Modelling Mechanics and Motion with Definitive Scripts

David R. Walton

January 2013

Abstract

Definitive scripts provide a powerful and intuitive way to express dependencies between observables, using definitions. A definition connects a number of observables, in much the way that links, both tangible and theoretical, connect physical systems. This paper investigates the connection between physical systems and definitive scripts further through the development of a number of empirical models using the JS-EDEN tool. The suitability of definitive scripts for modelling physical systems will be discussed in more detail, and finally potential applications and extensions of such models will be outlined.

1 Introduction

This report will investigate the development of a number of models of physical systems using definitive scripts.

One of the most distinctive features of definitive scripts is their ability to easily and concisely express relationships between observables. These relationships are then automatically maintained by the interpreter, without the need for interaction on the part of the modeller. The definitions in a definitive script can be used to link a model together in a direct and intuitive way. These links can be seen as analogous to the physical connections between parts of a machine, or to the the equations defining the motion of particles in Newtonian mechanics. For this reason, such scripts have the potential to be used to produce models in which the code corresponds directly and intuitively to such equations and physical connections.

This connection between definitive scripts and physical links is the motivation behind this modelling study. This report will describe the development of a number of models of physical systems using the JS-EDEN tool. These models will then form the basis for a discussion of the application of definitive scripts in general, and JS-EDEN in particular, to the production of such models. For information on how to run the provided models, see Appendix 6.1.

The models produced will be roughly divided into two categories, which make use of definitive scripts in different ways. The first type of model will

use definitions to describe the geometric relationships between moving parts in a mechanism. The second will make use of definitions to describe the forces acting on particles, in a model based on Newtonian particle mechanics. Scripts to produce the models will be included as an appendix to this report, and, where appropriate, parts of the script will be included here.

Other excellent empirical models have been developed which simulate physical [5] and mechanical [4] systems. This paper attempts to give a broad overview of approaches to the development of such models, and to provide and discuss some examples.

The focus in developing these scripts will be on producing elegant, human-readable code which makes use of the unique features of definitive scripts as often as is possible. For this reason, procedural code and injected JavaScript will be kept to a minimum. Features such as computational efficiency will be of secondary importance.

There is, in the author's opinion, a second reason to minimise use of procedural code in definitive scripts as far as possible. Generally, all parts of models used in EM are intended for interaction at any time. For this reason, the modeller should ideally be able to rewrite or redefine any aspect of the model during interaction, whenever she desires. When a script consists primarily of definitions, this is easily achieved, by typing new definitions into the interpreter. However, if the script contains lengthy `proc` and `func` constructions, these are in some sense "sealed" - their contents cannot be modified except by redefining the whole procedure. This restricts interaction, and this restriction is somewhat contrary to the spirit of Empirical Modelling (EM).

2 Approach 1: Defining Physical Links

The first models in this study make extensive use of definitions to geometrically describe the physical links between objects. As an example, consider the following simple script, which produces a model of a rigid pendulum:

```
1  ## The following script should be run in the latest version of JS-EDEN.
2  theta is mouseX % 360;
3
4  pivotX = 200; pivotY = 200;
5  rodLength = 100; weightRadius = 10;
6
7  weightX is pivotX + rodLength * cos(theta);
8  weightY is pivotY + rodLength * sin(theta);
9
10 rod is Line(pivotX, pivotY, weightX, weightY);
11 weight is Circle(weightX, weightY, weightRadius, "yellow");
12
13 picture is [rod, weight];
```

Lines 7 and 8 define the position of the weight on the pendulum in terms of the pivot position, length of the rod, and the angle `theta` of the pendulum from the horizontal. For simplicity, `theta` is defined in terms of the horizontal position of the mouse. Note the way in which lines 7 and 8 simply describe the geometric relationship between the parts of the pendulum, and the fact that this simple, readable code is enough to produce an interactive model.

A similar approach was used to produce the following models.

2.1 Engines from Definitive Notations

These models are simple moving diagrams of two very different combustion engines. The consistent, predictable motion of such engines makes them highly appropriate for modelling using this approach. Such models of engines have a wide variety of potential applications, including education and CAD/CAM. Diagrams of some of the components of the modelled engines are contained in appendix 6.2, and may be helpful in interpreting the code for these models.

2.1.1 Four-Stroke Engine

The first engine to be modelled is a four-stroke internal combustion engine. These widely used engines are named for the four stages which take place in each cylinder during an engine cycle [9], [3]. In chronological order, these are:

- **Fuel Intake** : Fuel and air are drawn into the cylinder as the piston moves downwards.
- **Compression** : The piston moves upwards, compressing the fuel/air mixture.
- **Expansion** : As the piston reaches the top of the cylinder, the spark plug fires. The fuel/air mixture burns and expands, forcing the piston downwards.
- **Exhaust** : The piston moves upwards, expelling the burnt mixture from the cylinder.

This model is a moving diagram of a single cylinder of such an engine. The script is designed to reflect both the geometric relationships between the engine parts and the cyclic nature of the engine.

The motion of the parts in the engine is determined by an observable `theta`. This is congruent to the angle of the cam relative to the horizontal. Since the camshaft rotates twice during a single engine cycle, `theta` $\in [0, 720)$.

This observable is used to determine the current engine stage, as follows:

- **Fuel Intake**: `theta` $\in [90, 270)$
- **Compression**: `theta` $\in [270, 450)$
- **Expansion**: `theta` $\in [450, 630)$

- **Exhaust**: `theta` $\in [0, 90) \cup [630, 720)$

Much of the rest of the script consists of describing the geometric relationships between the parts using definitions, and defining shapes to represent parts of the engine. However, this model also has the capability to move automatically when a button is pressed, and this required a small amount of procedural code. This code simply repeatedly calls a procedure which increases the value of `theta` according to the elapsed time. That is, `theta` is gradually increased using a variable time step.

2.1.2 Newcomen Steam Engine and Pump

The second engine to be considered is an early steam engine developed by Thomas Newcomen in 1712. Most Newcomen engines were used to power pumps to remove water from mines, and as such, this model is a diagram of a Newcomen engine attached to a simple dead weight pump.

Atmospheric steam engines operate in a fundamentally different way to internal combustion engines; rather than being powered by expansion, they are powered by contraction [10], [11]. Specifically, most steam engines operate roughly in the following way:

- A volume is filled with hot steam.
- A small amount of cold water is added to the volume.
- The water causes the steam to condense, reducing in volume dramatically.
- This creates suction, providing power.

The Newcomen engine makes use of a pivot, on one side of which is a weight, and on the other a piston in a cylinder attached to a boiler and a supply of cold water. An engine cycle has two stages:

1. **Filling** The weight pulls down on the pivot, pulling the piston up. This draws hot steam into the cylinder from the boiler.
2. **Contraction** A small amount of cold water is sprayed into the cylinder. The steam inside condenses, pulling the piston down. This pulls up the weight, ready for the next cycle.

In this model, the weight has an additional role, as it forms part of a dead weight pump. This pump consists of an internal volume filled with water, connected to an inlet and an outlet. Valves ensure that water can only enter the volume through the inlet, and only exit through the outlet.

When the weight falls, it drops into the water, displacing some and forcing it out of the outlet. When it rises again, water flows in through the inlet to fill the empty space. As the weight repeatedly rises and falls, water is pumped from the inlet to the outlet.

The code for this model is structured in a very similar way to that used in the four stroke engine model, and the same procedural code is used to provide automatic movement.

2.1.3 Discussion

This approach proved successful in producing models of some complexity, which move in a cyclic fashion. During the development of these model, the author first drew diagrams of the structure of the models, and then used these as a reference. The ability to use definitions allowed the code to correspond directly to the diagrams, making much of the development easy and intuitive.

There are, however, sections of these scripts which proved more difficult to write, and are now, in the author's opinion, overly verbose and difficult to read.

Perhaps the most verbose definitions in these models are those of the cam and connecting rod in the four-stroke engine, and of the beam in the Newcomen engine. These definitions use polygons to describe rounded shapes, and as such require a large number of points. Additionally, as all these parts are moving, every point must be defined in terms of other observables. This makes for long definitions, which could be considered unhelpful to future readers of the code.

One solution to this problem would be to define suitable drawing functions for these shapes. This would allow, for example, for the long `connectingRod` definition from the four-stroke model to be replaced with something similar to the following:

```
connectingRod is ConnectingRod(camX, camY, pistonX, pistonY, r, "black");
```

This would make for simpler, more readable code, make refactoring easier, and would also make the definitions for these shapes easily re-usable in future models.

3 Approach 2: Defining Theoretical Links

One feature of the previous models which might be considered unsatisfactory is the way in which the movement of all the components depends directly upon a single observable, which is in some sense separate from the model. In the case of the four-stroke engine, this observable is called `theta` and in the Newcomen model it is called `cycle`. This does not mirror real-world mechanisms, in which components obeying physical laws push, pull, twist and turn one another, (hopefully) moving in the desired way.

If we removed the connecting rod from a four-stroke engine, we would expect the piston to fall to the bottom of the cylinder. If we stopped adding fuel, we would expect the engine to stop after a short time. Applying the same changes to the four-stroke engine above will not have the expected effects; remove the connecting rod, and the engine surprisingly functions otherwise as normal. Remove the fuel, and the model is largely unaffected. In a compiled program, such changes could not be performed, and this could be considered unimportant. In EM, however, modellers have the ability (and, indeed are encouraged) to interact with models as they run, and adding, removing and redefining observables is common.

