SodaSumo

An Extension to SodaConstructor

Kenneth Pak Kiu Lam
Supervised by Dr Mary McGee Wood
29 April 2008

Abstract

SodaSumo – An Extension to SodaConstructor

(original title: Sodarace)
by Pak Kiu Lam, Kenneth
Supervised by Dr Mary McGee Wood
11 April 2009

SodaConstructor allows users to create their own 2D life-like models using masses, springs and muscles, and change their behaviour by modifying physical values like spring stiffness, gravity and friction.

SodaSumo aims at introducing interactions among the virtual models created by SodaConstructor. It implements collision detection between models and allows the models to simulate a Sumo-style wrestling game, where 2 opponents attempt to push each other out of the ring. It allows users to understand basic collisions and, more importantly, opens up a whole new platform for competition between human and machine intelligence.

Although a public API for SodaConstructor was scheduled to release in late 2007 [1], it did not happen. For the sake of completeness and usability of SodaSumo, extra effort had to be made to reverse-engineer the SodaConstructor physics engine from scratch. The result was a success, and models generated in SodaConstructor can be used in SodaSumo, in which they wrestle each other sensibly.

Acknowledgements

First and foremost, I would like to thank my parents for realising my childhood dream which was to study abroad.

I also thank Dr Mary Wood for supervising my work, providing encouraging comments and giving me a lot of freedom in this project.

Tim Diggins from Soda Creative has spent a lot of time answering my questions and I would like to thank him here.

Finally, hats off to the Sodaplay community for trying out SodaSumo and all the constructive ideas.

SodaSumo

1 Introduction

The computer has always been used to simulate our surroundings, from the effects of an earthquake to how water flows in a stream. It was not until recent years that we realise computer simulations can be exploited for educational uses. The Sodaplay series is a number of Java applications developed to simulate simple physics like magnetic forces, collisions, spring tensions, and so forth. They serve two major purposes: To allow users to learn about simple physics in an interactive way; to create a platform for human and machine intelligence (AI) to compete, and genetic algorithms (GA) to develop.

SodaSumo is developed bearing the same principles with the addition of one idea: to enhance interactions between models. This can be done by introducing collision detection between different models. Through this a new way of competition will be unleashed, which will allow deeper developments in GAs with the performance of offspring cross-affecting each other through direct interactions. Since SodaSumo is about the extension of an existing application, it is essential for readers to know more about how they work.

1.1 SodaConstructor

The SodaConstructor allows the construction of 2D models with points representing masses and lines representing springs. The presence of muscles generate forces required to move the models, allowing them to behave like machines or living things. When the run button is pressed, the physics simulation will start, with the likes of Hooke's and Newtonian laws taken into account.

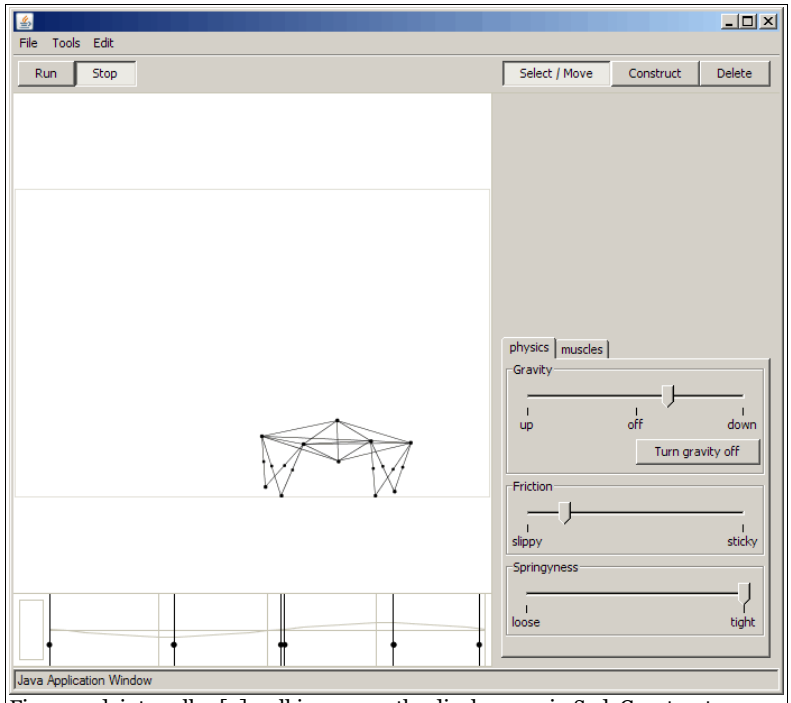

Figure 1: daintywalker[2] walking across the display area in SodaConstructor

The model will then move across the screen if built correctly (or move in any desired way). In the example of daintywalker [2], a model created by the Soda Creative team, it balances itself on its 4 legs, and walks back and forth driven by the 8 muscles which form the legs. In SodaConstructor, gravity, friction in springs and the “springyness”, which defines the stiffness of a spring, can be changed to alter the model's behaviour. An XML description of the model can be generated and exported by SodaConstructor, giving the possibilities of extensions.

1.2 Existing Work – Sodarace

After the release of SodaConstructor, the extension Sodarace was developed by the official development team for a race in moving speed between models. Five models can be loaded into the application simultaneously, which will attempt to move across an editable 2D terrain. A number of applications for the study of genetic algorithms have been developed by the community, and it is notable that some artificially altered models have beaten their original counterparts [3], which sparked heated debates on the forums.

Although the models appear in the same screen, they do not affect each other's motion and overlap. The same thing can be done by having the models “race” individually and comparing the number of frames taken. This is not a perfect way for the models to interact with each other, and the idea for 2 models to collide with each other spawned.

1.3 Project Proposal

SodaSumo is, like Sodarace, an extension to SodaConstructor which allows the behaviour of its models to be observed. It will allow two models to wrestle each other, after which, a score will be returned by a fitness equation, which evaluates how well the models are performing. The objectives of SodaSumo are:

1. import SodaConstructor models
    - understand how SodaConstructor describes its models with XML files
    - write an XML parser
2. make sure models behave like they do in SodaConstructor
    - understand how the SodaConstructor engine works
    - reverse-engineer the physics engine in SodaSumo (see note)
3. implement collision detection between models
    - write an algorithm for collision detection between moving point and lines
    - improve the performance of the processor-heavy component
4. animate the wrestling process
    - optimise drawing code to improve framerate
5. load models from www.sodaplay.com directly

With these objectives, it is hoped that the current SodaConstructor teaching tool will be improved through the addition of interactions between models. Furthermore, it should provide a new platform for virtual life models to develop and genetic algorithms to compete on.

Note: The SodaConstructor was not open-sourced in late 2007 as planned [1], and no source code was available for public use. The project plan had to be revised to include the studying and reverse-engineering of the SodaConstructor engine, which was a crucial component for generating the movement of the models.

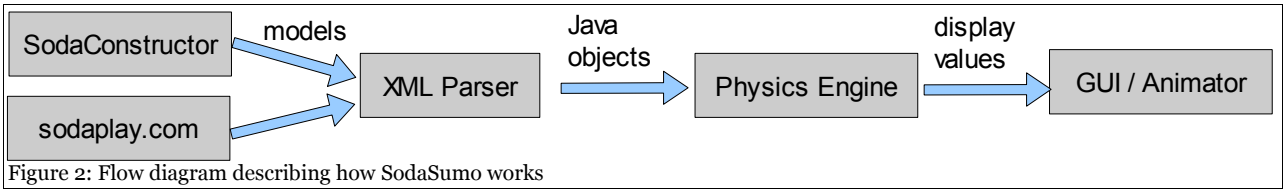

Figure 2: Flow diagram describing how SodaSumo works

1.4 Acronyms

For the purpose of clarity, SodaConstructor will be shortened to SC from now on. SodaSumo will take the acronym of SS and GA will represent genetic algorithm(s). Abbreviations common in computer science will be used in this report, e.g. AI for artificial intelligence and API for application programming interface.

1.5 Assumptions

It is assumed that readers have a good knowledge of simple trigonometry, mechanics and the Newtonian and Hooke's laws.

2 Background

This section covers detailed background information on the significance of SC and the use of virtual life models.

2.1 Simulations As a Learning Tool

Computer simulations are an important tool for engineers because they can be exploited at the early stages of the design process. Time and money can be saved by comparing different designs on the computer before moving on to create a working prototype. As the complexity of a problem increases, so does the time needed to plan, construct, evaluate and redesign elements of a system – the design-build-test cycle (DBT). With computer simulations, the cycle can be iterated easily, thus increases efficiency.

Similarly, scientists believe that by recreating authentic engineering practices, students can be stimulated and motivated to learn better. In research carried out by the Department of Educational Psychology of the University of Wisconsin, 12 middle-school students were invited to take part in a 10-hour virtual engineering work on the SodaConstructor. During the workshop, the students were allowed to explore on their own to make a standing structure. By switching between “Construct” and “Simulate” modes, the students could quickly edit their designs and obtain results. At first, most of them failed to succeed. But after multiple changes, a standing structure could be made by most of them. Before and after the workshop, the students were asked questions about “centre of mass”. By comparing the answers before and after the workshop, it was shown that the students gained understanding of the physical phrase “centre of mass” and were able to explain it in their own words. SC, seemingly a pointless game just for people to play around with masses and springs, is indeed a great tool to assist teaching.

In the example above, it shows SC can help students develop conceptual understanding of physics. The students ran through multiple iterations of the DBT cycle by switching between “construct” and “simulate” modes. Scientists believe that in a “microworld” such as SC, the learning process is enhanced because of three main ideas: Firstly, it contains sets of attributes which describe a particular domain (our physical world). Secondly, it allows users to repeatedly articulate ideas and obtain results (autoexpressivity). Thirdly, it allows creation, manipulation and test without much constraints, and users have the freedom to explore and sense of control (expressivity). With these the users will be motivated to explore, and more knowledge is thus obtained [4].

2.2 Artificial Life

2.2.1 Introduction

Artificial life focuses on synthesising “life-like” behaviours from scratch in computers. They can be used for studying biological evolution to reduce the amount of time needed. The problem with studying biological evolution naturally is that the process happens too slowly. Also, experiments can only be done with

organisms of small generation times, e.g. viruses and bacteria. Certain data can be difficult to obtain, and it is often impractical to make enough replicas for statistical accuracy.

There are a number of reasons why artificial life is preferred over biological life:

1. It provides an opportunity to seek generalisations about self-replicating systems. Organic forms have a common ancestor which relates to one evolving system (evolution theory). But with artificial life, we can study other forms of evolution.
2. It enables questions impossible to study with organic life forms to be addressed. In artificial life forms, every bit of memory can be altered without affecting the others. This kind of micromanaging is not possible in the natural system.
3. Speed. Questions can be addressed on a scale unattainable with natural organisms. With artificial life, a population of of 10000 organisms can have 20000 generations created per day, but the same process takes 10 years with bacteria, making artificial life the obvious choice.
4. Artificial organisms can be used to design solutions to computational problems where it is difficult to write explicit programs that produce the desired behaviour. With a simplistic view of evolution, other factors are ignored, which makes it more likely to give a large impact on problems not related to biology [5].

According to Daniel Dennett, evolution occurs whenever and wherever 3 conditions are met: replication, variation, and competition [6]. This fact allows evolution to be simulated on computer, normally achieved in the form of genetic algorithms.

