saces

A Simple Artificial Chemistry Experiment System

DIPLOMA 2006

Berne University of Applied Sciences
School of Engineering and Information Technology
Departement Information Technology

Authors
Anthony Aguillon, Daniel Noelpp

Supervisors
Dr. Peter Schwab, Dr. Thomas Hinze

Expert
Prof. Dr. Federico Flueckiger

Abstract

‘saces’ is a simple tool written in Java which visualizes chemical processes using 3D animation of molecules. It is intended as both an application in artificial chemistry and as an educational tool for students and teachers in chemistry. ‘saces’ simulates ideal gases in a rectangular reaction vessel. The molecules are hard, colored spheres.

An *artificial chemistry* is a man-made system that is similar to a real chemical system. There is a population of molecules, a set of reaction rules and an algorithm which executes the reactions. With artificial chemistry many applications are possible: artificial life, chemistry modeling, massively parallel execution of finite automata, graph rewriting and modeling solution processes of NP-complete problems.

Contents

Chapter 1

Introduction

1.1 Summary

‘saces’ is a simple educational tool which visualizes chemical processes and reactions using 3D animation of molecules. The paradigms of artificial chemistry are helpful when considering a compromise between costly physics based calculations and a more abstract approach to modeling. Artificial chemistry is also in itself a fascinating and important subject.

Adleman demonstrated that computations can be carried out using molecules and solved a *Traveling Salesman*[1] problem using real DNA in 1994. An artificial chemistry can be designed to model DNA computing and their operations. Other applications of artificial chemistry are parallel execution of finite automata, graph rewriting, artificial life and chemical and biology modeling. Molecules in artificial chemistry need not always represent physical molecules. Strings, graphs, bit vectors etc. are also good candidates for artificial molecules. Reaction rules then recombine them in diverse ways. Artificial chemistry is therefore more about speculative thinking than it is an attempt to model chemistry with physical realism. The *Conway’s Game of Life*[2] can be viewed as an extremely abstracted form of artificial chemistry.

Despite the level of abstraction dictated by artificial chemistry, ‘saces’ uses a more concrete model. The molecules are hard, colored spheres which move as an

[1] The problem statement is: Given a number of cities and the costs of traveling between them, what is the cheapest round-trip route that visits each city once and then returns to the starting city?

[2] It is a cellular automaton with two states: A cell can be either alive or dead. The rules are: $a)$ A cell with fewer than two and more than three neighbors dies. $b)$ A cell with exactly three neighbors comes to life. All births and deaths occur simultaneously in the grid [Sil05].

ideal gas in a rectangular reaction vessel. They may collide with each other. If a reaction applies upon collision, the educt molecules are removed from the simulation and the product molecules are added. Physical conservation laws for mass, energy and momentum can be modeled where necessary. A few sample experiments are available: an abstract explosion, a simplified detonating gas explosion, Brownian movement and others.

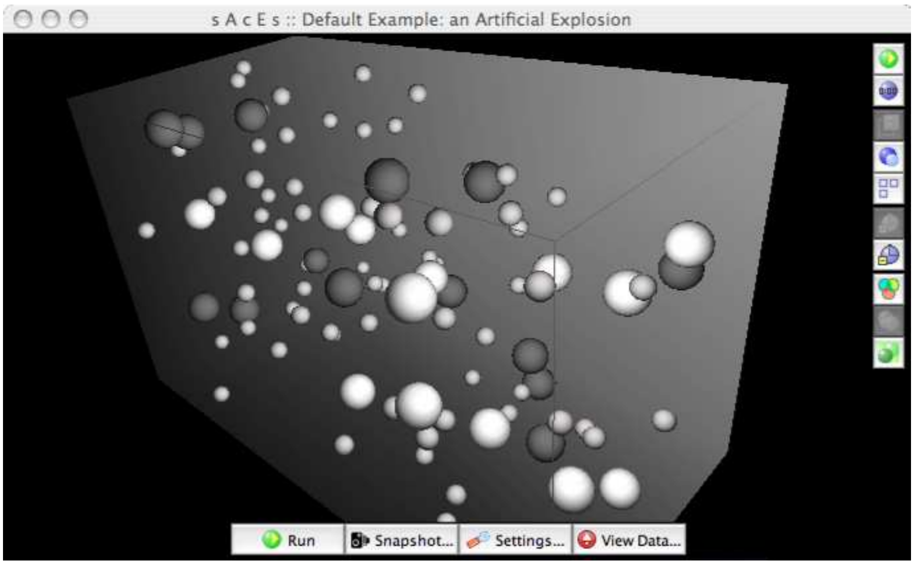


Figure 1.1: A screenshot of an 'artificial' explosion

More general-purpose experiments are possible. A Lotka-Volterra system models predator-prey interaction. The Lotka-Volterra experiment fluctuates as expected: The number of prey molecules decreases as the number of predators increases. The predators in turn begin to die out when most of the prey has been eaten, giving the prey species a chance to recover. The cyclic nature of predator-prey systems can therefore be visualized. A finite automaton experiment is also included. The states and the letters of the input alphabet are modeled as molecules. State transitions are reactions of which the new state is the product.

'saces' is developed using Java 5 and JOGL (Java OpenGL) and is available for Windows, Mac OS X and Linux[3]. JOGL is a native wrapper which provides hardware-accelerated 3D graphics to Java applications. Experiments are saved as XML files and validated with XML Schema.

[3] 'saces' is tested only on Windows.

1.2 Fields of the Thesis

- Theoretical Computer Science and Artificial Chemistry
- Software Engineering: OOA and OOD, Profiling and Optimization
- Software Development: Java 5 and OpenGL
- Natural Science: Physics and Chemistry

1.3 Software and Hardware

Software Java 5 and JOGL.

Hardware Platform neutral, but rather ressource-intensive.

A video card with OpenGL hardware acceleration and enough RAM (around 256 MB or more) is recommended.

Wheel button mouse For zooming in and out of the simulation space.[4]

1.4 Acknowledgments

We would like to acknowledge the assistance of everyone who supervised or took the time to review our work. We especially thank Dr. Thomas Hinze from the TU Dresden, Dr. Peter Schwab from the Berne University of Applied Sciences, and Prof. Dr. Federico Flueckiger from the University of Applied Sciences of Southern Switzerland for their invaluable input. We would also like to thank Dr. Olivier Biberstein for his advice, Ariana Aguillon, Yvonne Hegi, Matthias Noelpp and Victor Senn for the critical reviews. Last but not least, we would like to thank our friends and relatives for their support and patience.

[4] There is a keyboard mapping, but using the mouse wheel is more intuitive.

Chapter 2

What is Artificial Chemistry?

2.1 Rationale

As the name suggests, an *artificial chemistry* is a constructed, artificial world. Laws of physics and chemistry work as prototypes for the rules applied in an artificial chemistry. There is therefore no attempt to emulate nature to the highest degree of detail.

The famous *Conway's Game of Life* [Sil05] follows very simple rules yet exhibits intricate patterns and forms of quasi-life when it is played. The contrast between the simplicity of the game's design, and the complexity resulting from it, illustrates the usefulness of simplifying complex processes in order to better understand them. The *Game of Life* is Turing-complete.[1] The rules of the *Game of Life* are abstract and detached from natural science laws. Artificial chemistry uses similar abstraction by borrowing the concepts of molecular interaction (in a reaction vessel), and applying these concepts to model other systems. It does not attempt to model the real world with physical realism, as does *computational chemistry*.

In [DZB01], page 227, a broad definition is given:

> An artificial chemistry is a man-made system that is similar to a real chemical system.

Some might ask: Why invent artificial worlds? What are the benefits of "fantasy" worlds with rules not applying to the real laws of physics?

[1] A computational or logical system is called *Turing-complete* if it has a computational power equivalent to a universal Turing Machine. In other words, the Game of Life can in principle compute everything a computer can compute.

One reason is simplification. Simple models are easily understood and therefore easy to use. Keeping the model as simple as possible, also helps in keeping the focus on the questions at hand, instead of on the model. There is also a certain educational value. Students or people not versed in a specialized and difficult subject can investigate simpler, more intuitive models, which in turn encourages their understanding for the real-world systems the models represent. Lastly, one can compare abstracted and strictly controlled models with real-world models (or with experiments) and be surprised about the similarities and differences between the two.

In computational chemistry some properties of molecules (reaction behaviour, molecular "shape", total energy, dipole moment etc.) are modeled with quantum-chemical computational methods. Software has been developed which is based on many methods that solve the molecular Schrödinger equation. For example a package of Fortran programs, the Molpro quantum chemistry package [WKL$^+$03], can perform accurate *ab initio*[2] calculations for single large molecules. Large-scale numeric calculation are needed for many of these methods.

Artificial chemistry, however, is more about speculative thinking than about a realistic model of the world. The aims and intentions in using artificial chemistry are more important. One might want to explore computation models like DNA or molecular computing. Can some problems from theoretical computer science be solved using artificial chemistry? How do the time and space complexities behave asymptotically?

In DNA computing a *Traveling Salesman* problem has been solved using real DNA [Adl94]. Adleman demonstrated with the experiment the feasibility of carrying out computations at the molecular level. It is therefore possible to solve other well-known Computer Science problems like the satisfiability of logic expressions, the so-called SAT problem, in this way.

2.1.1 The SAT Problem

To see the connection between artificial chemistry and real chemical reactions, we take the SAT Problem as an example. First we explain the problem, then what happens if we tackle it with a massively parallel computer using DNA- or chemical computing.

[2] In quantum chemistry or quantum mechanics *ab initio* is understood as the solving of the Schrödinger equation using only the "first principles", using only the constants of Natural Science, for example. For a more complete treatment of *ab initio* see [Sce04].

A logic expression consists of boolean variables joined together with conjunctions and disjunctions. Some variables can be negated. An example is:[3]

$$(x_1 \vee x_2 \vee x_4) \wedge\ (\neg x_3 \vee x_4 \vee \neg x_5) \wedge\ (\neg x_1 \vee \neg x_2 \vee (x_3 \wedge \neg x_5))$$

The question is: Is it possible to find an assignment to the variables so that the evaluation of the logic expression yields the value *true*?

The SAT problem is NP-complete. Algorithms to solve SAT are based on the full enumeration of all possible assignments and their verification.[4] We therefore have to try every possible solution before we can answer for sure: “No, this logic expression does not have a satisfiable assignment”. Computers of today are not very good at coping with the SAT problem. The SAT problem is important however, because many practical problems (database queries, Artificial Intelligence and expert systems, Electronic Design Automation etc.) depend on SAT.

The complexity (expense in either time or memory) doubles with every additional variable. Suppose we have a computer that can solve a SAT problem with 25 variables in about one second. If we give it a SAT problem with twice as many variables, namely 50 variables, it will need about a year to find an answer! (We assume about 30 nanoseconds for a step, about 33 million steps for 25 variables, and these 33 million steps squared ($1.12 \cdot 10^{15}$) for 50 variables.)

If we have a good computer using chemistry or molecular biology for its calculations, we can trade memory for time. Instead of trying out all possible solutions one after the other, we try out all possible solutions at once and extract the solutions we require. Such a computation model allows a massively parallel execution of algorithms. Information can be packed much more densely using molecules in a test tube than using electromagnetic states in RAM. An operation can be applied to all the molecules simultaneously, for example by pouring an reagent into the test tube.

There is, however, still an important limitation when trading time for memory.

The SAT problem has exponential complexity. If we added an additional variable to a SAT problem, we would double the problem space, because there would be twice as many possible solutions to verify. Suppose we buy a computer which works twice as fast. We would only be able to solve SAT problems with *one* more variable in the same time: a rather pathetic improvement.

[3] The vee operator $\vee$ is the disjunction (‘or’). The wedge operator $\wedge$ is the conjunction (‘and’). And $\neg$ is the negation operator (‘not’).

[4] The NP in NP-complete comes from non-deterministic polynomial. A non-deterministic Turing machine would solve all problems in NP in polynomial time. Equivalently a deterministic Turing machine can *verify* the solutions of all problems in NP, in polynomial time. See footnote for finite automaton on page 44 for further explanation.

If we had a computer based on chemistry working a million times faster, then we could add twenty more variables to the SAT problem (the dual logarithm of a million is roughly 20). This is better, but not much better. If we want to add 100 more variables, we would run into problems. Suppose we need 2^{100} water molecules with a molecular weight of around 18, then they would weigh

$$\frac{18 \cdot 2^{100}}{N_A} \approx 38 \cdot 10^6$$

grams or around 38 tons (N_A being Avogadro's number). With a few more variables we would need more water than there is available on the earth. This is the limitation of the memory for time tradeoff.

It is not known whether a more efficient algorithm for the SAT problem exists or not. One of the greatest unsolved problems of Mathematics and Computer Science is the *P versus NP* problem [Coo03], [Dev02]. Most theoretical computer scientists "believe" that there are no efficient (polynomial-time) algorithms for NP-complete problems. But this has not been proven yet.

A practical way to cope with the SAT problem is to use heuristics (backtrack search). In practice, the SAT problem can be solved in reasonable time for many instances [Nad02].

2.1.2 SAT and Artificial Chemistry

Suppose someone designs a working computer with DNA. With simple reasoning it has been shown that even massive parallelity with many DNA strands is not "enough". Massive parallelity will just push the limit of the problem size by a few dozen units.

This insight leads us to suspect that problems more complex than the one discussed above are perhaps better formulated using artificial chemistry, and therefore suggest the analyzing of computation models as one of the applications of artificial chemistry.

A 'saces' experiment has been developed for an artificial chemistry of a finite automaton (see section 5.4, page 41). Many, if not all, NP-complete problems can be "run" using finite automata. If a certain final state is reached, the question is answered with 'yes'. We could therefore create a representation of a SAT problem as a 'saces' experiment.

A finite automaton, however, is not Turing-complete.

2.2 The General Form of an Artificial Chemistry

A formal definition of an *artificial chemistry* is given in [SBB$^+$00] and [DZB01]. A model of a reaction vessel or a domain containing objects or molecules and of reactions inside the vessel is assumed. The definition is (see [DZB01], section 2.1, page 227):

> An *artificial chemistry* can be defined by a triple (S, R, A), where S is the set of all possible molecules, R is a set of collision rules representing the interaction among the molecules, and A is an algorithm describing the reaction vessel or domain and how the rules are applied to the molecules inside the vessel.

Let us have a look at the molecules, the reactions and the algorithm.

2.2.1 The Molecules

The elements $s \in S$ are molecules. They can be objects (like strings of characters A, C, G, T as highly abstracted models of DNA strands) or numbers to solve mathematical problems. To solve the SAT problem, one can use boolean arrays or binary strings. A yes/no value in such a string is an assignment to a variable in the logic expression. The set S might even be infinite (as the set of natural numbers). This does not mean that the population of molecules inside the reaction vessel itself is infinite. Only the “choice” of possible elements of S is infinite.

The molecules can have additional parameters like position or speed inside the reaction vessel. These parameters are used in the reaction rules.

2.2.2 The Reactions (Collision Rules)

R, the set of reaction or collision rules, describes the interactions between the objects in the domain. A rule $r \in R$ can be written in the same notation for chemical reactions:

$$s_1 + s_2 + \cdots + s_n \longrightarrow s'_1 + s'_2 + \cdots + s'_m \text{ where } s_i, s'_j \in S$$

The elements s_i are educts of the reaction and the elements s'_j products. The educts react with one another and are removed from the population in the vessel. After the reaction, the products are added to the population. The “$+$” sign is not a mathematical operator here, but a separator between the reagent symbols.

Rules can have conditions and, as molecules, additional parameters. The most important condition for a rule to apply, is that all educts of the rule must be available or have collided. An additional condition could be a reaction probability (the reaction is executed only if a random number in range $[0..1]$ exceeds the reaction probability). Another rule parameter or condition is the activation energy (the rule is activated only if the kinetic energy of the educt molecules exceeds the activation energy).

The set of rules R can be infinite like the set of molecules S. Infinitively many rules can be constructed by a meta-rule, as the rule $a + b \rightarrow (a/b) + b$, for example. a and b are natural numbers and a is divisible by b. The products are the new natural numbers a/b and b. Such a meta-rule is useful for an artificial chemistry designed for finding *prime numbers* (see [SBB^{+}00], page 13).

2.2.3 The Algorithm

The algorithm determines which molecules react with one another, what is done with the reaction products and how to treat molecule and reaction parameters. A very simple artificial chemistry does not require the notion of space. Molecules which “collide” with one another are selected randomly and reaction rules are applied to the collided molecules. ‘saces’, however, uses a three-dimensional reaction vessel and molecules collide when their trajectories intersect.

Another approach is to use differential equations. Reaction rules are reformulated to contain stoichiometry factors. Such a rule can be understood as a recipe: “Add two parts of agent A, three parts of agent B etc., and you get one part of agent X and three parts of agent Z”. Instead of acting on single collisions, the algorithm calculates the concentration of the agents in the reaction vessel with differential equations (see [DZB01], page 229).

There are many different approaches to letting molecules react and to handle reaction products. Sometimes an alternative definition of artificial chemistry makes more sense, namely a tuple (S, I), where S is the same as before, a set of molecules and I a description of the interactions among the molecules. This definition avoids a separation between reaction rules and the algorithm if they are coupled tightly.

2.3 Applications in Artificial Chemistry

Because of artificial chemistry’s fairly elevated level of abstraction, it becomes possible to model a wide palette of formal problems. These models include selforganization and evolution, automatic proof, chaotic systems, graph rewriting and DNA computation.

Applications in artificial chemistry can generally be classified into three types: Information processing, modeling and optimization. All three application types rely on the one significant aspect of artificial chemistry which keeps it bound to natural chemistry, namely the metaphor of interacting molecules which are capable of recombination and procreation (the creation of new molecules).

2.3.1 Information Processing

We will look at two examples for information processing: *DNA Computing* and *Graph Rewriting*.

DNA Computing

DNA computing uses real DNA molecules for calculation. The data density of DNA and its double stranded nature, allows the possibility of massively parallel computation. This was demonstrated by Adleman in 1994 [Adl94] when he solved a *Traveling Salesman* problem using real DNA. Not only did this illustrate the possibilities of using DNA to solve a class of problems which are tedious to solve using traditional computing, it also demonstrated the unique characteristics of DNA when used as a data structure.

The approach usually taken when modeling DNA computing, is the so-called *Adleman-Lipton* paradigm, an extension of Adleman's work. This modeling technique consists of three basic steps, namely: The molecules are initialized according to the problem at hand (preparation phase). The molecules are then mixed together to generate solution candidates (assembly phase), and finally the solutions are picked out from among these candidates (detection phase). This basic algorithm exploits the massive parallelism of biochemical reactions to perform an exhaustive search.

Information processing applications in artificial chemistry can be stripped of the physical realism, assuming only the metaphor of molecular interaction. In [Tom04], an artificial chemistry for DNA computation is modeled using the *Adleman-Lipton* paradigm.

Graph Rewriting

Graph rewriting is also refered to as *graph grammars* in the context of formal language systems. The term *rewriting* used in computer science and logic covers a wide range of non-deterministic methods of replacing subterms of a formula with other terms. This is practical in equation solving. *Rewrite systems* being non-deterministic, do not dictate a strict substitution rule the way an algorithm would, but offer a set of possible substitution for each term. This is where the combinatorial properties of an artificial chemistry come in handy.

Graph rewriting is in itself abstract and can be applied to various models, some of which have already been investigated. One of these is the investigation of chemical networks using an artificial chemistry based on graph rewriting [BFS03]. Large chemical reaction networks were generated and their generic properties analyzed. Again, *generic* emphasizes the adequacy of an artificial chemistry for this kind of analysis.

2.3.2 Modeling

Modeling applications include the above mentioned investigation of complex and dynamic systems such as biological or evolutionary systems, social systems, consumer-producer dynamics (predator-prey systems) and parallel processes. An example of such an application is a *Lotka-Volterra* experiment for ‘saces’. The experiment models predator-prey interaction (see section 5.5, page 44).

2.3.3 Optimization

Optimization is closely related to evolutionary computing because both can be reduced to combinatorial problems, problems easily modeled by an artificial chemistry. Optimization consists of picking the best possible solution out of a set of candidates which in turn is a tool used by evolution to make a system sustainable. The *Traveling Salesman Problem* is an example of such an optimization problem. The optimal solution for such problems can only be obtained (in classical terms) using an exhaustive search. Many optimization problems are in fact NP-complete, making artificial chemistry an interesting approach when investigating possible ways in which to tackle them.

