**University of Bath**

**PHD**

# A modular physics methodology for games

Schanda, Florian

*Award date:*
2012

*Awarding institution:*
University of Bath

# A Modular Physics Methodology for Games

submitted by

Florian Schanda

for the degree of Ph.D.

of the

University of Bath

Department of Computer Science

2012

# Summary

Currently, games with rich environments allowing a wide range of possible interactions and supporting a large number of physical simulations make use of a large number of scripts and bespoke physical simulations, adapted to fit the needs of the game.

This thesis proposes a methodology that can be used to tie together various different physical simulations, both off-the-shelf and bespoke, such as rigid body physics, electrical and magnetic simulations to give something greater than the sum of the individual parts. We present a notation for designing the overall physical simulation and a means for the different parts to interact.

Experiments using an implementation of the methodology containing electricity, rigid body simulation, magnetics (including electro-magnetics), buoyancy and sound show that it is possible to model everyday objects such an electric motor or a doorbell. These object work 'as expected', without the need for special scripts and new, originally unexpected, interactions are possible without further modification of the experiment setup.

# Acknowledgements


- First, and foremost, for many reasons, my supervisor *Phil Willis*. He had unending patience and never did give up on me, even if I was really, *really* late. He also patiently listened to all my crazy ideas and helped me distill something useful out of them, he always had a positive thought, helpful insight or an encouraging word ready when it was needed. Thank you Phil.

- My examiners, *Steven Pettifer* (external) and *Joanna Bryson* (internal) for an interesting, intense and productive viva voce.

- My proof-readers: *Lucy Perry* (all), my father *Friedrich Schanda* (all), *Dalia Khader* (parts) and of course *Phil Willis* (all).

- I would like to thank *Eamonn O'Neill* and *Angela Cobban*. You may have forgotten what you did, but I did not.

- My comrades in arms, *Martin Brain*[1] and *Jessica Jones*. Without their support, nagging and distraction (with random side-projects) in the later years of this project I would have gone insane and this work would not have concluded.

- My mother *Ursula "Aïda" Schanda* and my father for their write-up stories that made me feel a lot better for not doing as much work as I should (apparently scrubbing leaves of a rubber plant is more entertaining than writing up a PhD) and offering a lot of support.

- *Dalia Khader* for being a good friend, great help with tutoring and for an entertaining motivational bet on who will finish writing up first, which I lost by more than a year. (Addendum: a lot more than a year now.)

- Two of my undergraduate lecturers which significantly shaped and influenced my understanding of computer science: *John Fitch* and *James Davenport*.

- Praxis for giving me 10 out of 15 days taken as holiday in order to finish writing up for free and then a 3 months sabbatical. In particular I would like to thank *Anna Mascetti* for suggesting and organising the free 10 days for me - thank you; it was appreciated!

- *Carina Murman* who helped a lot with tutoring and the occasional lecturing and general sanity as I could rant to somebody sympathetic about the absence of mixing taps in the UK.

- Comrades *Carl*, *Jan* and *Hagen* with whom I shared a house with in the second and third year of this project.

- Comrades *Cat* and *Emma* for being awesome friends. Thanks - many a good rant was had.


[1] Who just lost the game.

- Comrade *Doug* for running a pretty epic SR campaign. Also you were the first person I knew of roughly my age finishing a PhD; proving that it could be done.
- *Richard Stallman* for pointing out the evils of proprietary software and for creating the GPL. Please never stop what you are doing.
- Debian GNU/Linux, the GNU Project and the Linux Kernel. Everything produced and used in this project was based on free software.
- Comrade *Martin* for destroying the word "Furthermore" for me.
- Comrade computers *florian*, and later *axiom*, for hosting the repository of my work. In particular *florian* that survived being short-circuited with a 5p coin. Despite the magic smoke escaping, you continued to work to this day. You were with me since 2001.
- *Joanna Bryson* for providing me with my first free laptop, on the condition that I acknowledge her in my thesis. So there ;) (Back then it was not known that you were my internal examiner.)

# Contents

# Chapter 1

# Introduction

This dissertation will develop a methodology for a unified physics simulation intended for, but not necessarily limited to, games. The contribution of this dissertation is a methodology that provides a framework for integrating various existing physical simulations into a combined simulation, focusing in particular on describing the interactions between the different physical simulations. The hypothesis underlying this dissertation is:

> For game scenarios where unexpected solutions to a problem by the player are both encouraged and deemed to be a good thing and for simulations where a large number of physical simulations are necessary, using the methodology presented in this dissertation reduces the number of scripts required to model such a rich environment.

This dissertation also introduces a notation that offers a concise way of describing, or indeed specifying, each instance of each component of the methodology in an implementation.

## 1.1 Motivation

In computer games that involve physics beyond basic rigid body simulation and collision detection, everything outside basic dynamics is usually solved by scripting. Most user to world or world to world interactions are scripted special cases. For instance, a light switch turning on the light in a particular room will require a script to do just that. The player shooting at a light bulb and thus destroying it will require a script to correctly turn off the light in the 3D engine. A motion sensor or light barrier will also require a script to work correctly. Opening a tap to fill a glass of water will also require at least one script.

The objective is to do away with (most) of these scripts. Objects should behave in a natural way; the level designer can focus on building a world that just works, mainly due to emergent behaviour, as opposed to building a specific world around a particular solution the game designer wishes the player to figure out.

For instance, imagine a laboratory door that is held shut with a magnetic lock. A simple switch on the inside of the lab will allow the door to be opened.

The intended way to open it is to disengage the magnetic lock for a brief moment using the switch. Currently, such a functionality would be implemented with a script on the switch, which if activated, allows the door to be opened. This solution does prevent, or more specifically does not explicitly implement, various other creative solutions to opening the door.

For example, the player might want to cooperate with another player to push open the door; together they might be strong enough to overcome the force of the magnetic lock. Or the player might be more clever and try and find a lever and use the basic principle of leverage to pry open the door. Or the player might wish to remove the main fuse from the building and thus disrupt the power supply to the electro-magnet holding the door shut.

Currently, the level designer would have to account for each of these solutions by writing another script. The proposed methodology will allow the level designer to build the world in as much detail as wished (most likely still envisioning one or two preferred solutions to the problem) and then the player can either go along with the intended plot or they can find their own individual and potentially unexpected solution to the problem. For instance, the player might discover that blowing up the power plant for the city is also a valid, if extreme, solution to open our laboratory door.

An idea fundamental to this is that *this is how the reality works*. A game world that behaves mostly like the real world because it is constructed like it and adheres to most obvious physical rules can inspire the player to pursue more interesting solutions as opposed to the obvious and planned solution. Usually the player comes up with an idea based on their experience with the real world and most games impose rather arbitrary limits on what can be achieved (usually because no script anticipated the action); this disappointment is something a implementation of the proposed methodology can help to avoid.

It is assumed that creating one whole unified solution would be not realistic; thus the methodology is based on combining many specialised physical simulations and making them work together in order to provide an overall more complete picture. This also has the advantage that one part can be replaced by another depending on the needs of the application.

The potential freedom for the player, depending on how thoroughly the world is constructed, is the key motivation for the development of the methodology.

## 1.2 Document overview

This section describes how the rest of this thesis is structured.

- Chapter 2, *Physical simulations and games*, highlights some of the current approaches to physics simulation in games and virtual reality environments identifying the issues and challenges faced.
- Chapter 3, *Goals, assumptions and constraints*, illustrates the motivations of this work in the context of the previous chapter and formulate the main goals of the project.

- Chapter 4, *The methodology: concept and design*, provides an informal introduction to the methodology and outlines some design decisions.
- Chapter 5, *The methodology: definition*, then defines the four core components of the methodology more formally.
- Chapter 6, *The methodology: implementation, experiments and results*, describes experiments used to validate the hypothesis. It examines the implementation of the methodology that was created during the development of the methodology.

  The focus of this implementation is to be able to conduct a few experiments and demonstrate that the overall concept is functional. The chapter contains a few notes on each physics simulation implemented and illustrations of four experiments.
- Chapter 7, *Discussions and future work*, identifies work in three areas: future research expanding on the concepts introduced here, future research concerning physics simulations and notes on how a much better proof of concept implementation can be written.
- We conclude in Chapter 8.
- In Appendix A, *Electricity*, a graph theory based algorithm is developed that can be used to model some aspects of electricity in a game context. An implementation of this algorithm provides the electrical simulation that is used in most of the experiments presented later in this dissertation.

# Chapter 2

# Physical simulations and games

## 2.1 Overview of Game Types

The methodology introduced in this dissertation will be more useful for certain types of games. It is unfortunately difficult to present a complete classification system for games, assign each game a category and then state that for one set of these the methodology is more relevant. The main problem is that the field of computer games is vast and a large number of computer games have been created so far. Some of the broader categories, or genres, of games can be listed as follows:

- *Action* – These are games where reaction speed of the player is important. As the genre title suggests, fast paced action is more important than tactical or strategic thinking, leading some to refer to these games as "twitch" games. Classic examples of this are the original Pong [Ata72] game or a first-person shooter such as Quake3 [id99].

