveness of the new visualization function. This model is consisted of six classes and each one models a basic elements of a real-world Ping Pong game. For example, there is one Ball class to simulate the physical dynamics of the Ping-Pong ball and other classes like Player, BatActuator, BallActuator, Referee, Game to simulate a concrete Ping-Pong game. We set an example about
how to model complex hybrid systems in Acumen. This game was used in the first edition of a CPS (cyber-physical system) course at Halmstad University in the third period of the 2011/2012 academic year. Students designed their own players and participated in the later three tournaments of this game. Moreover, by comparing modeling such a hybrid system in Java3D and in Acumen3D, we find concrete evidence that modelling it in Acumen3D is easier in that of less code and less prerequisite knowledge and more supporting built in functions.

1.4 Outline of the thesis

- Chapter 2 introduces Acumen
- Chapter 3 introduces the design of Acumen3D and how we expect the user to use it
- Chapter 4 explained the implementation of Acumen3D, design choices and alternatives are presented. In addition, benchmarks and results are presented.
- Chapter 5 introduces the case study: the Ping Pong game in detail and a comparison between implementing the model in Acumen3D and Java3D are presented.
- Chapter 6 Conclusion and Future Work.
Acumen

Systems that evolve over a dense notion of time interact in complex ways that can confound both intuition and analytical methods [10]. This problem is acute for nonlinear differential systems, which include virtually all three-dimensional systems, for which solutions rarely have closed form descriptions. As a result, successful analysis and design of novel cyber-physical systems invariably includes an extensive experimental component that relies either on physical prototypes or on simulation. Today, both types of experimentation can be prohibitively costly and slow. Physical experiments incur material costs and pose challenges in control and reproducibility, measurement and instrumentation, and safety. Simulation has the potential to reduce these problems and to significantly accelerate innovation. But current methods either raise questions about fidelity or are extremely labor intensive. Using mainstream tools means depending on proprietary, black-box codes that come with a fixed set of component models and that offer only limited support for building custom models. Writing simulation codes by hand requires both effort and specialized expertise in mapping the high-level analytical models to executable codes, software implementation (including debugging and testing). These difficulties can be debilitating for novel and creative designers of cyber-physical systems.

Acumen [3, 2, 1] is a modeling language being developed with the goal of reducing the cost of building simulations as well as bridging the gap between several important efforts in modeling and simulation, hybrid systems verification, and synchronous languages.

1.5 Objects and Hybrid Laws

Acumen objects are introduced by defining a class for each kind of object, and then by creating instances of these classes at a particular point in model/simulation time. As an example, consider a device consisting of a battery and discrete controller that decides whether the battery should charge or drive a load. When charging, the voltage on the battery increases at a constant rate until it reaches its full capacity. Then, the battery stops charging and starts driving the load until the voltage is too low. When that happens, the battery switches back to charging. In Acumen, the device is modeled as follows:

```
class Contraption ()
    private v = 0; v' = 0; mode = 0 end
switch mode
    case 0 // Charge (until high)
        if (v < 0.8) v' [=] 1/2
    else mode = 1 end
    case 1 // Drive (until low)
        if (v > 0.2) v' [=] -v
    else mode = 0 end
end
end
```
All variables (such as \( v \), \( v' \), and \( \text{mode} \) in the first class) are implicitly functions of time. Variables introduced in the private section of the class are the state of any objects of this class. The value assigned to each variable is the initial value it has at the instant the object is created. A prime (\('\)) following a variable name denotes its derivative with respect to time. Thus, given an initial value for the variable \( v \), if \( v' \) is defined, then the value of \( v \) will be automatically determined for future values as well. The switch statement allows different sets of rules to govern the behavior of the system under different conditions. The case that applies is determined by the value of the variable \( \text{mode} \). When the value of \( \text{mode} \) is 0, then the first if statement is active. When the value is 1, then the second if statement is active.

The two if statements follow a parallel pattern: their true branch contains a continuous assignment (\( = \)), and the false branch contains a discrete assignment (\( = \)). The main difference between discrete and continuous assignments is that discrete assignments block the progress of logical time, in the sense that simulation cannot advance beyond a particular point in time until all discrete assignments have been performed. Continuous assignments, in contrast, happen continuously, pose no particular constraints on how the simulator advances time.

1.6 Class Main and the simulator parameter

In any Acumen model there must be a declaration for a class named Main. This class always represents the entire world being modeled. Each Main object is required to have a parameter (by convention called the simulator) that allows the user to express how the model should be simulated. Continuing the above example, we can write:

```plaintext
class Main (simulator)
    private mode = "Init" end
    switch mode
        case "Init"
            simulator.timeStep = 0.001;
            simulator.endTime  = 5.0;
            create Contraption ();
            mode = "Persist"
            case "Persist"
        end
    end
end
```

1.7 Formal class parameters, Plot and Trace table

We have already seen how private fields can be useful in endowing objects with state. One feature of such fields is that they must be declared locally in the class definition. Objects can also be endowed with state (and fields) using a slightly different mechanism that allows us to
initialize the value of the fields at the time the object is created rather than at the time the class is defined. This is precisely the purpose of formal class parameters. The following model is a bouncing ball example that uses parameters to allow us to create different instances of the bouncing ball with different initial values.

```plaintext
class BouncingBall (x, x', x'', g)
  x'' = -g;
  if (x < 0)
    x = 0; x' = -x'
  end
end

class Main (simulator)
private state = "Init" end
switch state
  case "Init"
    create BouncingBall (2, 0, 0, 9.8);
    create BouncingBall (10, 0, 0, 9.8);
    state = "Persist"
  case "Persist"
end
end
```

This model simulate an ideal bouncing ball, variable \(x\), \(x'\) and \(x''\) denotes the height, velocity and acceleration of the ball respectively. This model has a continuous equation and a discrete statement when a certain condition is satisfied. The continuous equation assigns the acceleration to always be minus gravity \((-g)\). The discrete statement models a sudden reversal of the direction (and the vertical position) as soon as the height \(x\) is below zero.

By default, simulating a model in Acumen produces a series of normalized plots. They are normalized based on the minimum and maximum values of the variable during the simulation. No scales are explicitly displayed but the user can determine the value of any point on a graph by selecting it with the mouse. The series shows the objects and their variables presented in the order that they are created and for the duration that they exist with vertical scales normalized by the value range. For the model above, the resulting series of plots is as follows:
The first three bands correspond to $x$, $x'$, and $x''$ of the first object, and the second three are for those of the second object. Notice that the $x$ behavior of the two objects is very similar, the difference comes from the different initial position of the two balls. Simulating a model in Acumen also produces a trace table that keeps track of all the values at every time step throughout the simulation. It allows the developer to closely inspect the progress of the model. For the model above, the trace table is as follows:
1.8 Children and Locality reference in Acumen

Every time an object creates a new object, the second object is considered a child of the first. When the new object is created, it is added to the children list of the object that created it. As stated before, every model in Acumen has a Main class, which represents the whole world being simulated and all the other classes are children of the Main class. The parent/child relationship is important in Acumen because it determines who can read and write the fields of an object. In particular, an object can only read and write its own fields as well as those of its children. Thus, even communication between siblings must be explicitly coordinated by the parent.

While this is a strong restriction, it is a deliberate part of the Acumen design, and it is expected to yield benefits. The first is that it is expected to offer significant freedom in modeling precisely how different objects can interact. The second is that, by restricting who can modify the fields of an object, we can expect that the language will be highly amenable to parallel evaluation. The following example and its hierarchical graph shows the parent/child relationship in Acumen.

```plaintext
class Basket()
    private
        m1 = create Ball();
        m2 = create Ball();
    end
end

class Ball()
end

class Main(simulator)
    private
        b1 = create Basket();
        b2 = create Basket();
    end
end
```

![Hierarchical Graph](image)

Figure 2.3: Basket model and its hierarchical graph
2 Design

In a hybrid system model, almost all the information needed for visualization is already in place. It is enough in this setting to add very few annotation to fully specify a useful 3D visualization. For example, in a bouncing ball model, there is very likely a variable denotes the position of the ball. The only annotations left for specifying are the ball's size and color. Thus, our design goal is to allow the developer to use the data already in the model and express the few additional annotations in a very high level and abstract way and the visualization will be automatically generated.

2.1 Choose the suitable 3D API

Over the past few years, with the significant progress in both hardware and software, 3D computer graphics technologies have evolved to a level that makes it possible to use them in a variety of applications. Among those technologies, OpenGL, VRML and Java3D are the most popular and widely used ones. To choose one from them, I will first briefly introduce each one.