2.3.4 Summary

As we have seen in this chapter, applications in artificial chemistry are numerous and diverse in what they are capable of modeling. Most, if not all, of the applications presented here rely on simplified models which, despite their simplicity, result in complex behaviour. This is a trait common to complex and dynamic systems which can be studied with artificial chemistry. This is where simplification and the setting of limits become essential to the modeling process. Making assumptions or even educated guesses about a complex system is usualy more practical than attempting to model the system with utmost realism.

This paradigm is a very powerful analytical tool because its application does not lie in chaos, nor in order, but in the very colorful realm between the two. This is where real world problems originate, and where life itself originates. The simplification and limiting aspects of the modeling concepts described here are also fundamental to the paradigms offered by *Systemics* [NBH01].

Investigating the dynamics of complex systems such as life employs the concept of *emergence*. Emergence can be described as deducing a system's global properties by observing the interaction of the system's components. Again, the interaction is emphasized and not the component itself. This is because local interaction between components, each following certain simple rules autonomously, can effect the system's global behaviour, leading to the *emergence* of global properties. Directing investigation toward the system's components instead of their interaction would obscure the macro view completely. Simply put, a complex system such as life, owes much of its being (i.e. system properties) not to the properties of its components, but to the mutual function of its components. The basic characteristics of artificial chemistry therefore reflect the idiosyncrasies of life itself.

The suitability of artificial chemistry for modeling such systems lies in its ability to idealize a system to an extent at which all of the system parameters become controllable. By defining a distinct imaginary border around the core parameters of a complex system, the system can be investigated using formal methods. The results of such investigations, although based on assumption rather than fact, may yield surprisingly realistic results when compared to the empirical values obtained through observation of the complex system as a whole.

Chapter 3

Proposal of ‘saces’

As a part of this work, a small and simple tool is proposed: ‘saces’. The name is an acronym for *Simple Artificial Chemistry Experiment System* and is the code-name of the project. It does three-dimensional visual simulations of ideal gas processes. It is useful as both an educational tool and as an application in *artificial chemistry*.

3.1 Fact Sheet

The tool ‘saces’

- is a moderately realistic simulation of an ideal gas undergoing reactions,
- has a three-dimensional smooth visual animation of the chemical process,
- is useful as a simple educational tool and as an application of artificial chemistry,
- allows the specification of particles and chemical reactions,
- uses a space partioning algorithm for almost linear-time collision detection between molecules,
- uses Monte-Carlo methods for collision response and reactions,
- is developed in Java 5 using JOGL (Java OpenGL, see [Dav04]),
- allows replacement of steps in the simulation loop (Collision detection and response, handling reactions etc., see section 3.4 on page 22),

- runs on Windows, Linux and Mac OS X 10.4,
- is recommended to be used with a video card capable of OpenGL hardware acceleration and enough system RAM (256 MB or more).

Further technical details are described in chapter 6, page 48ff.

3.2 The Position of ‘saces’ in Artificial Chemistry

‘saces’ acts on a middle ground between the high abstractions of “true” artificial chemistry and the highly realistic models of computational chemistry. We wanted to simulate certain simple chemical processes and at the same time provide an appealing and instructive visual display. Some corners had to be cut, or in other words, we had to introduce abstractions and simplifications necessary to reach adequate animation frame rates.

Nevertheless, it is possible to play through artificial chemistry scenarios as well (see chapter 5, page 37).

Abstractions and Simplifications

1. The reaction vessel or test tube has flat rectangular walls. The space inside the reaction vessel has the form of a cuboid (see figure 3.1).

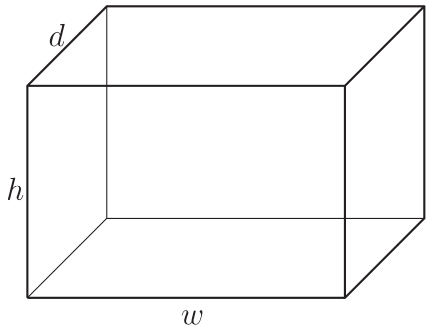


Figure 3.1: The shape of the reaction vessel in ‘saces’

2. Molecules are hard spheres. Collisions are elastic (except if reactions apply).

3. Ideal gases are assumed. There are no van der Waals forces and other intermolecular forces.

4. No quantum chemistry.

5. Collision detection in ‘saces’ is not water-tight. A few percent of all collisions are not detected. Particles overlapping to neighboring partitions are ignored in these partitions (see chapter 6, page 74).

6. A problem is the quasi-simultaneous collision of more than two particles. Particles handled after the first collision are ignored or handled in the next simulation loop iteration.

7. Fast particles might pass through each other before collision detection has the chance to detect a penetration. Collision detection only occurs at certain points in time (around 50 to 5 times per second).

8. Collision response determines the reflection angle randomly. Rutherford or other sophisticated reflection models are not implemented. (They could be programmed as an extension to ‘saces’, but perhaps at the expense of animation performance.)

9. The following types of reaction equations are supported:

   Transform of form $E_1 + E_2 \rightarrow P_1 + P_2$ (two educts transform to two products),

   Merge of form $E_1 + E_2 \rightarrow P$ (two educts merge to one product) and

   Decay of form $E \rightarrow P_1 + P_2$ (an educt decays spontaneously to two products).

   More complicated reactions can be composed out of these simple equations.

Most of these simplifications arise from the need to do the calculations at runtime. For details on how the calculations are implemented see chapter 6, page 48ff.

3.3 The Artificial Chemistry of ‘saces’

We analyze ‘saces’ according to the definition of artificial chemistry using the triple (S, R, A) as explained in the general form of artificial chemistry.

3.3.1 The Molecules

The elements of the set S are moving particles in a test tube. The particles are represented by spheres. They have the following parameters:

- position $\vec{p}$
- velocity $\vec{v}$
- its “species”, namely the particle class with the parameters
  - mass m,
  - bound energy E,
  - sphere radius r and
  - a display name and color

The initial number of particles and their particle classes are determined by the experiment configuration. The set S is finite, provided we ignore the molecule parameters.

3.3.2 The Reactions

The rules have one or two educts and one or two products with the following parameters:

- reaction probability p_r and
- activation energy E_r

A reaction rule is applied only if the kinetic energy of the colliding molecules exceeds the activation energy, and if the generated random number in the range $[0..1]$ exceeds the probability p_r. The set R is finite as well.

When considering energy conservation, an additional condition applies. Suppose the reaction is endothermic[1], and two fast particles do not collide head-on but slightly graze each other instead. In this case it is possible, that there is still not enough energy in the mass center reference frame. The impetus of the collision is not enough (see section 6.4.9, page 70).

3.3.3 The Algorithm

The algorithm takes into account the velocities and the positions of the molecules. It performs collision detection and collision response for each particle. The list of collision pairs is given to the reactions to be handled. A special case are decay reactions which occur quasi-spontaneously (using random numbers).

The visualization of the moving particles is not part of the definition of the artificial chemistry algorithm.

3.4 Main Features of ‘saces’

The most important parts of the tool are:

- A three-dimensional view of the rectangular reaction vessel
- The simulation process
- A data viewer window to view diagrams and a binary logger
- A dialog window to edit the experiment settings and chemical reactions

The Three-dimensional View of the Reaction Vessel

The main window displays a three-dimensional view of the simulation. The bounding box of the reaction vessel is painted. Molecules are animated within the reaction vessel (see figure 1.1, page 6). Two tool bars with buttons appear at the bottom and to the right of the main view. The user can start or stop the simulation, edit experiments (click on 🔧 ‘Settings…’), change the perspective etc.

[1] If the product particles have a total bound energy higher than the educt particles, then the energy difference is negative and the reaction endothermic. In the detonating gas example (see page 38) there are two reactions: a reaction producing water (steam) and freeing a lot of bound energy and the inverse, endothermic reaction.

The user can zoom in and out using the mouse wheel or arrow keys, and rotate the vessel using the virtual track ball[2] (drag the mouse inside the view).

The Simulation Loop

The simulation loop calculates the simulation parameters, displays the particles and saves data to a binary file. The simulation loop consists of the following steps:

1. Reposition the particles and reflect on the reaction vessel walls
2. Measure and log pressure, temperature etc.
3. Detect particle collisions
4. Apply merge and transform reactions, if necessary
5. Calculate collision response for the remaining collision pairs
6. Apply decay reactions, if necessary
7. Paint the scene

For a complete treatment of not only the simulation loop, but the whole simulation process, see section 6.3.2, page 56.

The Data Viewer and the Binary Log

Thermodynamical data such as temperature and pressure, mechanics such as particle count, a histogram of particle speeds and other data are presented as diagrams in the data viewer. The data viewer can be started as a separate process, or even on another workstation, if the binary log file is stored on the network. The binary log is used to transfer data to the viewer (for more details, see section 6.7, page 76).

The Experiment Settings Dialog Window

The experiment settings dialog allows the user to load, edit and save experiments (for more details, see the manual, page 27).

[2] The virtual track ball is a known three-dimensional user interface attributed to [CMS98]. See section 7.3, page 81 as well.

Chapter 4

The User Manual

This chapter is a description of the ‘saces’ user interface. We will begin with first steps to get the reader acquainted with ‘saces’. We will then describe the windows and their controls in more detail.

4.1 First Steps with ‘saces’

Getting ‘saces’

‘saces’ is delivered as a *.jar* file for all platforms. It is available on the ‘saces’ CD or on the project homepage `http://saces.yce.ch`. An executable for Windows *saces.exe* is available.

Java Runtime Environment Version 5

‘saces’ works with Java versions 5 and upward only. Java version 5 is available for download at `http://java.com/en/download/manual.jsp`. This is also necessary when using the Windows executable.

Getting JOGL (Optional)

Note! This step is not necessary when using the Windows executable. It is only required when using the *.jar* file.

‘saces’ uses the library file *jogl.jar* from JOGL and native libraries, which are different for each platform. They are available on the JOGL homepage or on the ‘saces’ CD.

1. Go to `https://jogl.dev.java.net` and download `jogl.jar`. Do not download the “Current nightly build”, but go to the “Downloads” section, click “Precompiled binaries and documentation” and select a release. Release 1.1 from June 2005 is stable. It is also available in the *jogl-1.1* subdirectory on the ‘saces’ CD.

2. Download the `jogl-natives-`<*platform*>`.jar` file, or use the libraries from the ‘saces’ CD.

3. **Windows** users unpack the file using *WinZip* and copy the *.dll* files to the same directory as the `saces.jar` file. For JOGL release 1.1 they are `jogl.dll` and `jogl_cg.dll`.

   Linux users enter `unzip jogl-natives-linux.jar` and copy the resulting *.so* files to the same directory as the `saces.jar` file. For JOGL release 1.1 they are `libjogl.so` and `libjogl_cg.so`.

   Mac OS X users start **Terminal** and enter:
   `unzip jogl-natives-macosx.jar` and copy the resulting *.jnilib* files to the same directory as the `saces.jar` file. For JOGL release 1.1 they are `libjogl.jnilib` and `libjogl_cg.jnilib`.

   Note that the `_cg` files are part of JOGL, but optional for ‘saces’.

Starting ‘saces’

Double-click the program icon.

A window with a black empty background appears. This is the main view. Click the ▶ ‘Run’ button at the bottom. A sample experiment is started. Molecules are displayed as colored spheres of various sizes. OpenGL lighting is applied to give a more realistic view.

Stopping the Simulation and Loading Another Experiment

Click the 🔧 ‘Settings…’ button while the simulation is running. This stops the simulation and shows the settings window.

Click the ‘Load. . . ’ button and load the *SimpleFiniteAutomaton.xml* sample experiment file. The file is available from the ‘saces’ CD or can be downloaded from the project homepage `http://saces.yce.ch`.

Tweaking the Experiment

Change the reaction vessel wall colors to a different color (click on ‘Box Color’ input field).

Change the initial count of the molecule R (click on the ‘Particle Classes’ tab and select the ‘Initial Count’ column of the row ‘R’ and edit the number). If you like, change the color of the particle, too.

Change the probability of the reaction $R + a \rightarrow S$ (click the ‘Reactions’ tab and set the probability of the reaction to a number between 0 and 1).

Starting the New Experiment

You can save the modified experiment or start the simulation right away (click the ‘Run’ button). The settings window disappears and the new experiment is started.

View Diagrams

Click the ‘View Data. . . ’ button, and you have a selection of diagrams. You can view how the number of particles of each class change over time, the speed distribution etc.

Dump the Binary Log

The binary log contains the data for the diagrams in binary form. To dump it in a form usable for further processing by other software, open a shell (DOS Prompt on Windows, Terminal on Mac OS X) and enter:

```
java -cp jogl.jar -jar saces.jar -tail SimpleFiniteAutomaton.xml
```

This dumps the contents of the binary log in text form. It works like the Unix `tail -f` command. This means the program waits at the end of binary log for more data from the running simulation. It can be cancelled using Control-C. If no data shows up, verify that the tail dumper is reading the right binary log.

The Snapshot

The snapshot saves velocity, position and corresponding particle class of all particles currently in the simulation in the binary log and is available for further processing. Click the ‘Snapshot. . . ’ button to make a snapshot.

4.2 The Main Window

The ‘saces’ main window consists of two toolbars and the display area containing the animation. The toolbar to the right controls a running simulation and the toolbar at the bottom enables access to the application and experiment settings.

Icon	Button	Function
▶/■	Run/Stop	Start or halt the simulation.
0:00	Reset	Reset the simulation to the beginning.
◻	Default Perspective View	Set view to the default perspective.
●	Particle View	Set view to the ‘Be a particle’ perspective: the point of view moves with a randomly selected particle.
⊞	Orthographic View	Set orthographic view with three outline views.
+	Speed Up	Speed up the animation.
−	Slow Down	Slow down the animation.
●	Low Detail	Set the graphical detail level to low (only edges).
●	Medium Detail	Set the graphical detail level to medium (no lighting).
●	High Detail	Set the graphical detail level to high (with lighting).

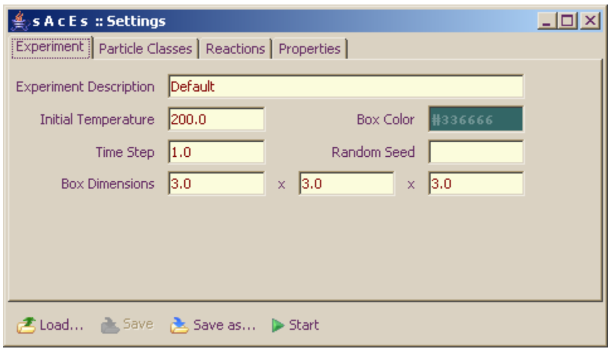
Table 4.1: Buttons of the vertical toolbar to the right

Icon	Button	Function
▶/■	Run/Stop	Start or halt the simulation.
◙	Snapshot	Save a snapshot to the binary log. A snapshot contains velocity, position and particle class of all particles.
🔧	Settings. . .	Open the settings dialog window to edit an experiment.
▲	View data. . .	Start the data viewer with a selection of diagrams.

Table 4.2: Buttons of the horizontal toolbar at the bottom

The user can zoom in and out of a running simulation with the mouse wheel or the cursor keys (cursor up and down) and rotate the perspective by dragging the mouse (except in the orthographic viewing mode). See also section 7.3, page 81 about the perspective view in OpenGL.

4.3 The Settings Dialog



Experiment Description: Default

Initial Temperature: 200.0

Box Color: #336666

Time Step: 1.0

Random Seed:

Box Dimensions: 3.0 x 3.0 x 3.0

Figure 4.1: Experiment settings window

With the settings dialog experiments can be loaded, saved and edited. There are four tabs: **Experiment**, **Particle Classes**, **Reactions** and **Properties** and a toolbar at the bottom.

Icon	Button	Function
📂	Load. . .	Load an experiment XML file.
📂	Save	Save back the edited experiment.
📂	Save as. . .	Save an experiment.
▶	Start	Start the experiment and close the dialog.

Table 4.3: Settings dialog toolbar buttons

Note the input validation of experiment data. It is not possible to leave an input field containing invalid data, like negative masses for particle classes, for example.

The experiment tab

The experiment tab allows to edit global experiment parameters.

Experiment Description A short description of the experiment.

Initial temperature The initial temperature with which the experiment will start (if Maxwell-Boltzmann distribution is used, see section 6.4.2, page 61). Must be non-negative. The value can be interpreted as a temperature in Kelvin, if the constants k (Boltzmann constant) and R (ideal gas constant) are configured accordingly. See section 6.4.5, page 63.

Time Step The simulation time step is a factor that influcences the speed of the simulation. Unlike the slow down and speed up functions which stay proportional to time (simulation time slows down and speeds up as well), the step factor specifies how many steps are made per iteration. The default here is the value 1, meaning that one iteration is equivalent to one step. Setting the value to 0.5 would half the distance a particle has moved in one iteration. This is a method for speeding up the simulation, independent of time.

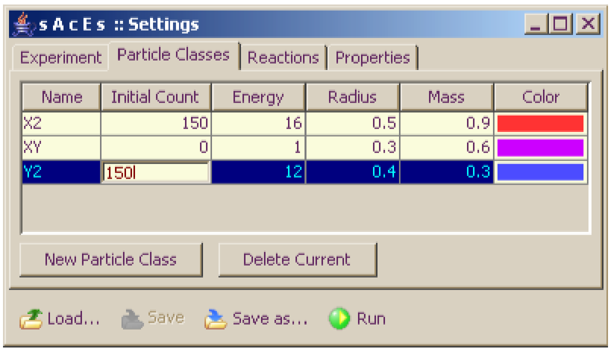
Box Color The color of the reaction vessel (bounding box). The user can click on the color field to access a standard color picker dialog.

Box Dimensions The width, height and depth of the reaction vessel (bounding box). Values for width, height and depth are usually between 1 and 50, although this is not a must. The bounding box volume must be large enough to accommodate the particles defined in the particle class tab.

Random Seed The random seed is used by the simulation when generating random numbers. The random seed parameter is optional. If a seed is specified, the experiment becomes repeatable. If not, a new random seed is calculated every time anew.

The Particle classes tab

The particle classes tab lets the user define the particle classes whose particles are to be simulated. There is a table with rows for each particle class and columns for the properties of them.



Name	Initial Count	Energy	Radius	Mass	Color
X2	150	16	0.5	0.9	🟥
XY	0	1	0.3	0.6	🟪
Y2	150	12	0.4	0.3	🟦

Figure 4.2: Experiment settings window: editing the particle classes

Name Define an unique name for a particle class.

Initial Count Define the initial particle count for a given particle class. Must be a non-negative integer.

Energy The bound energy of the particle class. Particles with high bound energy have high 'energy content'. Must be a positive number.

Radius The radius of the particle sphere. Note: If the distributor finds that the reaction vessel is too small (or the particles too big), an error message is shown with further instructions (like making the particles smaller). Must be a positive number.

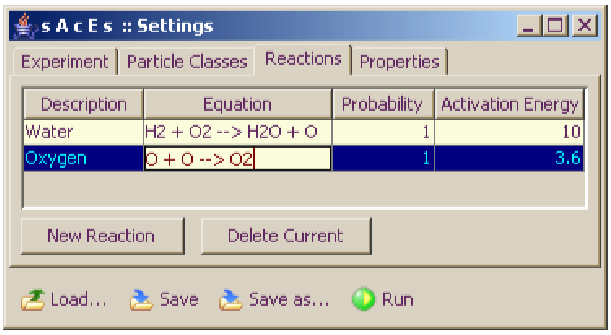
Mass The mass of the particle. Must be a positive number.

Color The color of the particle. Note that OpenGL lighting might produce color shades slightly different to the selected color.

To create a new particle class click **New Particle Class**. A new row appears in the table. To delete a particle class, select the row and click **Delete Current**. Note that the particle class must not be involved in reactions. Delete the reactions first, then the particle class.

The Reactions tab

Reactions are entered as equations in the form of: $\mathrm{O} + \mathrm{O} \rightarrow \mathrm{O}_2$. There is a table with rows for each reaction and columns for the properties of them.



Description	Equation	Probability	Activation Energy
Water	H2 + O2 --> H2O + O	1	10
Oxygen	O + O --> O2	1	3.6

Figure 4.3: Experiment settings window: editing reactions

Description A short description of the reaction.