- *Adventure/Puzzle* – These games do not require fast reaction speeds, but instead focus on solving problems. For adventure games this usually is navigating a character through a world, applying various tools found earlier in the game to progress. Iconic examples of this genre are the Indiana Jones [Luc89] games and Monkey Island [Luc90]. The precursor to these graphical adventure games were the text-adventure games; one of which can even be found in the GNU Emacs text editor [Sch83].

  In more 'pure' puzzle games, the problem is more important than any story or plot, a good example is the original Sokoban [Ima82] game. Many puzzle games do not include any plot at all, such as the Sudoku game, which is originally a pen-and-paper game found in newspapers, and The Incredible Machine [Sie92].

- *Role-Playing Game (RPG)* – The distinguishing feature of these games is a particular game-play mechanic. The player controls one or more characters and during the course of the game accumulates "experience points" (XP). Once a certain amount of XP has been gained, the player character "levels up", i.e. its capabilities in-game can be improved.

Story and plot are usually a central aspect for these games and the player may have considerable control over how the story progresses and ends. Notable examples of this genre include Nethack [ea89], Planescape Torment [AB99] and World of Warcraft [Ent04].

- *Simulation* – A simulation game usually features two of the following: complex game mechanics, detailed in-game economics and a high degree of realism. Flight simulators are simulation games, but this genre covers a variety of styles and also includes games such as SimCity [WM89].

- *Strategy* – Somewhat related to simulation games, these games emphasise strategic problem solving and planning. Strategy games can be roughly split into two styles, real-time and turn-based. In Real-Time Strategy (RTS) games the action develops in real-time and simultaneously (for all players) and decisions have to be made quickly. The RTS game that defined the genre was Dune II [Wes92], but many more have been created since then following the same basic formula.

  The most popular format for turn-based strategy games are the *4X* games. *4X* stands for e**x**plore, e**x**pand, e**x**ploit and e**x**terminate. Players take turns and are generally not restricted in the amount of time they can spend planning each move. These games are similar to many board games, such as Risk. Two notable genre-defining examples are Sid Meyer's Civilisations [Mic91] and the Master of Orion [Sim93] series.

- *Sport* – Sports games are also related to simulation games, but they are usually action driven and focus on the simulation of various sports. A good example is the golf game series Tiger Woods PGA Tour [Ele98].

  A recent development in the genre of sports games came with the Nintendo Wii console which to some extent physically simulated the sport. The best known example today is perhaps Wii Fit [Nin07].

The above six categories however are not mutually exclusive. Many games could equally fit into more than one of them. In fact the combination of action games with puzzle solving already forms its own sub-genre of Action-Adventure games. A good example of this sub-genre is the Metroid [Nin86] series in which the exploration of the world is the main objective. Games such as System Shock [Loo94] and DeusEx [Ion00] blend even more genres; both combine action, puzzle and RPG elements.

The games for which the methodology will be the most useful are games that feature detailed physical simulation and puzzle solving, as the variety of physical simulation and the ability use unconventional but sensible solutions are core features that the methodology offers. For this reason, primarily games that fall into these categories will be examined in this Chapter.

## 2.2 Introduction to physical simulation in games

Many computer games, in particular 3D action games, contain some element of physical simulation. Often this is just means to an end; basic things like "you cannot walk through a wall"

are necessary to support the game mechanics. Other games apply their physics simulation to other game objects as well resulting in the ability of the player to (for example) push boxes around or cause objects to fly away if an explosion is set off nearby.

The evolution of rigid body simulation in games can be observed when looking at the various titles id Software released. Doom [id93] did not feature much more than collision detection. Quake [id96] included vertical movement and basic effects such as an explosion pushing the player around. Doom 3 [id04] finally applied this simulation to most smaller objects (such as canned drinks or small crates) causing an explosion let off in a store room to knock over most objects.

Other action games have also followed this trend. For example the title Half-Life 2 [Val04] included a 'gravity gun' , which allowed the player to pick up and hurl objects around. The important part was that the gravity gun worked within the physical rules of the rest of the world.

A different genre of games making use of physics simulations are puzzle games. Perhaps the most illustrative example of this is the Sierra classic 'The Incredible Machine' [Sie92] . This game provided the player with various basic building blocks ranging from common objects like bowling balls and conveyor belts to the more bizarre such as a jack-in-a-box or monkeys on bicycles. Each object acted in a predictable fashion. For example the bowling ball followed standard gravity and fell down, but it was heavier than a tennis ball which became relevant if they were both placed on a see-saw. The player was then presented with a series of levels, which had some objective, such as get all tennis balls into the bucket, and was given a small number of items to place into the world. Once the items were placed the player started the simulation. If the goal was eventually fulfilled, the game progressed to the next level. If not, the world was reset and the player could adjust the positioning or combination of the placed items and try again.

The Incredible Machine was revolutionary in that there were usually many ways to achieve an objective and part of the charm of the game was to find unconventional ones.

Other physical simulations have also occasionally appeared in games. For example the Thief [Loo98] series based an important aspect of its game-play on sound: certain types of floor produced more sound when walked over, which in turn increased the chance that a guards would notice the player. Items were included that could mitigate (moss to cover particularly noisy ground such as metal) or take advantage of this ('magic' sound emitters attached to arrows that could be shot into a useful location to draw the guards away to investigate allowing the player to move past them unnoticed).

Another example of (non-rigid body) physical simulations used for game-play is System Shock II [LI99] which featured very basic electricity: certain doors needed power to be opened so the player had to obtain a 'power cell', find a suitable 'recharge station' before being able to operate the door. Although the power cell was nothing more than a glorified 'key-card', the basic idea was reasonable and once introduced into the game world it was intuitive to solve similar problems within System Shock II that required portable power sources.

The game Portal [Val07] demonstrates, in a unique way, that unexpected emergent behaviour

is a good property. The original "Source" game engine, also developed by Valve Corporation originally for Half-Life 2 [Val04], was found to have such a flexible physics engine (Havok [Hav00]) that, with only a small amount of effort, the concept of 'portals' could be implemented. The player was given the 'portal gun', which could create blue and orange portals. They were limited to creating a single blue and a single orange portal at a time; opening another would close the old one. Entering one portal would cause one to emerge from the other, preserving momentum. The game was based on presenting the player with various scenarios to escape from and it was not always obvious where the portals had to be placed and there was often more than one solution. This in turn gave rise to competitions on how to escape from each particular level with the least number of portals. In the case of the game Portal, the flexible underlying physics engine gave rise to an entirely new product. This innovative re-use of an existing system to do something new and originally unexpected is precisely what the methodology proposed addresses and encourages.

The remainder of this chapter will look at current solutions for the rigid body simulation problem (including collision detection), briefly discuss various available solutions, unified physics engines (containing more than one distinct simulation), dynamic game environments (which are necessary to support the more interesting physical simulations), level of detail concepts and other physical simulations besides rigid body simulations. We conclude by identifying some of the problems in the scenarios examined in the above introduction and some of the approaches below that will provide the motivation for the proposed methodology.

## 2.3 Rigid body simulation in games

Rigid Body Simulation (RBS) and collision detection is a well understood problem and an active area of research. As this work is not strictly focused on how the underlying rigid body simulation works, but rather uses it as an important component, this review will provide an overview of common approaches to rigid body simulation. The various implementations can be split up into roughly three main groups, pointed out below and described in detail in Section 2.3.1, 2.3.2 and 2.3.3.

- 2D rigid body simulations
- 3D rigid body simulations (which generally can also be constrained to act as a 2D simulation)
- Dedicated collision detection

The term *Dynamics simulation* will be used throughout this thesis and refers to a combination of rigid body simulation and collision detection. Such a simulation can possess some self-explanatory properties such as run-time performance or scalability. A particular quality that is useful for both rigid body simulations and collision detection is *stability*. This does not refer to the absence of run-time exceptions but to the simulation not 'exploding'. Since floating

point arithmetic is prone to introducing small errors that can accumulate quickly into something noticeable, good dynamics engines apply various dampening forces in order to eliminate such errors making the simulation more stable.

### 2.3.1 2D rigid body simulation engines

Although the majority of dynamics simulations are geared towards 3D rigid body simulation, there are a few dynamics projects that focus purely on a 2D simulation. As basic physics simulations are becoming more and more a standard feature in computer games, these physics engines are especially useful for games on platforms that do not support computationally expensive 3D graphics such as flash games or games for mobile phones.

One such engine is Box2D [Cat]. It supports many features and bindings for many languages. It is also available as a fixed point arithmetic version for platforms that do not support floating point arithmetic or where floating point operations are exceedingly slow.

Chipmunk [Lem] is a similar engine loosely based on concepts found in Box2D. It has a slightly different feature set but as it is newer has much less documentation available.

Both of these engines are available free of charge and under a free software license. Although most 3D engines can be adapted and constrained to provide simulations in 2D the above engines are attractive because they are much faster, more memory efficient and generally much more stable than their 3D counterparts. For example modelling a suspended chain attached to two poles is a common benchmark for the stability of rigid body simulation engines and many of the 3D engines mentioned below struggle with this. However both Box2D and Chipmunk can accurately model this scenario.