OpenGL (Open Graphics Library) is a standard specification defining a cross-language, cross-platform API for writing applications that produce 2D and 3D computer graphics. Since its introduction in 1992, it has become one of most widely used API, bringing thousands of applications to a variety of platforms. But the OpenGL programming is complex and impractical because it is relatively low-level. For example OpenGL does not provide high-level commands for describing models of three-dimensional objects. With OpenGL, one must build a desired model from a small set of geometric primitives—points, lines, and polygons. It is a difficult task for a non-profession person to design a 3D scene with it.

Direct3D is a proprietary API that provides functions to render 3D graphics, and uses hardware acceleration if it is available on the graphics card. It was designed by Microsoft Corporation for use only on the Windows platform. However Direct3D is mainly used in designing 3D games, which requires the programmer to become conversant with C++ language, thus it is difficult for most people.

VRML is a descriptive language of modeling and rendering 3D graphics and is widely used for web-based 3D visualizations. VRML supports easy definition of geometric shapes in hierarchical groupings and it also provides many advanced 3D graphics functions such as lighting models and surface materials. The main shortcoming of VRML is its relatively weak functionality, which make it difficult to cope with dynamic and complex environment.

Java3D is a full-featured 3D graphics API that has been evolving over the past decade. Java3D API is designed as a high-level, platform independent 3D graphics programming API, and is amenable to very high performance implementations across a range of platforms. Java 3D is built with the goals and advantages of Java in mind. A Java 3D applet or application will run
unmodified on computers running different operating systems, using different graphics hardware. It provides a platform-independent mechanism for developing 3D graphics. To optimize rendering, Java3D implementations are layered to take advantage of the native, low-level graphics API available on a given system. Being the extension of Java language in 3D field, Java3D is easy to learn and master, ordinary designer can efficiently develop complex application using it. We chose the 3D API based on three criterias:

1: Platform independence

Platform independence is a very important feature of Acumen and we do want to keep it with the extensional functionality--Acumen3D. An obvious advantage of using Java3D API is that it works on all Java-supporting platforms, and only one version of the code needs to be created, tested, and maintained. There is simply no other way to perform 3D visualization on such a wide variety of platforms, with so litter overhead for both the user and the programmer. Being the extension of Java language, Java3D has the natural advantage of platform independence over other APIs such as Direct3D and OpenGL.

2: Easy to use and master

Java 3D offers higher-level programming constructs when compared to these low-level APIs such as OpenGL and Direct3D. As a result, programmers familiar with Java can create sophisticated 3D applications in a shorter amount of time than they could expect to with lower-level graphics APIs. In other words, Java 3D allows developers to focus on what to draw, not how to draw [8].

3: Compatible with Scala

The current version of Acumen is written upon a programming language called Scala, which runs on the Java VM, and is compatible with Java so we can make full use of existing Java libraries or existing application code [11]. Thus we can import Java3D libraries quite easily and seamlessly. Based on the three reasons above, we choose Java3D API as the underlying 3D library to implement Acumen3D.
However, in order to use the external 3D API Java3D, a scheme for mapping from the source modeling data structure to special Java3D data structure is needed. A basic scene graph in Java3D is as follows:

BG (Branch group) serves as the root of a branch, and Transform Group (TG) is used for translating all the objects which belong to it. Geometry and appearance classes determine the outlook of the object and Java3D provides several primary geometries such as box, cylinder, sphere and cone. Acumen has a similar parent/child structure graph as described in the previous chapter. The scheme we choose is that every Acumen object for visualization will map into a Java3D object along with a transform group (TG) and a branch group (BG) attached to the virtual scene. One important reason is that Java3D only allows branch groups to be added or detached from a live scene not the geometric object. To better illustrate this point, let us consider the basket example in Figure 2.3, the Acumen class structure and the corresponding Java3D scene graph are shown below:

![Basic Java3D scene graph](image-url)
Figure 3.2: Mapping from Acumen class graph to Java3D scene graph

Every object in Acumen model, like the ball or basket, will map to a Java3D object as well as a branch group object and a transform group object.

### 2.2 Visualization data format

Integrating visualization into simulation requires that the data format follows the language's syntax and semantics. In addition, to satisfy the goal that very little modeling efforts should be devoted to building the visualization, abstraction is crucial. The ideal design allows developer to express the 3D visualization related information in a very high level and abstract form. For example, declaring a object's type is "Sphere" will generate a 3D sphere in the visualization automatically. To provide a basic set of 3D visualization functionalities, we summarized that there should be at least 5 fields for every 3D object and they are type, position, size, color, and rotation angle. Compare to other formalism for modeling hybrid systems, Acumen is unique in providing a clear binding time separation from the source language, as well as in its support for derivatives. Binding time makes every Acumen variable is implicitly function of time, its value throughout the whole simulation time will be stored and represented in plot view and trace table. Animation or 3D visualization are inherently binding to time (For example, a object's position
usually vary with respect to time), thus inspiring us to design the visualization data format to be just an ordinary Acumen variable.

In the thesis work, there have been two different formats. The first one expresses the five fields separately in the simulation model using five different reserved variables (_3DType, _3DPosition, _3DSize, _3DColor, _3DAngle) dedicated for 3D visualization. The other one declares the five fields in a more compact way, using only one reserved variable (_3D) that includes all the five fields. For example, if we want to put a cylinder at position [1,1,1] and do not care about its size and color, then the two corresponding formats are as follows:

<table>
<thead>
<tr>
<th>Design 1</th>
<th>Design 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>_3DType</td>
<td>_3D</td>
</tr>
</tbody>
</table>
| \[=\] "Cylinder"; | ["Cylinder", [1,1,1],[0.1,1],
| _3DPosition | [1,0,0], [0,0,0]]; |
| \[=\] [1,1,1]; | \[=\] [0,1,1]; |
| _3DSize | _3D      |
| \[=\] [0.1,1]; | \[=\] [0.1,1],
| _3DColor | [1,0,0], [
| \[=\] [1,0,0]; | 0.1,1], [1,0,0],
| _3DAngle | [0,0,0]; | [0,0,0], [0,0,0]]; |

![Figure 3.3: Static cylinder](image)

They result in the same visualization shown in Figure 3.3, and it is clear that the second format is less verbose. Moreover, using the first format makes it impossible to visualize multiple objects in one Acumen class. Because there is no way to represent multiple objects using one set of variables and one has to create multiple classes to visualize multiple objects. For example, a Ping Pong table is made up of a surface, four legs and a net. Using the first format will require us to make six classes in order to visualize the whole table, which is very inconvenient. The second format, however, can include multiple object's information in one class still using only one reserved variable. The table model and visualization are shown below:
Thus we choose the second visualization data format to express the visualization related information using only one reserved variable named \(_3D\). Also, any variables already in the simulation can be used to specify the visualization fields. In this example, the Table class has three initial parameters \(s_1, s_2, s_3\) to represent the size of the table surface, leg and net respectively. We can just plug in those variables into the corresponding \(_3D\) fields, for example, all table legs are "Box" with size \(s_2\) and with color gray.

We summarized before that there should be five basic attributes for specifying a 3D object. Now we will describe the meaning of each of them in a more detailed way. First, let us consider a dynamic example where a ball with unit mass attached to a linear spring. A linear spring is one that exert force linear to the distance it has been stretched. Assume that there is no gravity, and that the ball only moves along one line passing through the origin. Assume also that the other end of the spring is attached to the origin. This system can be modeled as follows:
Let $x$ denotes the distance between the ball and origin and it's 2D-plot is shown in Figure 4.1. It is clear that we should use the value of $x$ to visualize the movement of the ball. The complete model with 3D visualization are shown below:

```
class Main (simulator)
    private x = 0; x' = 2; x'' = 0 end
    x'' [=] - 4 * x
end
```

**Figure 3.6: Single object example**

We added the _3D variable to the model, and used the variable $x$ to consist of a vector $[x,0,0]$ to be the second element of _3D. Thus, we connect the variable already in the model to the visualization variable in a very easy way. The meanings of each element of variable _3D are as follows:

The first element is a string, which is used to specify the geometry of the 3D object to be drawn later in the animation. Now it supports four different primary geometries which are sphere, cylinder, box and cone respectively. As a result, a valid string for specifying a geometry should be “Sphere”, “Cylinder”, “Box” or “Cone”. In this example, it is “Sphere”. The second element is a triple, which specifies the position of the 3D object in the three dimensional Cartesian coordinate system. In this example, the triple is $[x,0,0]$ and the ball moves along X-axis. If we change it to $[0,x,0]$ or $[0,0,x]$, then the ball will move along Y-axis and Z-axis respectively. The third element determines the size of the 3D object, which can be a number, tuple or triple. In this case, the size of the ball is specified to 0.2. If it is a cylinder, its size can be assigned to $[0.1, 0.5]$, which denotes the length of the cylinder is 0.5, and width is 0.1. The
fourth element is also a triple, which represents the color of the 3D object using RGB color space. For example, \([1,0,0]\) represents the color red, while \([0,1,0]\) represents green. The last element is a vector with 3 variables which specifies the rotation of the 3D object in radian and the rotation is around the center of the object itself in counterclockwise manner.