Equation The reaction equation. There must be one or two particle class names (separated by the plus sign) before and after the arrow (the string ->), and at least three particle classes must be used. The particle classes must be already defined in the particle classes tab.

Probability The probability $p \in [0..1]$ of a reaction. Note that multiple reactions applying to the same particle classes before the arrow (the educts) are accepted by 'saces', but must be defined carefully. If, for example, the first reaction has the probability of 1, the following reactions of the same educts are never activated.

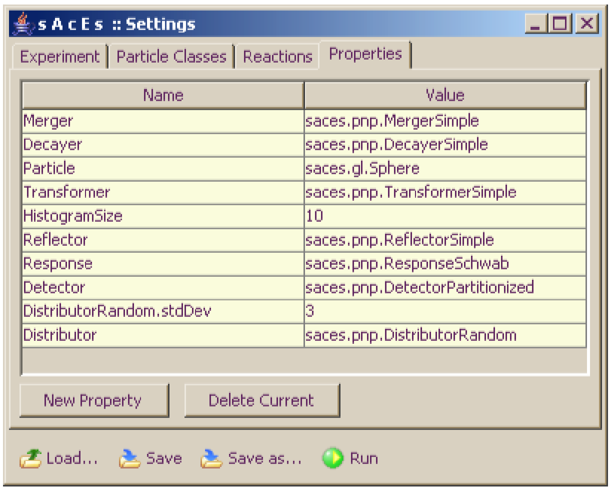
Activation Energy The minimal combined kinetic energy of the educts which the reaction requires to come into effect. Activation energy is optional, but if entered, it must be a positive number.

To create a new reaction click **New Reaction**. A new row appears in the table. To delete a reaction, select the row and click **Delete Current**.

The Properties Tab

Additional experiment settings are defined as name-value pairs, the experiment properties. Table 4.6, page 35 at the end of the chapter, lists all experiment properties of ‘saces’. The simulation plug-and-play uses the properties to find implementations of the simulation steps (see section 6.1.5, page 49), for example.

There are default values for many properties. If a property is not defined, ‘saces’ assumes the default.



Name	Value
Merger	saces.pnp.MergerSimple
Decayer	saces.pnp.DecayerSimple
Particle	saces.gl.Sphere
Transformer	saces.pnp.TransformerSimple
HistogramSize	10
Reflector	saces.pnp.ReflectorSimple
Response	saces.pnp.ResponseSchwab
Detector	saces.pnp.DetectorPartitionized
DistributorRandom.stdDev	3
Distributor	saces.pnp.DistributorRandom

Figure 4.4: Experiment settings window: editing properties

The table in the properties tab with rows for each experiment property has only two columns:

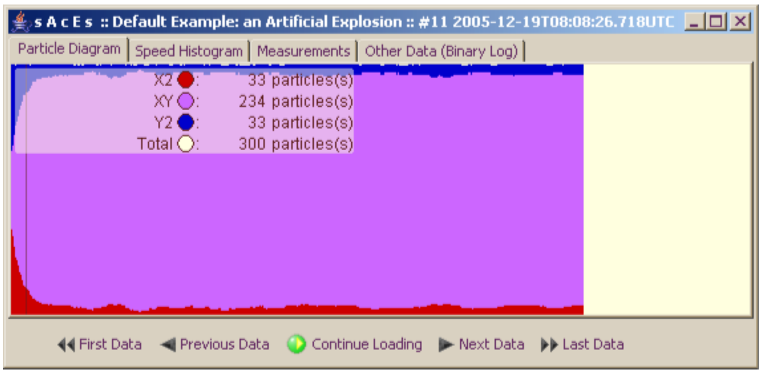
Name Define an unique name for a property.

Value The property value. Can be a string (like a class name for a plug-and-play property), a number (like the standard deviation for *DistributorRandom*) or a boolean value (to enable or disable validation of mass conservation).

To create a new property click **New Property**. A new row appears in the table. To delete a property, select the row and click **Delete Current**.

4.4 Data Viewer

The data viewer visualizes the data stream of a running simulation in form of diagrams. The data stream is read continuously from the binary log, split up and displayed as different sequences of data instances in diagrams. The particle diagram, for example, is a sequence of particle counts per partition. With the toolbar buttons it is possible to navigate forward and backwards in the sequence.



sAcEs :: Default Example: an Artificial Explosion :: #11 2005-12-19T08:08:26.718UTC

X2 ●:	33 particles(s)
XY ○:	234 particles(s)
Y2 ●:	33 particles(s)
Total ○:	300 particles(s)

Figure 4.5: Particle diagram of the data viewer

Icon	Button	Function
◀◀	First Data	Jump to the first data instance of the binary log.
◀	Previous Data	Move to the previous data instance in the binary log.
●/■	Continue/Stop Loading	Continue/Stop loading binary log.
▶	Next Data	Move to the next data instance in the binary log.
▶▶	Last Data	Jump to the last data instance of the binary log.

Table 4.4: Settings dialog toolbar buttons

The available views or diagrams are:

Particle Diagram Display the progression of the particle count for each particle class in time. The thin vertical line indicates at what time (position on the time axis) the displayed data was recorded. The exact counts are displayed in the upper left corner in a semi-transparent pane. See figure 4.5 above.

Speed Histogram Display the particle speed distribution for a given time. The speeds on the x axis are subdivided into smaller intervals. The y axis is the count of the particles whose speeds fall into the interval. The vertical line shows the average speed. Also displayed are the exact minimum, average and maximum speeds.

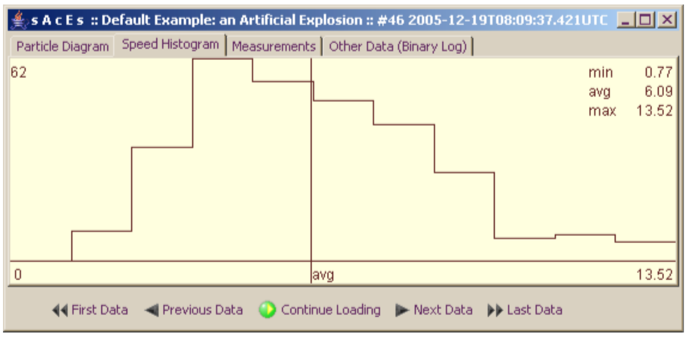


Figure 4.6: Speed histogram of the data viewer

Measurements Display temperature and pressure progression in time. The thin vertical line indicates at what time (position on the time axis) the displayed temperature and pressure was recorded.

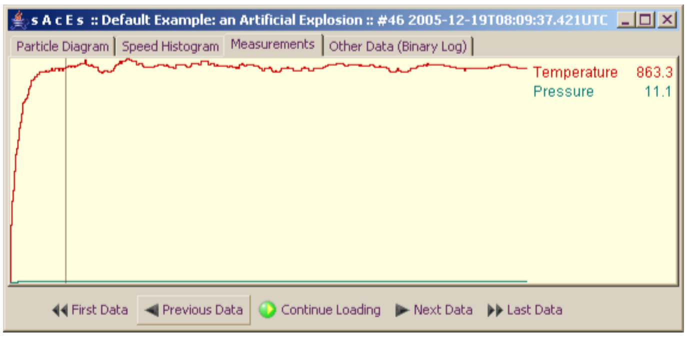
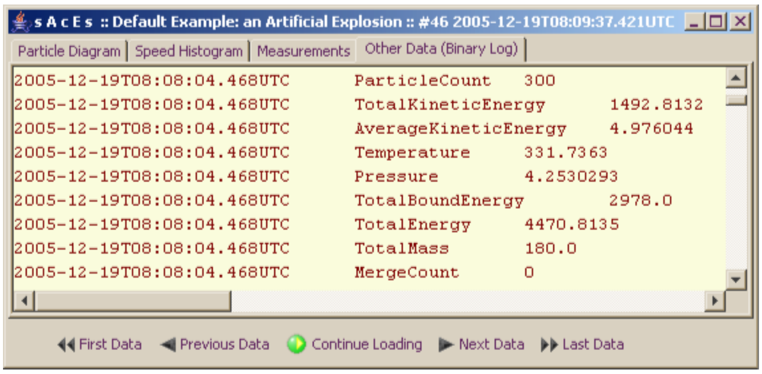


Figure 4.7: Measurements diagram of the data viewer

Other Data (Log) A lot of data in the binary log is saved as key-value pairs. For example, how the reaction vessel is being partitionized for an efficient collision detection (see section 6.6, page 74) is saved as a series of key-value pairs. They can be viewed in the view. All data instances have their own time-stamp printed in the UTC timezone. **Note** that the navigation buttons do not have an effect in the other data view.

Running the *Data viewer* and the simulation on the same host decreases animation performance. To avoid this, start a view-only instance of ‘saces’ with the command-line option *-view* from another workstation.



```
2005-12-19T08:08:04.468UTC      ParticleCount    300
2005-12-19T08:08:04.468UTC      TotalKineticEnergy      1492.8132
2005-12-19T08:08:04.468UTC      AverageKineticEnergy    4.976044
2005-12-19T08:08:04.468UTC      Temperature      331.7363
2005-12-19T08:08:04.468UTC      Pressure         4.2530293
2005-12-19T08:08:04.468UTC      TotalBoundEnergy        2978.0
2005-12-19T08:08:04.468UTC      TotalEnergy      4470.8135
2005-12-19T08:08:04.468UTC      TotalMass        180.0
2005-12-19T08:08:04.468UTC      MergeCount       0
```

Figure 4.8: Other data view of the data viewer

4.5 Command Line Arguments

(no arguments)	Load the default experiment.
-load *experiment.xml*	Load the specified experiment.
-view *experiment.xml* **[***saces.binlog***]**	Start the data viewer only (optionally using another binary log than specified in the experiment).
-tail *experiment.xml* **[***saces.binlog***]**	Dump the binary log of the experiment (works like the Unix `tail -f` command).

Table 4.5: Command line arguments to ‘saces’

Note that the command-line arguments only work with the *.jar* executable. To load the experiment *SimpleFiniteAutomaton.xml*, invoke (assuming the experiment file is in the same directory as the files *saces.jar* and the native library):

```
java -cp jogl.jar -jar saces.jar -load SimpleFiniteAutomaton.xml
```

Name	Default	Meaning
`BinaryLog`	`saces.binlog`	The file name of the binary log file.
`BoltzmannConstant`	1	The Boltzmann constant k (see page 62).
`Decayer`	`saces.pnp.DecayerSimple`	Name of Java class to handle decay reactions. The class must implement the interface `saces.pnp.Decayer`.
`Detector`	`saces.pnp.DetectorPartitionized`	Name of Java class to provide the collision detection. The class must implement the interface `saces.pnp.Detector`.
`DetectorPartitionized.particleCountPerPartition`	12	A hint to the partitionizer about how many partitions are to be created. The value is the average count of particles in each partition (see section 6.6 page 74).
`Distributor`	`saces.pnp.DistributorMaxwellBoltzmann`	Name of Java class to provide the initial distribution of the particles in an experiment. The class must implement the interface `saces.pnp.Distributor`.
`DistributorRandom.stdDev`	1	A parameter for the class `saces.pnp.DistributorRandom`, namely the standard deviation for the speeds distributed to the particles.
`HistogramMax`	0	The maximum value of the histogram. If it is 0 (zero), use the maximum speed encountered so far in the simulation.
`HistogramSize`	100	The number of histogram steps (the resolution of the histogram).
`IdealGasConstant`	1	The ideal gas constant R (see page 62).
`MeasureInterval`	2000	The time in milliseconds between measurements and histograms.
`Measurer`	`saces.pnp.MeasurerDefault`	Name of Java class to provide the measurements of temperature, pressure etc. The class must implement the interface `saces.pnp.Measurer`.

Table 4.6: Experiment properties and their meaning

Name	Default	Meaning
`Merger`	`saces.pnp.MergerSimple`	Name of Java class to handle merge reactions. The class must implement the interface `saces.pnp.Merger`.
`Particle`	`saces.gl.Sphere`	Name of Java class which implements a particle. The class must be able to render the particle in OpenGL.
`Reflector`	`saces.pnp.ReflectorSimple`	Name of Java class to provide the reflection of particles at the reaction vessel walls. The class must implement the interface `saces.pnp.Reflector`.
`Response`	`saces.pnp.ResponseSchwab`	Name of Java class to provide the collision response. The class must implement the interface `saces.pnp.Response`.
`Transformer`	`saces.pnp.TransformerSimple`	Name of Java class to handle transform reactions. The class must implement the interface `saces.pnp.Transformer`.
`ValidateActivationEnergy`	*false*	Is activation energy validated? If yes and a reaction's negative bound energy difference is greater than activation energy, an error message is shown.
`ValidateMassConservation`	*false*	Is mass conservation validated? If yes and a reaction violates mass conservation, an error message is shown.

Table 4.6 continued

Chapter 5

Some Experiments with ‘saces’

5.1 The Default Experiment, an Artificial Explosion

When starting ‘saces’ a sample experiment is loaded and initialized. This is very helpful for first-time users because they can start “playing” immediately, getting a feel for the application. This is the “Batteries Included” philosophy: The user is not required to do any additional preparations (buying batteries), to get a first positive experience: It works out of the box!

The sample experiment is a very simple artificial explosion with two invented atoms **X** and **Y** and the molecules $\mathbf{X}_2$, $\mathbf{Y}_2$ and **XY**. The first two have high bound energies and if they collide they react as follows:

$$\mathbf{X}_2 + \mathbf{Y}_2 \longrightarrow 2\mathbf{XY}$$

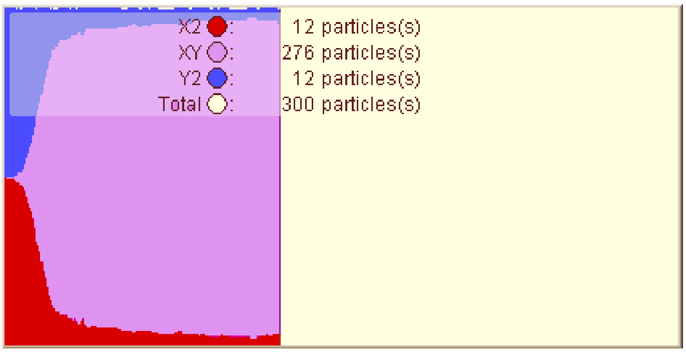
XY has low bound energy. The bound energy difference is converted into kinetic energy. Because there is a probability of 50% and an activation energy, the explosion does not start immediately. Only very few particles are fast enough to trigger the first reaction when they collide. The reaction products move away at high velocities. They collide with other particles, until more particles acquire enough velocity to react. A chain reaction starts. Many reactions happen in a short time. The average speed increases massively and with it the average kinetic energy. In the beginning it is around 0.23 and after a minute, when an equilibrium has been reached, it is at 11.8 (or around 50 times more). This would correspond to a temperature increase from 200 Kelvin[1] to 10000 Kelvin.

[1] The Kelvin is an unit of temperature and is measured with respect to the absolute zero, where molecular motion stops. The melting point of water ice at 0°C is 273.15K.

There is the inverse reaction:

$$2\mathbf{XY} \longrightarrow \mathbf{X}_2 + \mathbf{Y}_2$$

which can occur only if the two **XY** molecules collide with much impetus. After a while an equilibrium is reached (see figure 5.1).



X2 ●: 12 particles(s)
XY ○: 276 particles(s)
Y2 ●: 12 particles(s)
Total ○: 300 particles(s)

Figure 5.1: Particle diagram of the artificial explosion

A few percent of the $\mathbf{X}_2$ and $\mathbf{Y}_2$ molecules remain, because they are recreated by the inverse reaction.

5.2 Detonating Gas

We are trying to model detonating gas in ‘saces’. Detonating gas is a hydrogen-oxygen mixture, so that every two molecules of hydrogen $\mathbf{H}_2$ meet one molecule of oxygen $\mathbf{O}_2$. It explodes violently on ignition, forming water. It has a very high energy content per unit of weight. The reaction could be written as:

$$2\mathbf{H}_2 + \mathbf{O}_2 \longrightarrow 2\mathbf{H}_2\mathbf{O}$$

In reality the explosion is a complex set of reactions. We model, however, a much simpler set of reactions, involving atomic oxygen as follows:

$$\begin{aligned} \mathbf{H}_2 + \mathbf{O}_2 &\longrightarrow \mathbf{H}_2\mathbf{O} + \mathbf{O} \\ \mathbf{O} + \mathbf{O} &\longrightarrow \mathbf{O}_2 \end{aligned}$$

We assume that the molecules $\mathbf{H_2}$ and $\mathbf{O_2}$ have high bound energy, and the atom **O** even a higher bound energy; water has low bound energy:[2]

O	200
$\mathbf{H_2}$	80
$\mathbf{O_2}$	120
$\mathbf{H_2O}$	10

Similar to the artificial explosion, after some delay the first few reactions are triggered. These reactions do not set energy free, because atomic oxygen is generated. The bound energy difference $80+120-(10+200) = -10$ is even negative. Only the second reaction is able to free energy. Because it is a merge reaction, inelastic collision applies here. Inelastic collision conserves momentum, but not energy. While a transform reaction can convert bound energy to kinetic energy, so that the product particles are moving faster than educt particles, it is not as easy with merge reactions, if we want to conserve momentum (see section 6.4.7, page 65 for details).

The class *MergerEnergyConservation* abandons momentum conservation under the assumption that the momentum conservation violations of all merge reactions more or less cancel each other out. (see section 6.4.7, page 65).

Each time two oxygen atoms collide and merge to the oxygen molecule $\mathbf{O_2}$, bound energy is set free as added kinetic energy of the product molecule. The effect is similar to the simpler artificial explosion: The gas heats up massively, but with more delay, because the atomic oxygen particles have to be created first (see figure 5.2 for a particle diagram of the detonating gas experiment).

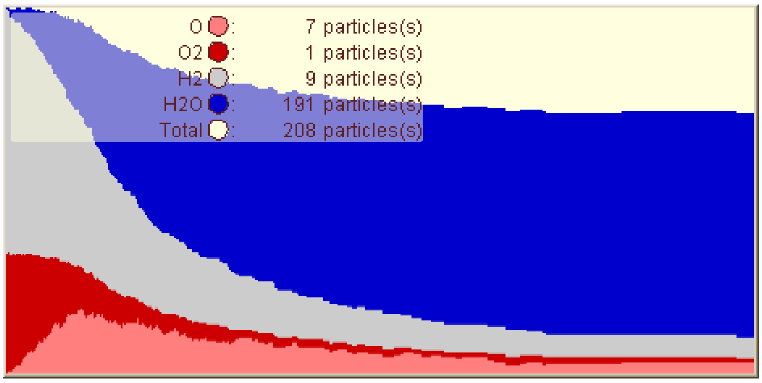


Figure 5.2: Particle diagram of the detonating gas experiment

[2] Bound energies of particles are edited in the particle classes tab of the Settings dialog using the column **Energy**.

An enhancement of the model would be to add more reactions like atomic oxygen and hydrogen to water:

$$\mathbf{O} + \mathbf{H}_2 \longrightarrow \mathbf{H}_2\mathbf{O}$$

and other reactions or even other molecules, ozone $\mathbf{O}_3$, for example.

5.3 Brownian Motion

Brownian motion describes the random, jittery motion of minute particles immersed in a fluid[3]. The mathematical model used to describe this random movement is one of the simplest stochastic processes in a continuous domain. We present a very simple ‘saces’ experiment (see figure 5.3):

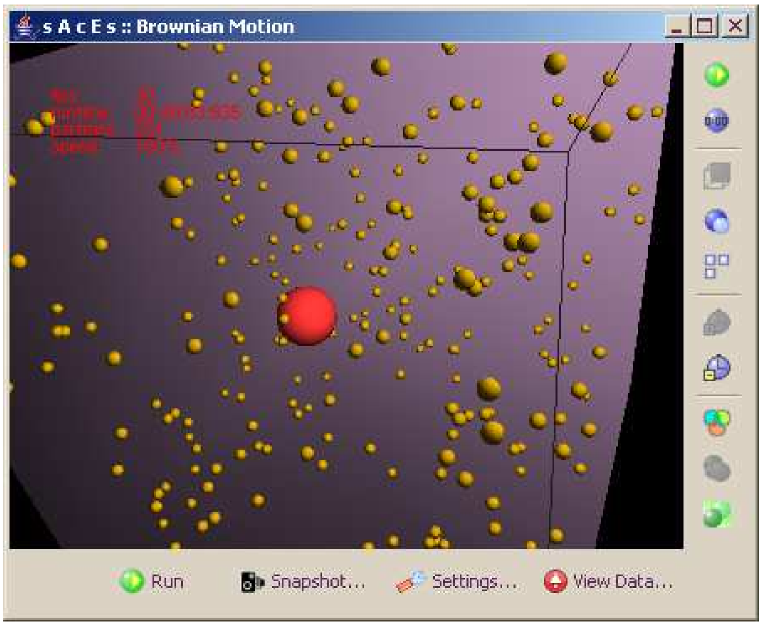


Figure 5.3: Brownian Motion in ‘saces’

A big sphere, representing the minute particle of the Brownian movement, and many small spheres, representing the fluid molecules, are animated. The big sphere is jittering and moving randomly around the inside of the cube.