### 2.3.2 3D rigid body simulation engines

Perhaps the most widely used free software rigid body simulation is Open Dynamics Engine (ODE) [StOCa]. It is a stable engine and it has already appeared in a few commercial titles as well. It has a broad feature set, good documentation and an active community; for these reasons it has been chosen as the main rigid body simulation engine to use throughout this work.

Another popular open source rigid body simulation engine is Bullet Physics [Cou]. It is considered to be stable and generally has a larger feature set than ODE including support for deformable soft bodies and cloth simulation. It has been used in a number of games but it is perhaps most famous for being integrated as the physics simulator of Blender [TB98] an open source 3D modelling and animation package.

Another library worth mentioning is Tokamak [Lam]. Its special feature is that it allows the dynamic fragmentation of objects on collision.

Finally OpenTissue [oCSotUoC] from the University of Copenhagen is a understated physics library with a vast number of features including two different approaches to fluid dynamics.

### 2.3.3 Dedicated collision detection

All of the above rigid body simulation engines also feature a collision detection framework; without collisions rigid body simulation is rather boring as objects would pass straight through each other. However there also exist a small number of external collision detection engines. A prominent open source example of such a library is GImpact [LtGC]. It is purely a collision detection library and as such is used by a number of other projects; most notably by Bullet Physics and ODE (as an optional replacement for the built-in engine).

The engines discussed so far always give an as accurate as possible solution in any given time-step. However that is not always desirable and there is some work that explores the concept of stochastic collision detection. For instance Gabriel Zachmann explores in his work [KZ03a, KZ03b] a collision detection approach where quality of results can be traded against the performance of collision queries. The idea is to give generally believable collision detection results, which are not necessarily completely correct, but do appear to be correct *enough*. Of particular note is that simulation accuracy can often be adjusted dynamically depending on how much CPU time is available in any given frame.

## 2.4 Notable commercial physics engines

There are also a few commercial 3D physics simulations available. For the domain of games there are two main products.

Havok [Hav00] is now distributed by Intel and it enjoys a large customer base and good reputation. It supports a large number of features attractive for game development such as cloth simulation and breakable objects. Recently a non-commercial free of charge version has been released but only for the Windows platform and source code is not available.

Another interesting solution is PhysX [AN]. Formerly known as Novodex it was bought by AGEIA who implemented it on top of a custom hardware solution. Due to a classic chicken and egg problem the technology never took off, as it was impossible to see what you were missing out on if you did not have access to the Peripheral Component Interconnect (PCI) extension board, and thus there was little incentive for developers to support it or gamers to buy it. Since then it has been bought by NVIDIA who are now aiming to implement it in top of their Compute Unified Device Architecture (CUDA) [NVI07] technology. (CUDA is a framework for performing computations on a Graphics Processing Unit (GPU). It is ideal for problems that are easily solved in parallel.) This approach shows more promise as most people playing games are likely to already have access to such hardware and the results can still be seen, albeit much more slowly due to CPU emulation, if they do not have the required hardware. PhysX supports a number of very interesting features most notably a liquid body simulation based on particles, Smoothed Particle Hydrodynamics (SPH). An example of this has recently been provided by NVIDIA [Cor08] which is based on techniques from the paper by Jos Stam on stable fluids [Sta99].

More generally, CUDA and its 'standard' cousin OpenCL [AK08] are an interesting ap-

proach that may be useful for other computationally expensive physical simulations apart from dynamics. CUDA is tailored specifically to NVIDIA's hardware, OpenCL intends to present a more platform independent approach.

## 2.5 Unified physics engines

There have been a few attempts at a 'meta' physics engine; frameworks that abstract away the implementation details of the various rigid body simulation and collision detection engines to provide a uniform Application Programming Interface (API).

For example Open Physics Abstraction Layer (OPAL) [FRS] started as such a high level interface to physics programming but at the time of writing it only supports ODE as a back-end. However it does support some features on top of ODE such as per-shape material settings and breakable joints. (The concept of a *joint* in rigid body simulation is defined in Section 2.7.1 below.)

An independent work with a similar name, PAL [Boe], also aims to provide a uniform interface and some extra features on top of many different physics engines. Currently it supports more than ten different underlying physics engines but unfortunately it is not available for any platform other than Windows.

## 2.6 Other interesting physical simulations

This section will look at three areas: electricity, procedural sound and liquids. These are physical simulations that do not commonly appear in games but could conceivably be useful for game-play mechanics and puzzle solving. They could also be considered good 'special effects', which contribute to the general immersiveness and predictability of the game world.

If they do appear they are usually scripted events or otherwise hard-coded and not the 'real thing'. Each of these simulations are actively researched in engineering or special effects contexts but the focus is usually not real-time simulation. Each subsection will describe how they are traditionally approached in games and highlight interesting current approaches that could enhance or substitute for how they are implemented in games; providing some idea what could be possible and how more common scripts could be eliminated.

### 2.6.1 Sound

#### In games

Most games produce some kind of sound, usually both background music and some sound effects, however this is all there is to it; both the music and the foreground sounds are basically just studio recorded sound clips played at appropriate times. For example when a gun is fired or a door is opened.

Usually games support stereo sound. Thus if a car is passing from left to right in front of the observer the sound should also pan from the left to the right channel. A more general

solution to this is full 3D sound, which can place a sound anywhere in 3D space and produce a more realistic experience. This requires certain hardware such as a 5.1 speaker system and the obvious downside of this is that such a setup is rather bulky and is certainly not portable. The OpenAL [StOcb] library provides an easy to use and widely used approach to 3D positional audio in games.

Some games have also featured some environmental audio effects. For example Unreal [ED98] divided up its game world into 'zones', each of which could be assigned one of a number of preset environmental audio effects such as echo and reverberation. It should be noted that these effects were assigned by the game designer and not automatically determined.

### Approaches to procedural audio

Sounds do not necessarily have to be sampled they can be generated and filtered on the fly. A brief overview is given for three areas that can be of interest to games.

**Positional audio with HRTF** A Head Related Transfer Function (HRTF) is an approach to provide 3D positional audio by just using ordinary headphones [Beg94]. They are unique to every person and basically describe how sound transmits from a point on your skull to each ear. Their main shortcomings are that they require some calibration to adjust for a particular person and are expensive to derive and use.

There are effectively three methods to simplify the HRTF to make it suitable for use in real-time audio [KMKK08, KM07]. Experiments seem to show that subjects can still locate a sound in 3D space with these simplifications applied and in certain cases the simplified versions seemed to provide better results than the original HRTF [Rav08].

**Environment** A different area of research is the procedural generation of environmental sound effects from the world geometry. Thus, given a set of materials and some geometry, a filter can be calculated that may provide an echo, hall or other such reverberation effects. To use an analogy it is effectively radiosity but for sound.

A different example of procedural audio is the Doppler effect. An approaching sound source will be perceived to be higher pitched than one that moves away from the observer. This effect is common in games and OpenAL directly supports this.

An example of such work is the effort to provide a complete virtual reality simulation of the Phillips Pavillion [LVF+09]. This work saw the the incredible acoustics from the building simulated allowing the viewer to look around and re-experience the performance of PoèmeÉlectronique. The reverberation of the building was calculated in an expensive pre-processing step; several gigabytes of data was generated as even head movement was taken into account.

**Sounds from collisions** Perhaps the most interesting aspect of sound synthesis for games is to model the sounds resulting from object collisions. Given the collision data generated by

rigid body simulation and some information about the materials of the objects it is possible to create a variety of sounds.

A framework fast enough for use in real-time simulations has been described by Doel and Pai [vdDP98]. A good demonstration of this is given by the same author in the form of a model of a rock in a wok [vdDKP01]. When the wok is tilted the rock rolls across its surface and produces a reasonably realistic sound. The impact sounds are pre-computed but procedural; the simulation itself runs in real time.

This work has spawned a library called JASS [vdD], implemented in Java. Another library working on the same principle is Phya [Men] but it is unfortunately only available for Windows.

A recent improvement to modelling sounds from collision data is proposed by Raghuvanshi and Lin [RL06]. It runs fast enough for use in real-time simulations and can deal with many objects simultaneously making the approach more attractive for games and the methodology proposed in particular. What makes their approach particularly attractive from the point of view of our methodology is that accuracy of the simulation is directly influenced by how much time is available each frame. One of their optimisations is to perform the sound equivalent of visibility culling: sounds of a frequency that would be mostly masked by another are not generated in the first place.

### 2.6.2 Electricity

#### In games

Electricity (or its effects) often features in computer games but usually in a minor role. For example a common use of electricity is a light that can be switched on and off by using a switch - except that in games there is usually no concept at all of 'electricity' - the light switch is simply connected to a script that in turn toggles an on/off flag on some light entity.

This use of 'electricity' appeared early in computer games; The Incredible Machine [Sie92] included a couple of objects that consumed and produced electricity and the first person shooter Duke Nuke'em 3D [3D 96] included light switches, which turned the lights in a room on or off, although it should be noted that the game did not feature a light model as such: the level designer had to create two versions of a room; one with surfaces lit and one with darker surfaces, toggling the light switch caused the game to swap which one of them was currently displayed.