It is very common to have more than one object being modeled at a single class, like the static Ping Pong table example shown before. We can still visualize such system by using only one variable, which is a vector of vectors, each one of them represents a different part of the system. To better illustrate this point, let us take a look at the Magic Box example; There are 6 objects with different appearances and they all move back and forth between their start point and origin as if they were attached to a linear spring. We also want every object’s size, color, rotation angle be varying all the time. The model and its animation are shown below:

```plaintext
x'' [=] -x;
// Sphere's size
ss [=] 0.2 * x;
// Box's size
bs [=] [0.3*x, 0.3*x, 0.3*x];
// Cone's size
cs [=] [0.2*x, 0.5*x];
// Rotation
rot[=] [x, x, x];
_3D [=]
["Sphere", [x, 0, 0], ss, [x, 0, 2-x], rot],
["Box", [0, 0, x], bs, [2-x, x, 0], rot],
["Cone", [0, x, 0], cs, [0, 2-x, x], rot],
["Sphere", [-x, 0, 0], ss, [x, 0, 2-x], rot],
["Box", [0, 0, -x], bs, [2-x, x, 0], rot],
["Cone", [0, -x, 0], cs, [0, 2-x, x], rot];
end
```

Figure 3.7: Magic box example

$x$ is the distance the current position to the origin, and its value is like a sine function. By using $x$ to represent the position, size, color and angle variables of each object, we achieved the desired behavior that every object’s position, size, color, rotation be varying continuously.
2.3 3D view and user interface

Many tools and languages for modeling hybrid systems use a separated configuration file that is dedicated for creating the virtual environment, which includes the background, lights and textures. These files can be written in many forms like VRML or CAD. The benefits are that one can create very sophisticated and life-like scenes and various textures for 3D objects. However, it is very time consuming and requires the user to have a solid understanding of the external 3D APIs even for visualizing a very simple system, which greatly impede its effective usage among non-professionals, especially for students studying hybrid systems. To address that, we make a design decision that Acumen's visualization system has a predefined virtual environment and the developer does not need to set it by himself. The 3D view tap is next to Trace view tap, and the empty scene is shown below:

![Figure 3.8: An empty scene with transparent unit box](image)

By default, the virtual scene contains a transparent box and three lines to give user a basic idea about where the origin of the three dimensional virtual universe is and the three coordinate axises. We also designed a “player” like user interface, which will allow the developer to interact with the animation in a number of ways, such as pause the animation, change the play speed, jump forward/backward. In addition, the virtual scene can be rotated and zoomed in/out by the user.
3 Implementation

The last chapter briefly describes this thesis work and a few important design choices are presented. In this chapter, we will explain the implementation in a more detailed way and a number of benchmarks are presented in the last section.

3.1 Collecting visualization data

Because the visualization related data is a normal variable in the simulation model, it is essential to extract it from the rest variables and store its value elsewhere for fast rendering. The basic strategy to do that is to look for the variable named "_3D" among all the other variables in the simulation, and copy its value to a special data structure or data core. In order to do that in Acumen or any other DSL (domain specific language), we first need to understand the AST (abstract syntax tree) of that language.

One type of data structure is used to express the most basic values like integers or strings in Acumen and it is called GroundValue.

```scala
/* ground values (common to expressions and values) */
sealed abstract class GroundValue
/* Example: 42 */
case class GInt(i: Int) extends GroundValue
/* Example: 4.2 */
case class GDouble(d: Double) extends GroundValue
/* Example: true */
case class GBool(b: Boolean) extends GroundValue
/* Example: "foo" */
case class GStr(s: String) extends GroundValue
```

For example, a string "Sphere" in the source code will be stored like `GStr("Sphere")` and 1.2 will be stored like `GDouble(1.2)` after parsing phase. Another important data type is called `Value`, which is used to represent not only basic literal values like integers or strings but also aggregations like vector or list.
sealed abstract class Value[+Id]
/* Example: 42 (or "a", or 4.2 ...) */
case class VLit[Id](gv:GroundValue) extends Value[Id]
/* Example: 1::3::4::nil */
case class VList[Id](l:List[Value[Id]]) extends Value[Id]
/* Example: [1,3,4] */
case class VVector[Id](l:List[Value[Id]]) extends Value[Id]

The _3D variable is actually a vector, for example if in the source code we have
_3D  =  ["Sphere",  [0,0,0],  1,  [1,1,1] , [0,0,0]], and its value after parsing will be stored as:

VVector (List(
  VLit(GStr("Sphere")),
  VVector(List(VLit(GInt(0)) VLit(GInt(0)),VLit(GInt(0)))),
  VLit(GInt(1)),
  VVector(List(VLit(GInt(1)), VLit(GInt(1)),VLit(GInt(1)))),
  VVector(List(VLit(GInt(0)), VLit(GInt(0)),VLit(GInt(0)))))

As described in chapter 2, Acumen stores all the variables in the simulation, and the data structures for storing them are as follows:

type CValue = Value[CId]
type CObject = Map[Name, CValue]
type CStore  = Map[CId, CObject]

CStore is a map holds the variables for every class, CId is the class's name and CObject is also a map holds all the variables inside one class. Name is the variable's name and CValue is its corresponding value with the type Value as shown before. Similar to this data type, the visualization data core is implemented as below:

/* Stores all the information for 3D-visualization */
type _3DStore = Map[CId, _3DClass];

19
/* Stores 3D-visualization information for a class */

type _3DClass = Map[Int,List[List[_]]]

_3DStore is a map which holds all the visualization related data, and _3DClass is also a map which holds all the visualization data inside one class. Because there can be multiple 3D objects inside one class so that each one has an integer as the key. The main function for collecting visualization related data is as follows:

def getData(s:CStore){
  for((id,o)<s.toList){
    /* Look for variable named _3D */
    for ((name,value) <- o.toList ){
      if(name.x == "_3D"){
        value match {
          case VVector(l) => {
            if (l.size == 0)
              throw _3DError(value)
            else
              l(0) match{
                /* If it’s only one object, 
                * _3D will start with a string or an int, 
                * example: _3D = ["Sphere",...,...]
                * _3D = [2,...,...]; */
                case VLit(_) => addTo3DStore(id, _3DData, List(value), 1)
                /* If it contains multiple objects, _3D will start 
                * with a vector. 
                * example : _3D = [["Sphere",[[]],[...]]
                * ["Sphere",[],[...]] */
                case VVector(some) => addTo3DStore(id, _3DData, l, l.size)
              case _ => throw _3DError(value)
            }
          case _ => throw _3DError(value)
        }
      }
    }
  }
}

This function goes through all the variables to find out those with name "_3D", and based on the type of the first element, two different methods were called to copy the values into the visualization data core.

### 3.2 3D visualization

After collecting visualization data from the simulation, the next step is to generate a 3D visualization or animation based on those data. First and foremost, we need to choose one suitable 3D API as the implementing tool. Second, implement a rendering mechanism to create an animation based on the visualization data core. At last, implement an easy-to-use GUI to allow the developer to better utilize the 3D visualization.