This provides the motivation for a second way of modelling mechanical systems. This involves defining the forces acting on objects in a mechanical system, and then using numerical methods to simulate their movement. This allows the model to evolve dynamically through time, without being controlled by an "external" variable (such as `cycle` or `theta` above). It also allows components to react to changes in observables in a more realistic way. It does, however, require the use of somewhat more procedural code.

3.1 Example: Pendulum Mk II

As a first example of this approach, the pendulum model has been altered so that its movement is controlled by a physical simulation. The code is now too long to include in the body of this paper, and is included in appendix 6.8. This simulation makes use of definitions to define the angular acceleration in terms of the angle. This acceleration value is then used in a simple forward Euler system to first find the angular speed and subsequently the position of the mass. This system uses a variable timestep.

Although the GUI for this model is limited to just "Reset" and "Run" buttons, users are able to use the interpreter window to change the values of observables such as `theta`, `angularSpeed`, `friction`, `mass` and `rodLength` and the model will react accordingly.

3.2 Bouncing Ball

The second model produced using this approach is a simple model of a round ball contained in a rigid box. This model was developed for the EMILE version of JS-EDEN, and, as it makes use of `Slider()`, will not work in the latest version.

This model is designed for interaction via changing the values of observables whilst the simulation is running. This is facilitated by the addition of sliders.

This model uses a similar general approach to that of "Pendulum Mk II", making use of EDEN definitions to maintain correct force, and hence acceleration values, and then using a forward Euler numerical method to move the ball appropriately. One notable difference is part of the numerical procedure, which accounts for collisions between the ball and the walls of the box. The interaction of a bouncing ball with a solid surface is highly complex, and takes place over just a few microseconds [6]. Thus, simulating this accurately would not be possible in this simple model.

The model uses a fairly good approximation, assuming a fixed coefficient of restitution C between the ball and the walls. Since the walls are assumed to be perfectly rigid, immobile and identical to one another, C is equal to the ratio of pre and post-impact speeds of the ball along the line of impact. Thanks to this, in the case of an impact, it suffices to negate the component of the balls velocity perpendicular to the wall, and multiply it by C.

This model essentially resolves ball-wall collisions by detecting intersections between the ball and the wall, moving the ball out of the wall if necessary, and

then negating the component of the balls velocity perpendicular to the wall, and multiplying it by C as described above.

This procedure works reasonably well, although it produces unrealistic behaviour at high speeds. This is an issue with numerical methods in general - the time step must be small compared to the distance moved in order to provide a good approximation. This can be a problem if the magnitude of the force and C are high, and friction is low for some time.

3.3 Attractive Forces

This model is an extension of "Bouncing Ball" which adds a second simulated ball, and an attractive (or repulsive) force between the two balls. This force is governed by an inverse square law, in a similar way to gravitational or electrostatic forces. Additionally, the balls are capable of colliding realistically with one another. Again, this model only functions in the EMILE version of JS-EDEN.

Adding the second ball did not in itself require many significant changes, other than the addition of some simple definitions accounting for the attractive forces between the balls. The most significant addition in this model is the addition of collision resolution between the balls. Again, an approximation was employed, assuming a fixed coefficient of restitution C_b between the two balls. However, since neither ball is fixed, resolving the collision is somewhat more complex.

Overall, collision resolution for the balls takes place as follows:

1. Detect intersecting balls.
2. Push balls apart along line of impact to remove intersection.
3. Update velocity of each ball as described below.

First, we make the simplifying assumption that the speed of the balls perpendicular to the line of impact is unchanged by the collision. It then remains to find the speed of the balls along the line of impact. This reduces the problem to resolving the collision of two balls moving along a line. The definition of coefficient of restitution and conservation of momentum may be used to derive the post-collision speeds of each ball (see appendix 6.5 for the derivation).

As with the "Bouncing Ball" model, this model functions poorly at very high speeds. There is an additional potential problem in that, at high speeds, the balls can pass through one another in an unrealistic way. This occurs if the balls move sufficiently far in a single timestep so that they do not intersect with one another before passing. Worse still is the case where the balls pass more than halfway through each other and then intersect.

Generally, this resolution procedure performs adequately, however. Again, theese problems only arise if the magnitude of the force and C are high, and friction is low for some time.

3.4 Discussion

These models provide simplified but effective models of 2-dimensional physical systems. Regrettably, no small amount of code had to be included in the `step()` procedure, making it less accessible to the user when interacting with the model. However, much of the model is produced using definitions and thus may be modified easily during interaction. As has been shown, the basic model "Bouncing Balls" was fairly easily extended to produce "Attractive Forces". Further extensions could also be simply carried out, for example by adding or removing particles, or altering the forces governing their movement. Some possible examples follow:

- Adding a third ball to simulate the three-body problem.
- Simulating a satellite orbiting a planet by using a single ball with an inverse square attractive force to a fixed point.
- Modifying forces so they are dependent on position (i.e. defining a force field). For example:
  - The force field around a bar or horseshoe magnet, to simulate magnetic attraction.
  - A spiral force field, to simulate the motion of floating balls in a whirlpool.
- Modifying forces so they are dependent upon time and position (requires more procedural code). For example:
  - Simulating the motion of floating balls in waves.
  - Simulating attraction to an electromagnet with varying input current.

Such models allow for relatively easy development and interaction, and as such would be well suited to, for example, education.

4 A Combined Approach

A natural question to ask is whether these two different approaches to producing mechanical models may be combined. The following model provides an example of how this might be achieved.

4.1 Pendulum Clock

This model is a fairly simple extension of the "Pendulum Mk II" model. It is designed to mimic the behaviour of a simple pendulum clock.

In a pendulum clock, a pendulum is attached to an escapement mechanism. This mechanism serves two purposes. Firstly, it provides a small impulse to the pendulum, keeping it moving. Secondly, each time the pendulum passes the

vertical position, a gear in the escapement moves forward one step, advancing the time on the clock face by one second.

This model does not attempt to simulate the relatively complex mechanism of the escapement, but simply mimics its behaviour. The model extends the pendulum model by adding a clock face. The update procedure advances the second hand whenever the pendulum passes the vertical, and definitions connect the minute and hour hands to the second hand, moving them appropriately:

```
minuteAngle is secondAngle / 60;
hourAngle   is minuteAngle / 60;
```

The unique features of definitive scripts allowed this extension to be implemented with relative ease. This relatively trivial addition to the pendulum model demonstrates the way in which these two approaches to producing mechanical models may be combined. This model could be extended, for example by including details of the escapement, weights and gearing. It would likely be most appropriate to model these components using “Approach 1”.

5 Summary

In this modelling exercise, a number of models for moving systems were produced which were designed in different ways. Some consisted of descriptions of the geometry and motion of the systems; simulating effects rather than causes. Other models attempted to describe the physical laws governing the motion of the systems; these models attempted to simulate causes in order to produce effects. A final model used elements from each approach.

5.1 Evaluation of Approaches

Both approaches have advantages and disadvantages, which became apparent during the modelling process. Some of these are summarised in the table below:

Approach 1.		Approach 2.	
Advantages	**Disadvantages**	**Advantages**	**Disadvantages**
Complex models can be produced with simple code, and little procedural code.	Models do not hold up well to redefinitions.	Models can respond well to some redefinitions.	Complex models may require large amounts of code, and more procedural code
Models behave predictably.	Models do not exhibit dynamic behaviour.	Models exhibit dynamic behaviour.	Models do not always behave predictably.
Modeller develops understanding of geometric relationships.	Modeller may not develop understanding of causes of behaviour.	Modeller develops understanding of causes of behaviour.	Modeller may not develop understanding of geometric descriptions of motion.
Well suited to repetitive movements.	Complex movement requires complex code.	Complex movement can arise from simple code.	Not well suited to scripting repetitive movements.

Models combining these approaches will adopt advantages and disadvantages from both sides of this table. Whether a geometric, physical or hybrid approach is most effective will depend greatly on the phenomena the modeller intends to explore, and the nature of the understanding she wishes to develop in the modelling process.

5.2 Applications and Extensions

As mentioned previously in this report, a major potential application for EM in general, and this type of model in particular, is in education [7], [8]. In the production of these models, the author gained a deeper understanding of geometry, Newtonian mechanics and hermeneutic computing, in addition to learning more about engines and clocks. Learning by developing a model has a power and immediacy to which passive learning cannot compare.

In modelling a mechanical device, a student could gain a better understanding of its functionality, in addition to deepening their understanding of geometric mathematics and programming. In modelling a physical system, they could also better understand the mechanical mathematics of Newtonian physics.

JS-EDEN is already a powerful tool, with a great deal of potential to be used in this way. However, some additions to the existing software could make it more suitable for educational purposes, removing some of the aspects of modelling which were time-consuming and perhaps of less educational value.

One way to potentially improve the experience for a student would be to

include libraries of functions to aid in the task at hand. For example, a student tasked with producing a model of an engine could be provided with a library of functions to draw pistons, connecting rods, valves etc. This would remove some of the more verbose and unreadable code, which the author found less pleasant to write. It could, however, be argued that writing functions to draw these observables has educational value; defining the pivoting beam in the Newcomen model, for example, required the use of trigonometry and vector arithmetic, and could be deemed educational. Any such library would have to strike a balance between making the modelling experience enjoyable, and not providing the student with too much of the model at the outset of the exercise.