System Shock II [LI99] featured a different set of electrical equipment: doors that required power to operate and rechargeable laser guns.

The original Quake [id96] featured a 'lightning gun' that had the interesting side effect of discharging all of its ammunition at once if used underwater, usually with fatal consequences for the user and anybody unfortunate enough to be close by. Again, this special effect was hard coded into the game logic.

More recently Half-Life 2 [Val04] featured some puzzles that were based around electricity, similar to the ones System Shock II had already done. The interesting part was more subtle: the game world included a few TVs that could be turned off by 'using' them (i.e. walking up

to them and pressing a special 'use' key, usually the space bar). However the player could also grab the TV with their gravity gun and move it away from its initial position causing the power cord to be ripped from the wall. A third, more violent, approach was to simply shoot the TV to turn it off. As above, all of these had to be anticipated by the game designers and scripts had to be written to support all three approaches.

One interesting case of non-scripted electrical simulation can be seen in the game Minecraft [Per10] and it illustrates one of the central points of this thesis well: a simple physical system, as long as it is consistent, gives raise to a large number of unforeseen interactions. In Minecraft it is possible to lay down "redstone" (which is effectively a wire) and electricity can pass down it for 15 tiles at most. It is also possible to place "redstone torches", these act as power generators with a twist: if power is supplied to them from another source they toggle on/off. Although the intention of the game designer was to allow players to wire up doors and set off TNT, some clever people have devised a way to construct all useful logic gates (and, or, xor, nand, not) and even clocks. Some have even constructed complex electronic circuitry, for example a 12x5 dot matrix display showing some scrolling text.

#### SPICE3: the traditional approach

Roughly speaking, there are two areas of electricity simulation: analogue circuit simulations and logic circuit simulations. Although there is a good deal of overlap between the two, in the context of this thesis the distinction is a useful one to make.

The traditional and definitive approach to electrical simulation is SPICE3 [otUoCaB] and its variants. It is an iterative solver; given a description of a circuit in a specialised language it will arrive at an answer after some time. The objective of SPICE3 is not speed but accuracy making it an ideal tool for electronic engineers. SPICE3 can model more or less any circuit, including logic circuits, although more specialised flavours of spice such as XSPICE [oEEotGIoT] and NGSPICE [NtNd] contain solutions to tackle this particular problem more efficiently.

### 2.6.3 Computational fluid dynamics

#### In games

Many games have included water and other fluids in some way. Larger bodies of water the player can swim or drown in are usually modelled as a static area (or volume in 3D games) designated as such. These areas are essentially 'flat' although often the illusion of a real body of water is created by applying a shader to it showing small waves or ripples in the surface. (A shader is a program that can be used to change the default rendering pipeline on modern graphics cards; they can be used to create special effects such as bump-maps or water ripples.)

This is the de facto method of creating large bodies of fluid in 3D games; the main improvements over time have been to apply more sophisticated shaders to the surface. Other dynamics based effects such as buoyancy have been added in more recent games.

Another common effect based on liquids are blood spatters. They are usually modelled by either 2D decals (sprites), which are drawn onto walls, or some kind of particle system or

both. Although a drop of blood and an ocean are fundamentally the same, two very different approaches are used to model them. Scale and the desired player interaction dictates which approach to use.

### Current approaches

Computational Fluid Dynamics (CFD) is a very difficult and expensive problem. Research in this area mirrors what is seen in games to some extent: different simulations exist for different scales and scenarios of the problem. General solutions tend to be *very* slow, run-time is usually measured in minutes or even hours per frame. This is perfectly acceptable for movies or in engineering contexts, but not for games.

Originally 2D approaches were used (2D because they effectively create a simple height map, which is then applied to a mesh) to simulate waves or splashes resulting from drops of liquid. There are currently three popular approaches to model open fluids in 3D. Firstly, the original approaches, which are based on solving the Navier-Stokes (NS) equations [KM90]. Secondly, SPH [Sta99], which are methods based on modelling fluids with particle systems. Finally, a more recent approach revolves around the Lattice-Boltzmann Method (LBM), which is an approach based on cellular automaton.

Generally the NS and LBM approaches measure their run-time in minutes and are therefore not suitable for real time applications such as games. There are also simplifications to the NS equations for shallow water that have been implemented in hardware [HHL$^+$05] and although they achieved a performance increase of about a factor of 15 to 30 the run-time was still measured in minutes.

These simulations are usually concerned with a body of liquid confined to some volume; it is a different problem to interact with the body of water or to 'direct' it [FF01].

The SPH approach [Sta99] however can yield interactive frame rates in certain scenarios and can easily be implemented partly in hardware thanks to modern graphics cards and the ability to run almost arbitrary programs on them. As mentioned previously recent NVIDIA cards also offer some capability of physical modelling and include facilities to run SPH simulations [Cor08].

The above simulations are general and yield good and accurate results in many scenarios; their only issue with respect to games is their speed (or lack thereof).

There are however more special case simulations that either make some sort of assumption about the problem or impose some limitations on what can be modelled. Thuerey has proposed solutions to a number of interesting special cases such as bubbles and foam [TSS$^+$07] or modelling breaking waves in a 2D shallow water simulation [TMFSG07]. These special case solutions are based on a 2D surface simulation but allow certain 3D open fluid effects whilst being fast enough for real-time animation. This is discussed in more detail in Section 2.7.1.

## 2.7 Dynamic worlds

Usually 3D game worlds are largely static; effectively a detailed 3D wallpaper. Only a few designated objects in it can move and change. This is largely due to the efficient data structures that can be used for rendering such static worlds; for example a Binary Space Partition (BSP) and octrees. However these data structures are not intended for accommodating change to the world and are even worse if this change happens every frame (for example as seen with a falling box or a deformable object such as a table cloth).

The two main reasons these data structures are useful for quick rasterisation (or rendering) of a scene are:

- Back to front sorting of triangles (making a depth buffer unnecessary).
- Fast visibility queries, allowing efficient culling. (Visibility queries are useful to avoid rendering geometry that cannot be seen in the first place.)

Traditionally, a complex and mostly static data structure is necessary for this but Scott Saulters demonstrates a GPU based approach to the problem [Sau04]. It is based on partitioning the world into simple cubes and then querying the number of pixels potentially rendered by using a particular (and now common) OpenGL extension. This in turn can be used to determine the visibility of each world cell. This approach is shown to be more or less equivalent to Quake 3's [id99] own visibility algorithm with the advantage of allowing for a completely dynamic world. (Quake 3's visibility approach requires an expensive off-line pre-computation pass over a game world, often lasting several minutes.)

Other games, in particular non-first person games, often use different data-structures for rendering the current scene. For example in so-called 2.5D games where game-play is 2D (for example a side-scroller), but the world is still rendered using a 3D graphics engine, a much more dynamic world can be presented. A recent and very relevant (for more than just its graphics) example is Little Big Planet [Mol08]. This game, in a way similar to The Incredible Machine, allows the user to create complex levels and objects out of simple building blocks. These components behave according to the materials they are made of. One of the main ideas behind Little Big Planet is this freedom; sharing your creations with others is a central aspect of the game and is further evidence that high replay-ability and ability to customise due to consistent and powerful physical simulation can be a good thing for game.

Another interesting example of a game featuring a dynamic world is Minecraft [Per10], already previously mentioned for its interesting electricity simulation. In Minecraft the entire world can be de-constructed and re-arranged during game-play, in fact this forms the central game-play element of Minecraft. The player can collect and mine various materials, for example dirt and sand using shovels, stone and iron using pick-axes and wood using axes. These materials can then be re-used to build traps, fortresses and any other object limited by only the players creativity. While day-time is peaceful, at night monsters will spawn all over the level making. Although difficult, it is possible to fight them directly; however it is much more efficient to

build defensive structures during day to help survival at night. Although the graphics are rather simplistic, this allows for a completely dynamic world.

### 2.7.1 Level of detail

The concept of Level of Detail (LOD) is widely used in computer graphics. It refers to the technique of rendering a less accurate or less complex version of a model or a scene the further away from it the viewer is; the reasoning is that the simpler model is faster to draw and it is impossible to distinguish the difference at long distances. A simple everyday example would be mipmaps in texture mapping (although they primarily serve to reduce aliasing artifacts) or using a lower resolution mesh for a height-mapped terrain. There are some challenges involved, mainly when to use LOD and how to shift from one type of rendering to another. Ideally this transition is seamless but 'popping' artifacts (when switching from one representation to another) are still commonplace.

The concept of LOD also applies to physical simulations, for which we shall use the term Level of Simulation Detail (LOSD). It refers to the use of a different, more specialised, simulation approach within a more general simulation. This different approach could have a number of properties depending on the circumstance. It could be a more complex, faster or more stable approach.

A common example of this is 'joints' in rigid body simulation engines. For instance, a doorway could be modelled by connecting a door to the door frame with modelled hinges. This would be a low-level model and it would be time consuming both to model in the first place and to simulate, as collision detection is expensive. A common improvement in this case is to still model the door, door frame and hinges but not perform any collision detection within the hinge model. Instead a *joint* is added that describes a mathematical relationship between two objects in the scene. Normally two objects in 3D have six degrees of freedom (translation in x, y and z and rotation about the x, y and z axis). In the case of a joint modelling the behaviour of a door hinge, the relative movement between the door and the door frame is restricted to only one degree of freedom: rotation around the y axis.