One type of these node objects specifies the geometry and appearance of virtual objects that are visible in the virtual universe. Some simple geometries such as box, cylinder, sphere, cone are provided by Java3D utility classes. For example, to add a sphere to the scene, one can write the code below:

```java
scene.addChild(new Sphere(1.0f, appearance));
```

That will add a sphere with size 1 and a certain appearance (color, texture) to the scene. The other type of node objects are invisible to the user but nevertheless necessary for setting up the scene graph. These objects specify the position of the virtual objects, the observer’s view of the scene, the lighting and so on. By default, Java3D puts all objects at the virtual universe’s origin. To modify their position and orientation in the universe, Java3D uses a class of objects called transform groups. Each TG (transform group) performs one or more transformations such as scaling, translation, or rotation upon its children objects. Thus all its children objects will be transformed in the same manner. For example, to move all the objects belong to a transform group called TG to a point, say (1, 1, 1); One can use the function called setTranslation like this:

```java
TG.setTranslation(new Vector3f(1.0f, 1.0f, 1.0f));
```

### Rendering Mechanism

Visualization data core has all the information needed for building an animation. The rendering mechanism is to translate those data to the corresponding Java3D code to actually create the animation. Basic strategy is go through the data core at every time step and translate every object to the specified position and with the specified outlook (size, color).
def renderCurrentFrame(){
for((id,map) <- _3DDateBuffer){
  for((objectNumber,buffer) <- map ){
    val frame = (currentFrame-bufferFrame(buffer.head)).toInt
    if(firstFrame(buffer) <= currentFrame  &&
      lastFrame(buffer) >= currentFrame){
      transformObject(List(id,objectNumber),app.trans,buffer,currentFrame)
      checkLook(List(id,objectNumber),lastLook,buffer,
        currentFrame,buffer(frame))
    }
    else {
      deleteObj(List(id,objectNumber))
    }
  }
}
}

This function go through the data core, and checks if the object should be rendered at the moment (currentFrame). If so, translate the object to the position specified in the data core; Otherwise, delete that object. Moreover, to achieve the behaviour described at the MagicBox example, we check if the object's outlook (type, size, color) has changed; If so, delete that object and created a new one with the new outlook.

Adding an object
The basic function we use to add object to scene is a function called addChild(). We first create the appearance and material classes with the color information from the data core then create the geometry object with the size and type information. For example, to add a cylinder to the live scene, the Java3D codes will be executed are as follows:

```java
val color  = bufferColor(list)  // Get the color and size of the object
val size  = bufferSize(list)
val mat = new Material()        // Define the material of the object
mat.setAmbientColor(new Color3f(color(0).toFloat,color(1).toFloat,color(2).toFloat))
```
mat.setDiffuseColor(new 
    Color3f(color(0).toFloat,color(1).toFloat,color(2).toFloat))
mat.setShininess(100);
val ap = new Appearance()
ap.setMaterial(mat)
scene.addChild(new Cylinder(abs(size(0).toFloat),abs(size(1).toFloat), ap));

Moving and rotating an object
To move an object to a certain position in space, we just need to translate the corresponding 
Transform Group using functions like `setTranslation(position)` and `rotX()` with 
the position and rotation information from the data core. As a result, when we need to translate 
and rotate an object, we first get the position value and rotation angle value from the data core 
then call the corresponding Java3D functions to do the translation and rotation.

```scala
val tempPosition = bufferPosition(buffer(frame)) // The angle of the object at that frame
val tempAngle  = bufferAngle(buffer(frame))
var transform  = new Transform3D()
var transAngle = new Transform3D()
    transAngle.rotZ(tempAngle(2))
    transAngle.rotY(-tempAngle(1))
    transAngle.rotX(tempAngle(0))
transform.setTranslation(new Vector3f((tempPosition(0)).toFloat,
                                      (tempPosition(1)).toFloat,
                                      (tempPosition(2)).toFloat))
```

`tempPosition` and `tempAngle` are the position and rotation values from the data core. Based 
on them, Java3D functions like `rotX` and `setTranslation` are called to actually move the 
object to the specified position and rotate it as required.

We also implemented two scala actors [11] to coordinate the animation display. One is called the 
timer, the other is called the renderer. The renderer holds the visualization data core, and will 
only update the virtual scene after receiving the "go" message from the timer. The timer will 
send the message "go" to renderer at a frequency determined by the current play speed. So if
the play speed is 0.5x for example, then the timer will send the message at a frequency that are half as the default one. In addition, the timer will stop sending message if the pause button has been pressed.

3.3 The Graphical User interface (GUI)

To allow the developer to interact with the 3D visualization. We built an “player” like GUI (graphical user interface), which will allow the developer to interact with the animation in a number of ways, such as pause the animation, change the play speed, jump forward/backward, etc. A snapshot of the interface is as below:

![GUI snapshot](image)

**Figure 4.1: GUI snapshot**

There are in total five buttons, they are “play”, “pause”, “faster”, “slower”, “stop”. “Play” button will restart the 3D animation from beginning and “pause” and pause the animation, click it again will resume the animation. “Faster” and “slower” buttons are to change the play speed of the animation, and the current play speed along with the current play time will appear at the right part of the player. “Stop” button will stop the animation and reset the viewing to the default setting.

Acumen3D supports zoom in/out and rotate operations to the virtual scene, which will allow user to see all the different perspectives of the visualization. Zoom in/out can be achieved simply by using mouse wheel or using keyboard shortcut. To rotate the virtual scene, just hold the left click and drag the mouse.

Just like in any general media player, Acumen3D has a slider that is called “timeline” to indicate how much the animation has been played. In addition, user can drag the knob of the timeline to
jump forward or backward of the animation, just like watching a movie. Additionally, when the animation is being paused, one can still drag the knob with the scene being static to capture some critical moments, e.g., when a ball impacts on a bat and its direction changed. Providing such external functionalities, we hope to facilitate the designing and debugging of cyber-physical models.

![Timeline](image.png)

**Figure 4.2: Time line**

### 3.4 Benchmarks and result

To validate Acumen3D's functionality as well as the performance, a number of benchmarks were developed.

#### 3.4.1 Breaking ball

Acumen object can be created dynamically, to test how well it is handled with 3D visualization, the BreakingBall example was developed. In this model, when the ball touches the ground, it will break into two smaller balls and go to different directions. The snapshots of the 3D-visualization are as follows:

```java
class BreakingBall(z, z', x, x', s,c)
private
z''=0;
_3D=['Sphere', [x,0,z], s, c,[0,0,0]];
end
z''[=]-9.8;
_3D[=] ['Sphere', [x,0,z], s, c,[0,0,0]];
if z < 0 && z' < 0
    create BreakingBall(z, -z',x, -1.1*x',0.9*s, [abs(x) , abs(2*x), 5*x]);
create BreakingBall(z,-0.9*z',x, x',0.9*s, [abs(z'), abs(x), s ]);
```

```
There are around 480 balls over 8 seconds, all were generated in run-time. It shows Acumen3D supports dynamically generating objects. Breaking ball object is initialized with the position \((x, z)\), velocity \((x', z')\), size and color. When the height \(z\) is smaller than 0 and the direction is towards the ground, then the if statement is active and creates two breaking ball with different initial states and the old ball will be terminated by the statement `terminate self`.

### 3.4.2 Pendulum

Rotation is very common in mechanical systems, but mainstream 3D libraries normally using matrix multiplication to achieve it, which can bring unnecessary trouble. For example, to rotate around X-axis, one has to use a 4*4 matrix

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & \cos(\alpha) & -\sin(\alpha) & 0 \\
0 & \sin(\alpha) & \cos(\alpha) & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

to perform such rotation.

We thought it can be simplified in Acumen3D, and the rotation of an object is been represented by an 1*3 vector just like its position or color. For example, if the rotation is around X-axis for 45 degrees, the resulting rotation variable will be \([1,0,0]\) and nothing more. The following two benchmarks are all related to rotation, the first one is a normal pendulum example, the second is an inverted pendulum stabilized by a LQG (linear quadratic gaussian) controller.

The pendulum model is an idealization of a real pendulum using the following assumptions:

- The rod on which the ball swings is massless, inextensible and always remains taut;
- Motion occurs only in two dimensions
- The motion does not lose energy to friction or air resistance.

The differential equation which represents the motion of a simple pendulum is:

\[
\frac{d^2 \theta}{dt^2} + \frac{g}{l} \sin \theta = 0
\]

where \(g\) is acceleration due to gravity, \(l\) is the length of the pendulum, and \(\theta\) is the angular displacement. To visualize the model, however, a couple of calculations need to be performed.
first. Assume the rod is attached to the origin, then we need derive the center position of the rod with respect to the angle $\theta$ and the length $l$. In this case the position is:

$$P = [0.5*l\sin(\theta), 0, -0.5*l\cos(\theta)];$$

The next thing is the rotation variable, as described before, Acumen3D only uses one vector to perform rotation. Assume the pendulum is hanging in X-Z plane and is rotating around Y-axis, we then can derive the rotation variable to be $[\frac{\pi}{2}, \theta, 0]$. Notice that $\frac{\pi}{2}$ is come from the fact that by default a cylinder is at X-Y plane, so we need rotate it around X-axis for half $\pi$ first. Then we can visualize the rod as a cylinder as follows:

$$_3D = ["Cylinder", \ p,\ [0.02,1],[0,1,0],\ [\frac{\pi}{2},\theta,0]]$$

The complete model as well as snapshots are shown below:

```plaintext
class Pendulum(s1,s2,red,green)
private
    theta = 1; theta' = 0; theta'' = 0;
    l = 2; Pi = 3.1415926; zero = [0,0,0];
    p = [0,0,0]; angle = [0,0,0];g = 9.8;
    _3D = [["Cylinder",[0.5,0,0],s1,        
            [0,1,0],angle],
            ["Sphere", [0,0,0], s2,        
            [1,0,0],zero]];   
end
theta'' [=] -(g/l)*sin(theta);
    p [=] [0.5*l*sin(theta),0,-
        0.5*l*cos(theta)];
    angle [=] [Pi/2,theta,0];
    _3D[=]["Cylinder",p, s1,green, angle],
            ["Sphere", 2*p,s2, red, zero]];\end
```

Figure 4.4: Pendulum example
3.4.3  Inverted pendulum

An inverted pendulum is a pendulum which has its mass above its pivot point and is often implemented with the pivot point mounted on a cart that can move horizontally. Whereas a normal pendulum is stable when hanging downwards, an inverted pendulum is inherently unstable and must be actively balanced in order to remain upright. This feature makes it a classic problem in dynamics and control theory and is widely used as a benchmark for testing control algorithms. In this model, the inverted pendulum is stabilized by a simple LQG (linear-quadratic-gaussian) controller, which will exert force upon the cart. Theta is measured clockwise from the vertical axis. The complete physical equations of the system are from[]. The simple controller we used in this example is modeled as follows:

\[ F = 64.0348 \theta + x + 4.36673 x' + 19.7313 \theta' \]

Like the previous example, we need to calculate the central position and rotation variable to visualize the system, which is shown below:

\[ p = [x + l \times 0.5 \times \sin(\theta), 0, 0.15 + l \times 0.5 \times \cos(\theta)]; \]

\[ _3D = \text{"Cylinder"}, p, [0.01,1], [0,1,0], [\pi/2, -\theta, 0]. \]

Notice because the pendulum has been inverted, so the rotation is with the opposite direction to the normal pendulum, and we use \(-\theta\).
The equations for calculating second derivatives of $x$ and $\theta$ are derived from the Lagrangian equations of this system and are transformed to ODEs. From the 3D visualization, we can tell that LQG controller can balance the system.

### 3.4.4 Walking robot

In complex mechanical systems, a whole model often is consist of several sub-systems, which hold certain connection between each other. For example, and robot arm's position is depend on joint on the shoulder. So it's natural to build the visualization based on sub-systems, and the connections should also holds in the animation. To test the consistency of visualizing such systems, one benchmark called WalkRobot was developed, and it is consist of three main parts, head, upper part and lower part. The upper part consist of two arms, two joints and one beam in between and is modeled as follows:
class upperPart(p)
    private
    p'=[0,0,0];
    armAngle = 1; armAngle' = 0;
    armAngle'' = 0;
    leftP = [0,0,0]; rightP = [0,0,0];
    offset = [0.6,0,0];
end

_3D [ ] ["Cylinder", p,[0.2,1.2], [1,1,1], [0,0,3.14/2]],
["Sphere", p-offset, 0.25, [0,0,0.5],[0,0,0]],
["Sphere", p+offset, 0.25, [0,0,0.5],[0,0,0]],
["Cylinder", leftP,[0.1,0.8], [1,1,1], [3.14/2 - armAngle,0,0]],
["Cylinder", rightP,[0.1,0.8], [1,1,1], [3.14/2 + armAngle,0,0]]);

armAngle'' [ ] -10*sin(armAngle);
leftP [ ] p + [0.6,-0.05,-0.4] + [0,-0.45*sin(armAngle), (0.45- 0.45*cos(armAngle))];
rightP[ ] p + [-0.6,-0.05,-0.4] + [0,0.45*sin(armAngle), (0.45- 0.45*cos(armAngle))];
end

Figure 4.6: Robot's upper part

P is the central position of this part, which is in the middle of the beam. And both the two bearing and the two arm's position (leftP and rightP) are based on p. Together, they form a upper body of a simple robot. To make the robot looks like walking, we also made the two arms be swinging, using the same concept from the normal pendulum example. The same has done to the head part and lower part, and together we assembled the robot as follows:
class Robot(p1,p2,p3)
private
  t=0; t'=1;
  head = create Head(p1);
  upperPart = create upperPart(p2);
  lowerPart = create lowerPart(p3);
end
head.p'  []  [0,-0.2,0];
upperPart.p' [] [0,-0.2,0];
lowerPart.p' [] [0,-0.2,0];
end

Figure 4.7: Walking robot example

By specifying the speed to be the same, the robot will looks like moving as a whole. The result animation shows that even though we define the 3D-visualization variables separately in different classes, the final animation is correct and seamless.

3.4.5 Newton's cradle

Let's take a look at another interesting example, which models Newton's cradle, an apparatus with five balls hanging from strings. Each one of them is essentially a pendulum model, but with initial attached point position and angle displacement. If the system is initialed with one of the balls (m0) displaced as shown in Figure 4.8 at upper right. When the ball collides with the ball next to it, a series of transient states results. At that time, the rightest ball transfer its momentum to the next ball and so forth. At last, the one at the very left (m4) get the momentum and starts
swinging. We can easily modify the initial displacement of the system in order to validate its correctness. Figure 4.9 shows another example, where three balls has the same initial angle displacement.

```java
class Cradle()
    private
        m0 = create Pendulum([1,0,0],1);
        m1 = create Pendulum([0.5,0,0],0);
        m2 = create Pendulum([0,0,0],0);
        m3 = create Pendulum([-0.5,0,0],0);
        m4 = create Pendulum([-1,0,0],0);
        temp = 0;
        mode = "fly";
    _3D=["Box",[0,0,0.1],[2,0.5,0.2],
            [0.2,0.2,1.0],[0,0,0]]
end

Figure 4.8: Newton's cradle example one
Three balls (m0, m1 and m2) were initialized with the same angle 1 and the results are shown above. The complete model is at the appendix section.

3.4.6 Three mass system

In the monograph Languages and tools for hybrid systems design [13] by Carloni, Passerone, Pinto and Sangiovanni, one example was used throughout the book, a system of 3 point masses. We consider a system where three point masses, $m_1$, $m_2$ and $m_3$, are disposed on a frictionless surface (a table) of length $L$ and height $h$. Mass $m_1$ has initial velocity $v_1$ while the other two masses are at rest. Mass $m_1$ eventually collides with $m_2$ which, in turn, collides with $m_3$. Consequently, mass $m_1$ falls from the table and starts bouncing on the ground. Each collision is governed by the Newton’s collision rule and the conservation of momentum. Let $v_i$ and $v_i'$ denote the velocity before and after the collision, respectively. Then Newton’s rule states that $v'_1 - v'_2 = -\varepsilon (v_1 - v_2)$, where $\varepsilon$ is called coefficient of restitution. The conservation of momentum is the other rule that determine the velocity after the impact: $m_1(v'_1 - v'_2) = m_2(v_2 - v'_2)$. A collision between $m_1$ and $m_2$ happens when $x_1 \geq x_2$ and $v_1 \geq v_2$, in which case the velocities after collision are:

$$v'_1 = v_1 \frac{(m_1 - \varepsilon m_2)}{m_1 + m_2} + v_2 \frac{m_2(1 + \varepsilon)}{m_1 + m_2}$$

$$v'_2 = v_1 \frac{(1 + \varepsilon)m_2}{m_1 + m_2} + v_2 \frac{m_2(m_2 - \varepsilon m_1)}{m_1 + m_2}$$
Adam duracz modeled this system in Acumen and using the visualization system to build the 3D animation. The collision equations above are models as follows:

```java
if m1.x >= m2.x && m1.x' > m2.x' // C12
    td = m1.xd;
    m1.xd = td * (m1.m - c_rest * m2.m) / (m1.m + m2.m) + m2.xd * (m2.m * (1 + c_rest) / (m1.m + m2.m));
    m2.xd = td * ((1 + c_rest) * m1.m / (m1.m + m2.m)) + m2.xd * (m2.m - c_rest * m1.m) / (m1.m + m2.m);
end;
```