The content of the library would also depend upon the focus of the modelling exercise. As an example, consider the `step()` procedure in the “Bouncing Ball” model. If one of the intended aims of an exercise was to teach a student to write procedural code, she could be tasked with writing this part of the script herself. If the exercise intended to focus more on mathematics and geometry than programming, this procedure could be provided to her as part of a library.

Another potential application for such scripts would be in CAD. The features of definitive scripts made the development of the engine models relatively straightforward and fast, and a comprehensive library of drawable shapes could have made it even faster. Development speed could be further augmented with the addition of a GIMP-like drawing GUI, with a toolbox of common engine parts. For more unusual shapes or definitions, the option to code using definitive scripts would also be available. With the addition of a vector datatype and some vector manipulation functions, definitive scripts could also be readily applied to produce three-dimensional mechanical models. The use of definitive scripts would also allow the models to be fairly easily updated and changed. Charles Care’s “Planimeter” model [4] has demonstrated that definitive scripts may be employed to develop complex three-dimensional mechanical models.

5.3 Conclusion

Despite using a relatively small amount of quite readable code, the models produced during this report are surprisingly rich, offering great scope for interaction, modification and extension. They also contributed greatly to the author’s understanding of the systems involved. This demonstrates some of the potential of the JS-EDEN tool to develop deep and fulfilling models of mechanics and motion.

References

[1] Js-eden: Emile at warwick department of computer science. `http://jseden.dcs.warwick.ac.uk/emile/`.

[2] Js-eden: Latest at warwick department of computer science. `http://jseden.dcs.warwick.ac.uk/latest/`.

[3] Video taken from inside a 4-stroke engine cylinder. http://www.liveleak.com/view?i=73e_1192001762.

[4] Charles Care. Planimeter model. http://empublic.dcs.warwick.ac.uk/projects/planimeterCare2005/.

[5] Ben Carter. Billiards model. http://empublic.dcs.warwick.ac.uk/projects/billiardsCarter1999/.

[6] R. Cross. The bounce of a ball. *American Journal of Physics*, 67:222, 1999.

[7] A. Harfield and M. Beynon. Empirical modelling for constructionist learning in a thai secondary school mathematics class. *The Ninth International Conference on eLearning for Knowledge-Based Society*, 2012.

[8] A. Harfield, M. Beynon, and R. Myers. Web eden and moodle: an empirical modelling approach to web-based education. In *Proceedings of the Eighth IASTED International conference on Web-Based Education*, pages 16–18, 2009.

[9] M. Keveney. Four-stroke engine animation and explanation. http://www.animatedengines.com/otto.html.

[10] M. Keveney. Newcomen engine animation and explanation. http://www.animatedengines.com/newcomen.html.

[11] Dionysius Lardner. *The Steam Engine Familiarly Explained and Illustrated*. 1851. Available as a free ebook at Google Books.

6 Appendices

6.1 Running the Models: JS-EDEN and EMILE

At the time of writing, two versions of JS-EDEN may be accessed via the Warwick Department of Computer Science webpage. One is the most recent version, located at [2]. The other is a version of JS-EDEN designed for the EM Interactive Learning Experiment (EMILE), located at [1]. Wherever possible, the models in this report use the latest version of JS-EDEN, however some models require features present only in the EMILE version. Models written for one version are not always compatible with the other. Reasons for this include:

- Angles in the EMILE version are expressed in radians, and in the latest version are expressed in degrees. This means models using trigonometric functions may exhibit unexpected behaviour if moved between versions.

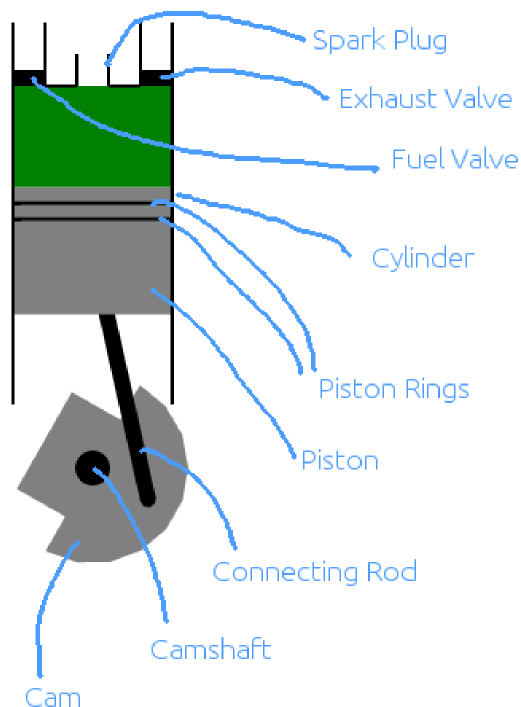
- The EMILE version includes a number of extra functions, including `Slider()`, `Matrix()` and `Vector()`. Models using this function will not function in the latest version[1].

Unless explicitly stated otherwise, all scripts in this report were designed for the latest version of JS-EDEN, and should be executed by accessing [2] using Google Chrome (The Google Chrome browser may be downloaded at `http://www.google.co.uk/chrome`), copying the code into the EDEN interpreter window, and pressing the 'submit' button. Incompatibilities will be noted both in comments in the script itself, and in the report where appropriate.

Please note that JS-EDEN is, at the time of writing, still considered a work in progress. As such, bugs in the tool may be encountered during use, or future changes could render the attached scripts obsolete. If problems running the models are encountered, please contact the author at `d.r.walton@warwick.ac.uk`.

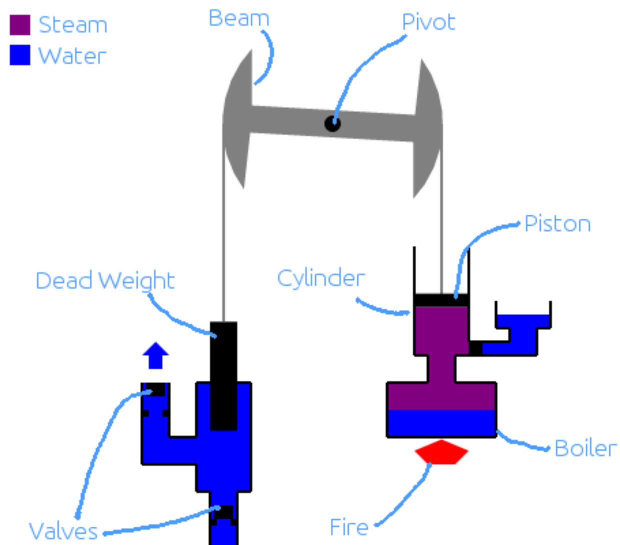
6.2 Diagrams of Engine Components

6.3 Four-stroke Engine



[1] Running such a script in the latest version will likely produce an "ERROR number 1: Cannot call method 'call' of undefined" error. Note, however, that there are other reasons why this common error message may occur.

6.4 Newcomen Atmospheric Engine



6.5 Derivation of Velocity Equation

Suppose two balls b_1 and b_2 with coefficient of restitution C collide with one another. Let:

- m_1 and m_2 be the masses of the two balls.
- u_1 and u_2 be their speeds along the line of impact, immediately before the collision.
- v_1 and v_2 be their speeds along the line of impact, immediately after the collision.

Then, from the definition of C, we have:

$$C = \frac{v1 - v2}{u1 - u2} \tag{1}$$

Additionally, by conservation of momentum, we have:

$$u_1 m_1 + u_2 m_2 = v_1 m_1 + v_2 m_2 \tag{2}$$

Rearranging (1) gives:

$$v_1 = C(u_1 - u_2) + v_2$$

Substituting this into (2) and performing some manipulation gives:

$$v_2 = \frac{u_1 m_1 + u_2 m_2 + m_1 C(u_2 - u_1)}{m_1 + m_2} \tag{3}$$

Similarly, expressing (1) in terms of v_2 and substituting into (2) gives:

$$v_1 = \frac{u_1 m_1 + u_2 m_2 + m_2 C(u_1 - u_2)}{m_1 + m_2} \tag{4}$$

6.6 Four-stroke Engine

```
## *** Four-Stroke Engine ***
##
## NOTE: The following script is written for the latest version of JS-EDEN.
## This may be found at:
##     http://jseden.dcs.warwick.ac.uk/latest/
##
## This script WILL NOT RUN CORRECTLY on the EMILE version
## of JS-EDEN, located at:
##     http://jseden.dcs.warwick.ac.uk/emile/
##
## This is due to the fact that more recent versions of JS-EDEN measure
## angles in degrees, wheras EMILE uses radians. This script could be
## adapted for EMILE by simply changing angles to radians as appropriate.
##
## The following script produces a simple model of a cylinder of a
## four-stroke engine.
##
## Run this script by copying and pasting the contents of this file into the
## EDEN interpreter window. The simulation parameters may then be adjusted
## using the sliders. To run the engine for a short period of time, press
## the "Run" button.

## *** Simulation Parameters ***

## Constant to control simulation speed.
speed = 72;

theta = 90;

camLength = 40;
rodLength = 160;

camShaftX is 100;
camShaftY is 310;