Other common examples of such joints are ball bearing joints (position is fixed but rotation is not) and slider joints (movement is constrained to a single axis).

Various physics engines also have other special purpose joints. For example Box2D [Cat] provides gear joints and a special joint to model rope pulleys.

The concept of LOSD is by no means restricted to rigid body simulation but is also applicable to other physical simulations such as CFD. As mentioned above, Thuerey explores this concept in his work: for example he demonstrates a faster approach to model complex interactions with a liquid by blending between a 2D surface model and a full 3D model of the fluid at a certain boundary away from a point of interest [TRS06].

Thuerey also explores a real time model of bubbles in a body of liquid [TSS+07]. Although the overall result could be achieved with a complex and computationally expensive 3D fluid simulation, the approach achieves real time performance by using three separate and simple

simulations and blending seamlessly from one into another. They are a shallow water simulation for the (relatively) small waves the bubbles cause, an SPH based simulation for the foam and a simulation for the upwards movement of the bubbles taking into account the vortices caused by the bubbles.

As a third example, normal 2D based shallow water simulations cannot model wave breaking but are a lot faster than full 3D models. It is possible to detect a breaking wave and by adding particles and some extra geometry on the fly to give a convincing illusion of a breaking wave [TMFSG07]. In this instance a normal simulation is augmented to give the appearance of a more sophisticated simulation.

In the more general case the problem of determining when to switch from one simulation to another or when to use a more restricted version of a simulation is a challenging problem. It is further explored in Section 7.1.5.

## 2.8 Other related work and concepts

### 2.8.1 Virtual Manufacturing at Bath

The Virtual Manufacturing project [WBTB93, BTBW95] undertaken at the University of Bath was an early and successful attempt to model a variety of interactions of a physical nature in a virtual world. A virtual workshop was provided and could be used to cut and process metal in a number of ways, with the ultimate aim of creating a more complex component. This required dynamic updates to the model of the object from operating on it with virtual tools.

It should be noted that the initial inspiration for the LOSD approach came from follow-up work of the above, which used a variety of standard parts to yield more than one new (but physically plausible) behaviour [Wil04]. These parts consisted of cogs that could be arranged in various ways and, when turned, the correct rotations resulted. The cogs were made from one of two materials, only one of which was conductive. Thus, electrical paths could be created with the cogs at the same time, with the correct mechanical and electrical behaviour emerging. However the construction area was only a fixed grid and all gear wheels were of identical size.

### 2.8.2 Physics Sandbox

A project that is worth mentioning is Phun [Ern]. It is an interesting '2D physics sandbox'. It consists of a custom rigid body simulation engine, a collision detection framework and a simple SPH based fluid simulation. Its main focus is to make it easy to construct simple worlds and experiment with physical simulation more or less in the spirit of the Incredible Machine but with 'more real' physics. Like the proof of concept implementation presented in this thesis it is restricted to 2D as this has a number of benefits: it is easier to interact with the world as a user, easier mathematics and much easier visualisation.

### 2.8.3 Pipelines

A completely separate concept that has inspired our design to some extent is that of UNIX and in particular shell pipelines. The philosophy of 'do one thing, but do it well' is a good one (which compares to the plug-in architecture). The ability to then combine these tools in mostly any order via pipelines is what gives the shell its flexibility (which compares roughly to the methodology's model of interactions).

### 2.8.4 Presence and Engagement

The two terms, presence and engagement, are often used when describing a virtual reality environment.

- *Presence* – will group the closely related terms "suspension of disbelief" or "absorption" [TA74]. This describes the illusion created by the game, the feeling of "being there". A break in presence (BIP) occurs whenever the player is reminded of the fact that everything is just a game [SS00]. BIPs can occur in-game, i.e. "why can I not move *this* box, but all the others"; or can be caused by outside factors, i.e. somebody asking you to turn the volume down. The concept of presence is relevant to this dissertation.
- *Engagement* – can be defined as the players willingness to play and do well in a game. This concept is often mentioned along with presence, but is not relevant to this dissertation.

Having a physics model that does not allow a variety of interactions between the player and the world to take place with can cause BIPs. More generally, behaviour that is unexpected or simply missing will cause BIPs as it reminds the player of the inadequacies and limitations of the physical simulation used.

### 2.8.5 Notation

The notation introduced in Chapter 5 borrows some concepts and its general 'look' from the formal specification language Z [Spi89].

## 2.9 Conclusion

Evidently there exist a large number of dynamics engines, both commercial and as free software. Their availability and quality suggests that re-inventing this particular wheel is not really a worthwhile endeavour. The modularity of the proposed methodology has been inspired by this as the option to swap out different underlying simulations is a valuable one.

The areas of the other simulations in the context of computer games are not nearly as well established as dynamics however, as mentioned earlier in this chapter, many new and exciting approaches have appeared recently. This is true in particular for fluids and sound. The area of electricity is not explored as well as it could be in the context of games. This motivated

the development of an electricity (and simplified magnetics in order to model electromagnets) simulation for this work.

Providing an arbitrary wrapper library that contains (usually) dynamics and one or two other simulations (usually some variant on SPH for liquids and some specialised subset for dynamics such as cloth simulation) is as such not a hard problem, which in turn perhaps explains why there is little research on the topic. It should be noted that wrapper libraries and more feature rich integrated solutions, such as Havok, can be of use to somebody using the methodology as they can make implementing its components much easier.

It is also important to distinguish the proposed methodology from such a wrapper library; the focus of the methodology is to provide an architecture in which different simulations can work together and terminology to express and describe these interactions. It is not a mere summation of its component parts. It is a consequence of this that an application of the methodology will yield a framework which does fit under the banner of 'wrapper library' or 'unified physics engine', but it is not the primary objective. What sets this framework apart from other meta libraries discussed earlier in this chapter is its concept of interactions.

Another common theme amongst the games mentioned earlier is that much of their physics beyond rigid body simulation is scripted. A key motivation for this work was to reduce scripting in particular the repetitive scripting, and also to reduce the amount of forethought required allowing the designer to focus on building the world and not worry too much about 'what if' scenarios.

Games with more than one ending or more than one preset path are usually very enjoyable and popular and are played well past their time as they offer something new every time. A key benefit of successful use of the proposed methodology should help creating such a game partly by allowing unconventional approaches from the player and partly by strongly encouraging the game world to be constructed in a way that allows such unconventional thinking.

## 2.10 Addendum

After the submission of this dissertation, a paper composed of subsets of Chapters 5 (The methodology: definition) and 6 (The methodology: implementation, experiments and results) has been published in the Workshop on Virtual Reality Interaction and Physical Simulation VRIPHYS (2010) [SW10].

# Chapter 3

# Goals, assumptions and constraints

This chapter will describe a hypothetical game scenario in Section 3.1 that will be used as an example in the subsequent sections. The high level goals of the methodology will be described in Section 3.2. The list of assumptions is given in Section 3.3. We conclude in Section 3.4 with some challenges to be faced and a number of desirable properties for the methodology.

## 3.1 An example scenario

The scenario used throughout this chapter is as follows: the objective is to gain access to a secure facility behind a fence. The obvious path is to open a gate set into the fence, however a power failure prevents using any of the gate-moving machinery.

**Intended solution** The solution as set out by the game designers is to identify the cause of the power failure, restore power and then operate the machinery to open the gate. The problem causing the power failure is a section of damaged wiring from the main generator, which is located nearby, to the gate. A suitable wire to replace the damaged part may be obtained by searching nearby houses.

**Consequences** Restoring power to the gate also has secondary consequences; some of the outer security systems of the facility are also powered from this generator. Restoring power to the gate will also activate parts of the security system.

**Alternative solutions** Here is a small selection of alternative solutions, assuming the world is built in such a way as to support them:

- Find a different metal object (a piece of the chain-link fence or a metal rod) in order to patch over the damaged part of the wire.

- Find a portable power generator and restore power using that.
- Find a large truck and drive it through the gate, braking it open.
- Find enough debris and build a ladder in order to scale the fence.

## 3.2 Goals of the methodology

The two primary goals of the methodology should be identified first; they are closely related to one another. All other goals either ensure the primary goals are met or are a direct consequence of this. The primary goals are:

1. An implementation of the methodology should both encourage the design of and make it easier to create games that can be solved in more than one way and allow unconventional approaches by the player.
2. The methodology should provide a generic way of describing interactions between different physics simulations.

The overall approach should be flexible enough to allow most of what was discussed in the previous chapter to be potentially integrated. The methodology must be general enough and make few assumptions about what kind of physics can be used. This includes physics that are not based on reality such as 'magical' or 'sci-fi' effects. A good example of this would be the previously mentioned gravity gun.

The methodology should also not be tied to any particular implementation, allowing more than one physics engine to be used to solve a particular problem. When implemented correctly this means that the different underlying physics simulations can be potentially swapped out. Furthermore the methodology should not imply a *particular* implementation. Using the methodology must not dictate how a simulation operates and should not prevent problem-specific shortcuts and optimisations to be used.