2.2.2 Genetic Algorithms

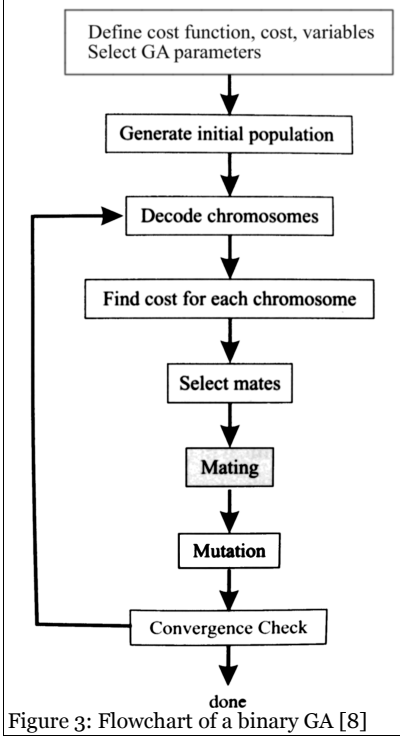

Figure 3: Flowchart of a binary GA [8]

Genetic algorithm (GA) is an optimization and search technique [8] inspired by natural selection where strong individuals are more likely to win in a competing environment. In a GA, the following components are essential: attributes to define an organism, a task for organisms to perform, and a cost function to tell how well they do (fitness). First of all, a pool of organisms are initialised or created randomly, usually in the form of a binary chromosome. Then, the application evaluates the organisms on a given task based on the cost function. Next, organisms are selected to breed. The genes of selected parents (usually the best 2) are crossed, giving a new child whose fitness is evaluated. Random mutations may be introduced. The organisms with worst fitness are removed, and the process is repeated until a child with the desired fitness is created [7]. Figure 3 shows the flow of a binary genetic algorithm, in which attribute variables of the organisms are stored in binary, with each digit representing one characteristic in the organism.

Cost function: It generates output from a set of input variables, and can be in the form of an experiment, a mathematical function, or a game. The cost of a stronger organism is lower and vice versa. Since it has to be applied to all organisms in a population, optimization of the function is important. Irrelevant information is to be taken off, and variables that are too small to take an effect on the output will be neglected. For example, in the equation $f(w,x,y,z)=2x+3y+z/10000+\sqrt{w}/200$, z and w can be ignored because they are too small. With fewer variables in the cost function, the algorithm can be sped up.

Variable encoding and decoding: A way of converting continuous variable values to binary values and vice versa has to exist. Values from a continuous function can be quantized (rounded off to the closest binary value) and saved as a chromosome (binary number).

Population generation: A population contains a total of (number of organisms X number of bits in a chromosome) bits, which is stored in the form of a matrix. Initialisation is done by filling the matrix with ones and zeros randomly.

Natural selection: The chromosomes are ranked from the lowest cost to the highest according to the cost function. The best are selected to survive while the worst are discarded. Normally, a fraction known as the selection rate is predefined to determine how much of the population will stay or go.

Mating: It is the creation of offspring from parents. Usually, it involves two parents producing two offspring. A crossover point, a kinetochore, is randomly selected in a chromosome, and the chromosome is split into two parts. In the case of two parents, one offspring can be created by combining the chromosome to the left of crossover point of the first chromosome, and that to the right of the crossover point of the second chromosome (Figure 4). With such crossover of chromosomes, offspring containing characteristics of both parents can be produced. It is possible to have more than one crossover point, where a larger number of different offspring can be produced.

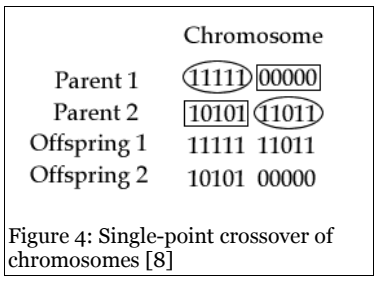

Figure 4: Single-point crossover of chromosomes [8]

Mutations: Bits in chromosomes are randomly altered. Apart from mating, this is another way in which a GA explores the cost surface. The more mutation occurs, the more freedom a population is given to explore new solutions. However, it also tends to reduce the chance of getting a consistent solution in general, so care has to be taken in choosing the percentage of mutations in the population. It is common to avoid applying mutations to the best organisms because we do not want to eliminate the elite performers.

Convergence check: It determines if an acceptable solution has been obtained after a number of generations. The desired cost can be compared with the lowest cost in the population. Also, the algorithm should be stopped when the cost of the best chromosomes remains the same after several generations [8].

The significance of genetic algorithms lies in its ability to automate the process of finding solutions of a complex problem, as well as providing a way of solving problems that cannot be solved by conventional numerical optimization techniques. Its application spans from art creation to flight planning [8]. Because of its simple requirements, it can be applied to almost any kind of problem without much effort. Apart from improving randomly created solutions, it can also work on creations made by human intelligence. Genetic algorithms also sparked a new way of programming called genetic programming (GP) which uses a genetic algorithm to write computer programs automatically.

2.3 Genetic Algorithms with SodaRace

The XML representation of SC models and the race time information in SodaRace allows artificial life to be created easily, with the XML tags enforcing the meaning of the variables and the race time forming the cost in a GA. Over the years, two types of optimization have been developed. The first one is called simulated annealing, where a GA is applied to an existing human-engineered racer (Fig. 5). The second one takes the approach of breeding models from scratch, where a GA randomly joins masses and springs. In simulated annealing, the variables of the models are randomly altered. The altered versions will be raced and timed to see if performance has been improved. Models which do not improve will be allowed to survive for a period of time, but their chance of survival decreases gradually [5]. On the other hand, breeding was realised by an Austrian AI group called Wodka. In the program also called Wodka, it allows users to breed Sodarace racers without worrying about the technical details involved in genetic algorithms [9]. In the application, sets of masses, springs and muscles are randomly joined together to form racers. The genetic algorithm breeds the quickest racers to form offspring and so on. At the beginning stages, the models look just like segmented sticks bouncing around. But as the population evolves, they take the shape of flippers. Wodka was open-sourced and its popularity increased quickly, letting the community experience genetic algorithms [5].

Figure 5: Left: Human-engineered daintywalker; Right: daintywalker optimized by a GA [5]

2.4 Previous work: Framsticks

Framsticks is an application which allows the study of evolution capabilities of 3D creatures in a simplified environment. It supports both direct (with fitness criterion defined) and spontaneous evolution (without fitness criterion defined). First released in 1996, Framsticks supports a wide range of applications, with no limitations on complexity and size of creatures. It is highly sophisticated in that it simulates real-world properties like static and dynamic friction, damping, action and reaction forces, energy loss on collision, gravitation, buoyancy in water, etc [11]. A Framsticks creature contains two main parts – the body and the brain. The body is a stick of two flexibly joined parts with properties like position, orientation, weight and friction. Moreover, custom properties can be defined, for example, the durability of joints on collusions. The brain is made of neurons which is a single processing unit, but it can also interact with the body as a receptor/effector. The presence of receptors allow Framsticks creatures to detect changes in the environment, look for food and follow targets. A predator and prey model can be formed, with the creatures looking for food and reproducing on their own. With FramScript, which is a scripting language designed for Framsticks, high-level (parameter adjusting) and low-level (custom procedures) scripting can be achieved in the experiments, which results in improved efficiency. Framsticks is

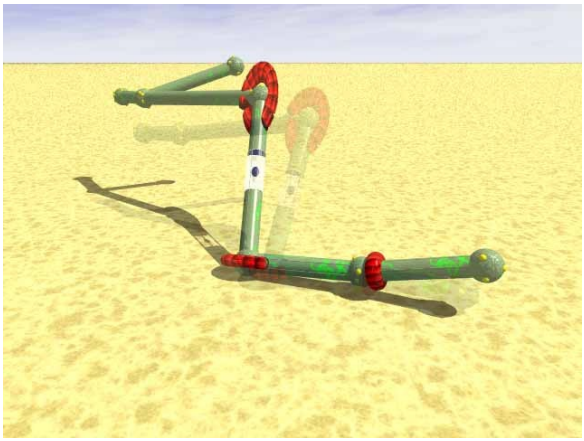

Figure 6: A jumping creature created in Framsticks [10]

a very useful tool for the study of artificial evolution and illustration of basic phenomena: genes and genetics, mutation, artificial selection, walking and swimming. The 3D models of life forms are not only a resemblance of real-world organisms, but also a testing environment for real robots, which can be built based on simulation results [5].

2.5 Collision Detection

Collision detection (CD) is a common and essential technique in computer games and simulations. It is useful for finding contacts and penetrations between different objects. Since CD has to be done for all objects at every simulation step, it is commonly the bottleneck of many applications and appears to be $O(N^2)$, which is very inefficient. To tackle this, computer scientists have come up with different ways to reduce the computation load, and CD is normally broken down into two major steps: broad-phase and narrow-base collision detections. Broad-phase detection is a crude way of detecting if collisions have occurred, usually by using geometric primitives like bounding boxes, cylinders and spheres. In the case of axes aligned bounding boxes (AABB), a bounding box with sides orthogonal to the world coordinate axes is drawn, which contains the entire object in question. Since it is block-shaped, 6 variables are needed to store the maximum and minimum points of x, y and z axles (Figure 7). When two objects are tested for contact, the two bounding boxes are first checked if any part of them are overlapping, by a simple comparison of the stored values. If not, the objects are pruned to avoid spending time for more precise contact detection. If the two boxes are indeed in contact, then narrow-phase detection will follow. Narrow-phase detection can be a collection of mathematical functions which return the separation distance, penetration depth and the closest points. Sometimes the exact time of collision has to be determined [12][13].

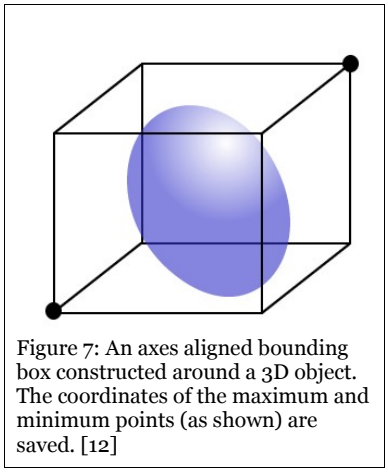

Figure 7: An axes aligned bounding box constructed around a 3D object. The coordinates of the maximum and minimum points (as shown) are saved. [12]

In general, the cycle of collision detection and response is as follows: 1) Calculate new positions 2) Check for collisions 3) Respond to collisions [14]. This essentially means that collisions are allowed to occur in the calculations, but the penetrations are reverted and bounces dealt with before the results are displayed as animation. For example, in a game where the player is not supposed to walk into a wall, the engine allows the player to move, then detects if the player is in a wall. If he is, he is moved back to his last position where collision has not occurred. This method of collision response is popular because there is only three variables to store (in 3D environments), which is the last position of the player, and the exact time of collision is not required [12]. In an environment with an enormous amount of objects, reducing the amount of calculations is crucial for performance.

2.6 Choice of Platform

The Sodaplay series of applications use Java as the platform for ease of use and compatibility reasons. With Java Virtual Machine installed, the same piece of Java program can be run on any hardware. E.g., the application “Newtoon” can be run on computers as well as mobile phones. Also, the Java Web Start

technology allows applications to be run with one click on the web browser. Furthermore, the classes in SodaSumo may be useful for future development of Sodaplay. For these reasons, SodaSumo makes sense to use Java as well.

Although Java 1.6 has been released, SodaSumo has been compiled with Java Development Kit 1.5 due to the fact that version 1.6 is not ubiquitous enough. Many users still have Java runtime 1.5.X installed and reported that SodaSumo could not be run on their computers when it was compiled with Java 1.6. For compatibility reasons, SodaSumo will be compiled with JDK 1.5.

3 Design

The aim of SodaSumo is to parse the XML representation of a SodaConstructor model and turn it into Java objects. Afterwards, the two separate models are translated to their respective positions on the screen, and will move as they do in SC. Lines and points from the same model will not collide but those from different models will. A time limit will be enforced, and when time is up, a score which can be zero representing a draw, or a positive integer for either model, will be displayed. The principle of this project is to create a user-friendly application so simple that even users who are unable to create their own working models can load models on display at www.sodaplay.com/play, so they can observe two models wrestle. This is easily done by entering the username and model name.