```

```
36 ## Booleans describing current state of engine.
37 fuelIn is (theta > 90 && theta < 270);
38 compression is (theta > 270 && theta < 450);
39 expansion is (theta > 450 && theta < 630);
40 exhaust is (theta< 90 || theta > 630);
41 spark is (theta > 450 && theta < 470);
42
43 ## *** Dynamic Observables ***
44
45 ## *UI*
46
47 ## Button to run the engine simulation for a short time.
48 runActive = true;
49 runButton is Button("Run", "Run", 200, 30, runActive);
50
51 ## Text describing status of the engine.
52 statusText1 is Text("1. Fuel and air enter the cylinder.",
53           200, 70,  fuelIn      ? "red" : "black");
54 statusText2 is Text("2. The fuel/air mixture is compressed by the piston.",
55         200, 90,  compression ? "red" : "black");
56 statusText3 is Text("3. The spark plug ignites the mixture and it expands.",
57         200, 110, expansion   ? "red" : "black");
58 statusText4 is Text("4. The burnt exhaust is expelled from the cylinder.",
59         200, 130, exhaust     ? "red" : "black");
60
61 camShaft is Circle(camShaftX, camShaftY, 10);
62
63 ## Defines the point at which the connecting rod meets the cam.
64 camX is camShaftX + (camLength * cos(theta));
65 camY is camShaftY - (camLength * sin(theta));
66
67 ## Piston movement and shape.
68 pistonX = camShaftX;
69 pistonY is camShaftY - (camLength * sin(theta)
70             + sqrt(rodLength*rodLength
71                 - camLength*camLength*cos(theta)*cos(theta)));
72
73 piston is Rectangle(pistonX - 50, pistonY - 40, 100, 80, "grey");
74 pistonRingTop is Line(50, pistonY - 30, 150, pistonY - 30);
75 pistonRingBottom is Line(50, pistonY - 20, 150, pistonY - 20);
76
77 ## Constants determining shape of the cam.
78 r_1 = 60;
79 r_2 = 35;
80 r_3 = sqrt(2) * r_2;
81
```

```
82  ## Polygon representing cam.
83  cam is Polygon([
84  camShaftX + r_1*cos(-theta - 90), camShaftY + r_1*sin(-theta - 90),
85  camShaftX + r_1*cos(-theta - 70), camShaftY + r_1*sin(-theta - 70),
86  camShaftX + r_1*cos(-theta - 50), camShaftY + r_1*sin(-theta - 50),
87  camShaftX + r_1*cos(-theta - 30), camShaftY + r_1*sin(-theta - 30),
88  camShaftX + r_1*cos(-theta - 10), camShaftY + r_1*sin(-theta - 10),
89  camShaftX + r_1*cos(-theta + 10), camShaftY + r_1*sin(-theta + 10),
90  camShaftX + r_1*cos(-theta + 30), camShaftY + r_1*sin(-theta + 30),
91  camShaftX + r_1*cos(-theta + 50), camShaftY + r_1*sin(-theta + 50),
92  camShaftX + r_1*cos(-theta + 70), camShaftY + r_1*sin(-theta + 70),
93  camShaftX + r_1*cos(-theta + 90), camShaftY + r_1*sin(-theta + 90),
94
95  camShaftX + r_2*cos(-theta + 90 ), camShaftY + r_2*sin(-theta + 90 ),
96  camShaftX + r_3*cos(-theta + 135), camShaftY + r_3*sin(-theta + 135),
97  camShaftX + r_3*cos(-theta + 225), camShaftY + r_3*sin(-theta + 225),
98  camShaftX + r_2*cos(-theta - 90 ), camShaftY + r_2*sin(-theta - 90 )
99  ], "grey");
100
101 ## Constant defining thickness of connecting rod.
102 r = 5;
103
104 ## Polygon representing connecting rod.
105 connectingRod is Polygon([
106 camX + (r / rodLength) * (cos(90 )*(pistonX - camX) - sin(90 )*(pistonY-camY)),
107 camY + (r / rodLength) * (sin(90 )*(pistonX - camX) + cos(90 )*(pistonY-camY)),
108 camX + (r / rodLength) * (cos(135)*(pistonX - camX) - sin(135)*(pistonY-camY)),
109 camY + (r / rodLength) * (sin(135)*(pistonX - camX) + cos(135)*(pistonY-camY)),
110 camX + (r / rodLength) * (cos(180)*(pistonX - camX) - sin(180)*(pistonY-camY)),
111 camY + (r / rodLength) * (sin(180)*(pistonX - camX) + cos(180)*(pistonY-camY)),
112 camX + (r / rodLength) * (cos(225)*(pistonX - camX) - sin(225)*(pistonY-camY)),
113 camY + (r / rodLength) * (sin(225)*(pistonX - camX) + cos(225)*(pistonY-camY)),
114 camX + (r / rodLength) * (cos(270)*(pistonX - camX) - sin(270)*(pistonY-camY)),
115 camY + (r / rodLength) * (sin(270)*(pistonX - camX) + cos(270)*(pistonY-camY)),
116
117 pistonX + (r / rodLength) * (cos(90 )*(camX - pistonX) - sin(90 )*(camY-pistonY)),
118 pistonY + (r / rodLength) * (sin(90 )*(camX - pistonX) + cos(90 )*(camY-pistonY)),
119 pistonX + (r / rodLength) * (cos(135)*(camX - pistonX) - sin(135)*(camY-pistonY)),
120 pistonY + (r / rodLength) * (sin(135)*(camX - pistonX) + cos(135)*(camY-pistonY)),
121 pistonX + (r / rodLength) * (cos(180)*(camX - pistonX) - sin(180)*(camY-pistonY)),
122 pistonY + (r / rodLength) * (sin(180)*(camX - pistonX) + cos(180)*(camY-pistonY)),
123 pistonX + (r / rodLength) * (cos(225)*(camX - pistonX) - sin(225)*(camY-pistonY)),
124 pistonY + (r / rodLength) * (sin(225)*(camX - pistonX) + cos(225)*(camY-pistonY)),
125 pistonX + (r / rodLength) * (cos(270)*(camX - pistonX) - sin(270)*(camY-pistonY)),
126 pistonY + (r / rodLength) * (sin(270)*(camX - pistonX) + cos(270)*(camY-pistonY))
127 ], "black");
```

```
128
129 ## Defines colours of various parts of the engine with reference to current state.
130 fuelColour    is fuelIn  ? "green"  : "white";
131 exhaustColour is exhaust ? "yellow" : "white";
132 sparkColour   is spark   ? "red"    : "white";
133 boreColour is (fuelIn || compression) ? "green"
134                  : ( expansion ? "red" : "yellow");
135
136 ## Coloured rectangles representing fluids in various parts of the engine.
137 fuelPipe is Rectangle(51, 30, 18, 40, fuelColour);
138 exhaustPipe is Rectangle(131, 30, 18, 40, exhaustColour);
139 sparkPlug is Rectangle(91, 50, 18, 20, sparkColour);
140 bore is Rectangle(51, 70, 98, pistonY - 110, boreColour);
141
142 ## Valves appear conditionally, dependent upon engine state.
143 fuelValve is !fuelIn ? Rectangle(50, 60, 20, 10) : null;
144 exhaustValve is !exhaust ? Rectangle(130, 60, 20, 10) : null;
145
146 ## *** Static Observables ***
147
148 ## Outline of engine.
149 leftLine        = Line(50,  30, 50,  270);
150 rightLine       = Line(150, 30, 150, 270);
151 fuelLine        = Line(70,  30, 70,   70);
152 exhaustLine     = Line(130, 30, 130,  70);
153 sparkLeftLine   = Line(90,  50, 90,   70);
154 sparkRightLine  = Line(110, 50, 110,  70);
155 fuelSparkLine   = Line(70,  70, 90,   70);
156 sparkExhaustLine = Line(110, 70, 130,  70);
157
158 picture is [cam, camShaft,
159         fuelPipe, exhaustPipe, sparkPlug, bore, connectingRod, piston,
160         pistonRingTop, pistonRingBottom,
161         fuelValve, exhaustValve,
162     sparkLeftLine, sparkRightLine, fuelSparkLine, sparkExhaustLine,
163     leftLine, rightLine, fuelLine, exhaustLine,
164     runButton, statusText1, statusText2, statusText3, statusText4];
165
166 ## *** Automatic Movement ***
167
168 ## Finds current time using the JS Date.getTime() function.
169 ## Note that this gives time in milliseconds since 1st Jan, 1970.
170 func getTime
171 {
172     return ${{ new Date().getTime() }}$;
173 }
```

```
174
175 ## Moves the simulation forward one time step.
176 proc step
177 {
178     after(1)
179     {
180         ## Timing. Note that timeStep is converted into seconds.
181         timeStep = (getTime() - currTime) / 1000;
182         currTime = getTime();
183
184         theta += timeStep * speed;
185         while(theta >= 720) theta -= 720;
186     }
187 }
188
189 ## Runs the simulation for a short period of time.
190 proc runSim : Run_clicked
191 {
192     runActive = false;
193     currTime = getTime();
194     for(i = 0; i < 5000; i++)
195     {
196         step();
197     }
198     runActive = true;
199 }
```