The methodology should also be a good sandbox for writing new, original physics simulations. In particular simulations that are dependent on other physics simulations (such as procedural sound generation) would benefit from this.

## 3.3 Assumptions for the methodology

The motivation and justification of the methodology depends on three key assumptions, which are described in this section.

### 3.3.1 Number of solutions and script complexity

Interactions with objects within the game world are generally limited to simple dynamics simulation and whatever the world creator anticipated and hard-coded with custom scripts. The

number of possible interactions with the game world scales roughly linearly with the number of scripts written.

Going back to the previous example from Section 3.1, most of the alternative solutions require both some forethought and a specific script (or scripts) written to support them. Of particular note is the 'solution' of driving a truck through the gate: when this is chosen it is likely that the player has not bothered to restore the power; this means the security system is not on-line. In a conventional game restoring the power is a prerequisite of encountering the security system; so from a game designer's view the security system would always be 'on' as it deemed to be impossible to walk to it without restoring the power.

The player could of course make use of this even if the power was restored as intended. Once the gate is opened it could be jammed preventing it from being shut again; the player could then back track and undo the repair, leaving the gate open and the exterior security system disabled. This is another solution that would have to be anticipated and catered for with a script. If the game world 'just works' as proposed then this solution also 'just works' without any extra effort from the world designer.

Generally, it assumed that having a large number of solutions to a problem is a desirable property for a game, as it has a positive impact on game-play. In particular, players will be encouraged to re-visit and re-play the game multiple times in order to try out different solutions. This in turn will keep the game 'alive' longer. (Although not related to this thesis, one could further argue that this is allows the commercial exploitation of game to go on for longer, be it via ad-support or extra add-on content, which can be released over time - with the game being actively played, the interest in it remains high for longer.)

A large number of solutions in turn conventionally requires a large number of scripts. Furthermore, the more these possible solutions have inter-dependencies, the more complex these scripts become. This is not unlike standard combinatoric explosions where having one more variable will double the effort required.

*The key assumption is that catering for many solutions with scripts quickly becomes expensive and relies heavily on the wisdom of the world designer to effectively prune and simplify.*

### 3.3.2 Different world building

Another assumption related to the previous one is that constructing worlds for use with the methodology will not cost more time than writing a multitude of scripts required for the same amount of interactions. The more attention the level designer pays to creating a physically plausible world, the more interactions with the game world will be possible.

This is also the primary trade-off of using the methodology: creating a physically plausible world that is detailed enough will likely require more time than creating a conventional static world and adding a few scripts. The idea is however that as one adds more and more scripts to make the world more 'alive' the point will be reached where building the world differently will be less effort.

*In essence the world building assumption states that after a certain point it will be easier*

*to create the world 'correctly' using the proposed methodology rather than making everything a special case.*

As an aside, creating a good world creation tool-set that allows efficient construction of worlds in this ways will be an interesting challenge.

### 3.3.3 No conflicting assumptions

This is not an assumption related to the hypothesis or methodology, rather it is an assumption placed on different physical simulations used in it. In other words, it is an issue of the implementer of an instance of the methodology to be aware of.

If any one simulation makes an assumption about something outside (but related to) its problem domain then special care will have to be taken to avoid conflicting assumptions.

Imagine a generic rigid body simulation with a constant acceleration due to gravity of $9.81ms^{-2}$ and a simulation for buoyancy that also assumes a value of $9.81ms^{-2}$. This is an example of a potentially conflicting assumption. It becomes an actual problem once the game environment moves to Mars or the Earth's moon and (by mistake) only one of the values is adjusted. Another possibility is that one of the simulations does not allow this value to be changed easily (it could be a hard-coded constant).

Thus one requirement of the methodology on its parts is that any simulation used should be free of assumptions outside its problem domain or if they exist, they must be configurable.

*This assumption is that most physics engines are general enough to operate independently from one another and have a sensible API.*

## 3.4 Challenges and desirable properties

### 3.4.1 Static worlds

In conventional 3D games the world around the player is more or less a 3D wallpaper. It is usually very rich in detail and 'aesthetically pleasing'. It can be viewed from different angles and positions (depending on player movement). However it is static and be cannot easily changed.

Such a static world does have some advantages. Most importantly for games, a static world can take advantage of clever data structures that allow the game engine to rasterise polygons more quickly and efficiently. Two popular data structures for this purpose are a BSP and octrees. Both of them have the advantage of allowing quick, back-to-front rendering of the scene. However both of them have a hard time dealing with changes to the world. In most cases a BSP has to be completely rebuilt while octrees suffer somewhat less.

Thus games generally divide the game world into two parts: a static world, also sometimes called a 'level', and a few dynamic actors, which represent everything non-static in the game world. Players, bullets, explosions, smoke and bottles are examples of objects that are usually not part of the static game world.

A world built with the methodology in mind should take the opposite approach. Most objects should be first class objects and not static scenery. Objects that traditionally would

have been scripted such as the gate from Section 3.1 should be 'built' so that they work as expected.

### 3.4.2 Predictability and consistency

Physics should be *predictable*, meaning that reasonable interactions should be possible and work as expected. In our example scenario the wire to obtain to operate the gates would conventionally be hard-coded into the system. It can be frustrating for the player when other plausible conductors that happen to be lying around (for example a metal rod or a piece of wire mesh fence) do not work; this invariably causes BIPs and can have an effect on player engagement as well.

A concrete example of a lack of predictability is Doom3 [id04] where it is possible to move boxes by either pushing them around or otherwise applying a force to them, for example by shooting at them. Thus it is reasonable to expect that shooting a desk lamp would also move it. However as this object was not special cased it cannot be moved by any means. Any such omission on the part of the world designer quickly destroys the player's suspension of disbelief particularly if the omission is inconsistent with others. A game world would appear more consistent if no small objects could be moved as opposed to a select few.

In summary a believable game world is one consistent within itself. It is also important that any plausible interaction is possible.

### 3.4.3 Identifying special cases

Certain physical simulations are special cases of a more general simulation or, more commonly, can be expressed in terms of another simulation. Buoyancy and explosions are two examples of physics that are usually integrated into the base dynamics simulation and would benefit from being abstracted away.

In this case both buoyancy and explosions manifest themselves as an extra force acting upon an object in the rigid body simulation. In the methodology these would be modelled as separate simulations and they use interactions to influence the other simulations. The challenge is to recognise such special cases and to efficiently encapsulate them; the benefit of doing so is that they are available even if a different dynamics simulation is used.

## 3.5 Constraints

The methodology proposed is intended to fill a particular niche, but it will not replace all scripting in games. Scripting due to plot and to carry the game's narrative forwards will remain in place, it is not clear to the author what, if anything, could ever replace this. An example of this could be the demolition of a building: the structure, fuses and charges could all be modelled using the methodology, but the decision of the bad guy to press the detonator at a most inconvenient time will be script driven.

The niche this methodology aims to fill is to make games that have a rich environment easier to write as these games currently require a large number of scripts and foresight of their designers. Games without such a detailed environment will derive much less use from this methodology since the assumption about script complexity (see 3.3.1) will not hold.

## 3.6 Summary

To summarise, the overall goal of allowing multiple solutions to a problem will be achieved by a combination of:

- A new approach to constructing the game world.
- Making more objects first class objects and generally allowing a more dynamic world.
- A large set of physics/physics interactions.

The overall goal of consistency will be achieved by reducing the number of special-case scripts. If everything follows the same set of rules the game world will be more predictable.

The next chapter will describe an approach to simulating simplified electricity for games. The motivations, goals and assumptions relevant to this approach are all contained within that chapter and are separate from the goals of the methodology. The reason for describing the electricity simulation in detail before describing the methodology is that the electricity simulation is often used as an example and many experiments discussed later are based on it.

The methodology itself is then developed in Chapters 4 to 6, drawing upon what was discussed in this chapter and Chapter 2.

# Chapter 4

# The methodology: concept and design

This chapter serves as an overview of the final design of the methodology given in Chapter 5; it describes both intended use cases and motivations for the particular of each component of the methodology.

## 4.1 Outline

The proposed methodology will partition the entirety of physical simulation into four parts. Properties are independent and passive components that simply store data used by the other parts. The other three components are the resolvers, derived property sets and interactions, all of which will be described in greater detail in the next few sections. Overall, the design has been heavily influenced by the following two desired aspects.

- **Modular** Any solution should support the implementation of various different kinds of physics simulations in a modular way, so that it is possible to write 'plug-ins' for various common simulations and build a library of commonly used parts. Dynamics simulation (rigid body simulation and collision detection) is a problem set for that many solutions exist, all with individual strengths and weaknesses. Three well known examples are: ODE [StOCa] (a free software package, originally developed by Russel Smith), Havok [Hav00] (a commercial solution, originally provided by Havok and now by Intel) and PhysX [AN] (a hardware based solution, originally provided by AGEIA and now by NVIDIA). The modularity intended to make it possible to write interchangeable plug-ins using such existing solutions.
- **Generic interactions** Tying separate simulations together in a generic way is very important and useful; it works towards the overall goal of having an environment that 'just works'. This particular idea motivated the creation of the vital 'interaction' part of the methodology.