[3] There is a story about Robert Brown studying pollen grains floating in water under the microscope.

A few limitations apply. The most important is that ‘saces’ cannot handle multiple collisions at once. If two small spheres collide with the particle almost simultaneously, collision detection might drop the second collision. Another problem is the scale of measure. To make the jittering movement visible, we had to model the small and big spheres with equal mass. In reality the fluid molecules move at a very high velocity, and that is why they have a visible effect on the particle at all.

5.4 Finite Automata

Finite automata are simple computation models that describe the class of regular languages.[4] An automaton (deterministic or indeterministic) can be formally described as a tuple of components (see [HS04], page 34). It is possible to use a directed graph, equivalently. You find an example in figure 5.4.

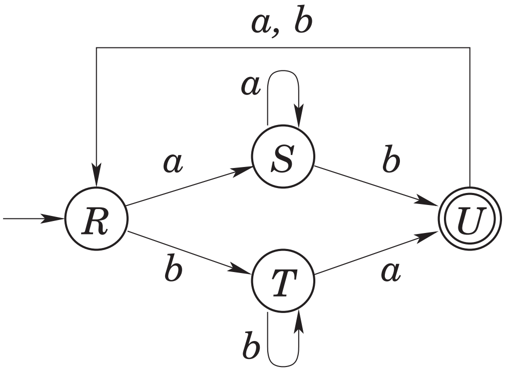


Figure 5.4: A deterministic finite automaton as a directed graph

This automaton has four states (R, S, T, U). State R is the initial state and U the final state. An automaton recognizes all strings leading from the initial to the final state. It can therefore be used to check the syntax of regular languages. The example automaton recognizes all strings described by the regular expression:

$$((aa^*b)|(bb^*a))((a|b)((aa^*b)|(bb^*a)))^*$$

An important question is whether the final state U can be reached at all.

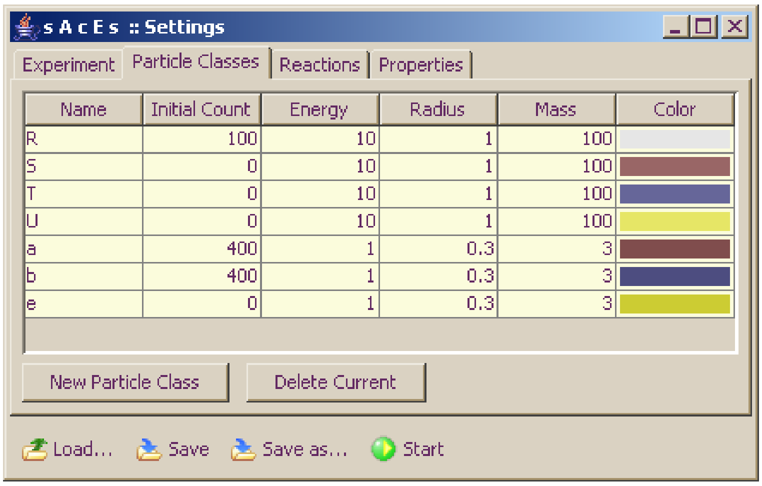
[4] A formal language is a set of finite-length words drawn from some finite alphabet (a set of symbols). An example word is the string $aabaa$. A regular language is described by a regular expression and is accepted by a finite automaton.

The automaton has been implemented as a ‘saces’ experiment *SimpleFiniteAutomaton.xml*. The artificial chemistry can be construed from an automaton with simple rules easily explained by the following example:

$$\begin{array}{rcl}
R+a & \longrightarrow & S \\
R+b & \longrightarrow & T \\
S+a & \longrightarrow & S \\
S+b & \longrightarrow & U+e \\
T+a & \longrightarrow & U+e \\
T+b & \longrightarrow & T \\
U+a & \longrightarrow & R \\
U+b & \longrightarrow & R
\end{array}$$

Note that mass conservation is impossible with this experiment. The third reaction $S + a \longrightarrow S$ cannot be expressed with mass conservation, if particle a has mass. Mass conservation can be ignored by ‘saces’. An alternative is to postulate a dummy particle x, if mass conservation is to be modeled.

The e molecule indicates a successful recognition. Aside from e, there are six different molecules R, S, T, U, a, b. We can define them as in figure 5.5:



Name	Initial Count	Energy	Radius	Mass	Color
R	100	10	1	100	⬜
S	0	10	1	100	🟫
T	0	10	1	100	🟦
U	0	10	1	100	🟨
a	400	1	0.3	3	🟫
b	400	1	0.3	3	🟦
e	0	1	0.3	3	🟨

Figure 5.5: Particles of the finite automaton experiment

We start the automaton with 400 instances of the letters a and b and with 100 instances of the initial state R. The bigger spheres are states, the smaller spheres letters. There are only the letters a and b (red and blue) and the state R (light gray).

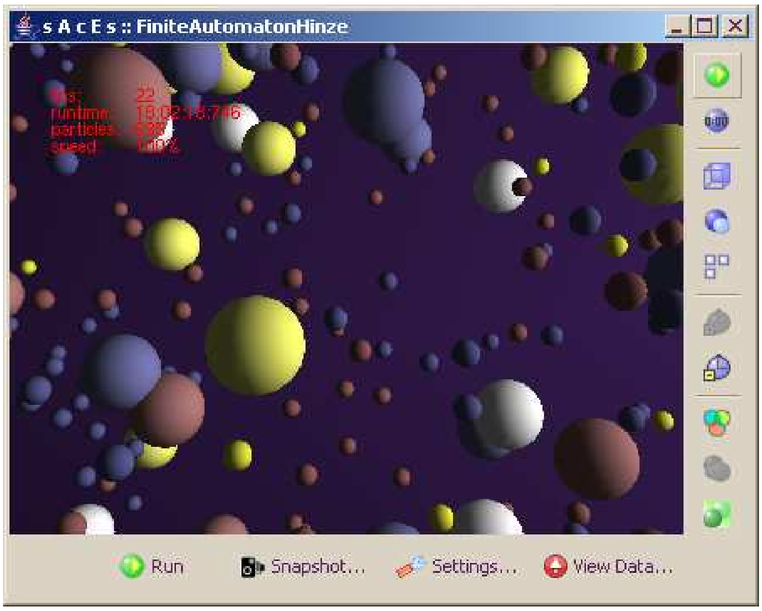
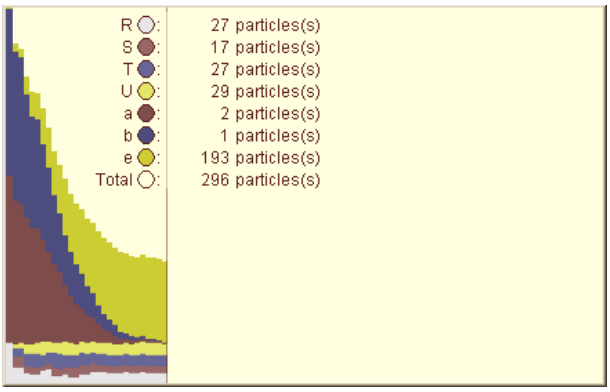


Figure 5.6: Running the finite automaton, particle view

After a few seconds, other states pop up, red for S, blue for T and yellow for U. A few small yellow e molecules indicate successful recognitions. See figure 5.6. Half a minute later, almost all letters a and b are consumed and the e letters abound.



R ○: 27 particles(s)
S ●: 17 particles(s)
T ●: 27 particles(s)
U ●: 29 particles(s)
a ●: 2 particles(s)
b ●: 1 particles(s)
e ●: 193 particles(s)
Total ○: 296 particles(s)

Figure 5.7: Particle diagram of the finite automaton experiment

A simple finite automata can therefore be simulated. Indeterministic automata are possible and implied, because it is the collision that dictates which edge is taken.

How to transfer the *Knapsack*[5] problem to a graph in polynomial time is shown in [HS04], page 227. If, and only if, there is a solution to the *Knapsack* problem, is the final state is reachable. This graph can be modeled in ‘saces’. If a desired state is reached, the problem is answered with ‘yes’.

Therefore, all NP-complete problems are representable as a finite automaton.[6] If many different molecules and their reaction equations can be designed exactly, a massively parallel approach to recognize[7] solutions to NP-complete problems could be modeled. It is supposed, that an algorithm which yields an exact solution requires exponential resources (see page 11, [Coo03] and [Dev02]). As already described in section 2.1.1, one could trade memory for time. The exponential growth of the problem space is managed by the sheer number of molecules available in a reaction vessel, at least to some extent.

A question remains to be answered: How many different particle species and reaction equations can be managed by ‘saces’ without problems?

5.5 A Lotka-Volterra System

Introduction

The purpose of the Lotka-Volterra simulation is to demonstrate the diversity of experiments possible with the ‘saces’ application.

The Lotka-Volterra Model can be used in simulating simple ecological predator-prey Systems. In its simplest form, it consists of only two actors: Prey that gets preyed on, and predators that predate. This scenario is of course idealized. It assumes that the prey is preyed upon by only one type of predator. In the real world this is hardly ever the case. The simplified model can however, deliver surprisingly realistic results when applied to questions about population dynamics in such systems. The model follows two basic growth rules:

[5] Problem statement: There are a set of items, each with a weight/cost and a value. Try to pack as many valuable items as possible in a bag without exceeding a given total weight/cost C. It is possible to attain at least the value V?

[6] An important consequence of theoretical computer science is that all NP-complete problems are polynomial-time reducible to each other (in fact it is part of the definition of NP-complete). If we show how to represent a specific NP-complete problem as a graph, we have done this for all NP-complete problems. Polynomial-time reducible means that there is a polynomial-time algorithm to translate a problem. A certain SAT problem can be translated to a certain *Knapsack* problem.

[7] If there is no such solution, the finite automaton would run forever. The final state would never be reached.

1. A prey population increases at a rate proportional to the number of prey, but is simultaneously reduced by predators at a rate proportional to the product of the numbers of prey and predators.

2. A predator population decreases at a rate proportional to the number of predators, but increases at a rate proportional to the product of the numbers of prey and predators.

The following screenshot illustrates the expected behaviour of the simulation as displayed by the data viewer.

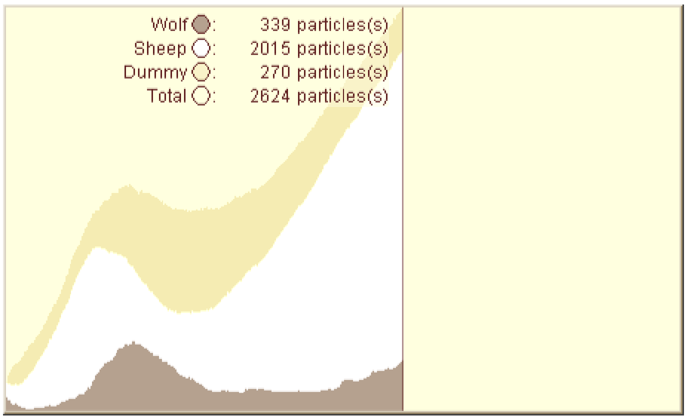


Figure 5.8: Wolves and sheep population diagram

The Lotka-Volterra example supplied with ‘saces’ simulates a population of wolves preying on sheep, and demonstrates the cyclic behaviour of such a system. The events occur in the following order.

1. The sheep population increases because there are not many wolves which prey on them.

2. The wolf population increases due to the increasing abundance of sheep.

3. The wolf population increases until there is not enough sheep to sustain the wolf population.

4. The wolf population decreases due to the lack sheep.

Vito Volterra and Alfred J. Lotka described this cyclic behaviour in population dynamics using two coupled differential equations, each one representing a species.

$$\frac{\mathrm{d}H}{\mathrm{d}t} = rH - aHP \ , \quad \frac{\mathrm{d}P}{\mathrm{d}t} = bHP - mP$$

$H =$ density of prey
$P =$ density of predators
$r =$ intrinsic rate of prey population increase
$a =$ predation rate coefficient
$b =$ reproduction rate of predators per 1 prey eaten
$m =$ predator mortality rate

Now that we have handled the formal definitions, let us take a look at how they are applied in the ‘saces’ experiment. The density of prey H and the density of predators P become the ratios:

$$N_{predator}/V_{box} \quad \text{and} \quad N_{prey}/V_{box}$$

where $N_{predator}$ is the number of predators, N_{prey} is the number of prey and V_{box} is the volume of the reaction vessel.

The intrinsic rate of prey population increase r becomes the probability that an individual prey reproduces within a certain time slice. This is represented in ‘saces’ by the reaction probability factor. Sheep population increase is modeled using a decay reaction occurring with a probability of 0.003. Wolf population increase is modeled using a transform reaction occurring with a probability of 1. A wolf therefore reproduces upon consuming a sheep.

Sheep reproduction (decay reaction) $Sheep \rightarrow Sheep + Sheep$

Wolf reproduction (transform reaction) $Wolf + Sheep \rightarrow Wolf + Wolf$

Predator mortality rate is handled by decay reaction that occurs with the probability of 0.007. The decay reaction produces two dummy particles.

Wolf mortality (decay reaction) $Wolf \rightarrow Dummy + Dummy$

While modeling wolf mortality, we run into a slight problem. The artificial chemistry of ‘saces’ provides a formal collection of three reaction types. Unfortunately, none of the three reaction types can be used to model particles which disappear without leaving a reaction product behind (the way a wolf would when it dies). Such a reaction would be in the form:

$Wolf \rightarrow -$

Dummy particles are used to compensate for the missing reaction type. When a wolf decays (dies), it produces two dummy particles and disappears. The dummy particles therefore function as placeholders in the decay reaction and are the educts of three helper merge reactions necessary to maintain the system. These merge reactions clean up the simulation space by consuming dummy particles, preventing an unwanted population growth of dummies. They have no other practical function in the simulation.

The helper merge reactions are:

$$\begin{aligned} Wolf + Dummy &\longrightarrow Wolf \\ Sheep + Dummy &\longrightarrow Sheep \\ Dummy + Dummy &\longrightarrow Dummy \end{aligned}$$

Chapter 6

The Internals of the tool 'saces'

6.1 Package Structure

Java lets developers group their classes into packages; this is more or less the equivalent of name spaces in C++. Using packages helps maintaining conceptual integrity at the least. At the most, it is useful for grouping function and data together in logical groups. The following seven sub-packages of the *saces* package are the most important: *app, app.gui, exp, file, gl, pnp* and *sim*.

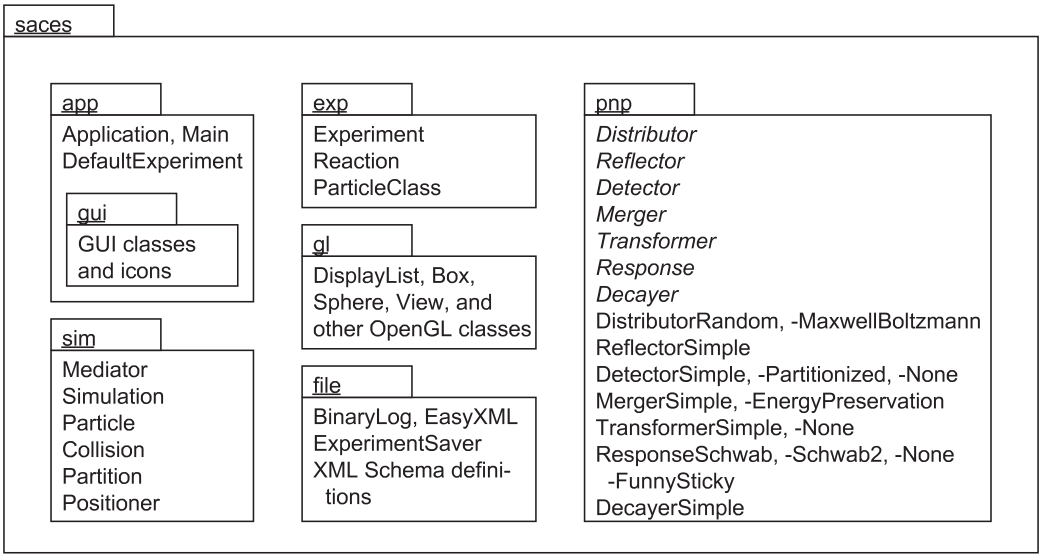


Figure 6.1: UML packet diagram for 'saces'

6.1.1 The UI Packages *saces.app* and *saces.app.gui*

The *app* package contains the program’s main entry point. It also hilds the *gui* package containing the application’s dialogs and the routines which handle application level user input. An example of this would be the loading of an experiment into saces.

6.1.2 The Experiment Model Package *saces.exp*

Experiment definition data is packaged herein. This encompasses the particle, reaction and the experiment space definition, i.e. everything which needs to be defined for a valid experiment. All these classes are immutable once created, to help maintain data integrity.

6.1.3 The Input/Output Package *saces.file*

The *file* package handles input and output. This includes ‘saces’ persistency and logging functionality. In ‘saces’, experiments are made persistent using XML. Reading and writing XML is handled here, as well as validation using XML Schema. Logging uses a binary protocol that is written to file and can be read by the viewer (preferably running on another host machine to avoid any performance loss when simulating). The file can then be read over a network share.

6.1.4 The OpenGL Package *saces.gl*

The *gl* package contains all the functionality required to render 3D scenes using OpenGL geometry definitions, transformations, lighting, material properties, view port calculations, viewing modes, detail level settings etc. No other package contains OpenGL code, aside from the mediator in the *sim* package. The Mediator serves as a hub where the simulation data meets the rendering capabilities of ‘saces’ (see figure 6.5, page 56).

6.1.5 The Plug-and-play Package *saces.pnp*

The plug-and-play package (*pnp* for short) consists of interfaces which define the simulation steps and their implementations like for collision detection and handling reactions etc. This package is named plug-and-play due to the possibility of interchanging the actual implementations of the simulation steps.

In other words, the implementations used for an experiment are fully configurable in XML. Let us take the wall reflection as an example. One implementation of the reflector interface might ‘heat’ the reaction vessel by increasing velocity after reflection. Another implementation might not reflect any particle at all, but instead position the particle on the opposite end of the box, as if the sides of the box would wrap around to form an unbounded wrapping simulation space. One experiment might require one type of reflection; another might require something quite different. Plug-and-play design enables developers to write their own custom implementations, tailored for the type of experiment they wish to examine.

6.1.6 The Simulation Package *saces.sim*

The *sim* package represents what happens when the experiment is run. It also contains the mediator, the positioner that calculates the location of every particle for a given step and the partition class that divides the simulation space into smaller cuboids for an optimized collision detection algorithm (see section 6.6, page 74 about space partitioning).

6.2 Data Architecture

‘saces’ distinguishes between experiment setup parameters and simulation state. Experiment setup parameters are constant (usually one does not change rules in the middle of a simulation) and its classes are located in package *saces.exp*. The simulation state is dynamic and its classes are located in the *saces.sim* package.

6.2.1 The Experiment Data Model

The experiment parameters consist of three classes in package *saces.exp*. They are class *Experiment*, class *ParticleClass* and class *Reaction* (see figure 6.2).

The Class *Experiment*

An *Experiment* instance provides global experiment parameters (like *initial* temperature in the reaction vessel). It is used as a starting point to access the experiment data. There are particle classes and reaction lists. There is only one instance per running simulation. A running simulation can therefore have only one setup.