6.7 Newcomen Engine

```
1  ## *** Newcomen Engine ***
2  ##
3  ## NOTE: The following script is written for the latest version of JS-EDEN.
4  ## This may be found at:
5  ##     http://jseden.dcs.warwick.ac.uk/latest/
6  ##
7  ## This script WILL NOT RUN CORRECTLY on the EMILE version
8  ## of JS-EDEN, located at:
9  ##     http://jseden.dcs.warwick.ac.uk/emile/
10 ##
11 ## This is due to the fact that more recent versions of JS-EDEN measure
12 ## angles in degrees, wheras EMILE uses radians. This script could be
13 ## adapted for EMILE by simply changing the angle measurements.
14 ##
15 ## The following script produces a simple model of a Newcomen steam engine,
16 ## running a dead weight pump.
17 ##
```

```
18  ## Run this script by copying and pasting the contents of this file into the
19  ## EDEN interpreter window. The simulation parameters may then be adjusted
20  ## using the sliders. To run the engine for a short period of time, press
21  ## the "Run" button.
22
23  ## *** Simulation Parameters ***
24
25  ## Constant to control simulation speed.
26  speed = 14;
27
28  ## Constants used to define pivoting beam.
29  pi = 3.14159265358979323846264338327950;
30  r_1   = 60.827625302982196889996842452021;   ## sqrt(60*60 + 10*10)
31  r_2   = 81;                                   ## slightly more than sqrt(60*60 + 50*50)
32  phi_1 = 9.4623222080256173911400705417420;    ## arctan(1 / 6)
33  phi_2 = 39.805571092265194006279489819298;   ## arctan(5 / 6)
34  phi_3 = 29.854178319198895504709617364473;   ## arctan(5 / 6) * (3 / 4)
35  phi_4 = 19.902785546132597003139744909649;   ## arctan(5 / 6) * (1 / 2)
36  phi_5 = 9.9513927730662985015698724548245;   ## arctan(5 / 6) * (1 / 4)
37
38  ## The 'cycle' variable governs the movement of the model.
39  ## 'cycle' lies in the interval [0,140).
40  cycle = 1;
41  filling is     (cycle < 70);
42  contraction is (cycle >= 70);
43
44  ## *** Dynamic Observables ***
45
46  ## *UI*
47
48  ## Button to run the engine simulation for a short time.
49  runActive = true;
50  runButton is Button("Run", "Run", 400, 310, runActive);
51
52  ## Text describing state of machine.
53  statusText1 is Text("Engine Status", 330, 70);
54  statusText2 is Text("1. The cylinder fills with steam from the boiler",
55    330, 90, filling ? "red" : "black");
56  statusText3 is Text(
57    "2. Cold water added to the cylinder causes the steam to condense, pulling piston down.",
58    330, 110, contraction ? "red" : "black");
59
60  statusText4 is Text("Pump Status", 330, 140);
61  statusText5 is Text(
62    "1. The dead weight falls, displacing water out of the pump and pulling the piston upwards
63    330, 160, filling ? "red" : "black");
```

```
64  statusText6 is Text(
65    "2. The piston pulls the dead weight up, drawing water into the pump.",
66    330, 180, contraction ? "red" : "black");
67
68  ## *Engine*
69
70  ## Defining content of cylinder (steam and water).
71  boreColour is filling ? "purple" : "white";
72  bore is Rectangle(230, pistonHeight + 10, 40, 280 - pistonHeight, boreColour);
73  splash is contraction ? Rectangle(230, 285, 40, 5, "blue") : null;
74
75  ## Defining the piston and its chain..
76  pistonHeight is filling ? (280 - cycle) : (140 + cycle);
77  pistonChain is Line(250, 120, 250, pistonHeight);
78  piston is Rectangle(230, pistonHeight, 40, 10);
79
80  ## Valves in the engine.
81  steamValve is filling ? null : Rectangle(240, 290, 20, 10);
82  waterValve is contraction ? null : Rectangle(270, 280, 10, 10);
83
84  ## *Pivoting Beam*
85
86  ## 'theta' is the angle of the pivoting beam from the horizontal.
87  ## 'thet2' is 'theta + 180', included to make the definition of the
88  ##    pivoting beam (slightly) less verbose.
89  theta is ((pistonHeight - 240) * 360) / (r_2*2*pi);
90  thet2 is theta + 180;
91
92  ## Point about which beam pivots.
93  pivotX = 170;
94  pivotY = 120;
95
96  ## The beam itself, and the pivot joint.
97  beam is Polygon(
98      [pivotX + r_1*cos(theta + phi_1), pivotY + r_1*sin(theta + phi_1),
99
100     pivotX + r_2*cos(theta + phi_2), pivotY + r_2*sin(theta + phi_2),
101     pivotX + r_2*cos(theta + phi_3), pivotY + r_2*sin(theta + phi_3),
102     pivotX + r_2*cos(theta + phi_4), pivotY + r_2*sin(theta + phi_4),
103     pivotX + r_2*cos(theta + phi_5), pivotY + r_2*sin(theta + phi_5),
104     pivotX + r_2*cos(theta        ), pivotY + r_2*sin(theta        ),
105     pivotX + r_2*cos(theta - phi_5), pivotY + r_2*sin(theta - phi_5),
106     pivotX + r_2*cos(theta - phi_4), pivotY + r_2*sin(theta - phi_4),
107     pivotX + r_2*cos(theta - phi_3), pivotY + r_2*sin(theta - phi_3),
108     pivotX + r_2*cos(theta - phi_2), pivotY + r_2*sin(theta - phi_2),
109
```

```
110     pivotX + r_1*cos(theta - phi_1), pivotY + r_1*sin(theta - phi_1),
111     pivotX + r_1*cos(thet2 + phi_1), pivotY + r_1*sin(thet2 + phi_1),
112
113     pivotX + r_2*cos(thet2 + phi_2), pivotY + r_2*sin(thet2 + phi_2),
114     pivotX + r_2*cos(thet2 + phi_3), pivotY + r_2*sin(thet2 + phi_3),
115     pivotX + r_2*cos(thet2 + phi_4), pivotY + r_2*sin(thet2 + phi_4),
116     pivotX + r_2*cos(thet2 + phi_5), pivotY + r_2*sin(thet2 + phi_5),
117     pivotX + r_2*cos(thet2        ), pivotY + r_2*sin(thet2        ),
118     pivotX + r_2*cos(thet2 - phi_5), pivotY + r_2*sin(thet2 - phi_5),
119     pivotX + r_2*cos(thet2 - phi_4), pivotY + r_2*sin(thet2 - phi_4),
120     pivotX + r_2*cos(thet2 - phi_3), pivotY + r_2*sin(thet2 - phi_3),
121     pivotX + r_2*cos(thet2 - phi_2), pivotY + r_2*sin(thet2 - phi_2),
122
123     pivotX + r_1*cos(thet2 - phi_1), pivotY + r_1*sin(thet2 - phi_1)],
124     "grey"
125 );
126 pivotJoint = Circle(pivotX, pivotY, 5);
127
128 ## *Pump*
129
130 ## Defining the dead weight, and its chain.
131 deadWeightHeight is (270 - r_2*(theta/360)*2*pi);
132 deadWeightChain is Line(90, 120, 90, deadWeightHeight);
133 deadWeight is Rectangle(80, deadWeightHeight, 20, 80);
134
135 ## Valves in the pump.
136 inValve is filling ? Rectangle(33, 310, 14, 10) : Rectangle(33, 320, 14, 10);
137 inValveLeft  = Rectangle(30, 330, 5, 5);
138 inValveRight = Rectangle(45, 330, 5, 5);
139
140 outValve is contraction ? Rectangle(83, 390, 14, 10) : Rectangle(83, 400, 14, 10);
141 outValveLeft  = Rectangle(80, 410, 5, 5);
142 outValveRight = Rectangle(95, 410, 5, 5);
143
144 ## Arrows indicating water flow through pump.
145 inArrow is contraction ? Polygon([90, 440, 100, 450, 95, 450,
146     95, 460, 85, 460, 85, 450, 80, 450], "blue") : null;
147 outArrow is filling ? Polygon([40, 280, 50, 290, 45, 290,
148     45, 300, 35, 300, 35, 290, 30, 290], "blue") : null;
149
150 ## *** Static Observables ***
151
152 ## Lines used to provide outline of Cylinder, water and steam tanks.
153 c_1 = Line(230, 210, 230, 290);
154 c_2 = Line(230, 290, 240, 290);
155 c_3 = Line(240, 290, 240, 310);
```

```
156  c_4 = Line(240, 310, 210, 310);
157  c_5 = Line(210, 310, 210, 350);
158  c_6 = Line(210, 350, 290, 350);
159  c_7 = Line(290, 350, 290, 310);
160  c_8 = Line(290, 310, 260, 310);
161  c_9 = Line(260, 310, 260, 290);
162  c_10= Line(260, 290, 320, 290);
163  c_11= Line(320, 290, 320, 270);
164  c_12= Line(320, 270, 330, 270);
165  c_13= Line(330, 270, 330, 250);
166  c_14= Line(290, 250, 290, 270);
167  c_15= Line(290, 270, 300, 270);
168  c_16= Line(300, 270, 300, 280);
169  c_17= Line(300, 280, 270, 280);
170  c_18= Line(270, 280, 270, 210);
171
172  ## Lines used to provide outline of Displacement pump.
173  d_1 = Line(100, 310, 110, 310);
174  d_2 = Line(110, 310, 110, 390);
175  d_3 = Line(110, 390, 100, 390);
176  d_4 = Line(100, 390, 100, 430);
177  d_5 = Line(80,  430, 80,  390);
178  d_6 = Line(80,  390, 70,  390);
179  d_7 = Line(70,  390, 70,  370);
180  d_8 = Line(70,  370, 30,  370);
181  d_9 = Line(30,  370, 30,  310);
182  d_10= Line(50,  310, 50,  350);
183  d_11= Line(50,  350, 70,  350);
184  d_12= Line(70,  350, 70,  310);
185  d_13= Line(70,  310, 80,  310);
186
187  ## Rectangles illustrating Water.
188  w_1 = Rectangle(290, 260, 40, 10, "blue");
189  w_2 = Rectangle(300, 270, 20, 20, "blue");
190  w_3 = Rectangle(270, 280, 50, 10, "blue");
191
192  w_4 = Rectangle(210, 330, 80, 20, "blue");
193
194  w_5 = Rectangle(70, 310, 40, 80, "blue");
195  w_6 = Rectangle(80, 390, 20, 40, "blue");
196  w_7 = Rectangle(30, 350, 40, 20, "blue");
197  w_8 = Rectangle(30, 310, 20, 40, "blue");
198
199  ## Rectangles illustrating Steam.
200  s_1 = Rectangle(210, 310, 80, 20, "purple");
201  s_2 = Rectangle(240, 290, 20, 20, "purple");
```

```
202
203 ## Pentagon to indicate position of fire. 'Flickers' as cycle progresses.
204 fire is Polygon([250, 350 + (cycle % 7) + (cycle % 2),
205     270, 360, 260, 370, 240, 370, 230, 360], "red");
206
207 ## Adding elements to picture (order is important).
208 picture is [w_1, w_2, w_3, w_4, w_5, w_6, w_7, w_8,
209     s_1, s_2,
210     pistonChain, piston, deadWeightChain, deadWeight, fire, beam,
211     bore, splash, steamValve, waterValve,
212     inValve, inValveLeft, inValveRight, inArrow,
213     outValve, outValveLeft, outValveRight, outArrow,
214     pivotJoint,
215     c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9,
216     c_10, c_11, c_12, c_13, c_14, c_15, c_16, c_17, c_18,
217     d_1, d_2, d_3, d_4, d_5, d_6, d_7, d_8, d_9,
218     d_10, d_11, d_12, d_13,
219     runButton,
220     statusText1, statusText2, statusText3,
221     statusText4, statusText5, statusText6];
222
223 ## *** Automatic Movement ***
224
225 ## Finds current time using the JS Date.getTime() function.
226 ## Note that this gives time in milliseconds since 1st Jan, 1970.
227 func getTime
228 {
229     return ${{ new Date().getTime() }}$;
230 }
231
232 ## Moves the simulation forward one time step.
233 proc step
234 {
235     after(1)
236     {
237         ## Timing. Note that timeStep is converted into seconds.
238         timeStep = (getTime() - currTime) / 1000;
239         currTime = getTime();
240
241         cycle += timeStep * speed;
242         while(cycle >= 140) cycle -= 140;
243     }
244 }
245
246 ## Runs the simulation for a short period of time.
247 proc runSim : Run_clicked
```

```
{
    runActive = false;
    currTime = getTime();
    for(i = 0; i < 5000; i++)
    {
        step();
    }
    runActive = true;
}
```