Before going into further detail, two physics simulations, which will be used as the main examples when developing and describing the parts of the methodology, will be examined.

The most common and useful simulation for many games is dynamics. A dynamics simulation usually considers a number of objects and simulates the movement of these objects under an arbitrary set of forces. The shape of an object is modelled by either a mathematical expression (such as the equation for a sphere) or by a triangular mesh. Dynamics also involves collision detection, so that objects do not intersect with each other. Unless the scene is set in free space, gravity is also required so that objects will fall to the floor, rest on a table, etc. Some other forces that may be relevant for games are wind or the shock wave of an explosion.

The second simulation will be a simplified electricity simulation, the specifics of which are detailed in Appendix A.

In the spirit of modularity, these two simulations, dynamics and electricity, can be split into a number of separate components. This will not only highlight some similarities and shared characteristics between the two, but also illustrates some of the reasons why the methodology was split into four parts.

### 4.1.1 Splitting up dynamics

Some aspects of a comprehensive dynamics simulation as outlined above can be cleanly separated into independent physics simulations. For example most dynamics simulations provide a simplified gravity model: a force proportional to an object's mass is applied to every object in the scene. This gravity model can be implemented separately to the core dynamics simulation, as it is simply an application of the same kind of force to all objects in the game world. This abstraction of gravity carries with it a number of benefits:

- The same gravity simulation can be used for different dynamics back-ends.
- The simple gravity simulation can be easily replaced by a more complex gravity simulation, possibly one with different goals and assumptions.

For instance, a game set in outer space would have no use for a constant gravity vector, but could require a more involved gravity model for stellar objects. Other phenomena like the shock wave of an explosion and wind can also be abstracted away into dedicated self-contained simulations. Like gravity, they are only an application of certain forces to objects and perform most of their work by interacting with the rigid body simulation.

Once these are abstracted away, the problem becomes a much purer one as it is only concerned with objects in a free space and a set of forces acting on these objects. Where these forces come from and what they represent is completely irrelevant.

The next step is to separate object collisions from the rest of dynamics. When two solid objects move towards each other they should clearly not intersect. At its core, rigid body simulation is only concerned with point masses and has no concept of the material or shape of an object. Depending on the dynamics package used, object intersections and points of contact are either calculated in a separate step or integrated in a single opaque simulation step. When

abstracted into a separate step, collision detection provides a list of contact forces (sometimes also known as contact joints). When applied, these forces will move all objects apart just enough to be touching instead of intersecting on the next simulation frame. This concept is generally known as 'reactive collision detection' and will cause objects to appear to collide, but not intersect.

Recognising that collision detection is ultimately just a set of forces and that calculating this set of forces can be done independently from the rest of dynamics simulation allows the two to be separated in an identical fashion to that seen above with explosions, gravity and wind.

To summarise, the big problem of dynamics can be split up into the following smaller problems:

- Rigid body simulation: the point mass simulation and resolution of forces.
- Collision detection: generation of contact forces.
- Simulation of gravity: generation of forces to be applied to each object.
- Simulation of wind, explosions, etc.: generation of forces to be applied to the relevant objects.

### 4.1.2 Re-factoring of the electricity simulation

It now considered how an electricity simulation can be split into parts. An electricity simulation for games would be designed to model obvious electrical circuits. For example connecting a live wire and a return lead to a light bulb should cause a functioning electrical circuit to be formed and result in a lit light bulb. Such a simulation requires two parts: the (abstract) electricity simulation itself and basic collision detection in order to recognise when a connection between two conductors occurs.

In the initial prototyping the collision detection library of ODE [StOCa] was used and the call-back function responsible for collision detection was extended to send 'Connected' and 'Disconnected' messages at the appropriate events. This arrangement took advantage of the already existing collision detection component and reused some of its output. However the code was also tightly integrated with that particular collision detection library. Thus if the dynamics simulation were to be replaced by a different one, the electrical connection code would either have to be adapted to the new library or two separate collision detection functions would have to be used at the same time.

The ideal solution for this is to define a more abstract interface of what a collision detection component should provide; in this instance both contact forces and a list of new and previous colliders (for connections and disconnections respectively). The electricity simulation can then function independently from the underlying collision detection, and is also easier to implement as it can be written in a purely abstract way. This in turn allows the system to be extended beyond the original scope of the electricity simulation, for example lightning arcs could be implemented by a separate simulation and the 'connection' (a lightning arc as opposed to a wire) would be handled in exactly the same way as any other connection.

## 4.2 The four components

In this section the four core parts (properties, resolvers, derived property sets and interactions) of the methodology are developed with reference to the above. Finally, the main loop of the entire simulation will be described, demonstrating how to combine all four parts.

Given a simulation (any kind) in some environment (such as a computer game) there are usually three distinct parts:

- A data structure that is used in the simulation associated with each object (data store).
- Either a stepping function (for clock driven simulations) or an update function (for event driven simulations), i.e. the main simulation logic.
- Interface code to the rest of the system. This could either tie the simulation into another simulation or even just to the user interface or network code (glue).

Three of the four components of the methodology will map closely onto these three concepts. The property layer will serve as a data store, the resolver layer will host the simulations themselves and the interaction layer will contain all of the glue code. (The interaction layer will contain more than just glue code, it is the integral part of the methodology.)

When given two separate simulations (such as dynamics and a SPH fluid simulation) there can be substantial overlap in code and functionality. In this example both simulations require a good collision detection system: the dynamics simulation requires collision detection so objects do not intersect, and the SPH simulation requires collision detection so that its particles do not fall through the objects in the game world (water should flow over a slope, not through the slope).

So far, three layers have been mentioned: the property layer, the resolvers and the interactions. The last part of the methodology aims to serve as a common ground for re-factoring and code reuse. The layer of derived property sets contains parts of the individual simulations, which can be abstracted out of the resolver layer, and are useful to other simulations, or for which there exist alternative approaches and implementations, for example: soft collisions vs. hard collisions. (The interaction layer will also provide the glue code between the derived property sets and the resolvers.)

### 4.2.1 Part 1: properties (data store)

Every object in a game world has a number of properties associated with it. They will define the behaviour and appearance of the object in the simulation. Properties do not necessarily have to serve a purpose in any simulation, such as the name of an object. Any number of properties can be defined for the purpose of convenience. Although there is no difference between one property and another they can be roughly categorised as follows:

- Real properties, for example: mass or position.
- Abstract properties, for example: a pointer to a data-structure.

- Convenience properties, for example: the name of an object.

Properties that model a tangible property such as the position or colour of an object will be referred to as 'real' properties. All other things that might be useful for the simulation but have no 'real-world' equivalent, such as a pointer to a data-structure, are called 'abstract' properties. In the methodology, properties are referred to by their type. For instance, the position property of an object could be implemented in any number of ways (for example either as a vector of floating point values or as a vector of fixed point values), however the rest of the system only cares about it being a 'position' property.

It should be noted that, realistically, an implementation of the methodology will pick one representation but the methodology itself does not concern itself with data types, representation or implementation language.

Each resolver in the system accesses a subset of all properties. For example the rigid body simulation resolver would be interested in position, rotation, velocity, angular velocity and mass of an object, whereas a simplified electricity model would only consider conductivity, electrical state ('on' or 'off') and a list of connections to other objects.

Certain properties are shared by many different simulations, such as the position or shape of an object. Some properties are unique to certain simulations and possibly even unique to a particular implementation of a simulation. Conductivity would be an example for the former, and a pointer to a data structure an example of the latter.

Properties should also be independent of one another. If the value of a certain property could be calculated (or derived) using different properties as an input, it should be part of a derived property set as outlined below in Section 4.2.3.

### 4.2.2 Part 2: resolvers (simulation)

A resolver is usually the core part of any given simulation. For instance in a dynamics simulation the rigid body resolver will calculate the effect of any forces that have accumulated for each point mass object during the last frame and move them to their new positions.

For an electricity simulation the corresponding resolver would update the electrical state of any given object, reacting to changes to the electricity graph (such as new connections and removed connections).

The purpose of a resolver is to represent and perform the core work of any given simulation. The effect of this work is that in the resolver updates certain object properties. To give a resolver work to perform (for example influencing the simulation by applying a certain force to an object) a group of functions called 'interaction functions' are used. Calling the interaction functions themselves does not modify any object properties, it merely queues up work to be performed on the next simulation step where all work is performed in a single self-contained step. A resolver has the following characteristics:

- Properties used (for both reading and writing).
- Resolver properties (properties unique to the resolver or simulation itself).

- Interaction functions.
- An algorithm (the work performed in a simulation step).

**Properties used** The list of properties used lists all properties the resolver will read from and write to. For instance a simple rigid body resolver would list the position, velocity, angular velocity, rotation and mass. If desired, a distinction can be made between properties read from and properties written to in order to aid multi-threading and scheduling.

**Resolver Properties** These properties are properties that are global and unique to a specific simulation. An example of such a property could be the amount of ambient lighting in the world.