The if statement checks the condition $x_1 \geq x_2$ and $v_1 \geq v_2$, $m_1.xd$ is $m_1$'s velocity before the impact. Part of model related to the visualization as well as snapshots from the 3D animation are shown below:

```java
class MovingMass
    (m,x,y,x',y',x'',y'',b_color,xd,ya)
private
    3D = ["Sphere",[0,0,0],0,[1,0,0],
        [0,0,0]];
    B_size = 0.1;
    offset = [-3,0,0];
    zero = [0,0,0];
end
_3D [] = ["Sphere",[x,0,y]+offset,
        b_size,b_color,zero];
end
```

Figure 4.10: Three masses system

The complete model is at the appendix section.
4 Case Study: A Model of Ping Pong

As a larger case study into the expressivity of this core language, we developed a simplified model of a Ping Pong game. This game was used in the first edition of a CPS (cyber-physical systems) course at Halmstad University in the third period of the 2011/2012 academic year [2]. I also made the same model of Ping Pong game purely in Java3D, and the comparison of modeling such systems using Java3D and Acumen3D will be present in the section 5.3.

4.1 Introduce the Ping Pong model

To simulate a real-world Ping Pong game, this model consider all the basic elements of the game. The following is a snapshot from the animation of a game between a team called WiffWaff and another called Paradox.

![Figure 5.1: Ping Pong game overview](image)

The model contains some elements that are visible such as Ping Pong ball, two bats and the table. However there are some elements or classes are invisible. For example, the bat actuator that constantly act upon the bat to control its trajectory and also the referee class that monitors the game all the time. A comprehensive abstract graph that contains all the components (classes) in this model is shown as follows:
The model we describe here consisted of nine classes spanning 618 lines of code. We will introduce selected portions of the game model, starting with the class Ball representing the ping-pong ball whose basic functionality is to bounce. In addition, its dynamics are affected by air resistance, proportional to the square of the ball’s speed and directed opposite to its velocity. Acumen also supports vectors and basic vector operations (e.g., inner product, cross product), allowing us to express the essential behaviour of the model as follows:

```
switch mode
    case "Fly"
        if dot(p,[0,0,1]) < 0 && dot(p',[0,0,1])< 0
            mode = "Bounce";
        else
            p'' [=] -k2 * norm(p') * p' + [0,0,-9.8];
        end;
    case "Bounce"
        p' = p' .* k_z; // Bouncing will lose some energy
        mode = "Fly";
    end
_3D [=] ["Sphere",p,0.03,[1,1,1],[0,0,0]];
```

Figure 5.2: Components of Ping Pong game
The dot operator takes into two vectors and computes the dot product between the two. The ball has two different modes or phases, one is a continuous mode when it is flying the other is a discrete mode when it is bouncing. By calculating the position of the ball (p in this case) and using that variable we can visualize the 3D Ping Pong ball in the model.

Another class represents a mechanical player. This is the class that students developed during the course of the class project. Early on in the project, the player was allowed to send a velocity signal directly to the bat, making it easier to design an ideal controller. For example, when the player wants to get close the ball, one general adopted strategy to control the velocity of bat is like that:

\[ v \text{ } [=] \text{ } \text{ballp} \text{ } - \text{ } \text{batp}; \]

The velocity of the bat (v) is based on the difference between ball’s positions (ballp) and bat’s position (batp), the closer the ball and bat, the slower the speed is. Similarly, when the player wants to get back to the starting point, the controller is like that:

\[ v \text{ } [=] \text{ } \text{startPoint} \text{ } - \text{ } \text{batp}; \]

Later on, the player could only send acceleration signals, and students had to write a controller that computes an acceleration signal a based on the velocity signal v they had already figured out how to compute. The controller for one of the players was expressed as follows:

```java
private
v = [0,0,0]; // Bat's speed
a = [0,0,0]; // Bat's acceleration (computed by controller)
estimatedBatV = [0,0,0]; // Predicted result of acceleration
estimatedBatV' = [0,0,0]; // Equated to computed acceleration signal
desiredBatP = [1.6*(-1)^n,0,0.2]; // Desired bat position (from derive)
desiredBatP' = [0,0,0]; // Desired bat speed (from naive)
end
// Compute the desired position based on the computed "control" speed
desiredBatP' [=] v;
// Keep track of effect of acceleration to estimate actual bat speed
estimatedBatV' [=] a;
// Basic controller to actuate acceleration
a [=] 13*(desiredBatP' - estimatedBatV) + 50*(desiredBatP - batp);
```

Here, desiredBatP and its derivatives are computed based on the v signal that is computed by a motion planning component (like the example before). In addition, an estimate of the anticipated result of the controller’s output is computed by the equation for estimatedBatV’. The acceleration a itself is computed by the last line, which constitutes a second order linear controller based on both measured and predicted components of the state of the bat as well as on the desired states.
Different player may have different return strategies, e.g., the team WiffWaff (shown Figure 4.1) always try to return the ball to the middle of the table; His opponent Paradox, however, tries to return the ball to land very close to the net. As stated before, this model contains some class that are invisible, Referee is one of them. The basic function of referee class is to observe the game, keep the scores and enforce the rules of the game. For example the referee checks whether the ball is out of boundary all the time, as the following code shows:

```java
if z<0.05 && z'<0 && status == "Normal"  // when the ball is near the ground
    if (abs(y)>0.78||abs(x)>1.53) && status == "Normal"
        reason = "BallOutOfBoundary";
    if bounced == false
        if x>0
            mode = "Player1Lost";
        else
            mode = "Player2Lost";
        end;
    end;
end;
```

Once the referee detects the ball is out of boundary, the mode will be changed accordingly to either “Player1Lost” or “Player2Lost” and the score also will be updated. The other two rules are the ball touches net and bounced twice.

To regulate the flow of information between components we introduce two styles of classes that we often refer to as observers and actuators. Two actuators are used in the model, one for the bat and one for the ball. The actuators make sure that the flow of information out of the player is regulated to enforce realistic constraints. BatActuator regulates the control signal between player and bat, putting a upper bound to acceleration signal and keeping track of the total energy that has been consumed.

```java
if norm(p') > 5
    p' = p'/norm(p') * 5;
end;
if norm(p'') > 100
    p'' = p''/norm(p'') * 100;
end;
energy'[] = sqrt(abs(norm(p')*norm(p'')))/2;
```

So the maximum velocity of the bat is 5, and maximum acceleration is 100. Energy plays an important role in this model, just like in a real game. When player used up all the energy, he can not move the bat anymore and the bat becomes a yellow box to indicate the situation.
BallObserver is used to limit the player’s ability to observe the state of the ball precisely, and uses discrete sampling to model a common constraint in a cyber-physical system.

BallActuator is used when impact happens between the ball and the bat. We use a three-dimensional perfect collision model to simulate such impact.

private
mode="Initialize";
v1 = [0,0,0]; // Input ball speed
v2 = [0,0,0]; // Output ball speed
v3 = [0,0,0]; // Bat's speed during the impact
angle = [0,0,0]; // Bat's normal vector
done = false;
action = 0;
end

case "Calculate"
v2 = v1 - dot(2.*(v1-v3),angle)*angle;
end

Game class connects all the components together, which is done using continuous assignments. Making sure the flow of information is correct. The following code fragment shows such continuous assignments:

<table>
<thead>
<tr>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ballob.p = ball.p;</td>
</tr>
<tr>
<td>player1.ballp = ballob.p;</td>
</tr>
<tr>
<td>player2.ballp = ballob.p;</td>
</tr>
<tr>
<td>player1.ballv = ballob.v;</td>
</tr>
<tr>
<td>player2.ballv = ballob.v;</td>
</tr>
<tr>
<td>batActuator1.p&quot; = player1.a;</td>
</tr>
<tr>
<td>batActuator2.p&quot; = player2.a;</td>
</tr>
</tbody>
</table>

Game class also is responsible for resetting the state of components when the game need be restarted. For example, the ball will be put back at the original position and with the initial speed, also player2 is about to serve.

case "Restart" // Put everything back to the starting point
ball.p = [0,0,0.5];
ball.p' = [5,1,-3];
bat2.p = [1.6,0,0.2];
bat1.p = [-1.6,0,0.2];
serveNumber = 2;
referee.bounced = false;
mode = "Player2Serve";
During the course, there has been 3 tournaments. Every tournament consisted of a series of simulations allowing each player two opportunities to play with each other player, taking once the role of the server and the other time the role of the receiving player. For example, the first tournament is made up of a total of 7x7-7 = 42 games. Each player played 7-1 = 6 games. All games were played with the same starting conditions as used in the game template provided for the designers of the players. The results of the 3 tournaments as well as the match videos are available on the acumen website [1].