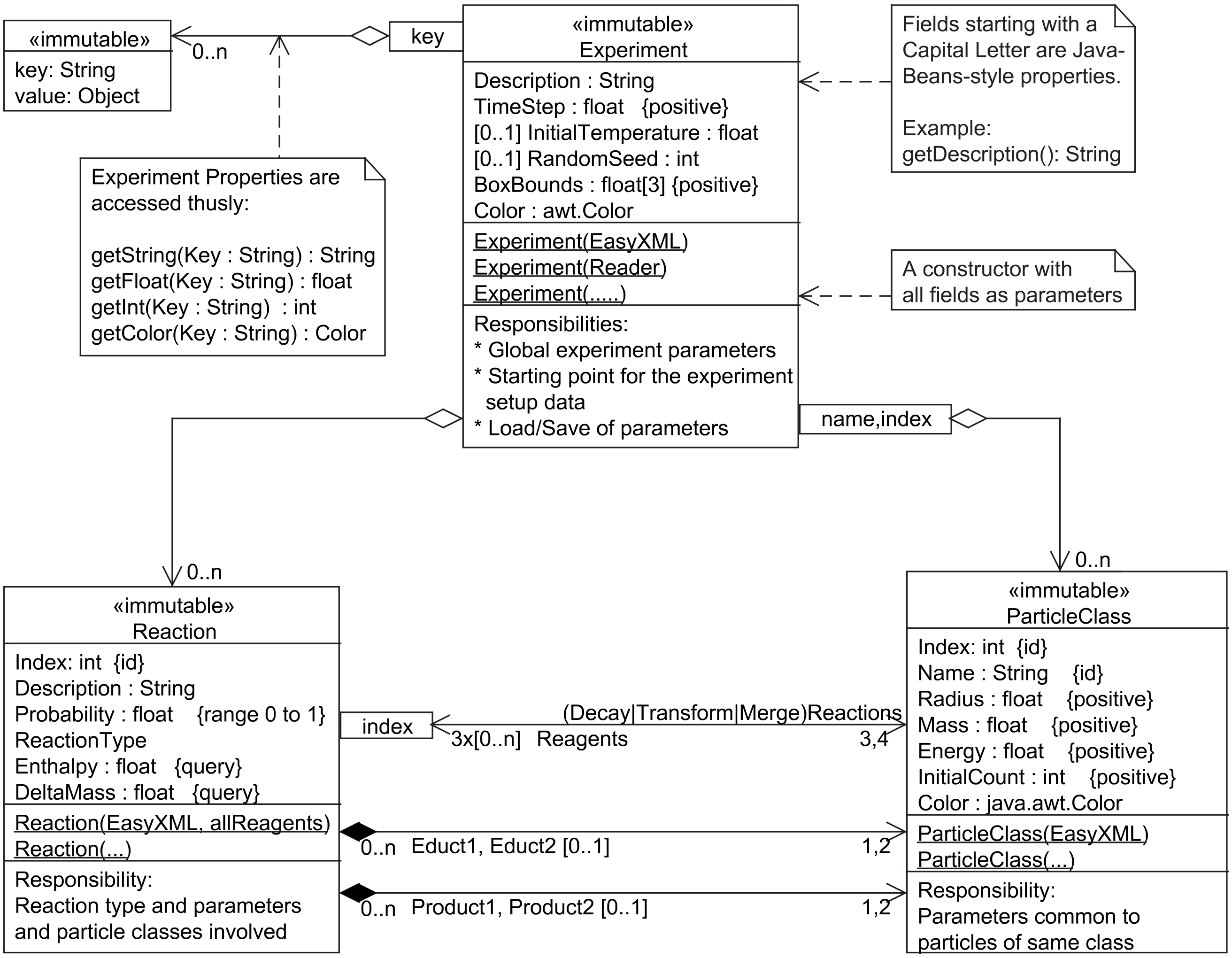

«immutable»
key: String
value: Object
0..n
key
«immutable»
Experiment
Description : String
TimeStep : float {positive}
[0..1] InitialTemperature : float
[0..1] RandomSeed : int
BoxBounds : float[3] {positive}
Color : awt.Color
Experiment(EasyXML)
Experiment(Reader)
Experiment(.....)
Responsibilities:
* Global experiment parameters
* Starting point for the experiment setup data
* Load/Save of parameters
Fields starting with a Capital Letter are Java-Beans-style properties.
Example:
getDescription(): String
A constructor with all fields as parameters
Experiment Properties are accessed thusly:
getString(Key : String) : String
getFloat(Key : String) : float
getInt(Key : String) : int
getColor(Key : String) : Color
name,index
0..n
0..n
«immutable»
Reaction
Index: int {id}
Description : String
Probability : float {range 0 to 1}
ReactionType
Enthalpy : float {query}
DeltaMass : float {query}
Reaction(EasyXML, allReagents)
Reaction(...)
Responsibility:
Reaction type and parameters and particle classes involved
index
(Decay|Transform|Merge)Reactions
3x[0..n] Reagents
3,4
0..n Educt1, Educt2 [0..1]
1,2
0..n Product1, Product2 [0..1]
1,2
«immutable»
ParticleClass
Index: int {id}
Name : String {id}
Radius : float {positive}
Mass : float {positive}
Energy : float {positive}
InitialCount : int {positive}
Color : java.awt.Color
ParticleClass(EasyXML)
ParticleClass(...)
Responsibility:
Parameters common to particles of same class

Figure 6.2: UML class diagram in package *saces.exp*

The Class *ParticleClass*

A particle class defines a particle *species*. Particles of the same class all have identical mass, for example. Mass and other parameters are stored in a *ParticleClass* instance.

The term “class” can be misunderstood by Java developers. A class is something like a blueprint that defines common characteristics of objects. This is also true for particles, but a particle class is not a Java class. It is a definition of common parameters for particles.

Each particle class has both an unique name and an index. We find *ParticleClass* instances starting from the *Experiment* instance using either the name or the index. In the simulation loop, finding the possible reactions of a particle must be efficient. A particle class therefore contains three reaction lists, one for each reaction type. These lists contain the reactions that the particle can be an educt of.

The Class *Reaction*

A *Reaction* instance defines a reaction in ‘saces’. It contains reaction parameters: the reagents, activation energy, probability of reaction etc.

There is a reagent list with three or four elements (depending on reaction type). The order of elements in the list is important. First the educts are listed, then the products.

Alternatively, access the *Educt1*, *Educt2*, *Product1* and *Product2* Java-Beans-style properties directly. The second educt or product are optional, of course. For *Merge* reactions, *Product2* is not defined and for *Decay* reactions, *Educt2* is not defined; that means the corresponding Java getter method returns a null pointer.

Reaction Type

There are three reaction types (see section 9, page 19): *Transform*, *Merge* and *Decay*. The *ReactionType* Java-Beans-style property can have one of these three values. With Java 5, enumerated types are possible. *ReactionType* is an enumerated type with these three values.

The Experiment Properties

Additional parameters are accessed as key-value pairs (see table 4.6, page 35). Experiment properties are important to specify additional data for the experiment in a flexible way.

6.2.2 The Experiment XML File

XML is used to persist experiments and XML Schema is used to validate experiment files before they are loaded. See figure 6.3 for a class diagram for the experiment XML files. Each class corresponds to an XML element, each member to an XML attribute. The types are specified exactly by XML Schema (for example probability values are to be clamped to the interval $[0..1]$).

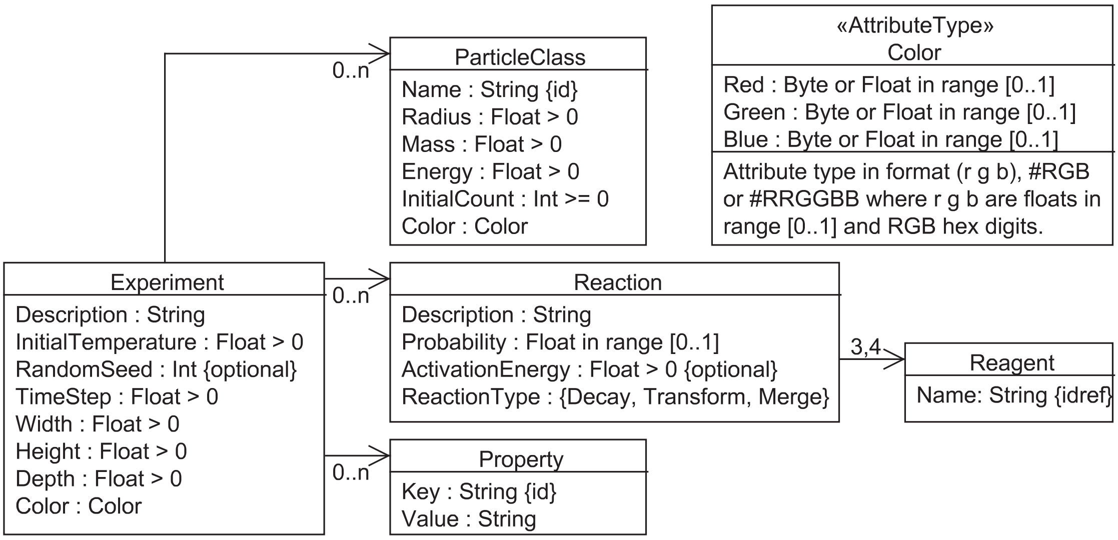

«AttributeType»
Color
Red : Byte or Float in range [0..1]
Green : Byte or Float in range [0..1]
Blue : Byte or Float in range [0..1]
Attribute type in format (r g b), #RGB or #RRGGBB where r g b are floats in range [0..1] and RGB hex digits.
ParticleClass
Name : String {id}
Radius : Float > 0
Mass : Float > 0
Energy : Float > 0
InitialCount : Int >= 0
Color : Color
0..n
Experiment
Description : String
InitialTemperature : Float > 0
RandomSeed : Int {optional}
TimeStep : Float > 0
Width : Float > 0
Height : Float > 0
Depth : Float > 0
Color : Color
0..n
Reaction
Description : String
Probability : Float in range [0..1]
ActivationEnergy : Float > 0 {optional}
ReactionType : {Decay, Transform, Merge}
3,4
Reagent
Name: String {idref}
0..n
Property
Key : String {id}
Value : String

Figure 6.3: XML Schema experiment definition as a class diagram

6.2.3 The Simulation State

A Side Note

Many design decisions explained in this section were influenced by performance considerations. We did not do so-called “premature” optimization. We optimized only if we discovered evidence of a performance problem. Two methods were used to discover such problems. The most general was complexity theory. For the naive, brute force collision detection, we calculated a time complexity of $O(n^2)$. Then, with a good Java profiler we analyzed running times and invocation counts per class and per method.

Simulation State

The experiment setup defines static, initial parameters and data for the experiment. Simulation state, however, is much more dynamic. It contains, for example, the velocity and position of all particles, which change a lot during the simulation.

The two most important classes in the *saces.sim* package are the *Simulation* and *Particle* classes (see figure 6.4). The *Partition* class is necessary for the collision detection algorithm *space partitioning* (see section 6.6, page 74).

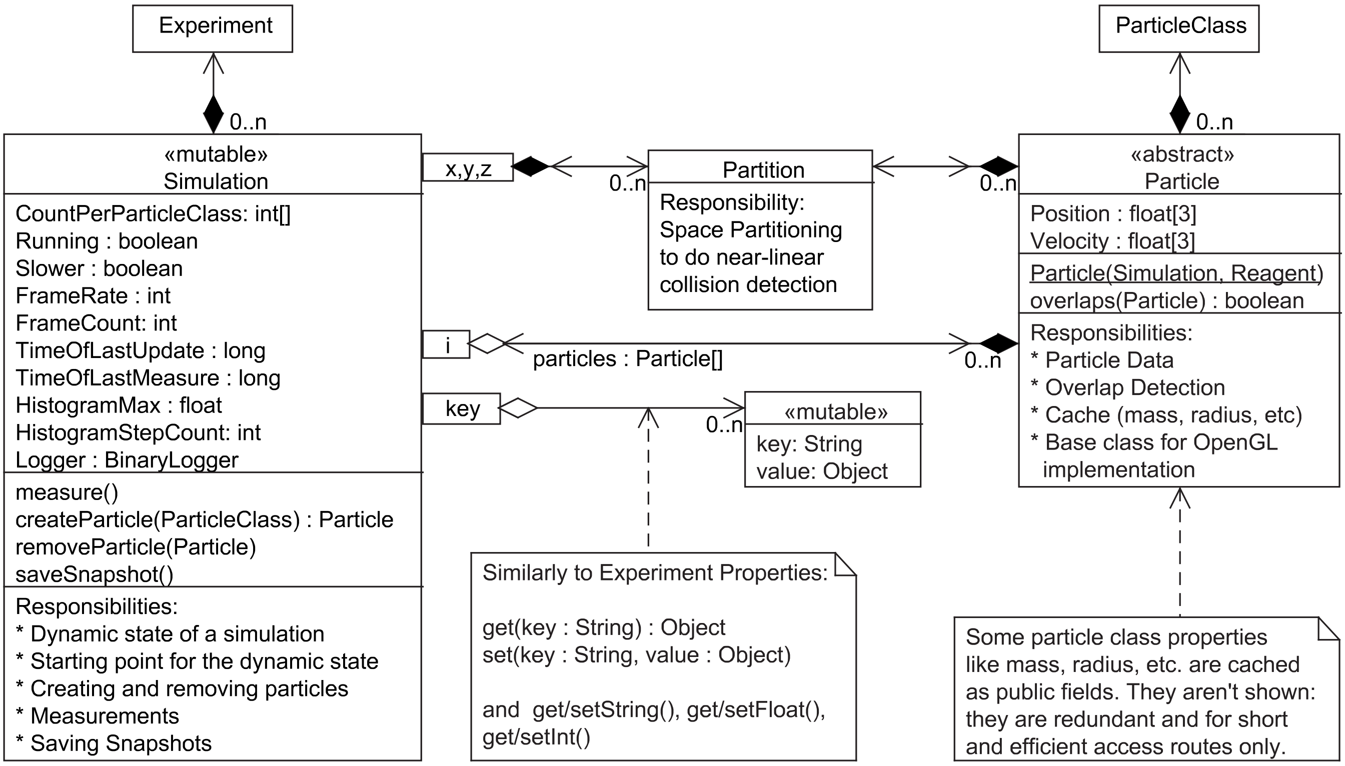

Experiment
ParticleClass
0..n
0..n
«mutable»
Simulation
CountPerParticleClass: int[]
Running : boolean
Slower : boolean
FrameRate : int
FrameCount: int
TimeOfLastUpdate : long
TimeOfLastMeasure : long
HistogramMax : float
HistogramStepCount: int
Logger : BinaryLogger
measure()
createParticle(ParticleClass) : Particle
removeParticle(Particle)
saveSnapshot()
Responsibilities:
* Dynamic state of a simulation
* Starting point for the dynamic state
* Creating and removing particles
* Measurements
* Saving Snapshots
x,y,z
0..n
Partition
Responsibility:
Space Partitioning
to do near-linear
collision detection
0..n
«abstract»
Particle
Position : float[3]
Velocity : float[3]
Particle(Simulation, Reagent)
overlaps(Particle) : boolean
Responsibilities:
* Particle Data
* Overlap Detection
* Cache (mass, radius, etc)
* Base class for OpenGL implementation
i
particles : Particle[]
0..n
key
0..n
«mutable»
key: String
value: Object
Similarly to Experiment Properties:
get(key : String) : Object
set(key : String, value : Object)
and get/setString(), get/setFloat(), get/setInt()
Some particle class properties like mass, radius, etc. are cached as public fields. They aren't shown: they are redundant and for short and efficient access routes only.

Figure 6.4: UML class diagram in package *saces.sim*

The classes contain many hotspots.[1] A very significant hotspot is the method *Particle.overlaps()*. A naive (quadratic) implementation of collision detection of ten thousand particles, it is called almost fifty million times per iteration. At 20 frames per second, this would be a billion (10^9) times per second.

Using *space partitioning*, it is called about sixty thousand times, which is still very ‘hot’. That is why we have public member fields in the classes instead of ac-

[1] A hotspot is a part of the program that is executed very frequently. The HotSpot Java compiler from Sun optimizes these parts.

cessor methods. This is not conforming to our Java Coding Convention [Amb00], which demands Java-Beans-style getter methods.

The Class *Simulation*

A *Simulation* instance contains the simulation's dynamic state (current temperature, particle counts per particle class etc.) and is used as a starting point to access the different parts of simulation state. The simulation holds a list of particles and the three-dimensional array of partitions for *space partitioning*. The *Simulation* instance is also responsible for the creation and removal of particles. This is necessary to keep the particle counts per particle class up to date.

The Class *Particle*

The particle is the main actor in the simulation. Where is the particle? How fast does it move in the reaction vessel? Particles are very dynamic: they move and change velocity often, they increase and decrease in numbers and even might disappear completely. The only thing which stays the same is its class or *species*.

The *overlaps()* method, a hotspot as already mentioned, tests whether the particle overlaps another particle. Because *overlaps()* requires the radius of the particles, the radius is cached as a public final member field. When a particle is created, the radius is copied from the particle class and is left unchanged until the particle disappears. Other critical particle class parameters are also cached in this way.

The class *Particle* is abstract. The *Sphere* class extends it and provides an implementation as a GLU (OpenGL Utility Library) sphere. There is no *Particle* constructor. It is the *Simulation* instance that creates and removes particles. An experiment property dictates which implementation class to use (see key `Particle` in table 4.6, page 36).

The Class *Partition*

Both *Simulation* and *Particle* instances maintain bidirectional references to the *Partition* instances. Such tight coupling should be avoided in good software design. However, short access routes in two directions are necessary. Whenever a moving particle crosses a partition boundary, particle lists in partitions must be updated. For details about *space partitioning* see section 6.6, page 74.

6.3 The Process Architecture

6.3.1 The Mediator

The mediator mediates between the GUI and the state of both OpenGL and simulation. For example, if the user clicks on the ▶ ‘Run’ button, the button event handler calls *Mediator.setRunning(true)*. The GUI does not interact directly with OpenGL or with the simulation state. User requests are mediated and delegated to the objects responsible for processing the request.

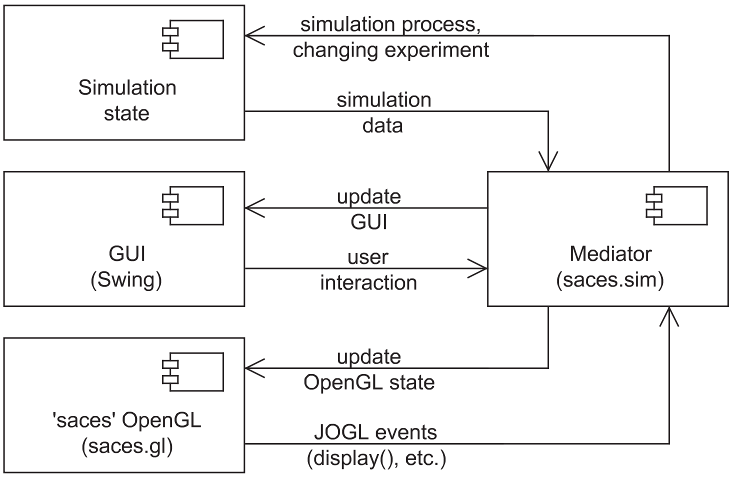


Figure 6.5: The data and control flow of the mediator

Another responsibility of the mediator is the forwarding of JOGL display events to the simulation loop. The JOGL framework uses an event-driven architecture. The mediator implements the interface *GLEventListener*. *display()* events are forwarded to the mediator which delegates the painting further to the *gl* package.

6.3.2 The Simulation Process

A simulation contains an endless loop where the simulation state is being calculated repeatedly as long as the simulation is left running (see figure 6.6).

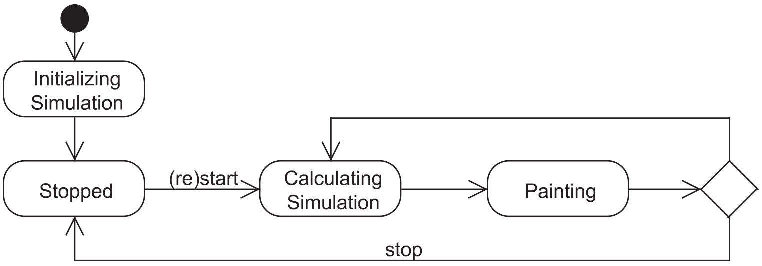


Figure 6.6: UML action diagram: a simple simulation process

This is a very simple example of a simulation process. When user starts the simulation, calculating and painting is done repeatedly till the user stops[2] the simulation. The actions *Initializing Simulations* and *Calculating Simulation* can be subdivided into more complex sub-processes (see figure 6.7).