3.1 Requirements Analysis

The following are the key requirements for the development of SodaSumo:

- It should be easy to use for users of a wide range of skill levels
  - it should automatically load models placed in the same directory
  - it should be compiled as an executable that can be run by double-clicking
  - the GUI should be simple with minimal number of buttons
  - users should be able to tell which wrestler has won, and by how much
- The physics and animation should be well implemented
  - the models should behave the same as in SodaConstructor
  - collision detection should prevent models from penetrating each other
- It should be stable and safe
  - it should report when user tries to load an XML file not generated by SC
  - it should restrict users to download XML files from sodaplay.com only
  - it should not crash in any circumstance

3.2 Type of application

Although Sodaplay programs are all web-based and can be loaded with one-click on the sodaplay website, SodaSumo is developed as a local application that is run from the user's computer. Once a working Java program is created, it is relatively easy to turn it into a web-based program, due to Java's high level of compatibility on different platforms. However, in a conversation with Tim Diggins, technical director of Soda Creative, it is made clear that a “special mechanism” is needed to load and store game progresses for applications to work on the Sodaplay website. No further information has been provided regarding the mechanism. To maintain progress of the project, SodaSumo will remain as a local application, which can be later extended to form a part of Sodaplay applications.

3.3 Class Structure

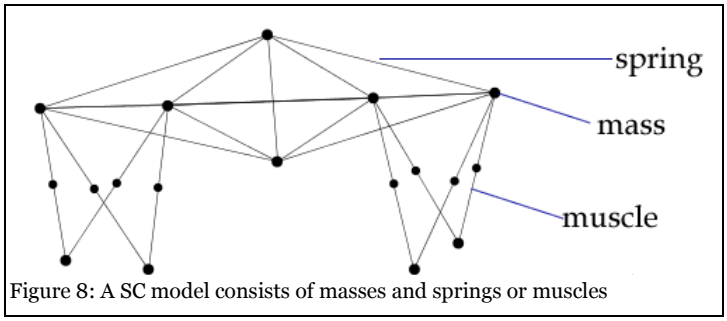

Figure 8: A SC model consists of masses and springs or muscles

In SC, a number of components can be created, mass, spring, muscle, fixed bar and node. Since fixed bars and nodes are always stationary and do not serve any purpose in a wrestling game, they are not allowed in SodaSumo. The project will exploit the object-oriented nature of Java to create a structure of program classes. In SodaSumo, there will be 2 models, each consisting of springs, masses and muscles. A file downloader, an XML parser and a physics engine will also be created. The GUI will act as the central class which contains the main method, and it will be used to call other classes when required. Figure 9 shows the class structure of SodaSumo. XmlParser and FileDownloader are separate from the main game engine because they can act as additional components in other programs. To strengthen the concept that three types of objects exist in each model, a model contains maps of masses, springs and muscles. Since a muscle is a special kind of spring, it extends the Spring class to inherit the common variables and methods.

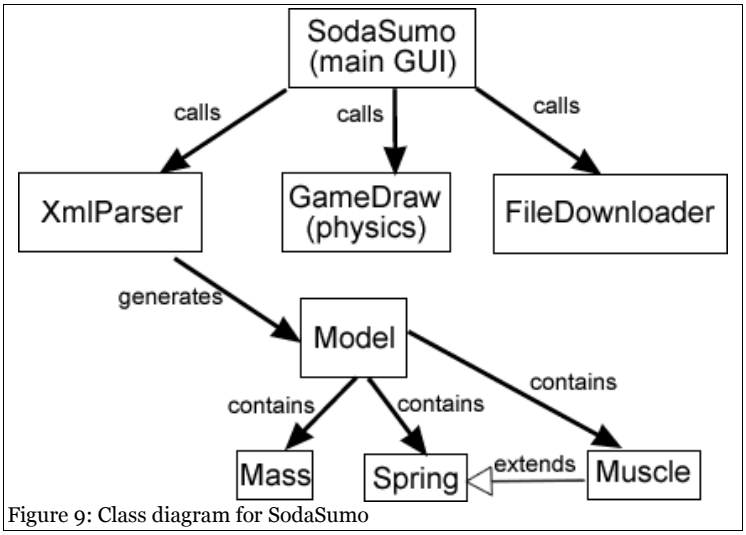

Figure 9: Class diagram for SodaSumo

3.4 XML Parsing

Since all the SC models are described and exported in XML, a parser is needed to extract the data into SodaSumo. For processing XML data in Java, a number of technologies are available: Simple API for XML (SAX) and Document Object Model (DOM) found in Java API for XML Processing (JAXP), and Java Architecture for XML Binding (JAXB).

SAX and DOM work by using callback methods called Content Handlers [19], which is a snippet of code passed to other methods as arguments [20]. They are not invoked by user or code, but called by the XML

parser at certain events, which appear in the XML data as tags. The data is scanned and broken into many discrete pieces. In the case of SAX, the parser takes the data sequentially as it appears in the document, and action requested in the callback methods is instantly taken. However, nothing is saved in memory, so the XML file cannot be updated. For DOM, a tree structure of all the data is built in memory, allowing random access and hence the manipulation of data. However, time is needed to allocate memory for the tree. JAXB allows Java classes to be mapped into XML data (marshall) and vice versa (unmarshall), without the need to implement a specific loading and saving routine. It is particularly useful for programs whose specification is always changing, since it can be time consuming to change the XML Schema to suit the altered Java structure [22]. The downsides of JAXB will be being too specific and tightly bound to the Java structure. It is also not possible for extensions using the same XML file to have different runtime classes.

For the purpose of parsing SC models, SAX is used. First of all, SodaSumo will be an application for loading SC models. All constructions and editions are supposed to be made in SC, so there is no need to update the XML files when they are loaded in SodaSumo. Speed is also very important for a web application. The creation of a tree of the data in DOM will increase the time required to load a document, so it is not desirable. Furthermore, there is no need for random access in SS as the whole model will be imported before anything is done. There is no point manipulating a single spring or mass of a model. With JAXB, unmarshalling and marshalling can be very efficient, but models have different uses in different applications (racing in Sodarace, wrestling in SodaSumo). In many cases the classes are defined separately. The re-usability of the classes are hardly exploited, so SAX remains to be the best choice for SodaSumo.

Decision 1: SAX is chosen to parse the XML files.

3.5 Underlying Physics

3.5.1 Mathematics

The animation of SodaSumo will base on the fact that the resultant force acting on an object will simply be the sum of all forces acting on it. This resultant force and other vectors can point in any arbitrary direction in 2D, which is undesirable for calculations. For the simplicity in calculations, all vectors are split into their x and y components, so only two directions will be considered. This can be done by finding the sine and cosine (Fig. 10) using Math.sin(angle) and Math.cos(angle) in Java, and then multiplying by the original vector. This requires the acute angle θ between the vector and the x-axis to be found.

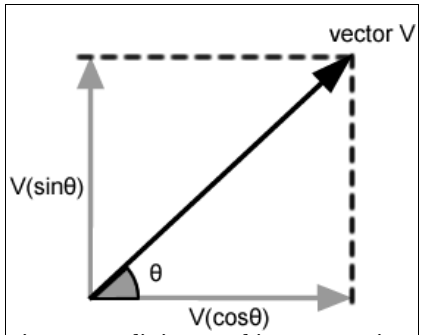

Figure 10: Splitting an arbitrary vector into x and y components

Decision 2: All vectors are split into their x and y components

3.5.2 Model Physics

Since SodaSumo is an extension to SC, the models have to behave the same. This requires the understanding of the SC physics engine and the reverse-engineering of it, together with how a model is formed.

Mass: It is a passive node which provides the weight of a model. It must be connected to at least one spring or muscle. Although it can be seen otherwise, a mass is assumed to have no area. According to Henry Jeckyll, the magnitude of the masses in kg is not important [18], since all masses are identical and they have the same inertial property.

Spring: It is a passive and weightless component (in physics studies, springs are usually assumed to be weightless) which behaves like a spring we can find in real life. It must be connected to two masses. When compressed, it exerts an outward force; when stretched, it exerts an inward force. The force is given by the Hooke's law:

$$F=-kx$$

k is the spring constant which describes the force per unit length of the spring.

x is the distance the spring is stretched or compressed from the equilibrium position. It is usually measured in metres, but we do not have to worry about it if all measurements in SodaSumo uses the same unit.

The minus sign says that the force generated by the spring is in opposite direction to the force exerted to it [16].

Muscle: It is a kind of spring which observes the Hooke's law. It is also an active component which extends and contracts in a Simple Harmonic Motion (SHM), which means its length relative to time can be described by a sinusoidal wave [17]. The muscle in SC works by varying the equilibrium length of a spring. Although such component does not exist in our physical world, it is essential for the generation of movement in the models. The equilibrium length, also known as the rest length of a muscle, is calculated by the following equation:

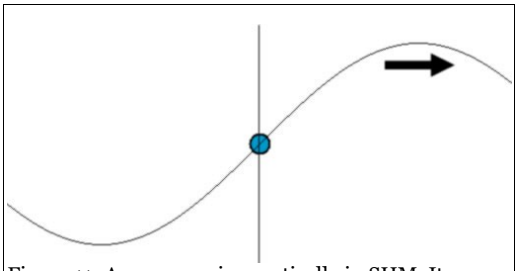

Figure 11: A mass moving vertically in SHM. Its motion is described by a sinusoidal wave moving towards the right [17]

$$L = L_o[1+\alpha\cdot\beta\cdot\sin(\omega t+\varphi)]$$

where:

L is the new rest length of a muscle, and $\mathbf{L_o}$ the unaltered rest length

α is the value of the relative amplitude of a muscle (between 0 and 1). It allows the amplitude of individual muscles to be changed.

β is the value of the relative amplitude for all muscles (between 0 and 1). It allows all muscle amplitudes to be tweaked altogether.

ω is the speed of the sinusoidal wave, measured in radians per unit time.

t is the time lapse in the engine.

φ is the phase lag in radians of the muscle, which allows muscles to have different lengths at the same time [18].

Since sine varies between -1 and 1, it can be deduced that the rest length of a muscle varies between 0 and 2

times the initial length.

A few changes have to be made before the equation can be used in SodaSumo.

- The phase lag is saved in SC as a double between 0.0 to 1.0, which has to be converted to radians.
- The same applies to the wave speed.
- The engine does not work in seconds of time but in frames, f.
- The user can save the model at any time. To avoid changing the muscle lengths suddenly when the models are loaded, the wave phase in which the simulation was last stopped was saved as a fraction λ (between 0.0 to 1.0). When the simulation starts again, the sine wave will be pushed forward by λ.

These change the muscle equation to:

$$L = L_o\{1+\alpha\cdot\beta\cdot\sin[(\omega f+\varphi+\lambda)\cdot 2\pi]\}$$

Decision 3: The equation for the rest length of a muscle has been adjusted.
Decision 4: Masses are assumed to have a mass of 1.0 kg.
Decision 5: Springs and muscles are assumed to be weightless.

3.6 Graphics Rendering

Java2D is a package in Java Development Kit which allows the rendering of simple graphics. Apart from drawing simple shapes like ellipses, lines, points and rectangles, it can also process effects like anti-aliasing, transformations, textures, clippings, etc. Java bindings for OpenGL (JOGL) is a more advanced hardware-accelerated technology for the use of OpenGL rendering. Not only can it render 3D graphics, it also allows simpler conversion of existing C programs to Java ones [24].

For SodaSumo, Java2D has been chosen to be the rendering method. The project bears the aim of replicating the SC physics and visuals so the users will feel a sense of consistency when designing their models on SC and putting them in a wrestling game. Since Sodarace and SodaConstructor uses Java2D for rendering, it makes sense to use the same rendering method in SodaSumo. In SS, hundreds of lines and small circles changing positions every frame are rendered. Anti-aliasing is turned on to smooth out the lines and edges of circles. Apart from these, no other effects are applied, and the animations are all in black and white. It is believed that Java2D, although simpler, will be more than suitable for the purpose of 2D animations. It will also increase the development time if we have to use a third-party API, trying to solve compatibility issues which may arise. For these reasons, SodaSumo will employ Java2D for rendering.

Decision 6: Java2D is chosen for graphics rendering

4 Implementation

The implementation of SodaSumo followed an iterative approach; at each stage of the development a working prototype was produced with additional features. The project started with creating a workable XML parser, then a parser plus a model displayer, followed by a parser with a model animator, and so on. This approach made sure that each additional component mingled well with the whole system, and that errors and bugs could be dealt with before the next stage is reached.