4.2 Modelling the Ping Pong game in Java3D

To compare the difference and to justify that Acumen3D allows the programmer to build a 3D visualization even easier. We developed the same Ping Pong game model as described before purely using Java3D.

In addition to the basic components in the Ping Pong model, Java3D requires some new classes to build a virtual scene. For example, in order to have a moving ball in the scene; First we need to use the appearance class and geometry class to have a 3D ball; Second, to make it move, a TG (Transform group), which will be the parent class of the ball, is needed. Notice one TG can only perform one transform (e.g. the translation, rotation of all its children objects are the same). So we need several such TGs in our program in order to visualize multiple moving objects (the ball, two bats). Like any Java3D program, there are things need to be setted, like the background, the light, and the viewplatform. All of those extra components are not directly related to the Ping Pong model, but have to be added. In Acumen3D however, those extra work is been taken care of before hand, allowing the programmer to really focus on the model he is trying to build.

4.3 Comparison

Following the discussion above, we will take a close look at the difference bring by those extra components. Take an example of the ball class in the model:

To initialize a ball in the model, Acumen3D needs to declare a new variable name _3D in the private session with the initial state of the ball, which includes position, color, size and angle. In Java3D, it need to use more external class like material class and appearance class.

<table>
<thead>
<tr>
<th>Java3D code</th>
<th>Acumen code</th>
</tr>
</thead>
<tbody>
<tr>
<td>public Ball(TransformGroup root){</td>
<td>_3D [&quot;Sphere&quot;,&quot;p&quot;,0.03,write,zero]</td>
</tr>
<tr>
<td>// appearance of the ball</td>
<td></td>
</tr>
<tr>
<td>Material m = new Material();</td>
<td></td>
</tr>
<tr>
<td>Color3f c = new Color3f();</td>
<td></td>
</tr>
<tr>
<td>m.setDiffuseColor(c);</td>
<td></td>
</tr>
<tr>
<td>m.setShininess(100);</td>
<td></td>
</tr>
<tr>
<td>Appearance appear = new Appearance();</td>
<td></td>
</tr>
<tr>
<td>appear.setMaterial(m);</td>
<td></td>
</tr>
<tr>
<td>TGBall.addChild(new _3D[=&quot;Sphere&quot;,&quot;p&quot;,0.03,write,zero])</td>
<td></td>
</tr>
</tbody>
</table>
To specify the color of the ball to be white, in Acumen3D, we only use one vector \([1,1,1]\); however, in Java3D, several classes like Material, Appearance as well as functions such as setDiffuseColor, setShininess are needed to do the same thing. Thus it requires certain understanding of Java3D to achieve the visualization.

Although Acumen3D only needs one line of code to initialize a 3D object, it contains the main information we want and care the most in a simulation. As described in Section 6, I explained what will be done in the core Acumen3D. And essentially, it will translate the acumen source code to the corresponding Java3D code as shown above. The reason I hide those procedures from the user is that it will allow the programmer to focus more on the model instead of the 3D animation.

Simulation progress

In Acumen, it is very convenient to translate physical equations into executable simulation code. For example, when the ball is flying, according to Newton’s law, its acceleration is determined by gravity and air resistance. The Acumen code is as below:

```acumen
case "Fly"
    if dot(p,[0,0,1]) < 0 && dot(p',[0,0,1])< 0
        mode = "Bounce";
    else
        p'' [=] -k2 * norm(p') * p' + [0,0,-9.8];
end;
```

It first checks if the ball will bounce that is the height is smaller than zero and the velocity is towards the ground. If not, the acceleration—second derivative \(p''\) is equal to the sum of gravity and air resistance. Then the position \(p\) will be calculated automatically at every timeStep and be stored at the trace map for looking up. Using continuous assignment, we can have a 3D visualization of this flying ball.

```acumen
_3D [=] ["Sphere",p,0.03,[1,1,1],[0,0,0]];
```

In Java3D, however, it’s not so convenient to translate physical equations to executable code. Integration step has to be performed manually the same as to the basic operators for vector. Main execution parts are as below:
Take another look at the BatActuator example, it should regulates the flow of information from player to bat. e.g., the velocity of the bat should never exceed 5, and the acceleration should never bigger than 100. The code below is the implementation of this class in Java3D and Acumen3D.
The implementation is the same, however with Acumen’s support for basic vector operation and automatically integration, it easier to build such physical model.

Final result of comparison

Now it is clear that there is less source code in Acumen3D to model and visualize the Ping Pong game. However, it may not be comprehensive by just comparing the length of the two problem-related codes (i.e., the simulation code). Because part of core Acumen is responsible for translating or interpreting the simulation code to executable Java3D code. And this part of code is used by any simulation model running in Acumen3D, we call it the reusable code. Below is the comparison table between modeling the PingPong game in Acumen3D and Java3D.

<table>
<thead>
<tr>
<th>Java3D code</th>
<th>Acumen code</th>
</tr>
</thead>
<tbody>
<tr>
<td>public class BatActuator{</td>
<td></td>
</tr>
<tr>
<td>public float[] batP = new float[3];</td>
<td></td>
</tr>
<tr>
<td>public double[] batV = new double[3];</td>
<td></td>
</tr>
<tr>
<td>public double[] batA = new double[3];</td>
<td></td>
</tr>
<tr>
<td>public float[] angle = new float[3];</td>
<td></td>
</tr>
<tr>
<td>double energy = 0.0;</td>
<td></td>
</tr>
<tr>
<td>double energyPrime = 0.0;</td>
<td></td>
</tr>
<tr>
<td>public void simulationStep (){</td>
<td></td>
</tr>
<tr>
<td>if (norm(batV) &gt; 5){</td>
<td></td>
</tr>
<tr>
<td>for (int i = 0; i &lt; 3; i++)</td>
<td></td>
</tr>
<tr>
<td>batV[i] = (batV[i]/norm(batV)) *5;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>if (norm(batA) &gt; 100){</td>
<td></td>
</tr>
<tr>
<td>for (int i = 0; i &lt; 3; i++)</td>
<td></td>
</tr>
<tr>
<td>batA[i] = (batA[i]/norm(batA))*100;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>energyPrime = Math.sqrt(norm(batV)*norm(batA))/2;</td>
<td></td>
</tr>
<tr>
<td>energy = energy + energyPrime*timeStep;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>class BatActuator(p1)</td>
<td></td>
</tr>
<tr>
<td>private</td>
<td></td>
</tr>
<tr>
<td>p = p1;</td>
<td></td>
</tr>
<tr>
<td>p' = [0,0,0];</td>
<td></td>
</tr>
<tr>
<td>p'' = [0,0,0];</td>
<td></td>
</tr>
<tr>
<td>angle = [0,0,0];</td>
<td></td>
</tr>
<tr>
<td>energy = 0;</td>
<td></td>
</tr>
<tr>
<td>energy' = 0;</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
<tr>
<td>if norm(p') &gt; 5</td>
<td></td>
</tr>
<tr>
<td>p' = p'/norm(p') * 5 ;</td>
<td></td>
</tr>
<tr>
<td>end;</td>
<td></td>
</tr>
<tr>
<td>if norm(p'') &gt; 100</td>
<td></td>
</tr>
<tr>
<td>p'' = p''/norm(p'') * 100;</td>
<td></td>
</tr>
<tr>
<td>end;</td>
<td></td>
</tr>
<tr>
<td>energy'=</td>
<td></td>
</tr>
<tr>
<td>sqrt(abs(norm(p')*norm(p'')))/2;</td>
<td></td>
</tr>
<tr>
<td>end;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Java3D</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>Reusable Code</td>
<td></td>
</tr>
<tr>
<td>Problem-related code</td>
<td>889</td>
</tr>
<tr>
<td>Total</td>
<td>889</td>
</tr>
<tr>
<td>Prerequisite knowledge</td>
<td>Mechanics, Ping Pong Rules, Java, Java3D library</td>
</tr>
<tr>
<td>Support for changing camera view</td>
<td>N</td>
</tr>
<tr>
<td>Support for pause, replay, and variable play speed</td>
<td>N</td>
</tr>
<tr>
<td>Support for inspection of precise values of variables (trace, plot inspection)</td>
<td>N</td>
</tr>
</tbody>
</table>