6.8 Pendulum Mk II

```
## *** Pendulum MkII ***
##
## NOTE: The following script is written for the EMILE version of JS-EDEN.
## This may be found at:
##     http://jseden.dcs.warwick.ac.uk/emile/
##
## At the time of writing, this script WILL NOT RUN on the latest version
## of JS-EDEN, located at:
##     http://jseden.dcs.warwick.ac.uk/latest/
##
## The following script produces a simple simulation of a pendulum. The
## pendulum consists of a point mass attached to an inflexible light rod
## rotating about a smooth pivot.
##
## Run this script by copying and pasting the contents of this file into the
## EDEN interpreter window. The simulation parameters may then be adjusted
## using the sliders. To run the simulation for a short period of time, press
## the "Run" button. The "Reset" button moves the ball back to its starting
## position and sets its velocity to zero.

## Buttons to control simulation.
resetButton is Button("Reset", "Reset", 400, 270, true);
runActive = true;
runButton is Button("Run", "Run", 400, 310, runActive);

## *** Simulation Parameters ***

theta = 100;

pivotX = 200;
pivotY = 200;
rodLength = 100;
weightRadius = 10;
friction = 0.01;
```

```
35
36  mass = 10;
37  g = 10;
38
39  angularSpeed = 0;
40
41  ## *** Dynamic Observables ***
42
43  weightX is pivotX + rodLength * cos(theta);
44  weightY is pivotY + rodLength * sin(theta);
45
46  ## * Here the angular acceleration of the mass is determined. *
47  momentOfInertia is mass * rodLength;
48
49  ## 1. Finding normal vector along rod.
50  pivotNormX is (pivotX - weightX) / rodLength;
51  pivotNormY is (pivotY - weightY) / rodLength;
52
53  ## 2. Finding normal vector perpendicular to rod.
54  perpNormX is  pivotNormY;
55  perpNormY is -pivotNormX;
56
57  ## 3. Projecting gravitational and frictional forces along perpendicular norm.
58  projForce is (perpNormX * 0) + (perpNormY * mass * g);
59  frictionForce is -friction * angularSpeed;
60
61  ## 4. Finding angular acceleration.
62  torque is (projForce + frictionForce) * rodLength;
63  angularAccn is torque / momentOfInertia;
64
65  ## Shapes representing rod and weight
66  rod is Line(pivotX, pivotY, weightX, weightY);
67  weight is Circle(weightX, weightY, weightRadius, "yellow");
68
69  picture is [rod, weight, resetButton, runButton];
70
71  ## *** Simulation Procedures ***
72
73  ## Finds current time using the JS Date.getTime() function.
74  ## Note that this gives time in milliseconds since 1st Jan, 1970.
75  func getTime
76  {
77      return ${{ new Date().getTime() }}$;
78  }
79
80  ## Given an acceleration, the following proc performs one step
```

```
81  ## of a forward Euler method to numerically solve for the position
82  ## of the pendulum.
83  proc step
84  {
85          after(1)
86          {
87                  ## Timing. Note that timeStep is converted into seconds.
88                  timeStep = (getTime() - currTime) / 1000;
89                  currTime = getTime();
90
91                  ## Angular speed updated using acceleration.
92              angularSpeed += angularAccn * timeStep;
93
94              ## Position updated using speed.
95              theta += angularSpeed * timeStep;
96
97          }
98  }
99
100 ## Resets the pendulum to its initial position, at zero angular speed.
101 proc reset : Reset_clicked
102 {
103         theta = 100;
104         angularSpeed = 0;
105 }
106
107 ## Runs the simulation for a short period of time.
108 proc runSim : Run_clicked
109 {
110         runActive = false;
111         currTime = getTime();
112         for(i = 0; i < 50000; i++)
113         {
114                 step();
115         }
116         runActive = true;
117 }
```