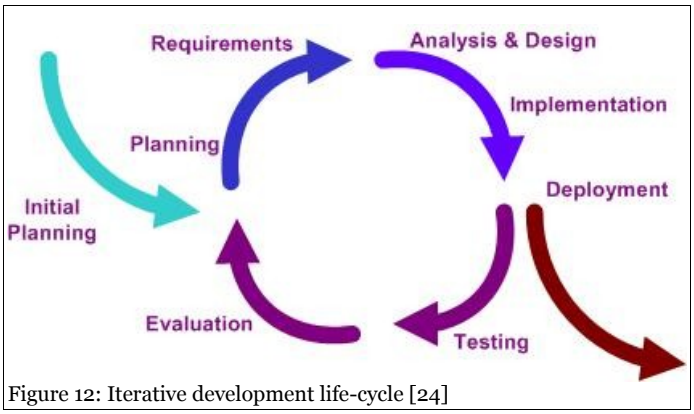

Figure 12: Iterative development life-cycle [24]

4.1 Model Parsing

As mentioned before, the parsing of XML files relies on SAX technology. The XmlParser class extends ContentHandler which is included in the Java API. A list of such methods can be found in Appendix B.

4.1.1 SAX Content Handlers

A number of methods are implemented for the parser to use when it comes across "events" in XML documents. Not all the methods are used in SodaSumo, but they must be present in the XmlParser class.
The following is a snippet of XML data describing a SC model. See Appendix A for the entire document.

```
<object id="Mass#3" type="com.sodaplay.soda.constructor2.model.Mass">
        <field name="vx">
          <double>-1.2464576578060318</double>
        </field>
        <field name="vy">
          <double>0.5105446232742532</double>
        </field>
      ...
```

In SAX, an event is fired every time a tag is reached, and the characters between tags also passed. In the above table, we can see that the object "Mass#3" will fire a number of events, namely "field" with an attribute "vx", "double" with characters -1.2464576578060318, etc. With that in mind, the XmlParser class is implemented so that the firing of separate events for one object will not affect the accuracy of the data describing the model. Also, we want to prevent users crashing the program by manually altering the document. This is done by constructing a "state machine" which oversees the data the parser is dealing with.

4.1.2 The State Machine

The state machine is a simple yet powerful tool to monitor the parsing of data. The machine is declared as an integer (int) which has one static starting point, “Start”, and each state has a specific int value. Every time a tag is reached, the machine checks what state it is in before anything is done. If the action does not match the correct state, an exception is thrown and the parser will stop its operation. Figure 13 demonstrates the state machine's working. The state machine is flexible enough so that set Vx, set Vy, set X and set Y do not need to be in order, or they can be absent (in such case the values are initialised to be 0). However, if actions for different objects are fired subsequently, an error message will be generated. For example, if Create Mass is followed by set MASS1, the error “manipulating empty object” will be triggered, because a spring has not been initialised. This makes sure the document generates objects and initialises data .

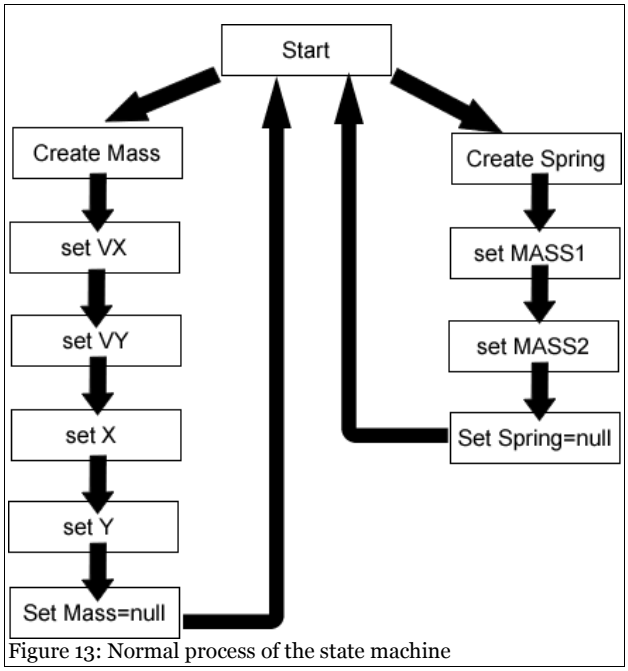

Figure 13: Normal process of the state machine

4.2 Model Translation

SC models are designed individually in their own window and they can be saved at any point of simulation. The problem is that the models can end up in any position on the world frame. If they are imported directly into SodaSumo, the two models may overlap right at the start. It is also for fairness that wrestlers start at a pre-defined position on both sides of the field. This is why translation is applied to both models as soon as they are parsed. We can do so by finding the maximum and minimum x coordinates of the model. If the model is on the left, then the maximum x coordinate is used, and vice versa. The max/min point of the model is compared with the horizontal mid-point of the display area, and a simple subtraction between the two will reveal the amount the models are shifted. The code goes through all the masses of the models and reset their x-coordinates accordingly. After this, the models will start in their default positions, and since they do not overlap, they can be viewed as being in the same frame of reference. This is demonstrated in Figure 14.

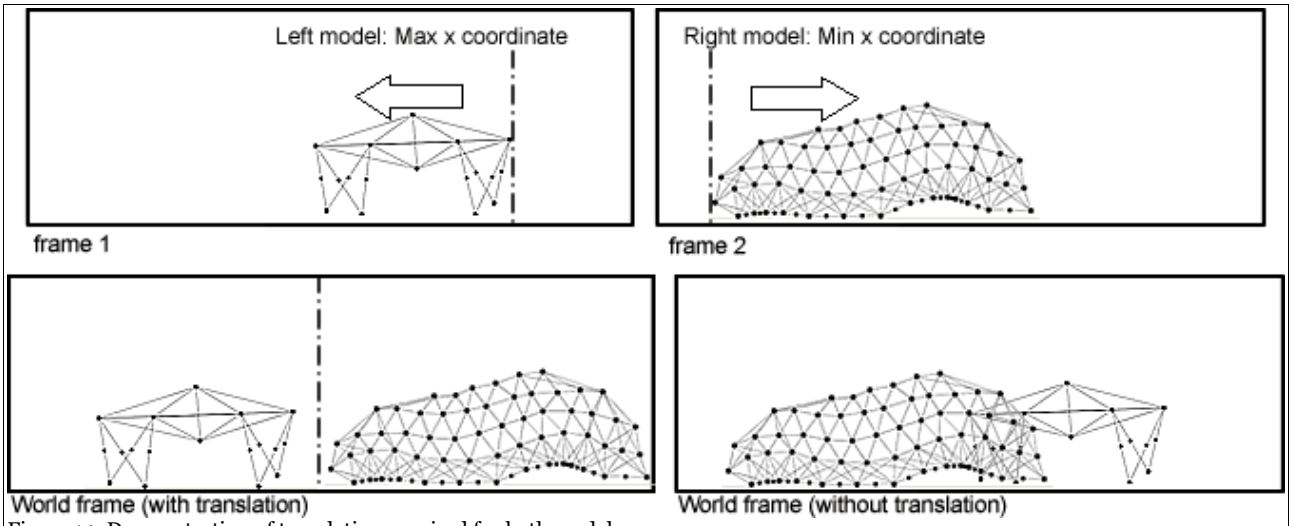

Figure 14: Demonstration of translation required for both models

4.3 Physics and Animation

The physical calculations and animations, together with collision detection, are all implemented in the GameDraw class. However, constituent objects in a model (mass, spring and muscle) all have their own classes to make use of the object-oriented nature of Java.

4.3.2 The Animation Loop

In an environment which only contains masses and springs (muscles), the only thing that can generate motion is the stretching and compression of springs. To calculate the change in position in each frame, we can loop through all the springs and muscles and apply the following equations:

F*rom F=ma (Newton's second law of motion) and F=-kx (Hooke's law),*

acceleration in this frame = (stiffness of spring x extension of spring) / mass

Since masses are all assumed to be 1 (See section 3.5.2),

acceleration in this frame= stiffness of spring x extension of spring

New velocity = Old velocity + acceleration

New position = Old position + New velocity [15]

In a computer simulated system, we assume that one frame – a small time step – is small enough that the change in acceleration in which is negligible, hence it remains constant. Acceleration is the change in velocity over time, so in the above equations, acceleration per frame will be the change in velocity in the current frame. The similar goes for displacement.

4.3.3 Calculation of Movements

To calculate the extension of a spring/muscle, we load the stored x and y-coordinate values of each mass (each spring/muscle must have 2 masses). Then, we calculate the vertical and horizontal displacements, and find the length of the spring by using

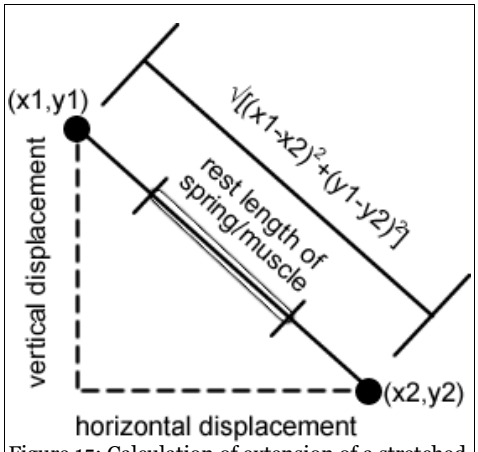

Figure 15: Calculation of extension of a stretched spring

Pythagoras' Theorem. The extension of a spring will be (current length – stored rest length of spring)(In the case of a muscle, the rest length which varies over time is calculated), which can be a positive (stretched spring) or negative (compressed spring) number. Figure 15 shows how the extension of a spring is calculated. For acceleration, consider an arbitrary stretched spring in Figure 16, where a is the acceleration of the 2 masses. The acceleration in the x and y directions will be acosθ and asinθ respectively. With the current velocity and position of a mass stored, the new position of the mass will be worked out. For a compressed spring, the forces will simply be the opposite of a stretched one. A minus operator can be applied when the extension is found to be less than zero.

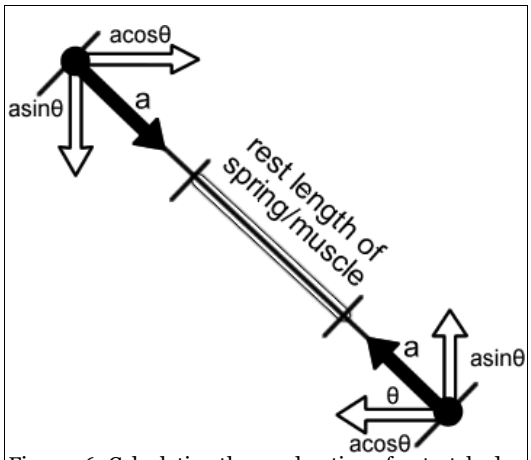

Figure 16: Calculating the acceleration of a stretched spring

The pseudo code for animation and physics is as follows:

Increase number of frames by one
for each model{
for each muscle{
find new rest length
calculate and save acceleration for both masses
}//muscle
for each spring
calculate and save accelerations for both masses
for each mass{
sum all accelerations in x and y directions
for each direction{
velocity=velocity+acceleration
position=position+velocity
update bounding box (needed for collision detection)
}//direction
}//mass
}//model

4.4 Collision Detection

4.4.1 Introduction

Two kinds of collision response methods are available: Posteriori and priori. In the posteriori case, we simulate the physics as usual, and check for penetrations after each time step. For the penetrating objects, we fix their positions by reverting them to a position where they do not collide. In the priori case, the trajectories of the objects are calculated precisely, and their moment of collision is known exactly [25]. That means penetrations are prevented, but a lot of processing power will be used for solving the time of impact. In SodaSumo, the posteriori method is used. This method will not give the most accurate results but will be sufficient for a simple wrestling game and a platform for analysing genetic algorithms.