We also notice in the problem-related code in Acumen3D, part of it is quite redundant. For example, every variable used in this model has to be declared first in the private session. In the future version of Acumen3D, which called Acumen3D++, we expect to add some dynamical initialization mechanism to get rid of such redundancy, which will result in a even less code model.
5 Conclusion and Future work

This thesis presented an extension of the Acumen language with support for 3D visualization and interaction during the execution of the simulation. We also developed a suite of benchmark models to evaluate functionality. The initial results presented in this work are promising, we can use it to visualize both trivial and more complex hybrid systems. Moreover, students can easily learn this new functionality of Acumen and use it to visualize and validate their design. In particular, the initial studies show that the proposed design meets the three goals that motivated this work in the first place, namely, that

1. Developers can easily use it to visualize the simulation models.

2. It provides the expected basic set of 3D visualization functionalities

3. It maintains the platform independence of Acumen

As a large case study into the expressivity of Acumen language and usability of the 3D visualization, we developed a simplified model of a Ping Pong game. By comparing modeling such a hybrid system in Java3D and Acumen3D, we find concrete evidence that modelling it in Acumen3D is easier in that of less code and less prerequisite knowledge and more supporting built in functions.

In this work we also found out several issues need to be addressed. First, Acumen3D takes quite a lot memory to run, finding a way to build the visualization more efficient is very important. Second, now Acumen3D only supports four basic type of geometry, we need to find a way to build customised geometry (using polygons).
6 References


7 Appendix

7.1 Newton's cradle

class Pendulum(init,theta)
private
theta' = 0; theta'' = 0;
l = 2; Pi = 3.1415926; radius  = 0.25;
p = [2*sin(theta),0,-2*cos(theta)]+init;
v = [0,0,0];
_3D = [
    ["Cylinder",[0.5,0,0],[0.02,1],[0,1,0],[3.14/2,1,0]],
    ["Sphere", [0,0,0], 0.1, [1,0,0],[0,0,0]]
];
end
theta'' = -(9.8/l)*sin(theta);
p = [l*sin(theta),0,-l*cos(theta)]+init;
_3D = [
    ["Cylinder", (p-init)/2 +init, [0.01,l],[0,0,1],[Pi/2,theta,0]],
    ["Sphere", p, radius, [0.6,0.6,0.3],[0,0,0]]
];
end

class NewTon()
private
m0 = create Pendulum([1,0,0],1);
m1 = create Pendulum([0.5,0,0],1);
m2 = create Pendulum([0,0,0],1);
m3 = create Pendulum([-0.5,0,0],0);
m4 = create Pendulum([-1,0,0],0);
temp = 0;
mode = "fly";
_3D = ["Box", [0,0,0.1],[2,0.5,0.2],[0.2,0.2,1.0],[0,0,0]];
end

switch mode
case "fly"
if (norm(m0.p-m1.p)<0.5 && ((m0.theta' <= 0 && m1.theta' >=0) ||
    (m1.theta' <=0&&m0.theta'<m1.theta')||
    (m0.theta' >0 && m1.theta' > m0.theta')))
    mode = "collide01"
end;
if (norm(m1.p-m2.p)<0.5 &&((m1.theta' <= 0 && m2.theta' >=0) ||
    (m2.theta' <=0&&m1.theta'<m2.theta')||
    (m1.theta' >0 && m2.theta' > m1.theta')))
    mode = "collide12"
end;
if (norm(m2.p-m3.p)<0.5 && ((m2.theta' <= 0 && m3.theta' >=0) ||
    (m3.theta' <=0&&m2.theta'<m3.theta')||
    (m2.theta' >0 && m3.theta' > m2.theta'))
    mode = "collide23"
end;
if (norm(m3.p-m4.p)<0.5 && ((m3.theta' <= 0 && m4.theta' >=0) ||
    (m4.theta' <=0&&m3.theta'<m4.theta')||
    (m3.theta' >0 && m4.theta' > m3.theta')))
    mode = "collide34"
end;
case "collide01"
    temp = m0.theta';
    m0.theta' = m1.theta';
    m1.theta' = temp;
    mode = "fly";
case "collide12"
    temp = m1.thetap;
    m1.thetap = m2.thetap;
    m2.thetap = temp;
    mode = "fly";
case "collide23"
temp = m2.theta';

m2.theta' = m3.theta';

m3.theta' = temp;

mode = "fly";

case "collide34"

temp = m3.theta';

m3.theta' = m4.theta';

m4.theta' = temp;

mode = "fly";

end

class Main(simulator)

private mode = "init" end

switch mode

case "init"

create NewTon();

mode = "persist";

case "persist"
end

class Main(simulator)

private mode = "init" end

switch mode

case "init"

create NewTon();

mode = "persist";

case "persist"
end

class Main(simulator)

private mode = "init" end

switch mode

case "init"

create NewTon();

mode = "persist";

case "persist"
7.2 Three mass system

class Main (simulator)
    private
        mode = "init";
    end
    switch mode
        case "init"
            simulator.endTime = 10;
            create ThreeMasses();
            mode = "persist"
        case "persist"
    end;
end

class MovingMass (m,x,y,x',y',x'',y'',b_color,xd,ya)
    private
        _3D = ["Sphere",[0,0,0],0,[1,0,0],[0,0,0]];
        b_size = 0.1;
        offset = [-3,0,0]; zero = [0,0,0];
    end
    _3D [] = ["Sphere",[x,0,y]+offset,b_size,b_color,zero];
end

class ThreeMasses ()
    private
        // Constants
        g = 9.8; c_rest = 0.95; c_rest_x = 0.9; c_rest_y = 0.9;
        L = 3; h = 1; b_size = 0.1; t_size = 0.1;
        grey = [0.25,0.75,0.75];
        td = 0; tdp = 0;
        offset = [-3,0,0]; zero = [0,0,0];
        // Initial conditions
m1 = create MovingMass(1.0,0.5,1,1,0,0,0,[1,0,0],1,0);
m2 = create MovingMass(2.0,1.5,1,0,0,0,0,[0,1,0],0,0);
m3 = create MovingMass(1.5,3.0,1,0,0,0,0,[0,0,1],0,0);
end

if m1.x >= m2.x && m1.x' > m2.x' // C12
    td = m1.xd;
    m1.xd = td * (m1.m - c_rest * m2.m) / (m1.m + m2.m)
        + m2.xd * (m2.m * (1 + c_rest) / (m1.m + m2.m));
    m2.xd = td * ((1 + c_rest) * m1.m / (m1.m + m2.m))
        + m2.xd * (m2.m - c_rest * m1.m) / (m1.m + m2.m);
    tdp = m1.x'
    m1.x' = m2.x';
    m2.x' = tdp;
end;
if m2.x >= m3.x && m2.x' > m3.x' // C23
    td = m2.xd;
    m2.xd = td * (m2.m - c_rest * m3.m) / (m2.m + m3.m)
        + m3.xd * (m3.m * (1 + c_rest) / (m2.m + m3.m));
    m3.xd = td * ((1 + c_rest) * m2.m / (m2.m + m3.m))
        + m3.xd * (m3.m - c_rest * m2.m) / (m2.m + m3.m);
tdp = m2.x';
m2.x' = m3.x';
m3.x' = tdp;
end;
if m1.x >= L && m1.y > 0 && m1.x' > 0 // F1
    m1.ya = -g;
end;
if m2.x >= L && m2.y > 0 && m2.x' > 0 // F2
    m2.ya = -g;
end;
if m3.x >= L && m3.y > 0 && m3.x' > 0 // F3
    m3.ya = -g;
end;
if m1.y <= 0 && m1.y' < 0 // B1
    m1.x' = c_rest_x * m1.x';
m1.y' = - c_rest_y * m1.y';
m1.y = - m1.y;
end;
if m2.y <= 0 && m2.y' < 0 // B2
    m2.x' = c_rest_x * m2.x';
m2.y' = - c_rest_y * m2.y';
m2.y = - m2.y;
end;
if m3.y <= 0 && m3.y' < 0 // B3
    m3.x' = c_rest_x * m3.x';
m3.y' = - c_rest_y * m3.y';
m3.y = - m3.y;
end;
end