6.9 Bouncing Ball

```
1   ## *** Bouncing Ball ***
2   ##
3   ## NOTE: The following script is written for the EMILE version of JS-EDEN.
4   ## This may be found at:
5   ##     http://jseden.dcs.warwick.ac.uk/emile/
6   ##
```

```
7  ## At the time of writing, this script WILL NOT RUN on the latest version
8  ## of JS-EDEN, located at:
9  ##     http://jseden.dcs.warwick.ac.uk/latest/
10 ##
11 ## The following script produces a simple simulation of a ball contained
12 ## in a box. A constant force acts on the ball, controlled by the user
13 ## via sliders.
14 ##
15 ## Run this script by copying and pasting the contents of this file into the
16 ## EDEN interpreter window. The simulation parameters may then be adjusted
17 ## using the sliders. To run the simulation for a short period of time, press
18 ## the "Run" button. The "Reset" button moves the ball back to its starting
19 ## position and sets its velocity to zero.
20
21 ## Sliders to control parameters.
22 ## Mass of ball (kilograms).
23 massSlider is Slider("Mass", 0.1, 1, 0.1, 1, 1, 400, 70);
24 mass is Mass_value;
25
26 ## Horizontal & vertical components of external force on ball (Newtons).
27 ## Note that a +ve vertical force acts downward.
28 xForceSlider is Slider("HorizontalForce", -20, 20, 1, 0, 1, 400, 110);
29 yForceSlider is Slider("VerticalForce", -20, 20, 1, 10, 1, 400, 150);
30 forceX is HorizontalForce_value;
31 forceY is VerticalForce_value;
32
33 ## Controls magnitude of friction on ball (Nsm^(-1)).
34 ## frictionalForce := velocity * friction * (-1)
35 frictionSlider is Slider("Friction", 0, 1, 0.1, 0.002, 1, 400, 190);
36 friction is Friction_value;
37
38 ## Coefficient of restitution between ball and the walls.
39 restitutionSlider is Slider("Restitution", 0, 1, 0.01, 1, 1, 400, 230);
40 restitution is Restitution_value;
41
42 ## Buttons to control simulation.
43 resetButton is Button("Reset", "Reset", 400, 270, true);
44 runActive = true;
45 runButton is Button("Run", "Run", 400, 310, runActive);
46
47 ## Speed of ball is initially set to zero.
48 speedX = 0;
49 speedY = 0;
50
51 ## Friction is modelled as a force opposite in direction
52 ##   and proportional in magnitude to the current speed.
```

```
53 frictionForceX is friction * (-speedX);
54 frictionForceY is friction * (-speedY);
55
56 ## Calculating net force on the ball.
57 netForceX is forceX + frictionForceX;
58 netForceY is forceY + frictionForceY;
59
60 ## Derived from f = ma.
61 accelerationX is netForceX / mass;
62 accelerationY is netForceY / mass;
63
64 x = 200;
65 y = 200;
66 r = 10;
67
68 ball is Circle(x, y, r, "orange");
69
70 ## Walls of box.
71 lineL is Line(100, 100, 100, 300);
72 lineR is Line(300, 100, 300, 300);
73 lineT is Line(100, 100, 300, 100);
74 lineB is Line(100, 300, 300, 300);
75
76 picture is [ball,
77         lineL, lineR, lineT, lineB,
78         massSlider, yForceSlider, xForceSlider,
79         frictionSlider, restitutionSlider,
80         resetButton, runButton];
81
82 ## Finds current time using the JS Date.getTime() function.
83 ## Note that this gives time in milliseconds since 1st Jan, 1970.
84 func getTime
85 {
86     return ${{ new Date().getTime() }}$;
87 }
88
89 ## Given an acceleration, the following proc performs one step
90 ## of a forward Euler method to numerically solve for the position
91 ## of the ball.
92 proc step
93 {
94         after(1)
95         {
96                 ## Timing. Note that timeStep is converted into seconds.
97                 timeStep = (getTime() - currTime) / 1000;
98                 currTime = getTime();
```

```
99
100                 ## Speed updated using acceleration.
101             speedX += accelerationX * timeStep;
102             speedY += accelerationY * timeStep;
103
104             ## Simulates bouncing off walls.
105             if(y > 300 - r)
106             {
107                     y = 300 - r;
108                     speedY = (-speedY) * restitution;
109             }
110             if(y < 100 + r)
111             {
112                     y = 100 + r;
113                     speedY = (-speedY) * restitution;
114             }
115             if(x > 300 - r)
116             {
117                     x = 300 - r;
118                     speedX = (-speedX) * restitution;
119             }
120             if(x < 100 + r)
121             {
122                     x = 100 + r;
123                     speedX = (-speedX) * restitution;
124             }
125
126             ## Position updated using speed.
127             x += speedX * timeStep;
128             y += speedY * timeStep;
129
130         }
131 }
132
133 ## Resets the ball to its initial position, at zero velocity.
134 proc reset : Reset_clicked
135 {
136         x = 200;
137         y = 200;
138         speedX = 0;
139         speedY = 0;
140 }
141
142 ## Runs the simulation for a short period of time.
143 proc runSim : Run_clicked
144 {
```

```
145         runActive = false;
146         currTime = getTime();
147         for(i = 0; i < 5000; i++)
148         {
149                 step();
150         }
151         runActive = true;
152 }
```

6.10 Attractive Forces

```
1  ## *** Attractive Forces ***
2  ##
3  ## NOTE: The following script is written for the EMILE version of JS-EDEN.
4  ## This may be found at:
5  ##     http://jseden.dcs.warwick.ac.uk/emile/
6  ##
7  ## At the time of writing, this script WILL NOT RUN on the latest version
8  ## of JS-EDEN, located at:
9  ##     http://jseden.dcs.warwick.ac.uk/latest/
10 ##
11 ## The following script produces a simple simulation of two ball contained
12 ## in a box. A constant force acts on the balls, controlled by the user
13 ## via sliders. An additional attractive force between the balls may be
14 ## applied by the user, and the balls can collide with one another and the
15 ## walls of the box.
16 ##
17 ## Run this script by copying and pasting the contents of this file into the
18 ## EDEN interpreter window. The simulation parameters may then be adjusted
19 ## using the sliders. To run the simulation for a short period of time, press
20 ## the "Run" button. The "Reset" button moves the ball back to its starting
21 ## position and sets its velocity to zero.
22
23 ## Sliders to control parameters.
24 ## Mass of ball (kilograms).
25 massSlider1 is Slider("Mass1", 0.1, 1, 0.1, 1, 1, 500, 70);
26 m1 is Mass1_value;
27 massSlider2 is Slider("Mass2", 0.1, 1, 0.1, 1, 1, 750, 70);
28 m2 is Mass2_value;
29
30 ## Horizontal & vertical components of external force on ball (Newtons).
31 ## Note that a +ve vertical force acts downward.
32 xForceSlider1 is Slider("HorizontalForce1", -20, 20, 1, 0, 1, 500, 110);
33 yForceSlider1 is Slider("VerticalForce1", -20, 20, 1, 0, 1, 500, 150);
34 forceX1 is HorizontalForce1_value;
35 forceY1 is VerticalForce1_value;
```

```
36  xForceSlider2 is Slider("HorizontalForce2", -20, 20, 1, 0, 1, 750, 110);
37  yForceSlider2 is Slider("VerticalForce2", -20, 20, 1, 0, 1, 750, 150);
38  forceX2 is HorizontalForce2_value;
39  forceY2 is VerticalForce2_value;
40
41  ## Controls magnitude of friction on ball (Nsm^(-1)).
42  ## frictionalForce := velocity * friction * (-1)
43  frictionSlider1 is Slider("Friction1", 0, 1, 0.1, 0, 1, 500, 190);
44  friction1 is Friction1_value;
45  frictionSlider2 is Slider("Friction2", 0, 1, 0.1, 0, 1, 750, 190);
46  friction2 is Friction2_value;
47
48  ## Coefficient of restitution between ball and the walls.
49  restitutionSlider1 is Slider("Restitution1", 0, 1, 0.01, 1, 1, 500, 230);
50  restitution1 is Restitution1_value;
51  restitutionSlider2 is Slider("Restitution2", 0, 1, 0.01, 1, 1, 750, 230);
52  restitution2 is Restitution2_value;
53
54  ## Magnitude of attractive forces.
55  attractionSlider is Slider("Attraction", -10, 10, 1, 5, 1, 750, 270);
56  attraction is Attraction_value * 1000;
57
58  ## Magnitude of attractive forces.
59  restitutionSlider is Slider("Restitution", 0, 1, 0.01, 1, 1, 750, 310);
60  restitution is Restitution_value;
61
62  ## Buttons to control simulation.
63  resetButton is Button("Reset", "Reset", 500, 270, true);
64  runActive = true;
65  runButton is Button("Run", "Run", 500, 310, runActive);
66
67  ## Speed of ball is initially set to zero.
68  speedX1 = 0;
69  speedY1 = 0;
70  speedX2 = 0;
71  speedY2 = 0;
72
73  x1 = 200;
74  y1 = 200;
75  r1 = 10;
76  x2 = 300;
77  y2 = 200;
78  r2 = 10;
79
80  ## Friction is modelled as a force opposite in direction
81  ##   and proportional in magnitude to the current speed.
```

```
82  frictionForceX1 is friction1 * (-speedX1);
83  frictionForceY1 is friction1 * (-speedY1);
84  frictionForceX2 is friction2 * (-speedX2);
85  frictionForceY2 is friction2 * (-speedY2);
86
87  ## Attractive force between balls.
88  distance is sqrt((x1 - x2)*(x1 - x2) + (y1 - y2)*(y1 - y2));
89  attractForceX1 is (attraction * (x2 - x1)) / (distance*distance*distance);
90  attractForceY1 is (attraction * (y2 - y1)) / (distance*distance*distance);
91  attractForceX2 is (attraction * (x1 - x2)) / (distance*distance*distance);
92  attractForceY2 is (attraction * (y1 - y2)) / (distance*distance*distance);
93
94  ## Calculating net force on the ball.
95  netForceX1 is forceX1 + frictionForceX1 + attractForceX1;
96  netForceY1 is forceY1 + frictionForceY1 + attractForceY1;
97  netForceX2 is forceX2 + frictionForceX2 + attractForceX2;
98  netForceY2 is forceY2 + frictionForceY2 + attractForceY2;
99
100 ## Derived from f = ma.
101 accelerationX1 is netForceX1 / m1;
102 accelerationY1 is netForceY1 / m1;
103 accelerationX2 is netForceX2 / m2;
104 accelerationY2 is netForceY2 / m2;
105
106 ball1 is Circle(x1, y1, r1, "orange");
107 ball2 is Circle(x2, y2, r2, "green");
108
109 ## Walls of box.
110 lineL is Line(100, 100, 100, 400);
111 lineR is Line(400, 100, 400, 400);
112 lineT is Line(100, 100, 400, 100);
113 lineB is Line(100, 400, 400, 400);
114
115
116 picture is [ball1, ball2,
117         lineL, lineR, lineT, lineB,
118         massSlider1, yForceSlider1, xForceSlider1,
119         frictionSlider1, restitutionSlider1,
120         massSlider2, yForceSlider2, xForceSlider2,
121         frictionSlider2, restitutionSlider2,
122         attractionSlider, restitutionSlider,
123         resetButton, runButton];
124
125 ## Finds current time using the JS Date.getTime() function.
126 ## Note that this gives time in milliseconds since 1st Jan, 1970.
127 func getTime
```

```
128  {
129      return ${{ new Date().getTime() }}$;
130  }
131
132  ## Given an acceleration, the following proc performs one step
133  ## of a forward Euler method to numerically solve for the position
134  ## of the ball.
135  proc step
136  {
137          after(1)
138          {
139                  ## Timing. Note that timeStep is converted into seconds.
140                  timeStep = (getTime() - currTime) / 1000;
141                  currTime = getTime();
142
143                  ## Speed updated using acceleration.
144              speedX1 += accelerationX1 * timeStep;
145              speedY1 += accelerationY1 * timeStep;
146              speedX2 += accelerationX2 * timeStep;
147              speedY2 += accelerationY2 * timeStep;
148
149              ## Balls bouncing off each other.
150              if(distance < r1 + r2)
151              {
152                      ## Finding normal vector for line of impact.
153                      lineOfImpactX = (x2 - x1) / distance;
154                      lineOfImpactY = (y2 - y1) / distance;
155
156                      ## Removing intersection between balls.
157                      intersection = distance - (r1 + r2);
158                      x1 -= lineOfImpactX * (intersection / 2);
159                      y1 -= lineOfImpactY * (intersection / 2);
160                      x2 += lineOfImpactX * (intersection / 2);
161                      y2 += lineOfImpactY * (intersection / 2);
162
163                      ## Finding projected impact speeds along line of impact.
164                      u1 = lineOfImpactX * speedX1 + lineOfImpactY * speedY1;
165                      u2 = lineOfImpactX * speedX2 + lineOfImpactY * speedY2;
166
167                      ## Finding post-collision speed along line of impact.
168                      ## See Appendix 6.5 of the report for an explanation
169                      ## of these expressions.
170                      v1 = (u1*m1 + u2*m2 + (m1 * restitution)*(u2 - u1)) / (m1 + m2);
171                      v2 = (u1*m1 + u2*m2 + (m2 * restitution)*(u1 - u2)) / (m1 + m2);
172
173              ## Finding normal vector perpendicular to line of impact.
```

```
174             perpToImpactX = lineOfImpactY;
175             perpToImpactY = -lineOfImpactX;
176
177             ## Finding projected impact speeds perpendicular to impact.
178                     p1 = perpToImpactX * speedX1 + perpToImpactY * speedY1;
179                     p2 = perpToImpactX * speedX2 + perpToImpactY * speedY2;
180
181                     ## Finding post-collision velocities.
182                     speedX1 = lineOfImpactX * v1 + perpToImpactX * p1;
183                     speedY1 = lineOfImpactY * v1 + perpToImpactY * p1;
184                     speedX2 = lineOfImpactX * v2 + perpToImpactX * p2;
185                     speedY2 = lineOfImpactY * v2 + perpToImpactY * p2;
186             }
187
188             ## Balls bouncing off walls.
189             if(y1 > 400 - r1)
190             {
191                     y1 = 400 - r1;
192                     speedY1 = (-speedY1) * restitution1;
193             }
194             if(y1 < 100 + r1)
195             {
196                     y1 = 100 + r1;
197                     speedY1 = (-speedY1) * restitution1;
198             }
199             if(x1 > 400 - r1)
200             {
201                     x1 = 400 - r1;
202                     speedX1 = (-speedX1) * restitution1;
203             }
204             if(x1 < 100 + r1)
205             {
206                     x1 = 100 + r1;
207                     speedX1 = (-speedX1) * restitution1;
208             }
209
210             if(y2 > 400 - r2)
211             {
212                     y2 = 400 - r2;
213                     speedY2 = (-speedY2) * restitution2;
214             }
215             if(y2 < 100 + r2)
216             {
217                     y2 = 100 + r2;
218                     speedY2 = (-speedY2) * restitution2;
219             }
```

```
220             if(x2 > 400 - r2)
221             {
222                     x2 = 400 - r2;
223                     speedX2 = (-speedX2) * restitution2;
224             }
225             if(x2 < 100 + r2)
226             {
227                     x2 = 100 + r2;
228                     speedX2 = (-speedX2) * restitution2;
229             }
230
231             ## Position updated using speed.
232             x1 += speedX1 * timeStep;
233             y1 += speedY1 * timeStep;
234             x2 += speedX2 * timeStep;
235             y2 += speedY2 * timeStep;
236         }
237 }
238
239 ## Resets the ball to its initial position, at zero velocity.
240 proc reset : Reset_clicked
241 {
242         x1 = 200;
243         y1 = 200;
244         x2 = 300;
245         y2 = 200;
246
247         speedX1 = 0;
248         speedY1 = 0;
249         speedX2 = 0;
250         speedY2 = 0;
251 }
252
253 ## Runs the simulation for a short period of time.
254 proc runSim : Run_clicked
255 {
256         runActive = false;
257         currTime = getTime();
258         for(i = 0; i < 5000; i++)
259         {
260                 step();
261         }
262         runActive = true;
263 }
```