4.4.2 Mechanism

Using the posteriori method, SodaSumo needs to detect for penetrations between two models, and fix the positions of masses (fixing the masses linked to a spring/muscle will mean moving the latter as well). A penetration is signalled by the intersection of lines from opposite models (Figure 17). They can be detected by exhaustively going through all the pairs of springs, and checking if they intersect. If penetrations are detected and removed as soon as it happens, the models will not overlap. However, penetration detection is an extremely slow process. As mentioned in Section 2.5, a pruning stage is needed to remove pairs of lines which must not intersect.

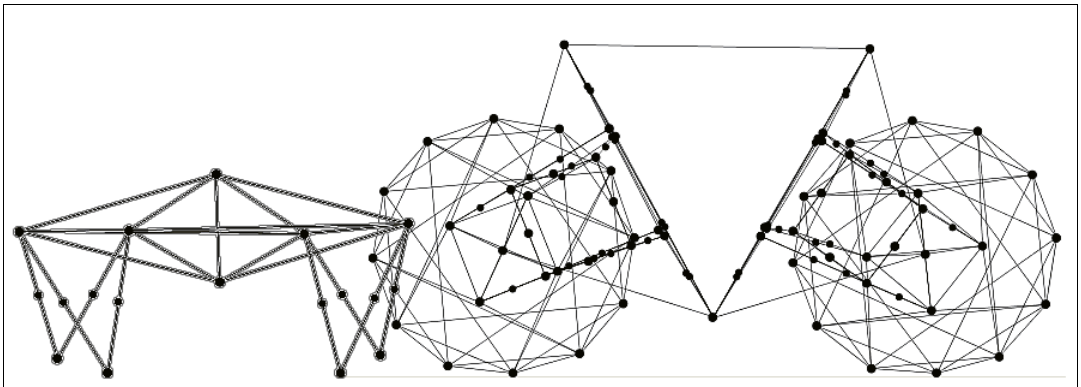

Figure 17: Penetration between "daintywalker" and "monster_truck". Note that daintywalker has been thickened with an external software to distinguish the two models.

Broad phase detection

A bounding box is kept for each model which is updated in each frame. After the mass positions are calculated, the bounding box of both models are compared. If there is space between the two bounding boxes, there is absolutely no chance that the two models are overlapping, so nothing further is done. This is the first stage of broad phase pruning. However, as the models move closer to each other, the bounding boxes will overlap in almost every frame. Even so, the models may not be colliding at all (Figure 18). In addition, only a small portion of the lines will collide (the front ones). For these reasons, a second stage of broad phase pruning has been added. The algorithm works by iterating through the masses of one model and the springs/muscles of the other, and comparing the positions of the three masses. For example, we iterate through the masses of the left model, and for each mass, we iterate through the springs and muscles of the right model. If both of the springs' x-coordinates are bigger than the mass's, then no penetration will occur. This stage involves only 2 comparisons but will eliminate most of the lines (at the back of the model). Figure 19 shows that when the x-coordinates of a reference mass and springs are compared, a large amount of lines will be pruned. However,

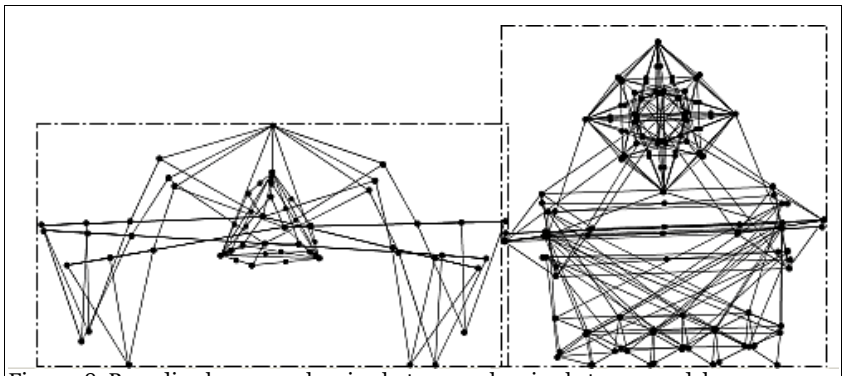

Figure 18: Bounding boxes overlapping but no overlapping between models

if the mass's x-coordinate does lie behind any one of the springs' x-coordinate, narrow phase detection which is expensive will be inevitable.

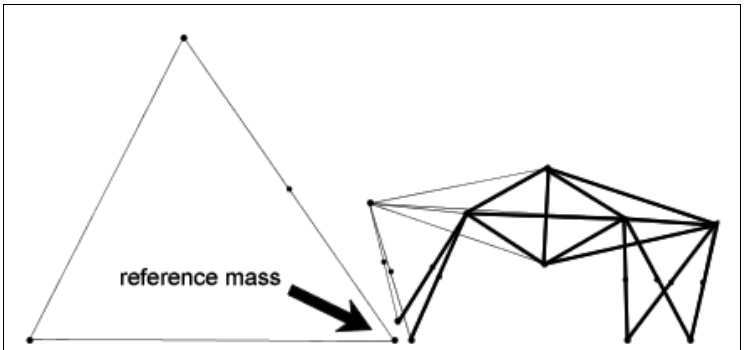

Figure 19: Broad phase pruning: the lines in bold are pruned from further calculations

Narrow phase detection

After pruning the impossible candidates, expensive intersection tests will be done. In the collision engine, we have to deal with different cases that lines overlap with each other. By comparing the position of lines and points in two subsequent frames, we will find out if a mass of one model has crossed the lines of another. There are two ways for this to happen: (1) A moving point cutting through a line during its movement. This can be detected by joining the positions of the mass in the last frame and the current frame. A line of its path will be drawn which is compared with the line from the other model. Using the method provided in Java, the two lines will be tested for intersection. If they do, then a collision has occurred (Fig. 20). (2) In the second case, we assume that the mass doesn't move, and the line moves to sweep through the point formed by the mass (Fig. 21). This type of collision cannot be detected by the first method, because no path can be drawn by a stationary mass. The current detection requires a lot more calculations to happen. We make use of the fact that the equation of a straight line is y=mx+c, and that plugging in any arbitrary point in 2D space will either give a negative or positive number. If the point is on the line, the result will be zero. By utilising this fact and the slope of the line, we can judge whether a point is on the left or right of a line (Fig 22). Comparing the state of a point between subsequent frames, we can decide whether a point

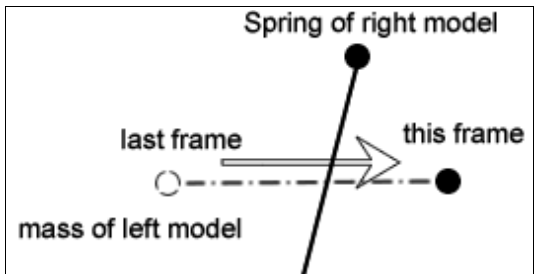

Figure 20: Intersection when line is stationary

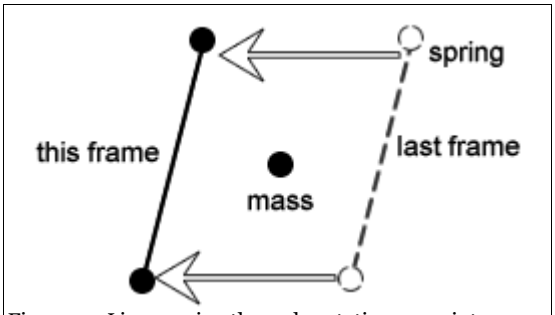

Figure 21: Line passing through a stationary point

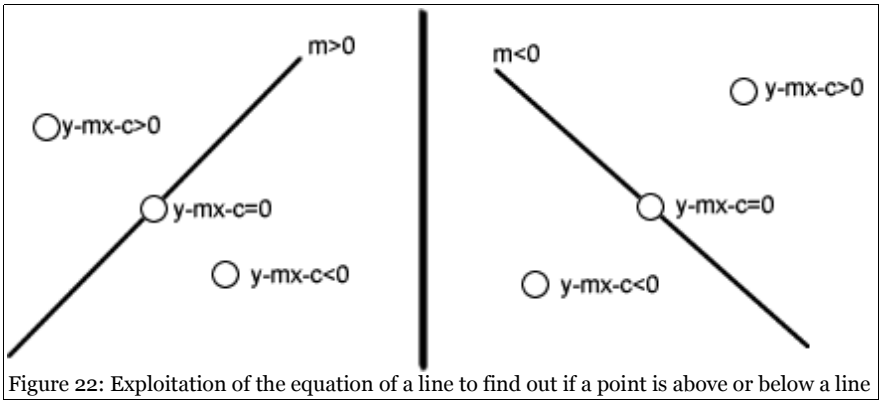

Figure 22: Exploitation of the equation of a line to find out if a point is above or below a line

have been swept over by a line. Extra caution is made for extreme cases, i.e. horizontal and vertical lines, where the slopes are zero and infinite respectively. For a very steep line, the slope can be very small too. However, with the use of double value which contains 16 decimal places, we can guarantee the detection is reliable. After these two phases of collision detection, all sorts of collision will be detected.

4.4.3 Collision Response

In every collision, three masses (one from one model, the other two from a spring/muscle) are involved. One approach to collision response will be taking potential energy, moment of inertia, centre of mass, torque and momentum into account, but the physics engine is bounded by the amount of time available for research and development. In SodaSumo, we only revert the masses back to their positions in the last frame. Then, we assume the total kinetic energy ($\frac{1}{2}$ mv^2) before the collision is evenly distributed to make the velocities of the three masses after collision. A multiplier is applied to reduce the remaining kinetic energy to 95% of original to simulate the energy loss to friction. In addition, it is assumed that the component in the left model bounces to the left, and vice versa.

4.4.4 Maintaining a constant framerate

Physics generation and collision detection are operations which vary greatly in processing speed, due to the different orientations of springs and masses in each frame. Most importantly, the pruning in collision detection may shorten the time required for calculations when the two models are far apart. It may also be vastly lengthened if few objects are pruned. To maintain a constant framerate, we make use of a thread which sleeps for a varying amount of time, depending on the time required for physics calculations. The following is the pseudo code:

1. *Save timestamp*
2. *Do physics and collisions*
3. *Find time lapse in step 2*
4. *Compare time lapse with delay required (20 milliseconds: 50 frames/second)*
5. *if required delay>time lapse, sleep for (20 - time lapse) milliseconds*
6. *else do next frame immediately* [26]

The thread with varying sleep time makes sure the animation pauses for the same amount of time after each frame. However, the correct implementation of constant framerate will also rely on the efficiency of the code. It can be determined that the else case above is undesirable, because the program is running late – thread has slept for a longer time then required. In this case the animation will suffer from lag – a delay in movements, or even a small pause. The next section will focus on techniques used to make sure the else case is run as little as possible.

4.4.5 Improving Performance

Although a thread with varying sleep time has been implemented, it is not enough to maintain a constant framerate if calculation time exceeds the delay required frequently. A number of techniques have been introduced to improve the average framerate:

- **Caching**
  During the course of physics calculations, a large number of variables are loaded from the model, and some of them reused regularly. By storing them as temporary variables, they will not be reloaded from the model, which will save time.
- **Static variables**
  Constants in SodaSumo are declared as static variables so they can be pre-compiled to reduce overhead.
- **Global variables**
  Variables that are used in every frame are declared as global variables, because it takes more time for Java to initialise a variable than use an existing one.
- **Reduce the number of for loops**
  Using a Java profiling tool called PerfAnal, it is found that most of the CPU time is spent on the method for drawing the scene, which contains three "for loops" for drawing masses, springs and muscles. By reducing the number for loops to one, efficiency of the program can be improved.
- **Use Integers instead of Strings**
  In the XML document, the names of the objects are stored as Strings, e.g. "Mass#21", "Spring#1". By stripping the type of object from the name and parsing only the Integer behind it, loading time for each component should significantly improve, because String manipulation is much slower than Integer manipulation in Java.

The program is tested with 2 huge models with over 700 springs and 200 masses each. By making them walk across the screen and printing the time required for physics calculations to system output, it is found that performance has improved by 6 milliseconds each frame after the above enhancements – a 20% improvement.