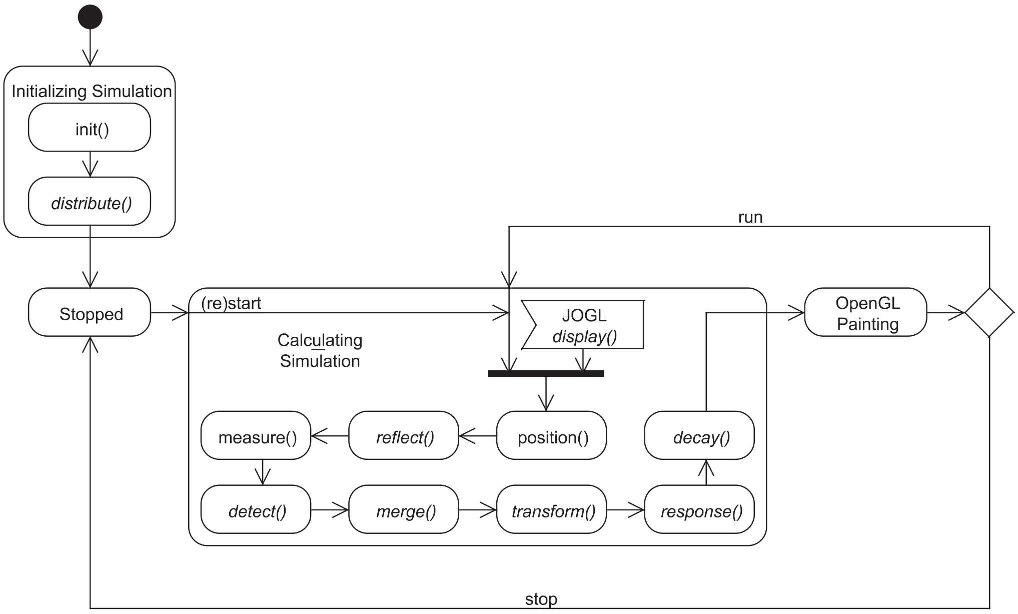


Figure 6.7: UML action diagram: a more complex simulation process

What the steps do exactly is described in detail in the section 6.4, page 59ff. An important aspect is data flow between the steps. *Transform* reactions need to know which particles have collided, but *Merge* reactions remove them from the simulation. This is an example of how the steps influence each other.

[2] This is not exactly true. Even if the simulation is stopped, there are paint events. For example they are necessary if a part of the window has been obscured by another window.

We tried to use the *Visitor* pattern [GHJV94] for the steps interface design. A particle was visited by a *TransformReactionVisitor* instance if it was to be used in a *Transform* reaction, for example. We discovered very soon that the pattern produced a lot of method calling overhead. Calling a method for every single particle that underwent a reaction, was not acceptable due to performance issues. After we switched to a “do-it-all-at-once” approach using arrays, performance increased considerably. All *Transform* reactions are handled within *one* call of the *transform()* method of the *Transformer* interface. Figure 6.8 shows a data-flow diagram.

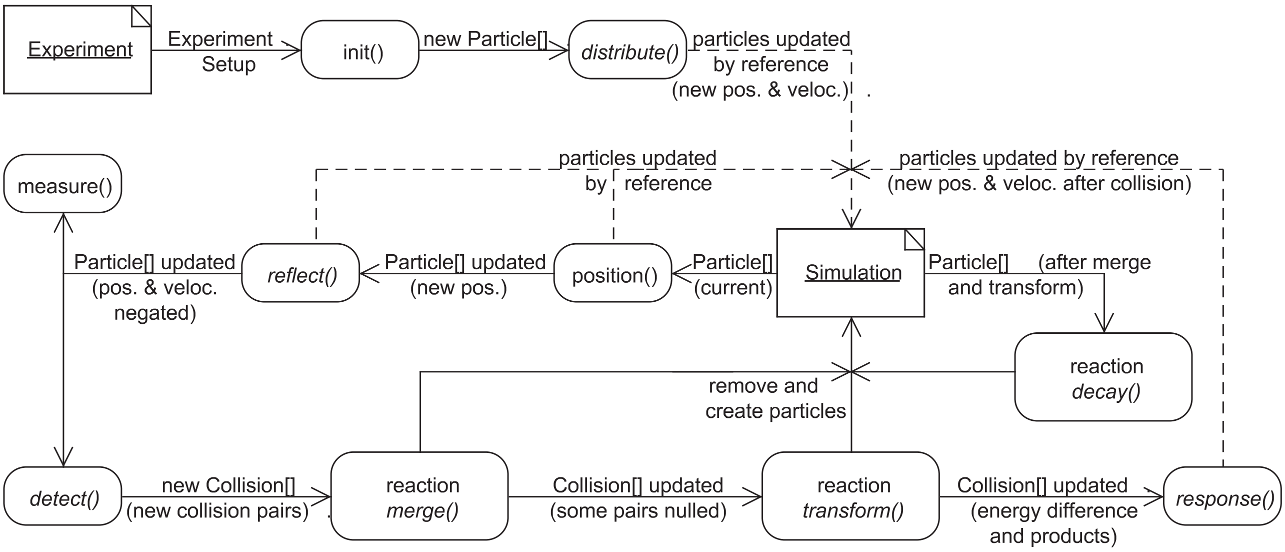

Experiment
Experiment Setup
init()
new Particle[]
distribute()
particles updated by reference (new pos. & veloc.)
measure()
particles updated by reference
particles updated by reference (new pos. & veloc. after collision)
Particle[] updated (pos. & veloc. negated)
reflect()
Particle[] updated (new pos.)
position()
Particle[] (current)
Simulation
Particle[] (after merge and transform)
reaction decay()
remove and create particles
detect()
new Collision[] (new collision pairs)
reaction merge()
Collision[] updated (some pairs nulled)
reaction transform()
Collision[] updated (energy difference and products)
response()

Figure 6.8: Data flow diagram: how data flows in the simulation process

Therefore particle arrays and collision pair arrays are created and passed around instead of invoking methods for every single particle. Steps in italics are available for the plug-and-play architecture. The data flows are:

Particle Arrays are passed to *distribute()*, *position()*, *reflect()*, *measure()* and *detect()*. They are passed to *decay()*, as well, but as last step in the simulation loop.

Updating Particles by Reference is done by the steps *distribute()*, *position()*, *reflect()* and *response()*.

Removing Educts and Creating Products is done by the reaction steps (a bit bigger) *merge()*, *transform()* and *decay()*.

Collision Pairs are created by *detect()* and passed through to *merge()*, *transform()* and finally *response()*.

Modifying Collision Pairs is done by *merge()*, which removes some pairs and by *transform()*, which augments a collision pair with additional information: bound energy difference and reaction products.

Simulation Properties can be read and written by all steps.

Binary Log is available for all steps to write data, especially *measure()* needs the binary log.

Initializing data flow is implicit through the mediator instance. Used by *init()*.

position(), for example, takes the current particles as an array and modifies particles by reference. *measure()* does not modify anything. It simply writes to the binary log and saves simulation properties. The reaction steps *merge()* and *transform()* take collision pair arrays. They create product particles and remove educt particles directly to and from the simulation state.

In a future version of ‘saces’, parts of the simulation process still too slow can be re-implemented as native methods in C. Let us say that *detect()* needs to be re-implemented. A class *DetectorPartitionizedNative* can be written that loads the native library and delegates *detect()* to it. This is why arrays are used in the data flow. C can handle arrays better than complicated Java lists. In Java, array access is also faster than list access. The only problem is that it is not possible to increase array capacity dynamically. With a carefully designed data-flow architecture, the need to add elements to arrays has been avoided.[3]

For the signatures of all simulation process steps, see table 6.1. The table contains the interface names as well without the package name *saces.pnp*. The table shows only data flow by parameters, return values and direct adding or removing of particles *(simulation out)*. A special case is *init()*, because data flow is implicit through the mediator instance (necessary to initialize first).

6.4 The Steps of the Simulation Process

The next sections contain the detailed description of the simulation process steps starting with *init()* and ending with *decay()*.

[3] *createParticle()* in the *Simulation* class is used to add product particles to the simulation. This works, because the particles are saved in a list. The list is copied to an array at the beginning of every simulation iteration. The simulation steps use the array, instead of the list. The cost of copying the list to the array is counterweighted by the costs of list access of all steps.

Step	Interface	Signature and Data flow
init()	part of the mediator	`void init()` *implicit:* simulation from mediator *implicit:* new particle array is created in simulation
distribute()	*Distributor*	`void distribute(Particle[], Simulation)` *in:* particle array and simulation *out:* array with *updated* particles $(\vec{p}, \vec{v})$
position()	*Positioner* (in package *saces.sim*)	`void position(Particle[], Simulation)` *in:* particle array and simulation *out:* array with *updated* particles $(\vec{p})$
reflect()	*Reflector*	`void reflect(Particle[], Simulation)` *in:* particle array and simulation *out:* array with some *updated* particles ($\vec{p}, \vec{v}$ negated)
measure()	part of the *Simulation* class	`void measure(Particle[], Simulation)` *in:* particle array and simulation

detect()	*Detector*	`Collision[] detect(Particle[], Simulation)`
		in: particle array and simulation
		return: collision pairs array

merge()	*Merger*	`void merge(Collision[], Simulation)` *in:* collision pair array and simulation *out:* collision pair array with *deleted (null)* elements *simulation out: removed* and *new* particles
transform()	*Transformer*	`void transform(Collision[], Simulation)` *in:* collision pair array and simulation *out:* collision pair array with more information for *response()* *simulation out: removed* and *new* particles
response()	*Response*	`void response(Collision[], Simulation)` *in:* collision pair array and simulation *out:* collision pair array with same elements but where particles have *changed* velocity (and perhaps position)
decay()	*Decayer*	`void decay(Particle[], Simulation)` *in:* particle array and simulation *simulation out: removed* and *new* particles

Table 6.1: Data flow and signatures of the simulation process

6.4.1 Initializing the Experiment — *init()*

The particles must be created. Plug-and-play classes must be dynamically loaded and instantiated using Java reflection. Particle counts must also be set. If there is a problem, an error message box is shown (see section 6.5, page 72).

Because ‘saces’ is always started with a default experiment, *init()* and *distribute()* are run before any user interaction. This should not pose problems. The default experiment is carefully designed to not produce any problems. It is located unchangeably within the *.jar* file.

The method *init()* is part of the mediator and not available for enhancement as a plug-and-play step.

6.4.2 Initial Distribution of the Particles — *distribute()*

After the particles have been created, they are assigned position and velocity. The *Distributor* interface consists of one method: *distribute()*. The distributor uses the particle array passed to the method to set the position and velocity of the particles.

DistributorRandom is a distributor class. *DistributorRandom* distributes position uniformly and velocity components normally using the standard deviation saved in the `DistributorRandom.stdDev` experiment property.

The *DistributorMaxwellBoltzmann* class is a distributor implementation which makes a Maxwell-Boltzmann distribution. It distributes the position as does *DistributorRandom*, but applies Maxwell-Boltzmann distribution to the velocity components. The standard deviation is calculated using : $\sigma = \sqrt{kT/m}$, where k is the Boltzmann constant (defined as the experiment property `BoltzmannConstant`), T the initial temperature of the experiment and m the mass of the particle to be distributed. *DistributorMaxwellBoltzmann* makes smaller particles move more quickly than heavier particles. This is because their velocity squared is inversely proportional to their mass.

6.4.3 Repositioning of the Particles — *position()*

After each run of the simulation loop, a certain amount of real time has passed. The particles are at new positions calculated using the velocities:

$$\vec{p}' = \vec{p} + \Delta t \cdot \vec{v}$$

This is an extremely simple step and we saw no need to provide a possibility to enhance *position()*. This step is therefore one of the two steps not available to the plug-and-play architecture.

6.4.4 Reflecting at the Reaction Vessel Walls — *reflect()*

position() did not worry whether the particles left the reaction vessel boundaries. This is the task of *reflect()*. It negates per component both velocity v_σ and position p_σ, if the position component is outside the range $[r \ldots b_\sigma - r]$ where σ is either x, y or z. Particle radius r is considered.

$$v_\sigma \leftarrow -v_\sigma; p_\sigma \leftarrow -p_\sigma + \begin{cases} 2r & \text{if } p_\sigma - r < 0 \\ 2b_\sigma - 2r & \text{if } p_\sigma + r > b_\sigma \end{cases}$$

With changing the sign, even the effect that the particle already has traveled a tiny distance beyond the boundary is considered and the new position is set retroactively. This works because it is assumed that the boundaries are rectangular and orthogonal to each other (see section 3.2).

A possible improvement of this step would be the heating or cooling of the vessel walls—this is why *reflect()* is pluggable.

6.4.5 Measurements of Physical Values — *measure()*

People use their senses to perceive things. Physicists use measurement instruments to measure things. In 'saces' we do measurements in the *measure()* step. Values like particle count, total mass of the experiment, temperature and pressure are written to the binary log in regular intervals defined by the `MeasureInterval` experiment property. See table 6.2 for an overview of the values measured.

Binary Logger Key	How to calculate the value
ParticleCount	n
TotalMass	$m_{tot} = \sum m_i$
TotalBoundEnergy	$E_{bound,tot} = \sum E_i$
TotalKineticEnergy	$E_{kin,tot} = \sum m_i \vec{v_i}^{\,2}/2$
AverageKineticEnergy	$\overline{E_{kin}} = E_{kin,tot}/n$
TotalEnergy	$E_{tot} = E_{bound,tot} + E_{kin,tot}$
Temperature	$T = 2\overline{E_{kin}}/3k$
Pressure	$p = nRT/V$

Table 6.2: Values measured

Temperature and pressure are macroscopic values and need a little more explanation.

Temperature

In the kinetic theory of thermodynamics, temperature is directly proportional to the average kinetic energy in a system. We postulate that the particles have only three degrees of freedom, one for each velocity component of the three dimensions. There are no degrees of freedom for rotation or oscillation, because we assume the particles are hard spheres with mass concentrated in the center i.e. an ideal gas. For temperature T we calculate:

$$T = \frac{2\overline{E_{kin}}}{fk}$$

where $f = 3$ is the number of degrees of freedom.

We can try different values of the Boltzmann constant k (canonically it is 1.3806510^{-23}JK^{-1}) to get “plausible” values of temperature. In fact, the Boltzmann merely reflects a preference for expressing the average kinetic energy in Kelvin. We are therefore free to redefine k in the chemistry of ‘saces’ considering that particles are bigger and much slower than in reality.

Pressure

Pressure in an ideal gas can be described by the equation $pV = nRT$, where R is the ideal gas constant (canonically it is 8.31447JK^{-1}mol^{-1}) and n is the mole (in ‘saces’ the number of particles). The gas constant is a conversion factor between the gas units of energy, temperature and molecule number. We also can redefine R to fit our experiments. We calculate pressure as follows:

$$p = \frac{nRT}{V}$$

The Boltzmann and ideal gas constants are experiment properties and available for tweaking (see table 4.6, page 35).

Speed distribution

Another task of *measure()* is creating speed distributions, also called speed histograms. The speed distribution subdivides the speed $|\vec{v}|$ of the particles into equal intervals. The number of intervals is determined by the experiment property `HistogramSize`. The speed interval stretches from 0 to the value of the experiment property `HistogramMax` or to the maximum speed in the simulation. The minimum and average speeds are also saved. The histogram is viewable in the speed histogram tab of the data viewer (see page 33).

Tight Coupling with *Simulation*

measure() is not part of the plug-and-play architecture. It is coupled a bit too tightly with the *Simulation* instance. Because it is responsible for calculating the frame rate, *measure()* cannot be replaced. We have decided not to make *measure()* available to the plug-and-play architecture for this reason. In a future version of ‘saces’ the measure step could be integrated better into the plug-and-play architecture.

6.4.6 Collision Detection — *detect()*

Without collisions ‘saces’ would not make sense at all. There would be no reactions and not even the simple Brownian Motion experiment would be possible. The Brownian Motion experiment does not execute reactions, but particles do collide.

Collision detection can be complicated. The most straight-forward approach uses a doubly nested loop that compares each particle with every other particle. It has quadratic time complexity and is not acceptable for an efficient simulation of many particles. Working with a very simple geometric object such as the sphere, makes the necessary calculations less complicated. When do two spheres ‘collide’, that means, intersect? They intersect if the following inequality holds true:

$$|\vec{p}_1 - \vec{p}_2| \leq r_1 + r_2$$

where $\vec{p}_i$ are the position vectors of particle 1 and 2 and r_i their radii. This is implemented by the method *overlaps()* in class *Particle*.

For collision detection in games, where more complicated objects can occur, more complicated algorithms are necessary, but on the other hand we have to contend with particle numbers ranging in thousands in ‘saces’. Broad phase collision detection must therefore be very efficient while narrow phase collision detection is non-existent because we work with spheres only.

For testing and experimenting purposes, we have developed two implementations of collision detection *DetectorNone* and *DetectorSimple* besides the default, efficient *DetectorPartitionized*. The first collision detector does nothing, i.e. it lets the spheres pass through each other. The other one is the naive, quadratic-time approach.

The default implementation *DetectorPartitionized* uses a highly optimized almost linear-time approach described in detail in a separate section 6.6, page 74. It uses *space partitioning*. Naive collision detection is done in every partition separately.

6.4.7 Merge Reactions — *merge()*

Merge reactions are reactions of the form $P_1 + P_2 \longrightarrow P_3$. That means two educts P_1 and P_2 merge together to produce P_3. In effect, we have an inelastic collision. Assuming molecules consist of atoms, the reaction product is a lump of atoms glued together: inelastic collision!

This is a problem if we want to model energy conservation. Inelastic collisions have an energy excess, even when not considering the bound energy difference. When a car crashes into a tree, the complete kinetic energy of the car is converted into deforming the car and perhaps also the tree. In ‘saces’, however, we do not deform particles. We want to keep energy. This is not possible if we also keep momentum. We have following equations:

$$\begin{aligned} \text{Momentum conservation:} \quad & m_1\vec{v}_1 + m_2\vec{v}_2 && = m_3\vec{v}_3 \\ \text{Energy conservation:} \quad & m_1\vec{v}_1{}^2/2 + m_2\vec{v}_2{}^2/2 && = m_3\vec{v}_3{}^2/2 \end{aligned}$$

The first equation for momentum conservation already solves the product velocity:

$$\vec{v}_3 = \frac{m_1\vec{v}_1 + m_2\vec{v}_2}{m_3}$$

The equation system for momentum and energy conservation is therefore over-determined and does not have a solution in most cases. There are several pragmatic approaches: Abandoning one of the two conservation laws, changing bound energy, redistributing and energy excess.

Abandoning Energy Conservation *MergerSimple* conserves only the momentum with the equation above. This is a simple approach and useful for all applications which need not model energy conservation.

Abandoning Momentum Conservation *MergerEnergyConservation* conserves energy and abandons momentum. The direction of the resulting velocity vector is parallel to the momentum after collision, but its length is determined by energy conservation:

$$\begin{aligned} \vec{v}_3 &\parallel m_1\vec{v}_1 + m_2\vec{v}_2 \\ |\vec{v}_3| &= \sqrt{\frac{m_1\vec{v}_1{}^2 + m_2\vec{v}_2{}^2}{m_3}} \end{aligned}$$

This approach assumes that the momentum violations of all merge reactions more or less cancel out each other. Therefore the total momentum of all particles should not change a lot.

Changing Bound Energy The energy excess is added to the bound energy of the product particle. The next time the particle collides this temporarily frozen energy can be freed.

This approach is not implemented because it has implications on the simulation process. The collision response and other reactions need to additionally consider the frozen energy. While this approach seems to be quite possible, it would exceed the limits of the diploma.

Redistributing Energy Excess The excess energies of all merge reactions in a simulation iteration is summed up and uniformly distributed to all particles of the simulation. Again, this approach would exceed the limits of the diploma and is therefore not implemented.

Handling Collision Objects

Merge reactions modify the collision object array in marking merged collisions as *alreadyMerged*, effectively removing them from the array.[4] The product is created, the educts removed and velocity and position of the product is calculated by one of the methods already described.