6.11 Pendulum Clock

```
## *** Pendulum Clock ***
##
## NOTE: The following script is written for the EMILE version of JS-EDEN.
## This may be found at:
##     http://jseden.dcs.warwick.ac.uk/emile/
##
## At the time of writing, this script WILL NOT RUN on the latest version
## of JS-EDEN, located at:
##     http://jseden.dcs.warwick.ac.uk/latest/
##
## The following script produces a simple simulation of a pendulum clock. The
## pendulum consists of a point mass attached to an inflexible light rod
## rotating about a smooth pivot. The clock is
##
## Run this script by copying and pasting the contents of this file into the
## EDEN interpreter window. The simulation parameters may then be adjusted
## using the sliders. To run the simulation for a short period of time, press
## the "Run" button. The "Reset" button moves the ball back to its starting
## position and sets its velocity to zero.

## Buttons to control simulation.
resetButton is Button("Reset", "Reset", 400, 270, true);
runActive = true;
runButton is Button("Run", "Run", 400, 310, runActive);

## *** Simulation Parameters ***

theta = 95;

pivotX = 200;
pivotY = 200;
rodLength = 100;
weightRadius = 10;
friction = 0;

mass = 10;
g = 10;

angularSpeed = 0;

clockCentreX = 400;
clockCentreY = 200;

## *** Dynamic Observables ***
```

```
45
46 weightX is pivotX + rodLength * cos(theta);
47 weightY is pivotY + rodLength * sin(theta);
48
49 ## * Here the angular acceleration of the mass is determined. *
50 momentOfInertia is mass * rodLength;
51
52 ## 1. Finding normal vector along rod.
53 pivotNormX is (pivotX - weightX) / rodLength;
54 pivotNormY is (pivotY - weightY) / rodLength;
55
56 ## 2. Finding normal vector perpendicular to rod.
57 perpNormX is  pivotNormY;
58 perpNormY is -pivotNormX;
59
60 ## 3. Projecting gravitational and frictional forces along perpendicular norm.
61 projForce is (perpNormX * 0) + (perpNormY * mass * g);
62 frictionForce is -friction * angularSpeed;
63
64 ## 4. Finding angular acceleration.
65 torque is (projForce + frictionForce) * rodLength;
66 angularAccn is torque / momentOfInertia;
67
68 ## Setting clock hand angles.
69 secondAngle = 0;
70 minuteAngle is secondAngle / 60;
71 hourAngle   is minuteAngle / 60;
72
73 ## Shapes representing rod and weight.
74 rod is Line(pivotX, pivotY, weightX, weightY);
75 weight is Circle(weightX, weightY, weightRadius, "yellow");
76
77 ## Clock face.
78 clockFace is Circle(clockCentreX, clockCentreY, 50, "white");
79 secondHand is Line(clockCentreX, clockCentreY,
80         clockCentreX + 50*cos(270 + secondAngle), clockCentreY + 50*sin(270 + secondAngle),
81 minuteHand is Line(clockCentreX, clockCentreY,
82         clockCentreX + 40*cos(270 + minuteAngle), clockCentreY + 40*sin(270 + minuteAngle),
83 hourHand   is Line(clockCentreX, clockCentreY,
84         clockCentreX + 30*cos(270 +   hourAngle), clockCentreY + 30*sin(270 +   hourAngle),
85
86 picture is [rod, weight, resetButton, runButton,
87     clockFace,
88         secondHand, minuteHand, hourHand];
89
90 ## *** Simulation Procedures ***
```

```
91
92  ## Finds current time using the JS Date.getTime() function.
93  ## Note that this gives time in milliseconds since 1st Jan, 1970.
94  func getTime
95  {
96      return ${{ new Date().getTime() }}$;
97  }
98
99  ## Given an acceleration, the following proc performs one step
100 ## of a forward Euler method to numerically solve for the position
101 ## of the pendulum.
102 proc step
103 {
104         after(1)
105         {
106                 startTheta = theta;
107
108                 ## Timing. Note that timeStep is converted into seconds.
109                 timeStep = (getTime() - currTime) / 1000;
110                 currTime = getTime();
111
112                 ## Angular speed updated using acceleration.
113             angularSpeed += angularAccn * timeStep;
114
115             ## Position updated using speed.
116             theta += angularSpeed * timeStep;
117
118             if((startTheta < 90 && theta > 90) ||
119                     (theta < 90 && startTheta > 90))
120                             secondAngle += 6;
121         }
122 }
123
124 ## Resets the pendulum to its initial position, at zero angular speed.
125 proc reset : Reset_clicked
126 {
127         theta = 100;
128         angularSpeed = 0;
129 }
130
131 ## Runs the simulation for a short period of time.
132 proc runSim : Run_clicked
133 {
134         runActive = false;
135         currTime = getTime();
136         for(i = 0; i < 50000; i++)
```

```
137         {
138                 step();
139         }
140         runActive = true;
141 }
```