5 Results

This section demonstrates the workings of SodaSumo. Since this is a physics simulation game, this section will make use of a large amount of screenshots to demonstrate the results.

5.1 Graphical User Interface

SodaSumo has been developed to be a fully graphical program, so no command-line knowledge will be required from the users. As soon as the file is run, a window will pop up, showing the 2 main areas: the display area and the control panel. The display area is in charge of the animation between the two models. To the right of it, a control panel with a list of buttons can be found (Fig. 23).

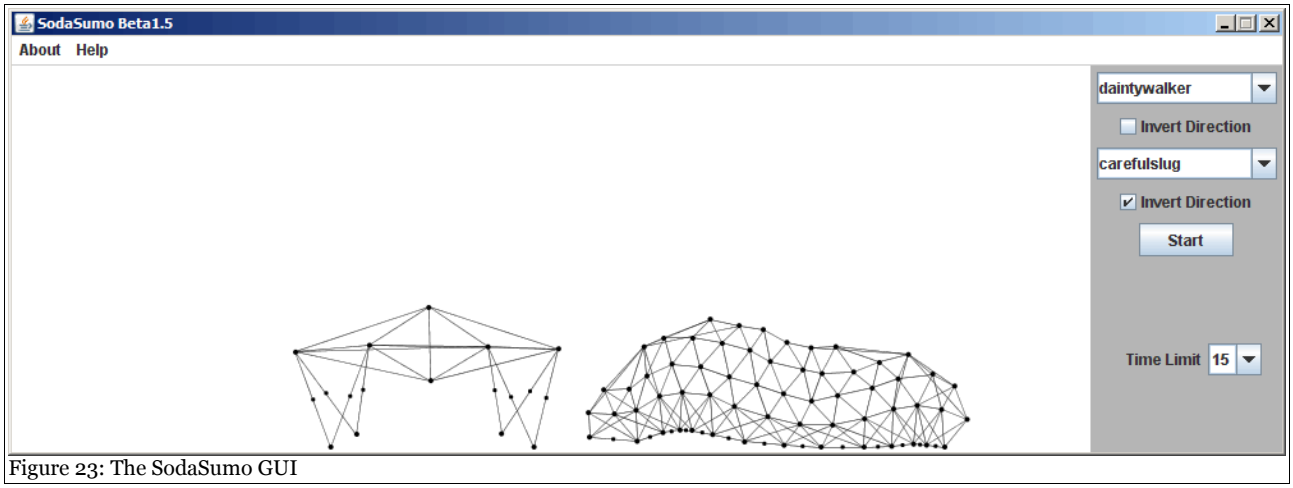

Figure 23: The SodaSumo GUI

Figure 24: The control panel

The most popular buttons in the control panel, model selection, direction inversion and start/stop button are placed close together, so users can make quick adjustments. Users will not have to move the cursor a lot to use the program. The time limit button is placed near the bottom of the control panel as users will not adjust it all the time. Therefore clarity of the panel is maintained. It is important to have a simple GUI because the users can be primary school children without much computing knowledge. Apart from mouse controls, the GUI can be navigated with the keyboard, which allows advanced users of the computer to navigate through the settings quickly.

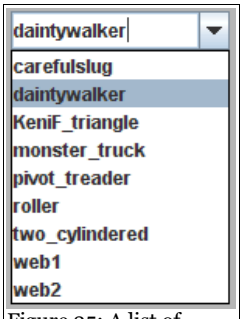

Figure 25: A list of available models are loaded automatically

The SodaSumo GUI, though simple, features a number of advancements. As soon as the program is run, the files in the directory in which it is placed is checked for XML files. Assuming they are SC models*, the file names are trimmed of their extensions (.xml) and placed into the two drop-down lists (combo-box) for the ease of loading (Fig 25).
They are loaded into both boxes so users can select which side the model is on. With this feature, users will not have to spend time typing in the file names.

The GUI also supports downloading of models from http://sodaplay.com/play directly. The user types the username of the model, followed by a forward slash and the model name into the drop-down list. After pressing enter, the model will be added to both lists and available for loading later. The users are restricted to download XML files from sodaplay.com, so the chance of getting malicious files are eliminated. This will be good for young users who have little knowledge of protecting their computers. The SodaSumo GUI is advanced so that the web loading function in it is quicker than SC, the official application. In SC, loading one model takes about 4 seconds. But in SS, loading 2 models only takes a second or less.

To conclude, it is a GUI that features both functionality and simplicity, two crucial requirements in modern GUI design. At the same time, it has improved in terms of model loading speed.

*to reduce overhead of loading the GUI, the files are not verified at this stage.

5.2 Game Flow

The resulting SodaSumo is a wrestling game between two models which can be repeated for an arbitrary number of times.

1. User selects left model and right model
2. User presses start
3. Two models loaded, animation starts
4. Models start to collide
5. Animation stops when time limit is reached or user presses Stop
6. Go to 1

   (user can change the move direction of the models at any time)

The following screenshots briefly describe the full process of the animation:

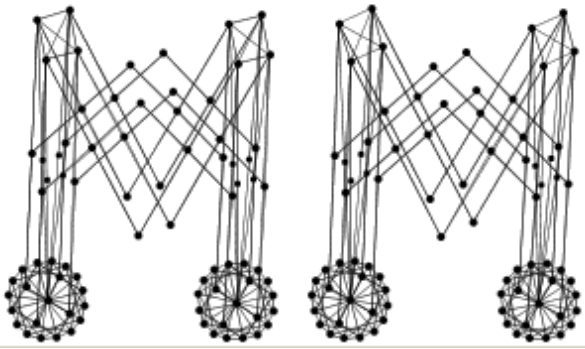

Figure 26: Animation started. A model can be made to wrestle itself. In this example, “roller” rolls forward with its wheels.

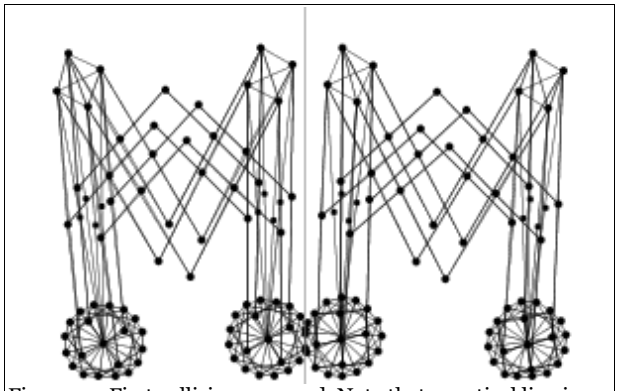

Figure 27: First collision occurred. Note that a vertical line is automatically drawn to show the point the models need to cross to win.

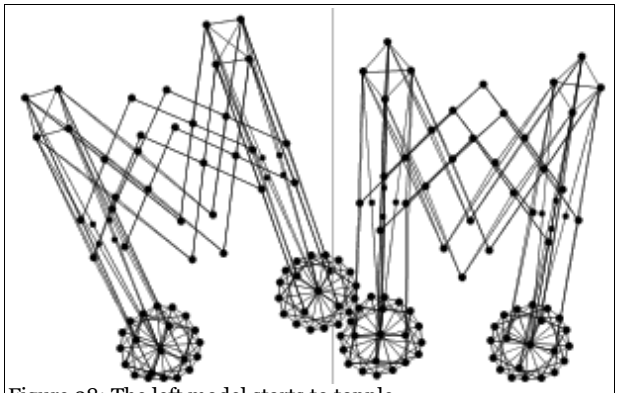

Figure 28: The left model starts to topple.

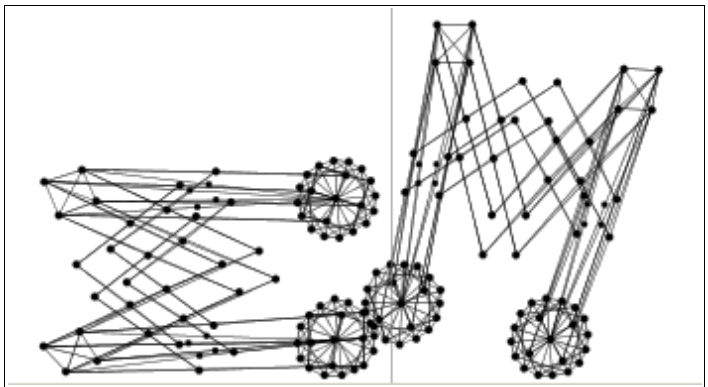

Figure 29: The right model starts to topple too.

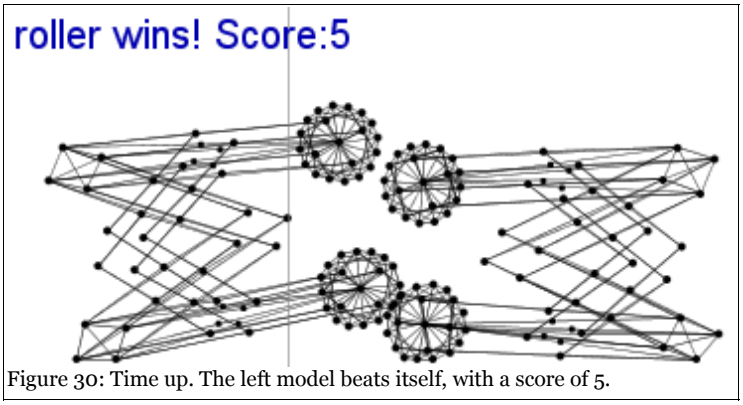

Figure 30: Time up. The left model beats itself, with a score of 5.

5.3 Scoring

As shown above, a score is returned by the gaming engine. It depends on a number of factors: the first contact point of the two models, how much "territory" a model has gained, and how much of it is lost by another model. In the case of a really close game, where the final displacement of both models are small, a draw game will be returned (Fig 31). The score is important as users will be anxious to know how well their models do.

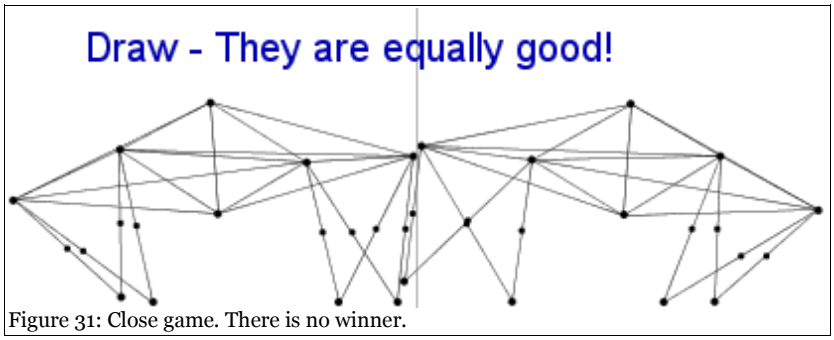

Figure 31: Close game. There is no winner.

6 Testing and Evaluation

SodaSumo has been made to prevent crashes, which means even if the user does something he/she is not supposed to do, the program will act sensibly. A number of tests have been carried out to check the reliability of the program, followed by the evaluation of the usefulness of the game. The inaccuracies of the program are then listed and explained for reference.

6.1 Program Reliability

6.1.1 Testing the XML Parser

SodaSumo assumes all XML files in the same directory are SodaConstructor models. However, in reality, it may not be the case. A well written program should not take anything for granted, and be ready for the worst. The following tests were carried out to test the XML parser:

#	Test Details	Result	Status
1	Everything parsed by the SAX parser is printed out to system output and compared with the original file	All tags and strings in SodaConstuctor models are taken correctly	Passed
2	A test XML file with wrong order of closing tags is parsed	A pop up reporting the cause of error is generated. After that, the program runs normally	Passed
3	A valid non-SodaConstructor XML file is parsed	An error message is generated, telling the user a model cannot be verified. Program runs normally afterwards	Passed
4	A model containing a node/fixed bar is parsed (stationary bars/nodes are not supposed to appear in SodaSumo)	An error message is generated, telling the user "objects not permitted in SodaSumo are found"	Passed

The above tests confirm that the XML parser is working normally.