6.4.8 Transform Reactions — *transform()*

Transform reactions are reactions in the form of $P_1 + P_2 \longrightarrow P_3 + P_4$. The *transform()* method does not do the reflection calculations of the reaction itself, this is the task of *response()*. *transform()* only verifies whether a transform reaction applies for a collision:

[4] The element is not nulled because this is not possible directly inside a Java 5 for-each loop `for (Collision collision : collisions) /* ... */`

TransformerSimple looks up the transform reactions of both educts and tries for each available reaction the probability, the activation energy and the collision energy (if bound energy difference is negative, see page 71), and when everything is in order, the reaction is applied and the collision object is augmented with the products and the bound energy difference. The educt particles are removed from the simulation and the product particles added. Note that in contrast to Merge reactions both momentum and energy can be conserved.

TransformerNone does nothing and ignores the collision object array. This is for experimenting purposes.

6.4.9 Collision Response — *response()*

When two particles have collided and perhaps even reacted, their new velocities need to be calculated. Collision response parameters are (available in the *Collision* instance):

Velocities $\vec{v}_1$ and $\vec{v}_2$ of the collided particles

Positions $\vec{r}_1$ and $\vec{r}_2$ of the collided particles

Masses m_1 and m_2 of the collided particles

Bound Energy Difference $\Delta E = E_1 + E_2 - E_1' - E_2'$ is the difference of the bound energy before and after the reaction. A negative energy difference means that the reaction is *endothermic*, or that an input of energy is required for the reaction. The opposite is the *exothermic* reaction, a reaction that releases heat.

Elastic Collision of Hard Spheres with Energy and Momentum Conservation

There are different approaches to collision response.

One approach solves momentum and energy conservation equations. *CollisionConserving* uses a thorough but inefficient method to calculate the response. The equations are:

$$\begin{array}{rcll}
\vec{p} & = & m_1\vec{v}_1 + m_2\vec{v}_2 & \text{(momentum before collision)} \\
\vec{p}\,' & = & m_3\vec{v}_3 + m_4\vec{v}_4 & \text{(momentum after collision)} \\
\vec{p} & = & \vec{p}\,' & \text{(momentum conservation)} \\
m_1v_1^2 + m_2v_2^2 & = & m_1v_3^2 + m_2v_4^2 & \text{(energy conservation)} \\
\vec{d} = |\vec{r}_1 - \vec{r}_2| & \| & \vec{p} - \vec{p}\,' & \text{(impulse transfer)}
\end{array}$$

With the three equations of momentum and energy conservation and the impulse transfer, a definite solution can be calculated. This represents a classic hard sphere collision in three dimensions. We ignore mass changes ($m_1 = m_3$ and $m_2 = m_4$) and bound energy difference ($\Delta E = 0$) for the moment. We proceed as follows:

1. Work with relative distance $\vec{d}$. We can think particle 1 as being in the origin at the time of the collision.

2. Use the velocity of particle 2 a as reference frame . This way we can think sphere 2 as resting.

3. Rotate the reference frame such that the x and y coordinates of particle 2 become zero. Rotate around the z-axis by angle $\theta_2, \cos(\theta_2) = d_z/|d|$ and around the y-axis by angle $\phi_2, \tan(\phi_2) = d_y/|d|$. The rotation matrix is:

$$\begin{pmatrix} \cos(\theta_2)\cos(\phi_2) & -\sin(\theta_2) & \cos(\theta_2)\sin(\phi_2) \\ \sin(\theta_2)\cos(\phi_2) & \cos(\theta_2) & \sin(\theta_2)\sin(\phi_2) \\ -\sin(\phi_2) & 0 & \cos(\phi_2) \end{pmatrix}$$

4. These steps simplify the situation. The z components are the only velocity component that changes (because impulse change is parallel to the z axis). The z component of the new velocity of particle 2 is:

$$\vec{v}_4|_z = \frac{2\vec{v}_1|_z}{1 + m_2/m_1}$$

5. Finally reverse step 2 and 3.

It is possible to consider bound energy difference and mass changes. The formula of step 4 gets more complicated, but the steps stay the same.

The problem with this approach is that it uses trigonometric functions for the rotation matrices. The Java implementation of the trigonometric calls are precise but slow. The hardware instructions of sin and cos on x87 FPU have quite significant errors outside the range $[-\pi/4 \dots \pi/4]$[5] and are not used by the Java Virtual Machine, The implementations provided by Java are a lot slower (see [Sun03]).

The approach of elastic collision with momentum and energy conservation therefore turned out to be rather slow. A reimplementation as a native method in C would be helpful but remains to be implemented. *CollisionConserving* was implemented only as a test class and never has been retrofitted to the plug-and-play architecture. It is therefore not available in ‘saces’.

[5] In Java the expression `Math.sin(Math.PI)` does not yield 0 but a very small non-zero value because of rounding errors of π. This value is different after the fifth digit for fsin of the x87 FPU. Then the FPU only works within the range $\pm 2^{63}$.

Central Collisions

In [Hin05a] a collision as described in [HS04], page 155, was proposed:

$$\begin{aligned} \vec{v}_3 &= \frac{m_1 - m_2}{m_1 + m_2}\vec{v}_1 + \frac{2m_2}{m_1 + m_2}\vec{v}_2 \\ \vec{v}_4 &= \frac{m_2 - m_1}{m_1 + m_2}\vec{v}_2 + \frac{2m_1}{m_1 + m_2}\vec{v}_1 \end{aligned}$$

where $\vec{v}_1$ and $\vec{v}_2$ the velocities of particle 1 and 2 before collision, $\vec{v}_3$ and $\vec{v}_4$ after collision; and m_1 and m_2 their masses. Inelastic collision is:

$$\vec{v}_3 = \frac{m_1}{m_1 + m_2}\vec{v}_1 + \frac{m_2}{m_1 + m_2}\vec{v}_2$$

This is a simplified elastic central collision. When we wanted to add mass changes and bound energy difference to this model, we ran into problems. When we discussed them with our diploma supervisor Dr. Schwab, he suggested an alternative approach.

Collision with Random Scattering

With the reference frame of the mass center of the two particles, the problem is reduced to a head-on collision. After the collision the particles move away from each other exactly in two anti-parallel trajectories. The scattering angles can be determined randomly. The approach is:

1. Calculate velocity of mass center $\vec{v}_{cm} = \frac{m_1\vec{v}_1 + m_2\vec{v}_2}{m_1 + m_2}$
2. Translate to the mass center reference frame $\vec{w}_1 = \vec{v}_1 - \vec{v}_{cm}, \vec{w}_2 = \vec{v}_2 - \vec{v}_{cm}$
3. Get a random unit normal vector $\vec{n}, |\vec{n}| = 1$ as a scattering direction. (For each component of $\vec{n}$ find a uniformly distributed random number in range $[-1 \ldots 1]$. Then normalize the vector as follows: $\vec{n}_N = \vec{n}/|\vec{n}|$.)
4. The resulting velocities are $\vec{v}_1{}' = \vec{v}_{cm} + \vec{n}_N|\vec{w}_1|$ and $\vec{v}_2{}' = \vec{v}_{cm} - \vec{n}_N|\vec{w}_2|$.

This method does not use any trigonometric functions, three square roots and three random numbers and is faster than elastic collision with hard spheres. Without energy difference and mass changes it is an elastic collision.

A side note. In the first implementation we forgot the minus sign for the resulting velocity of the second particle. The velocities should be antiparallel because after a collision the particles move away from each other (in the reference frame of mass center). With two normal vectors pointing in the same direction we get a very interesting and quite a funny effect. After collision the particles are glued to each other and build compounds. We decided to keep this mistake as a separate class *ResponseSchwabFunnySticky*.

Energy Difference and Mass Changes

While the first approach with a Monte-Carlo method works well and is available as *ResponseSchwab*, it does not consider mass changes and bound energy difference. For the successor *ResponseSchwab2* handling energy difference and mass changes we use:

$$\frac{1}{m_3}\sqrt{\frac{m_3 m_4}{m_3+m_4}(|\vec{w}_1|^2(m_1+m_2)\frac{m_1}{m_2}+2\Delta E)}$$

in the place of $|\vec{w}_1|$ and:

$$\frac{1}{m_4}\sqrt{\frac{m_3 m_4}{m_3+m_4}(|\vec{w}_2|^2(m_1+m_2)\frac{m_2}{m_1}+2\Delta E)}$$

in the place of $|\vec{w}_2|$. *ResponseSchwab* becomes a special case with $m_1 = m_3$, $m_2 = m_4$ and $\Delta E = 0$.

How did we find these terms? We start with energy conservation and use momentum vectors instead of velocity vectors:

$$E = \frac{|\vec{p}_1|^2}{2m_1} + \frac{|\vec{p}_2|^2}{2m_2} = \frac{|\vec{p}_3|^2}{2m_3} + \frac{|\vec{p}_4|^2}{2m_4} - \Delta E$$

where $\vec{p}_1$ is the momentum of particle 1 and $\vec{p}_3$ the momentum of the same particle after collision. Note the term ΔE which adds energy difference to the collision.

Isolating $|\vec{p}_3|^2$ and using $\vec{p}_1 = -\vec{p}_3$ and $\vec{p}_2 = -\vec{p}_4$, because in the mass reference frame the momentum vectors of the two particles before and after collision are antiparallel and of same length, we get:

$$|\vec{p}_3|^2 = \frac{m_3 m_4}{m_3+m_4}(|\vec{p}_1|^2\frac{m_1+m_2}{m_1 m_2}+2\Delta E)$$

Then we divide the momenta by the masses of the particles, calculate the square root, and we have the desired terms.

Negative Bound Energy Difference for Endothermic Reactions

Negative bound energy difference should not make the terms inside the square roots negative. We have found that particles with high velocities grazing at each other can yield negative terms. The concept ‘activation energy’ does not help here to avoid impossible energy scenarios. Important is not only the high kinetic energy, but also some sort of collision impetus or collision energy. That is why *transform()* has an additional check after activation energy to avoid this problem.

Design by Contract

Because the math is at times rather difficult to realize in Java code and tiny typos can have catastrophic or, even worse, unnoticed small effects, `assert` is used to formulate pre- and post conditions and invariants. This is *Design by Contract* ™, a methodology based upon the metaphor of a legal contract.[6] Physical conservation laws are very good contracts: easily defined and verified. Energy and momentum before and after collision are compared, and if they differ, an *AssertionError* is thrown and the ‘saces’ is halted.[7] At times it was quite frustrating to find out what went wrong, but if all was well the asserts were left in the code, maintaining conservation laws. Note, however, that some asserts assume mass conservation.

6.4.10 Decay Reactions — *decay()*

Particles can decay with specified decay reaction probabilities.

In *DecayerWithEnergy* activation energy is not considered in a decay reaction. But a decay reaction cannot be activated if there is not enough bound energy:

$$E_e \geq E_{p_1} + E_{p_2}$$

Or, in other words, there are no endothermic decays.

Excess energy is transformed into kinetic energy. That means the decayed particles are faster than the educt particles (assuming mass is same before and after). The collision response approach with random scattering can be reused here. The steps:

1. Create a collision pair and the educts

[6] The object-oriented Eiffel programming language was created to implement *Design by Contract* (DBC for short). However, the ideas behind DBC are applicable to many programming languages. Bertrand Meyer is the initial designer of the Eiffel method and language.

[7] Only if asserts are enabled by the `-ea` option to the Java virtual machine.

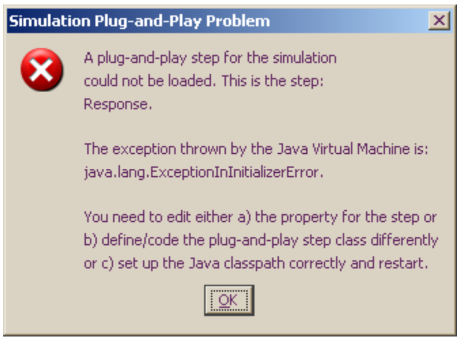
2. Set the educt positions and velocity to the ones of the product
3. Set the two educt masses of the collision pair both to half the educt mass $m_e/2$.
4. Calculate bound energy difference $\Delta E = E_e - E_{p_1} + E_{p_2}$.
5. Call *response()*.

Other approaches perhaps need to calculate the scattering angles differently and cannot pass the pair to the collision response. That is why *decay()* is the last step of calculating the simulation. *DecayerWithEnergy*, however, reuses *ResponseSchwab2* by composition. A private *ResponseSchwab2* member variable is used.

6.5 How to Implement the Plug-and-play Classes

The idea, in principle, is simple. Just write a class which implements the interface of your choice. What the method does exactly, is up to you. There are a few intricacies, however.

1. The class should be in the classpath of ‘saces’. Usually this means using the `-classpath` option of the `java` command. Please refer to the JDK documentation from Sun Microsystems. A simple solution is using package-less classes in the same directory as *saces.jar*.



Simulation Plug-and-Play Problem

A plug-and-play step for the simulation could not be loaded. This is the step: Response.

The exception thrown by the Java Virtual Machine is: java.lang.ExceptionInInitializerError.

You need to edit either a) the property for the step or b) define/code the plug-and-play step class differently or c) set up the Java classpath correctly and restart.

Figure 6.9: An error message for a plug-and-play problem

2. The classes are loaded with Java reflection. With reflection many things can happen. If loading or running a simulation an error message is shown like in figure 6.9, consult the list of possible plug-and-play problems below to find out what you can do.

3. The step must run fast. There is only a few milliseconds per step available.
4. The data flow (see figure 6.8, page 58) must be respected.

The List of Possible Plug-and-play Problems

ClassNotFoundException if the Java classloader did not find the class. The user must set the classpath or move the class to the same directory as the *.jar* file. If the class has package, the class file must be put into the corresponding tree of subdirectories as Java requires it.

ClassCastException if the instance could not be casted to the given interface. The class must implement the corresponding interface. For example, to implement the *detect()* step, write a class that implements the interface *saces.pnp.Detector*.

ExceptionInInitializerError if during class initialization an exception has been thrown. Try to write a test program that creates the instance and see what happens.

InstantiationException if the class is abstract or an interface. The class must be a concrete class.

IllegalAccessException if the constructor is not public. There must be a public constructor without parameters.

Linkage Errors (rare) if the class file has a wrong format or could not be verified and other linkage problems.

No Property Key (a bug: defaulting of properties failed) if there is no such experiment property. Try to define the property with the correct key in the ‘Properties’ tab of the ‘Settings’ window as a workaround of this bug.

6.6 Space Partitioning

The naive, brute force collision detection is a double loop:

```
for each particle i in the reaction vessel do
  for the remaining particles k after particle i do
    does particle i intersect particle k?
```

and needs $n(n+1)/2$ iterations. The time complexity of $O(n^2)$ is not acceptable for a large number of particles.

Partitioning is based on the idea that we need to check only particles that are near to each other. It does not make sense to check if two particles intersect, if they are in the opposite corners of the reaction vessel.

‘saces’ uses a variant of *Binary Space Partitioning*, a technique for collision detection in games. A BSP tree is a recursive, hierarchical partitioning of an n-dimensional space [N$^+$95]. BSP works well with varying densities of objects. Where there are more objects, the partitioning is finer. *Space Partitioning* of ‘saces’ is not recursive and hierarchical. In a reaction vessel the particles are usually evenly distributed and we do not need the hierarchy.[8]

DetectorPartitionized first partitions the cuboid of the reaction vessel into smaller sub-cuboids. With a constant number of particles per partition N (usually a low number between 10 and 50), the collision detection becomes:

```
for each particle i do
  find the partition of particle i

for each partition p do
  for each particle i in partition p do
    for the remaining particles k in partition p do
      does particle i intersect particle k?
```

The first loop finds out in which partition a particle is located. The particles are moving and do not stay forever in the same partition. It has n iterations. The second loop has n/N iterations and its inner loop is the naive collision detection, only that it works on N particles of the partition. We have

$$n + \frac{n}{N}\frac{N(N+1)}{2}$$

[8] There is a problem however. Collisions at partition boundaries are not detected. This happens if two particles intersect but their centers are in different partitions.

iterations. If we fix $N = 12$, we get $7\frac{1}{2}n$ iterations for collision detection, therefore a time complexity of $O(n)$. But there is a big overhead, and we are not able to see whether it is really linear. Anyway, *DetectorPartitionized* seems to be better for 300 particles and more [9] (see tables 6.3 and 6.4).

Particle Count	Running time of step (ms)	Particle throughput per ms
10	0.5	22.2
30	0.8	35.7
100	8.5	11.7
300	73.5	4.1
1000	751.8	1.3

Table 6.3: Running times and particle throughput of *DetectorSimple*

Particle Count	Running time of step (ms)	Particle throughput per ms
10	0.7	13.9
30	5.6	5.4
100	43.8	2.3
300	54.7	5.5
1000	134.1	7.5
3000	315.8	9.5
10000	911.5	11.0

Table 6.4: Running times and particle throughput of *DetectorPartitionized*

It is important that the number of particles per partition is fixed (in fact it is the experiment property `DetectorPartitionized.particleCountPerPartition`). If there are more particles, the partitioning will be finer. We calculate the number of partitions $n_P = n/N$ and try to find a subdivisioning of the reaction vessel cuboid in three dimensions that yields the number n_P of partitions. This is not always possible, because not all numbers are factors of three whole numbers, but this approach gives adequate numbers:

$$\begin{aligned} n_x &= \lfloor \sqrt[3]{n_P} + \frac{1}{2} \rfloor \\ n_y &= \lfloor \sqrt{n_P/n_x} + \frac{1}{2} \rfloor \end{aligned}$$

[9] The measurements were done with a Java profiler. Profiling imposes an overhead slowing down the simulation performance. In practice we had 50% or better running times than the measurement values in the table.

$$n_z = \lfloor \frac{n_P}{n_x n_y} + \frac{1}{2} \rfloor$$

The results of this step are written to the binary log. Suppose we have 2000 particles and want 12 particles per partition on average. The requested partition count is 166, the number of subdivisions is 6, 5, 5 and therefore we get 150 partitions and 13.3 particles per partition.

6.7 The Binary Logger

What is the sense of a beautiful visual simulation, if no hard data is available? What if the user wants to know how many particles of which particle class are currently in the reaction vessel?

To keep such a time-critical application such as ‘saces’ animating smoothly, data logging must be as efficient as possible. We decided to exploit Java’s well-defined internal representation of binary data. In dropping the conversion to text, we gain performance.

We also want to decouple simulation and data visualization. We started with the Observer pattern which provided a good decoupling between data ‘production’, the *Subject* and its *Observers*, the ‘consumers’ of data. But performance-wise the decoupling was not so great. If a CPU-intensive observer registered with the simulation, animation performance suffered. We decided to leverage the file system. Operating systems of today have highly optimized file systems with lazy buffering. Writing data to files does not mean that the software must wait till the hard disk spun up and wrote every bit. Another advantage are network-oriented file systems. We get distributed processing almost for free. A workstation is visualizing a simulation, another workstation accesses the binary log over the network and displays diagrams. Invoke (see table 4.5, page 34 about command line arguments):

```
java -cp jogl.jar -jar saces.jar -view experiment.xml saces.binlog
```

The structure of the binary logger is simple. It uses a record-oriented binary format with eight different records and a timestamp for every record. The binary logger emits a stream of records whose sizes are multiples of 8 bytes, and whose ordering in the stream is free. A histogram record can be followed by a reaction record, then by a snapshot etc. The only exception is the first record of a binary log which should be the MAGIC record. The ‘saces’ binary log is marked as such with this record. All records start with an eight-byte magic number, whose last nybble[10] is the record identifier and an eight-byte time stamp: the number of milliseconds since midnight, January 1st, 1970, as produced by the Java method `currentTimeMillis()`.

The only types written to the log are the record identifiers (8 bytes), the time stamps (8 bytes), integers (4 bytes), floating point numbers (4 bytes), strings in UTF-8 format with a two-byte length tag as written by the method *writeUTF()* as defined in the interface *DataOutput*, and zero bytes to pad to a multiple of 8 byte.

The binary logger assumes that the data viewer has access to the experiment definition. Without the experiment definition the binary data is uninterpretable, because only indices to the list of particle classes are written, for example.