6.1.2 Testing the Animation

#	Test Details	Result	Status
1	The direction of the models are changed when the animation is in progress	They change direction accordingly without error	Passed
2	A code has been inserted to count the number of frames in a second	The framerate maintains more or less the required value, 50 frames per second	Passed
3	The sodaplay community has been invited to compare the physics between SC and SS	None of them reported noticeable difference between two engines	Passed
4	A model which contains solely of a straight line is loaded	The model collides will the straight line normally	Passed
5	A model is intentionally stretched (springs to generate huge pulling force) to create an "explosive" effect when it is started. This is known to cause problems in earlier versions of SodaSumo	The model slows down and returns to normal	Passed

The above tests confirm that the animation is as close to that in SC as possible.

6.1.3 Testing the GUI

#	Test Details	Result	Status
1	The time limit is changed when the animation is in progress	The time limit is updated instantly and animation stops immediately if the time lapse is bigger than it	Passed
2	No XML files are placed in the same directory as the game	An error message pops up, but there is no change in functionality. Loading from web works the same	Passed
3	Model XML files are removed from the directory when the program is running	An error message reports the absence of files, which will not trigger other errors when restored	Passed
4	A model link which does not exist is entered	An error message reports that the file cannot be found and the entry from the combo box removed to avoid confusion	Passed
5	A model link is entered when there is no internet connection	An error message reports that the web location cannot be found and removes the entry from the combo box	Passed

The above tests prove that the GUI is stable enough to catch errors.

6.2 Evaluation of Game Usefulness

A successful game needs to separate the good from the bad, and maintain a good level of fairness. This section will investigate these in detail.

6.2.1 Fairness of SodaSumo

In SodaSumo, fairness means no model will be benefited by what side of the "battlefield" it is on. This is evaluated by having a number of models (chosen at random) wrestle itself in 60 seconds. The scores are given in the following table:

Left Model Score	Model #	Right Model Score
35	daintywalker	
	carefulslug	1
20	roller	
40	wheel	
draw	climber	draw
	car	3
11	monster_truck	
5	two_cylindered	
	pivot_treader	11
	my_first_walker	12
2	diplopoda	
71	basic_biped	
draw	beast	draw
	dainty_evolved	17
2	sdnm	
	overjointed	1
8	moon_lander	
	leggy_point	3

	hangingmotor	6
	foldingwalker	6

From the above table, it can be seen that the models, when made to wrestle another instance of itself, have the same chance of winning no matter which side it is on. However, it can be seen that the left models tend to have a higher score when they win. This could be due to the fact that the community has a habit of saving their models when they are moving to the right (a requirement in the game Sodarace). For the right model, although the muscles change direction at the start of the animation, time is required to accelerate the moving masses to the left. The right model tends to have a lower velocity on the first contact, leading to poorer average score. In practice, it is advised that the models swap sides after each round to eliminate this kind of unfairness.

6.2.2 Competitiveness of SodaSumo

The last section only shows that there is no bias on any of the two models. However, it can still mean SS chooses the winner at random. To show that SS is a truly competitive game, it has to be shown that there is a hierarchy of models ranging from weak to strong. Five models have been chosen for a wrestling league for this purpose, and their scores recorded. (Score for model 2 is -1 * score of model 1 found in the table).

Model 1	Model 2	Score for m1 (left)	Score for m1 (right)
daintywalker	carefulslug	18	4
climber	daintywalker	51	70
climber	monster_truck	19	10
climber	roller	2	-33
climber	carefulslug	35	32
daintywalker	monster_truck	-43	-67
daintywalker	roller	-41	-34
carefulslug	monster_truck	6	-36
carefulslug	roller	-37	-57
monster_truck	roller	1	-23

The total score of each model is as follows:

Model	Accumulated Score
roller	+222
climber	+186
monster_truck	+89
carefulslug	-213
daintywalker	-284

It is clear that a hierarchy of “wrestling ability” has been formed, showing that some effort is needed to build a good wrestler, and that SodaSumo does work to distinguish different models. SS has satisfied the requirements of a successful game, because the outcome of the game does not depend solely on luck. This also gives SS the status of a good evaluation platform for genetic algorithms, where “survival of the fittest”

holds.

6.3 Inaccuracies and Errors

A number of assumptions have been introduced in SodaSumo which inevitably lead to inaccuracies in the simulations. They are illustrated below.

6.3.1 The Speed Limit

In early versions of SodaSumo, it is observed that models can "explode" on collision or without proper reason. On investigation, it is found that this is due to springs being able to extend without limit, hence creating a large force which spreads across the model. Also, due to the large force created and the absence of physical structure in the "springs", the springs can flip over (Fig 32) unlike springs in real life. A simple

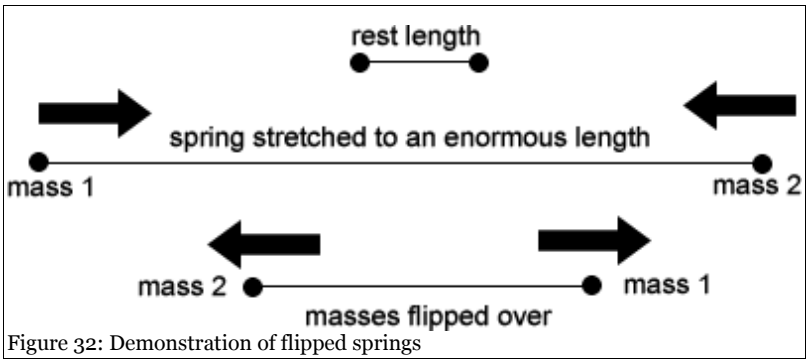

Figure 32: Demonstration of flipped springs

workaround will be adding a SPEED_LIMIT constant in the physics engine, so that masses cannot move too far in each frame. This is imposed on x and y direction separately. Whenever a velocity component is bigger than that number, it is set to SPEED_LIMIT instead. This works fine for preventing "explosion" of models, and is found to exist in SC as well. However, this is also a source for inaccuracy, because the masses of a spring will only move in a constant speed when they are stretched a lot. Also, since SPEED_LIMIT is only imposed on the x and y axis separately, it is possible for masses to move faster than SPEED_LIMIT itself. Assume that our speed limit is 10. If a mass is moving in the x and y directions at 10 units/frame respectively, we will get a resultant speed of 14.14 by Pythagoras' Theorem. This effect is undetected in SodaSumo, which can lead to unfairness when one model tends to move diagonally relative to both axes.

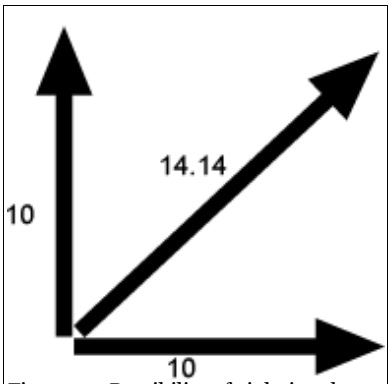

Figure 33: Possibility of violating the maximum speed limit in SodaSumo

6.3.2 Inaccuracies in Collisions

Due to time constraint, a lot of assumptions have been made in the physics engine of SodaSumo for the ease of implementation. The following physical quantities have been neglected:

- **Potential energy**

  The height of the colliding objects are ignored. This will not have a big impact as the distance

between the masses and the ground are small.

- **Moment of inertia**
  This is the tendency to prevent an object from rotating. When a mass hits the non-central point of a spring, the spring should rotate due to the torque created. However, the springs in SS are assumed to be a point mass, and do not rotate on collision.
- **Momentum**
  In a closed system, momentum must be conserved under all circumstances. Having been neglected from calculations, it is possible that momentum is not conserved in SS.
- **Time of impact**
  In physics, the time of impact varies according to the material used. In SS, the time of impact must be one frame for every collision.

With these ignored from the physics engine, it is advised that SodaSumo be used as a tool for competition but not sophisticated physics experiments.

6.4 Possible Improvements

It may have been observed that in SodaSumo, walls are absent on both sides of the "battlefield". This is because models have an arbitrary width. Some models are too wide to fit in the space provided, and the addition of walls will tamper their movement. If the time limit is set too high, one or both of the models may have already moved out of the display area while the timer is still counting (Fig 34). To tackle this, a "panning" function may be added where the user can move the "camera" sideways to capture the action. Also, the engine can allow users to "zoom out" to capture more on the screen, though the models will appear smaller than before. This will require scaling to be applied to the models.

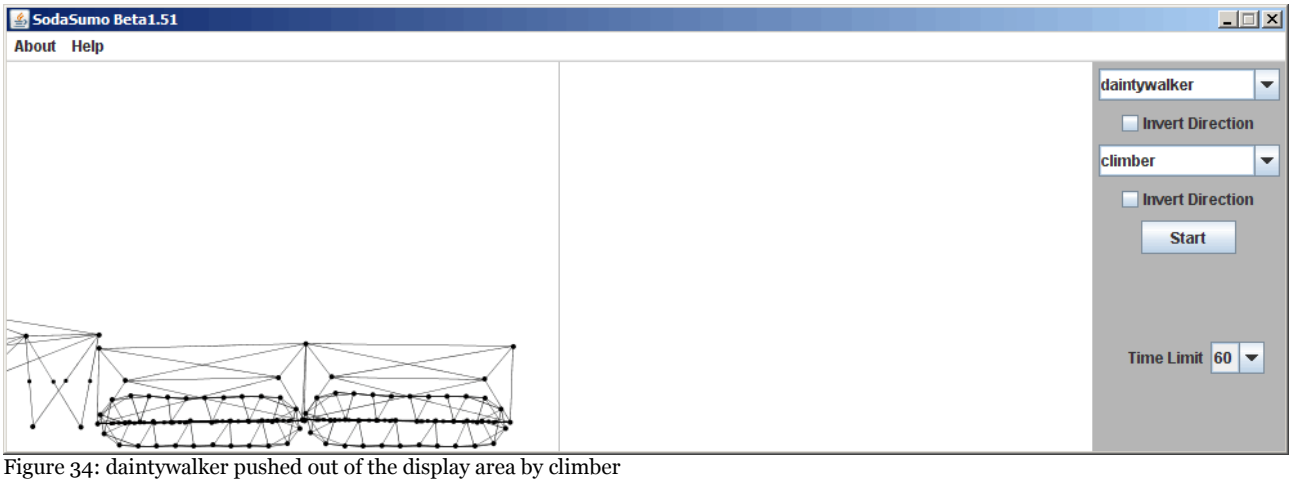

Figure 34: daintywalker pushed out of the display area by climber

7 Conclusion

7.1 SodaSumo – An Extension to SodaConstructor

SodaSumo has been developed to increase the number of possible interactions between the virtual models made with the SodaConstructor. Before, the models can only race each other in a terrain where they overlap like they do not physically exist. The aim of virtual life has always been simulating what biological life can do, and one of the most basic things they do is contact physically. SodaSumo has been completed with a collision detection engine capable of differentiating models in terms of their wrestling ability. It allows users a new way of exploring virtual models and improving them through repeated experiments. It will also be interesting to compare racers with wrestlers and analyse their performances.

A XML parser has been created to load SC models into SodaSumo. First, it validates that the file is from SC by looking for two unique strings in the document, then builds the model according to the description. The result of the parser is encouraging, as it is quicker than the official counterpart. The animation boasts a replication of the official engine with the help of the community written paper by Henry Jeckyll. This will provide a feeling of consistency as users will not need to spend time getting used to the new physics. The performance of the animation is also promising, which runs at a lucid 50 frames per second at most times. The GUI is made as simple as possible for the likely user which is primary school students. Advanced users will find efficiency from navigating the GUI with the keyboard. SodaSumo can be a great tool for users to compete and have fun, and also apply genetic algorithms for the analysis of virtual life forms (see Section 7.2). In conclusion, the project aims have been achieved.

7.2 Future Work: API for Genetic Algorithms (GA)

As aforementioned, it was originally planned that with the acquisition of the source code of SodaConstructor, the development time of SodaSumo could be shortened and an API for the use of genetic algorithms could be made, which was one of the main aims of virtual life forms. However, the open-sourcing of SC did not realise in late 2007 as planned [1]. As a result, a program which could import SC models and generate their physics had to be built from scratch. A lot of effort has been spent studying and replicating the physics engine of SC, which eventually proved successful. However, no work has been done for the sake of genetic algorithms due to time constraint. Limited effort has paved the path for genetic algorithms though, for example, the returning of a score for the models can act as an evaluation value for fitness of the models. It is inevitable that developers of the API for GA will need to rely on the source code of SodaSumo.

SodaSumo is a breakthrough for the development of GAs. In traditional virtual environments, the population works on a common task, and are evaluated by a function. In SS, the population will be performing against one another, and each opponent for them is a different task to work on. The final cost of each member is cross-related to all other members of the population. SS will be a platform for investigating new forms of genetic algorithms where organisms are no longer individuals, but a community. In order to outperform

other opponents, individuals will need to know more about their neighbours, or even learn to counter them. This will be an interesting topic in the development of Artificial Intelligence. To achieve the above, wrestlers should be loaded quickly and automatically with the API. It is also essential to allow speeding up the simulation so users do not have to wait a long time for results. It should have a mechanism for users to import their genetic algorithms, which describe how offspring will be produced. More details for genetic algorithms have been provided in Section 2.2.2.

References

[1] *Soda Constructor*, Wikipedia | 28 March 2008, 15:44
http://en.wikipedia.org/wiki/Soda_constructor

[2] "*daintywalker*", Ed Burton | 28 March 2008, 18:11
http://sodaplay.com/creators/ed/items/daintywalker

[3] *Artificial Life Models in Software* by Andrew Adamatzky and Maciej Komosinski
Springer-Verlag London Limited (2005) Chapter 5

[4] *SodaConstructing Knowledge Through Exploratoids* by Gina Navoa Svarovsky and David Williamson Shaffer
Journal of Research in Science Teaching, Department of Educational Psychology, University of Wisconsin (29 March 2005, Volume 0, No. 0, Pages 1-21)

[5] *Artificial Life Models in Software* by Andrew Adamatzky and Maciej Komosinski
Springer-Verlag London Limited (2005) Chapters 1, 2, 5, 6

[6] *The New Replicators*, Dennett D, Encyclopedia of Evolution, Oxford Univ. Press. (2002)

[7] *Genetic Algorithms* by K.F.Man, PhD, K.S.Tang, PhD, S.Kwong, PhD.
Springer-Verlag London Limited (1999)

[8] *Practical Genetic Algorithms*, by Randy L. Haupt and Sue Ellen Haupt
John Wiley & Sons Inc. (2004) Second Edition, Chapters 1-2

[9] *Goals*, Wodka official website | 5 April 2008
http://wodka.sourceforge.net/index.html

[10] *Sample Results of Evolution* – movies, Framsticks official website | 5 April 2008
http://www.framsticks.com/a/al_ewol.html

[11] *About the Project – objectives and scope*, Framsticks official website | 5 April 2008
http://www.framsticks.com/a/al_project

[12] *Physics-Based Animation*, by Kenny Erleben, Jon Sporring, Knud Henriksen, Henrik Dohlmann.
Charles River Media, Inc. (2005)

[13] Real-time Rendering, by Thomas Akenine-M:oller and Eric Haines
A K Peters, Ltd, USA (2002)

[14] *Symbolism on 3D Interactive Graphics*, by Mirtich, B. and Canny, J.F
(1995) P. 181-188, 217

[15] *Underlying Physics,* by Ed Burton | 5 Feb 2008
http://show.zoho.com/public/edburton/cinekid2007 Slide 12

[16] *Hooke's law*, Wikipedia | 8 April 2008
http://en.wikipedia.org/wiki/Hooke's_law

[17] *Simple Harmonic Motion*, Wikipedia | 8 April 2008
http://en.wikipedia.org/wiki/Simple_harmonic_motion

[18] *The physics behind the SodaConstructor*, Henry Jeckyll (2002) | 9 April 2008
http://web.archive.org/web/20031003174249/http://www.sodaplaycentral.com/sodamath/paper.php

[19] *Java & XML*, by Brett D McLaughlin

O'Reilly & Associates, Inc. (2001)

[20] *Callback (computer science)*, Wikipedia | 10 April 2008
http://en.wikipedia.org/wiki/Callback_(computer_science)

[21] *Java Architecture for XML Binding*, by Ed Ort and Bhakti Mehta (March 2003)
http://java.sun.com/developer/technicalArticles/WebServices/jaxb/#introjb | 12April2008

[22] *Java Architecture for XML Binding,* Wikipedia | 12 April 2008
http://en.wikipedia.org/wiki/JAXB

[23] *Iterative and incremental development*, Wikipedia | 15 April 2008
http://en.wikipedia.org/wiki/Iterative_and_incremental_development

[24] *Java OpenGL*, Wikipedia | 16 April 2008
http://en.wikipedia.org/wiki/Java_OpenGL

[25] Collision Detection, Wikipedia | 16 April 2008
http://en.wikipedia.org/wiki/Collision_detection

[26] Keeping a Constant Framerate, Java World | 2007
http://www.javaworld.com/jw-03-1996/animation/Example2Applet.html

Appendix

A. XML code of a SodaConstructor model

The following is the XML description of the model “KeniF_triangle”. Every time a model is imported, a document of the same format is parsed.

```
<object id="ConstructorApplication#1" type="com.sodaplay.soda.constructor2.ConstructorApplication">
 <field name="model">
  <object id="Model#1" type="com.sodaplay.soda.constructor2.model.Model">
   <field name="autoReverse">
    <boolean>true</boolean>
   </field>
   <field name="comment">
    <string />
   </field>
   <field name="friction">
    <double>0.05</double>
   </field>
   <field name="gravity">
    <double>0.3</double>
   </field>
   <field name="gravityDirection">
    <int>1</int>
   </field>
   <field name="height">
    <int>430</int>
   </field>
   <field name="metadata">
    <object id="Metadata#1" type="com.sodaplay.soda.constructor2.model.Metadata">
     <field name="pairs">
      <list class="java.util.Vector" />
     </field>
    </object>
   </field>
   <field name="name">
    <string>model</string>
   </field>
   <field name="nodes">
    <list class="java.util.Vector">
     <object id="Mass#1" type="com.sodaplay.soda.constructor2.model.Mass">
      <field name="vx">
       <double>-2.0882535107713296</double>
      </field>
      <field name="vy">
       <double>-1.2115243293454647</double>
      </field>
      <field name="x">
       <double>173.60420625497596</double>
      </field>
      <field name="y">
       <double>189.22208924389815</double>
      </field>
     </object>
     <object id="Mass#2" type="com.sodaplay.soda.constructor2.model.Mass">
      <field name="vx">
       <double>-0.011855015067731404</double>
      </field>
      <field name="vy">
       <double>0.5734082342855786</double>
      </field>
      <field name="x">
```

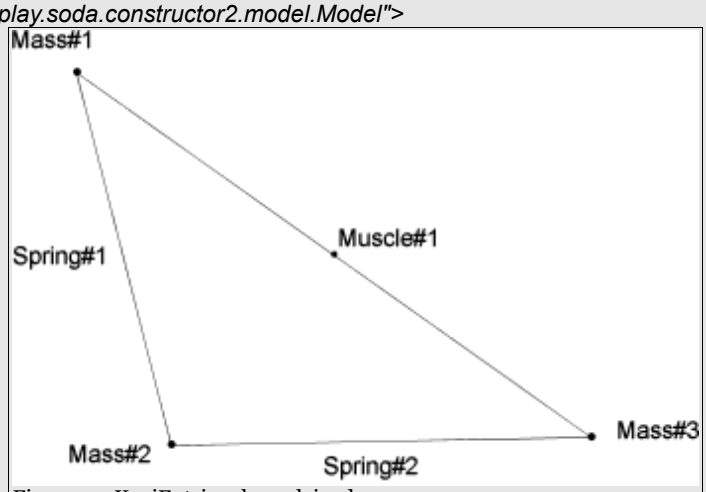

Figure 35: KeniF_triangle explained

```
        <double>219.24398123840555</double>
       </field>
       <field name="y">
        <double>0.0</double>
       </field>
      </object>
      <object id="Mass#3" type="com.sodaplay.soda.constructor2.model.Mass">
       <field name="vx">
        <double>-1.2464576578060318</double>
       </field>
       <field name="vy">
        <double>0.5105446232742532</double>
       </field>
       <field name="x">
        <double>432.92796876509817</double>
       </field>
       <field name="y">
        <double>0.8104115354068946</double>
       </field>
      </object>
     </list>
    </field>
    <field name="springs">
     <list class="java.util.Vector">
      <object id="Spring#1" type="com.sodaplay.soda.constructor2.model.Spring">
       <field name="a">
        <ref refid="Mass#2" />
       </field>
       <field name="b">
        <ref refid="Mass#1" />
       </field>
       <field name="restLength">
        <double>199.73617544495946</double>
       </field>
      </object>
      <object id="Spring#2" type="com.sodaplay.soda.constructor2.model.Spring">
       <field name="a">
        <ref refid="Mass#3" />
       </field>
       <field name="b">
        <ref refid="Mass#2" />
       </field>
       <field name="restLength">
        <double>216.4147516593543</double>
       </field>
      </object>
      <object id="Muscle#1" type="com.sodaplay.soda.constructor2.model.Muscle">
       <field name="a">
        <ref refid="Mass#1" />
       </field>
       <field name="amplitude">
        <double>1.0</double>
       </field>
       <field name="b">
        <ref refid="Mass#3" />
       </field>
       <field name="phase">
        <double>0.2512363996043521</double>
       </field>
       <field name="restLength">
        <double>210.6773334099998</double>
       </field>
      </object>
     </list>
    </field>
    <field name="springyness">
     <double>0.2</double>
```

```
    </field>
    <field name="surfaceFriction">
      <double>0.1</double>
    </field>
    <field name="surfaceReflection">
      <double>-0.75</double>
    </field>
    <field name="wave">
      <object id="Wave#1" type="com.sodaplay.soda.constructor2.model.Wave">
        <field name="amplitude">
          <double>0.5</double>
        </field>
        <field name="direction">
          <string>forward</string>
        </field>
        <field name="phase">
          <double>0.010000000000002665</double>
        </field>
        <field name="speed">
          <double>0.01</double>
        </field>
      </object>
    </field>
    <field name="width">
      <int>657</int>
    </field>
  </object>
 </field>
</object>
```

B. SAX Content Handlers

These methods are implemented in the XmlParser class:

```
public void characters(char ch[], int start, int length)
        // Information found between tags, in the form of a string
public void startElement(String uri, String localName, String qName, Attributes atts)
        // Called when an opening tag is reached, e.g. <object>
        // Attribute is the data followed by the opening tag
public void endElement(String uri, String localName, String qName)
        // Called when the closing tag is reached; not used in SodaSumo
public void startPrefixMapping(String prefix, String uri)
        // not used in SodaSumo
public void endPrefixMapping(String prefix)
        // not used in SodaSumo
public void ignorableWhitespace(char[] ch, int start, int length)
        // Called when white space is encountered
public void processingInstruction(String target, String data)
        // not used in SodaSumo
public void setDocumentLocator(Locator locator)
        // not used in SodaSumo
public void skippedEntity(String name)
        // notifies when an entity is skipped; not used in SodaSumo
public void startDocument()
        // Called when the document starts
public void endDocument()
        // Called when the end of the document is reached
```

C. SC Models Used in This Report

The following models have been used in this report. They can be loaded from SodaSumo by entering "creator/model" in the drop-down box.

Creator	Model	Figure
ed	daintywalker	1, 5, 8, 14, 17, 19, 23, 31, 34
ed	carefulslug	14, 23
mmaarrkkuuss	mmaarrkkuuss (roller)	26-30
karulo	monster_truck	17
kenif	kenif_triangle	19, 35
matthew102000	pivot_treader (climber)	34
lasse1234	diplopoda	18
mmaarrkkuuss	castle	18