The MAGIC Record (Nybble 0)

The magic record starts with these bytes `73416345732100f0` in hexadecimal. This magic number is ASCII for the string `sAcEs!`, followed by a null byte and a final byte `0xf0` in hexadecimal. The last nybble (0 for the MAGIC record) identifies the record type. The magic is chosen so that ‘saces’ binary logs are easily recognized and to facilitate resynchronization in case of data corruption (that record sizes are multiples of 8 bytes also helps here). The MAGIC record is the only record consisting of only 16 bytes, the magic and the time stamp.

The REACTION Record (Nybble 1)

The REACTION record has been dropped because it has turned out to be too expensive to log every single reaction to the binary log.

The PARTICLECLASSES Record (Nybble 2)

The PARTICLECLASSES record contains the count of the particles per particle class. If an experiment has nine particle classes, $4 \cdot 9 = 36$ bytes and 4 additional zero bytes to pad up to the length of 40 bytes are written to the binary log. The particle classes are ordered same as in the experiment definition.

[10] A nybble is half a byte or 4 bits. The name is a play with words bit and byte — a nibble is a small bit and a nybble a small byte. A nybble is written with only *one* hexadecimal digit and can contain numbers from 0 to 15.

The SNAPSHOT Record (Nybble 3)

The SNAPSHOT record contains the count of all particles, then for each particle its particle class index and six floating-point numbers; three for the position and three for the velocity. It is a very big record. For 1000 particles, $16 + 4 + (4 \cdot 7 \cdot 1000) + 4 = 28020 + 4$ bytes (or more than 27 kilobytes) are written.

The 📷 ‘Snapshot’ button causes such a snapshot to be saved to the binary log, where it can be converted to text and be processed by other software.

The HISTOGRAM Record (Nybble 4)

Speed histograms show how particle speeds are distributed. The record has four floating point values for the minimum, average and maximum speed and for the top speed of the histogram. Maximum particle speed and upper limit of the histogram do not need to coincide. Then the number of histogram speed subdivisions and the number of particles in the speed subdivisions are written. Suppose there are 10 subdivisions between speed zero and histogram top, then $16 + 4 \cdot 4 + 4 + (4 \cdot 10) + 4 = 80$ bytes (including 4 bytes of padding) are written.

The PROPERTY Records (Nybbles d, e and f)

A name-value pair can be written to the binary log. The name is written by the method *writeUTF()*. After the name the value follows. It can be of type int, float or String, depending on the record type. Padding is also applied here.

The simulation process can write any data to the binary logger. Especially the PROPERTY records are flexible and the data is easily identified by its name. The partitioning algorithm, for example, writes how many partitions were requested, the number of partitions for each dimension and the estimated average particle count per partition as five PROPERTY records with the names “RequestedPartitionCount”, “PartitionCount[X]”, “PartitionCount[Y]” “PartitionCount[Z]” and “AverageParticleCountPerPartition”.

Chapter 7

Java OpenGL (JOGL)

7.1 A Bit of History

OpenGL grew out of the IRIS GL specification written by Silicon Graphics and is now regulated by the OpenGL Architecture Review Board (ARB). The ARB exists since 1992 and consists of several companies with the likes of Apple, ATI, Dell, Hewlett-Packard, IBM, NVidia, SGI and Sun Microsystems. Microsoft, one of its founding members, left the ARB in 2003. This is not surprising considering Microsoft's decision to not fully support the OpenGL standard in Windows Vista. Nevertheless, OpenGL remains one of the most robust, portable, widely used 3D graphics API. It is also cherished by many developers for its simplistic, straight forward interface.

'saces' uses JOGL, the OpenGL port for Java. JOGL does not add any functionality to the OpenGL standard. Structurally, it is merely a JNI mapping of the OpenGL functions to Java classes. This is an advantage for developers already familiar with the OpenGL modus operandi. The Java OpenGL code therefore bears a striking resemblance to OpenGL, used in the C programming language, for which it was originally conceived. OpenGL is not object oriented and functions basically as a state machine. The object-oriented Java wrapper does not change this operative characteristic of OpenGL.

Java3D, the object-oriented 3D API written by Sun, was also evaluated as a candidate for 'saces' 3D rendering. As mentioned in the project documents, the decision not to use Java3D was based on the fact that it appears to have been abandoned by Sun. Other than that, it is a very large and complicated API implying steep learning curve. JOGL on the other hand, is as intuitive as OpenGL.

The following chapter will discuss the algorithms and implementations concerned with the graphical representation of ‘saces’. This chapter may make a lot more sense to a reader familiar with the general usage and internals of OpenGL, although the authors have tried to describe the algorithms and implementation in a straight forward manner, avoiding OpenGL technicalities that would only complicate certain explanations. This chapter will start out by describing the objects that need to be drawn by OpenGL for the application, and then we will discuss the viewing modes, lighting and the detail levels implemented by ‘saces’.

7.2 ‘saces’ and OpenGL

Display Lists

Before we proceed with the actual objects, it might be useful to understand how ‘saces’ draws the objects. This is where display lists come in. Display lists are used by OpenGL to increase drawing performance. OpenGL does this by compiling the routines necessary to draw the desired object. As the reader might guess, display lists cannot be modified once they have been compiled. Their main usage is therefore limited to objects that do not change their geometrical appearance, such as the spheres and bounding box in ‘saces’. Positioning of the spheres, plus the shading and lighting, are done outside the display list. Using a display list for the cuboid (bounding box) does not yield drastic performance boosts because of the cuboid’s very simple geometry. The spheres are a little more complicated and use trigonometry to calculate the angles needed for surface subdivision. This is where a display list is advantageous, if not a prerequisite for smooth animations with numerous particles. Now, let us move on with the geometrical objects.

The Particles

The particles in ‘saces’ are in most cases represented by the GLU[1] sphere. The GLU sphere uses surface subdivision to approximate a sphere. The number of vertical (stacks) and horizontal (slices) subdivisions are the parameters used for adjusting the detail of the sphere. ‘saces’ uses sixteen stacks and slices for the approximation in medium and high detail mode. In low detail mode, the spheres are drawn with half the number of slices and stacks. This level of detail is useful when simulating more particles than the host machine can handle in medium or high detail.

[1] OpenGL Utility Library

The Cuboid

The cuboid represents the bounding box of the experiment, a rectangular reaction vessel wherein the particles are contained. It is coded as six polygons, sharing eight vertices. The wire frame representation is drawn on top of the polygons to accentuate the edges of the cuboid.

7.3 Viewing Modes

'saces' uses three viewing modes for visualization. Two of the three are frustum[2] perspective views with a viewing angle of forty five degrees. The third is an orthographic projection which draws three view ports: One from the top, front and side of the cuboid.

Perspective View

The default perspective view (and the particle view) allows the user to rotate the simulation space as if it were circumscribed by an invisible sphere with a radius identical to the distance from the cuboid center to any of its eight vertices. This is a user interface known as the virtual sphere, virtual track ball or cue ball and is attributed to [CMS98].

The virtual track ball algorithm functions by mapping 2D mouse cursor position to the invisible bounding sphere and then calculating the rotation of the cuboid by measuring the rotation angle of the sphere. The default perspective view also implements a zoom in and out function that moves the camera towards and away from the center of the cuboid with the mouse wheel or with the up and down keys.

'Be a Particle' View

The 'Be a particle' view (or particle view), lets the user observe the simulation from the view of a random particle moving around the reaction vessel. By default, the camera points to the center of the cuboid, although this can be changed by the user with the methods described in the virtual track ball section. Zooming is not implemented for the particle view.

[2] A frustum is the basal part of a solid pyramid or cone, formed by cutting off the top by a plane parallel to the base. It represents the viewing volume of a viewport.

Orthographic View

As previously mentioned, orthographic viewing is more of a schematic interpretation of the simulation. It draws the simulation space in three profiles, the way it would be drawn using paper and pencil. Since orthographic viewing does not use perspective, objects (particles) further away from the camera do not appear smaller than they actually are. This is the behavior one would expect from CAD/CAM applications.

7.4 Detail Levels

'saces' uses three detail levels to leverage performance/detail tradeoff. The three detail levels can be used in any of the three viewing modes described in the viewing modes section.

Low Detail

Low detail level strips the simulation of lighting, shading, z-buffer depth testing and surface all together. This may remind some readers of the antiquities vector based arcade games such as Lunar Lander from Atari (1979). Vector graphic display technology was in part conceived for the Apollo space program in an attempt to create a system capable of simulating the moon landing. Using a bare-bone display model such as this can increase the frame rate and helps keep the animation running smoothly when simulating a large number of particles. When using a high-end video cards however, performance difference might not be as drastic because rendering of the higher level detail (lighting, shading etc.) is hardware-accelerated.

Medium Detail

Medium detail level uses lighting, smooth shading and the z-buffer[3]. When using lighting, material properties[4] need to be assigned to all surfaces appearing to

[3] The z-buffer is a facility used for depth testing. Depth testing enables OpenGL to recognize which objects are visible and which are obscured by other objects in relation to the viewpoint. Without the z-buffer, objects which are drawn last, are simply drawn over existing objects.

[4] Material properties can be assigned to OpenGL primitives to define how these primitives react to lighting. Material properties include the color and intensity of the primitive's light emission and the material's shininess.

be lit. This, together with the smooth shading and depth testing functions, influences performance substantially. On newer machines and in most cases, however, performance should be acceptable in this viewing mode.

High Detail

Although visual reflections do not exist on an atomic level, together with lighting, they help increase the feeling for 3D depth perception.

High detail mode adds reflection to the inner walls of the cuboid and translucency to the outer walls. Removing three sides of the cuboid to expose the particles therein is an alternative to back face culling[5] used in medium detail level mode. The reason for choosing this alternative is as follows.

The costly thing about reflecting objects on a surface is that the objects must be drawn for each reflecting surface. In our case, this means drawing all the particles three times (once for each visible wall). And it does not stop here: The translucency of the outer walls can also be made visible by rotating the simulation space to its backside. Since translucency works pretty much the same way reflection does, this implies that the particles need to be drawn another three times. After drawing the entire cuboid in high detail level mode, each particle has been drawn six times and needs to be drawn one more time; where they really are (in the reaction vessel). Below is a sketch of the algorithm used by ‘saces’ for creating reflections and translucency. Quake[6] fans might recognize this basic technique used by John Carmack to implement the reflections in OpenGL Quake. It uses the stencil buffer[7] and alpha blending[8] to attain the effect.

First, the three inner walls of the cuboid are drawn to the stencil buffer instead of to the color buffer. This step is invisible because stencil buffer information is not implicitly used by OpenGL when rendering.

The particle geometries are then scaled and combined with the content of the stencil buffer for each wall. Scaling results in a mirror representation of the particles when calculated with the following factors:

[5] Culling can be used for hidden surface removal in 3D rendering. It can also be used to make the back or front face of a polygon invisible. This effect is used by ‘saces’ to hide the walls that obscure the inside of the cuboid

[6] Quake is a popular first person shooter game written by id Software

[7] The stencil buffer is an OpenGL facility that can be used for caching and combining off-screen pixel data.

[8] Alpha-blending is a technique that can be used to create transparency and translucency. An alpha value clamped between 0 and 1 can be interpreted as the opacity factor of the object.

$(-1, 0, 0)$ for x-axis, $(0, -1, 0)$ for y-axis and $(0, 0, -1)$ for z-axis mirroring. Not using the stencil buffer for the mirroring operation would result in three new sets of particles visible outside the cuboid. The stencil buffer therefore works the way a stencil would in the real world.

Now that we have drawn the reflections to the inner walls using the stencil buffer, we have to make the reflections visible. This is done by drawing the inner walls again, this time using alpha-blending to blend the mirrored particles with the new inner walls. The alpha value is clamped between zero and one. Using one as the alpha value would make the inner walls opaque, zero would make the inner walls invisible and the reflection effect would be mute. ‘saces’ blends with an 0.9 alpha value to attain the effect of a glassy type of reflection.

After we have completed the inner wall rendering, all that is left to do is to repeat the process for the outer walls to make them appear translucent. The front surface of the outer wall and the back surface of the inner wall are culled and drawn into the same position, resulting in one visible wall with the depth of one pixel.

A note on lighting: To make sure that the reflections look authentic, the particle materials have to be calculated using lighting before they are combined with the stencil buffer. The same goes for the cuboid.

7.5 Lighting

Lighting is a very important part of any 3D visualization. It lends the viewer of a scene an increased feeling of the position and interaction of the objects depicted therein. ‘saces’ uses only one light source when rendering the scene because any additional light sources would decrease performance drastically on non-accelerated video cards. The light source is located on one of the top corners of the box, giving the viewer a clearer impression of where a particle is located in relation to other particles during simulation.

7.6 OpenGL State Variables

The table below illustrates the OpenGL state variables[9] and their settings, used when rendering the simulation scene in low, medium and high detail as a reference.

Table 7.1: OpenGL state variables in the detail levels

	Low	Medium	High
Shade model	None	`GL_SMOOTH`	`GL_SMOOTH`
Lighting	Disabled	Enabled	Enabled
Culling	Disabled	Enabled	Enabled
Polygon mode	`GL_LINE`	`GL_FILL`	`GL_FILL`
Depth test	Disabled	Enabled	Enabled
Stencil test	Disabled	Disabled	Enabled
Normalize	Disabled	Disabled	Enabled
Blending	Disabled	Disabled	Enabled

[9] OpenGL, being a state machine, allows the programmer to set and query state variables. State variables have default values, and remain constant until modified.

Appendix A

Known Problems

As for any software system exceeding a certain size, 'saces' is not free from bugs and other problems. This appendix lists them.

- It is possible to save files with extensions other than *.xml*.
- Restarting experiments can confuse the data viewer.
- Decay reactions are not tested as carefully as other reactions. There are problems with energy conservation. Some experiments can cool down or heat up, even when they should not.
- An attempt is occasionally made to remove a non-existent particle from the simulation. If such a thing happens, a notification is printed to the standard output.
- There is a logical memory leak. Running 'saces' over extended time can slow down the workstation considerably.
- 'saces' was never tested on Linux.
- 'saces' has a serious problem on the Mac OS X; it crashes very soon.
- Decay reactions like $X \rightarrow Y$ or even $X \rightarrow -$ are not implemented.
- Input validation of probabilities automatically resets probabilities over 1 to 0 and beeps. (It should block the input field, not reset.)
- Some steps of the simulation process are very important and should not be replaced. Basic functionality of the process should not be pluggable.

- A defined random seed does not always make the experiment reproducible. Reproducibility also depends on the simulation performance. On slow workstation more collisions are missed than on fast ones.
- Some (older) video cards seem to have problems in high detail mode. The reflections are drawn, but also three sets of ghost particles.
- Some (older) video cards do not handle lighting correctly for a large number of particles. The light seems to move inside the reaction vessel for more and more particles.

  It is suggested to use low detail mode or to use a better video card.
- Unexpected Java exceptions are caught and presented to the user with the option to exit or continue. If continuing, ‘saces’ may behave erratically or even crash.

  It is suggested to exit.
- Partitioning for the collision detection is done only once, when the simulation starts. If the particle number increases dramatically this can lead to performance degradation. The subdivisioning of the reaction vessel should be recreated every few seconds.
- When viewing in Orthographic mode, the axis labels are sometimes obscured by the reaction vessel.

Bibliography

[Adl94] Leonard M. Adleman. Molecular computation of solution to combinatorial problems. *Science*, pages 1021–1024, 1994.

[Amb00] Scott W. Ambler. Writing robust java code: The AmbySoft Inc. coding standards for java v17.01d. AmbySoft Inc., `http://www.ambysoft.com/javaCodingStandards.html`, 2000. White Paper for Java software developers.

[BFS02] Gil Benkö, Christoph Flamm, and Peter F. Stadler. A graph-based toy model of chemistry. Institut für Theoretische Chemie und Molekulare Strukturbiologie, Universität Wien, 2002.

[BFS03] Gil Benkö, Christoph Flamm, and Peter F. Stadler. Generic properties of chemical networks: Artificial chemistry based on graph rewriting. *Advances in Artificial Life — Proceedings of the 7th European Conference on Artificial Life (ECAL 2003)*, pages 10–19, 2003.

[CMS98] Michael Chen, S. Joy Mountford, and Abigail Sellen. A study in interactive 3-d rotation using 2-d control devices. *Proceedings of SIGGRAPH*, pages 121–129, 1998.

[Coo03] Stephen Cook. The P versus NP problem. *Journal of the ACM*, 50/1:27–29, 2003.

[Dav04] Gene Davis. *Learning Java Bindings for OpenGL (JOGL)*. AuthorHouse, 2004.

[Dev02] Keith Devlin. *The Millenium Problems: The Seven Greatest Unsolved Mathematical Puzzles of Our Time*. Basic Books, 2002.

[DZB01] Peter Dittrich, Jens Ziegler, and Wolfgang Banzhaf. Artificial chemistries—a review. *Artificial Life*, 7:225–275, 2001.

[FG04] Corina Frutschi and Pascal Grossniklaus. Simulation ausgewählter DNA-Computing Operationen. Diploma, Hochschule für Technik und Informatik Bern, Abteilung Informatik, 2004.

[Fra03] Irmgard Frank. Chemische Reaktionen „on the fly“. *Angewandte Chemie*, 115:1607–1609, 2003.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and Johnson Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[Hin05a] Thomas Hinze. *Arbeitsmaterial zum Diplomprojekt Ein Experimentiersystem zur Simulation des Ablaufs chemischer Reaktionen*. Technische Universität Dresden, Institut für theoretische Informatik, Arbeitsgemeinschaft Rechnen mit Molekülen, 2005. Preliminary specification for the ‘saces’ tool.

[Hin05b] Thomas Hinze. *Beispielanwendungen der Künstlichen Chemie zur Lösung von Aufgaben aus Mathematik und Informatik*. Technische Universität Dresden, Institut für theoretische Informatik, Arbeitsgemeinschaft Rechnen mit Molekülen, 2005. Description how to run finite automata in an artificial chemistry.

[HS04] Thomas Hinze and Monika Sturm. *Rechnen mit DNA*. Oldenbourg Verlag München, 2004.

[Kne90] Fritz Kurt Kneubühl. *Repetitorium der Physik*. Teubner Studienbücher, 1990.

[LCF04] Rodrigo G. Luque, João L. D. Comba, and Carla M. D. S. Freitas. Broad-phase collision detection using semi-adjusting BSP-trees. Instituto de Informática, UFRGS, Brazil, 2004.

[LJ05] Remo Lehmann and Bojan Jambrešić. Simulation der Arbeitsweise eines DNA-Chips. Diploma, Hochschule für Technik und Informatik Bern, Abteilung Informatik, 2005.

[N$^+$95] Bruce Naylor et al. A FAQ about binary space partitioning trees. `http://www.faqs.org/faqs/graphics/bsptree-faq`, 1995. FAQ.

[Nad02] Alexander Nadel. Backtrack search algorithms for propositional logic satisfiability: Review and innovations. Master’s thesis, Hebrew University of Jerusalem, 2002.

[NBH01] Andreas Ninck, Leo Bürki, and Roland Hungerbühler. *Systemik: Integrales Denken, Konzipieren und Realisieren*. Orell Füssli Verlag, 2001.

[SBB+00] Andre Skusa, Wolfgang Banzhaf, Jens Busch, Peter Dittrich, and Jens Ziegler. Künstliche Chemie. *Künstliche Intelligenz*, 1/00:12–19, 2000.

[Sce04] Eric R. Scerri. Just how ab initio is ab initio quantum chemistry? *Foundations of Chemistry*, 6:93–116, 2004.

[Sil05] Stephen Silver. Life lexicon, release 24. `http://www.argentum.freeserve.co.uk/lex_home.htm`, 2005. List of Game of Life terms; see term *Universal Computer*.

[Sun03] Sun Microsystems. Performance regression in trigonometric functions (very slow strictmath). `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4857011`, 2003. Bug report for java:classes_lang.

[Tom04] Kazuto Tominaga. Modelling DNA computation by an artificial chemistry based on pattern matching and recombination. Tokyo University of Technology, Hachioji, Tokyo, 2004.

[WKL+03] H.-J. Werner, P. J. Knowles, R. Lindh, M. Schütz, et al. Molpro, version 2002.6, a package of ab initio programs. Birmingham UK, `http://www.molpro.net`, 2003.

[WNDS99] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide Third Edition*. Addison-Wesley, 1999.

[You04] W. Anthony Young. Comparison of collision detection algorithms. University of Waterloo, Canada, 2004.