**Interaction functions** These functions are used to accumulate work for a resolver to do in the next simulation step. Using the example of the rigid body resolver, there is an interaction function for adding an arbitrary force to an object. Interaction functions are not only essential for most simulations to work (such as in the case of dynamics, where the forces preventing objects from intersecting are supplied by calling the relevant interaction functions), but also provide a way of influencing and directing the simulation due to user input. They also provide a clean interface for different physics simulations to influence (or interact with) each other.

**The Algorithm** Each resolver implements some kind of simulation; the algorithm refers to the specific method used.

### 4.2.3 Part 3: derived property sets (re-factoring and abstraction)

The purpose of derived property sets is to provide a common place for re-factoring and abstracting common problems. Some functions over object properties are required by more than one simulation and thus it would be a waste to implement or perform them more than once. One example of this is collision information: the dynamics simulation needs it to prevent the rigid body resolver from moving objects into positions where they would intersect with one another and the electricity simulation needs it to determine which conducting objects are in contact with each other.

A 'derived' property can be viewed as a mathematical function. It has a number of arguments and performs some kind of calculation over those arguments and only those arguments, i.e. no global state is used and no side effects occur. The result of the calculation is a property that is dependent upon only the function's arguments.

Any number of such functions can be grouped together (if it makes sense for the implementation) to form a derived property set. For instance it would make sense to group all collision related functions into a single derived property set. This set will provide a number of derived properties such as:

- A list of contact forces.

- A list of objects that are currently colliding.
- A list of objects that have just started to touch in this frame.
- A list of objects that have collided in the previous frame but no longer touch in this frame.

A very different example of a derived property set is the creation of shadow volumes for dynamic lighting. Figure 4-1 below illustrates the concept of a shadow volume; the left image shows the original object and its four vertices and the right image shows the vertices of the resulting shadow volume. A shadow volume is at its base a copy of a polygonal object, but each vertex exists twice. The normal for each vertex is taken from the edge it is on. For example the vertex **b** in the left diagram of Figure 4-1 yields two vertices **b** and **b'** in the object's shadow volume. Vertex **b** remains at its current location as its normal is derived from edge **a-b** that points towards the light, vertex **b'** is projected away from the light source towards infinity as its normal (which is derived from edge **b-c**) points away from the light source.

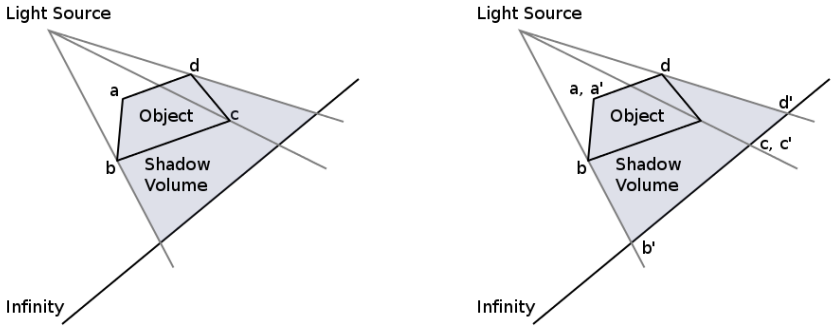


Figure 4-1: Illustration of a shadow volume. The image on the left shows the original object with each vertex labelled. The image on the right shows the 'doubled-up' vertices and their locations after projecting them away from the light source.

Each vertex that has a normal pointing away from the light source is then projected away from the light source to infinity. This new object can then be used to render shadows by rendering the scene in two passes: the first pass with only ambient lighting, and the second pass with lighting. For the second pass the shadow volume is rendered to the stencil buffer and texels that fail the stencil test (i.e. are inside the shadow volume) are then not rendered.

This shadow volume creation can be represented by a derived property set that is calculated from four properties: object geometry, position, rotation and the position of a light source. Generally shadow volumes are intended for rendering real time shadows, but they could also be used for occlusion queries. For example it is possible to model a light barrier for an alarm system by making use of the shadow volume property, the light sensor of the light barrier is 'on' if it is not contained within a shadow volume and 'off' otherwise.

In theory, rendering the game world itself can also be considered a derived property: its input would be all properties of objects relevant for visualisation such as shape, texture, etc. and the output is a rendered image.

In general, a derived property set has the following characteristics:

- A set of the object properties it uses.
- A set of the resolver properties it uses.
- A set of functions used to calculate the derived properties.
- A set of the derived properties it provides.

Derived property sets can be arbitrarily complex in what work they do, sometimes even more complex than a resolver. Most of them update at each time step (also known as the 'world step') however if none of the properties the derived property set depends on change, there is no need to update.

The difference between a derived property set and a resolver is that a derived property set will never change any object properties. It is simply a function over already existing properties and does not directly influence object properties. A resolver on the other hand specifically exists to change properties.

### 4.2.4 Part 4: interactions (glue)

The purpose of an interaction is simple: it allows one simulation to influence (or interact with) another. For example if a new electrical connection is made that causes a certain object such as a light bulb to change its state to 'on', the mechanism that instructs the light simulation to cause the light bulb to emit light is an interaction between the electricity simulation and the light simulation.

Sometimes interactions are also required to make certain simulations work in the first place as they consist of more than one component in the methodology. A good example of this is dynamics: in order to feed the contact forces calculated by the collisions derived property set back into the rigid body resolver an interaction is required.

The main purpose of interactions however is to connect different physics simulations together. For instance if two metal objects touch, an interaction calls the relevant interaction function in the electricity resolver to create an electrical connection and a different interaction could create the appropriate sound (or rather instruct the appropriate resolver to do so), which goes along with such a collision. Another example is an interaction that applies the forces generated in a magnetics simulation to the rigid body simulation so that there is a visible effect.

More formally, an interaction takes data from any of properties, resolver properties or derived property sets and calls interaction functions in a particular resolver. An interaction has the following characteristics:

- A set of properties, global properties and derived properties it reads from.
- A single resolver (on which interaction functions will be called).

Similar to a derived property set, interactions only read from properties and never modify them directly. The interactions are the only layer in the methodology that can use the interaction functions of resolvers.

## 4.3 Concluding remarks

This chapter introduced the four parts of the methodology and some context in that they apply.

If there is one point worth repeating, it would be that the methodology is a collection of ideas and approaches: it is not tied to a particular language or game or physics simulation.

The next chapter will provide a more formal definition of each of the four parts based on the above and will introduce a notation to describe an instance of them, along with some discussion of properties, advantages and limitations of the overall system. Chapter 6 will provide four concrete examples on how an implementation of the methodology works.

# Chapter 5

# The methodology: definition

This chapter will define the four components of the methodology: *properties*, *resolvers*, *derived property sets* (generally abbreviated to 'DPS') and *interactions*.

It is recognised that sometimes application-specific shortcuts are necessary for improved run-time performance of the system; this chapter discusses one such shortcut in detail.

## 5.1 Introduction

In the next Section, Section 5.2, the main loop of a program based on the methodology is given. The main loop is executed once per timestep. A *timestep* refers to a small fraction of time between one update and the next. It usually represents one frame and is thus in the order of $\frac{1}{20}$th of a second to $\frac{1}{60}$th of a second.

In the following four sections the four core parts of the methodology are then defined. The three main active layers of the system (resolvers, derived property sets and interactions) each contain one 'do-work' function called by the loop. These functions are named *advance*, *update* and *apply* for resolvers, derived property sets and interactions respectively and are referenced in the next section describing the main loop.

## 5.2 Main Loop

The main loop of a program based on the methodology is executed once per frame and usually represents one time-step. Depending on the individual requirements of the application, these time-steps are either constant or variable (in which case they depend on the average time it took to compute a number of previous frames). The main loop, in terms of the methodology, will consist of the following steps:

1. For each resolver, call *advance*.
   Some optimisations are possible at this point: certain resolvers may only be advanced every $n$ frames, other resolvers can be advanced in parallel.

2. For each derived property set in the system, *update* its derived properties.
   The same optimisations as above apply, with the exception that all derived property sets can be updated in parallel.

3. Finally, *apply* each interaction defined in the system.
   All interactions can be applied in parallel, but unlike the above two items all interactions are applied in every frame (but there may no work to do for any given interaction).

For most programs there is, of course, more to it. Event handling, rendering, networking, etc. are also part of such a main loop but omitted here as they do not strictly related to the methodology. A different view is to run the above three steps in place of the usual "run physics engine" step of a game engine.

It should be noted that the main loop only deals directly with three out of the four parts of the methodology. The properties layer is manipulated directly by the resolver layer; all other layers have read-only access to it. There is no logic in the property layer, it is simply 'dumb' data storage.

The generic main loop given above is not very interesting in itself, it simply provides the necessary glue to make the four parts, described in the next four sections, work together to provide the overall physical simulation.

## 5.3 Layer 1: properties

The first layer of the methodology is a passive layer It is used only to store data associated with objects in the game world. This 'per-object' data can represent both 'real' properties (such as the mass, shape or position of an object) and 'abstract' properties (such as a pointer to an internal data-structure).

Most real properties are designed to be shared by various different physics simulations, for example the position property is useful for both dynamics and magnetism. Some real properties on the other hand are used by only one physics simulation, for example conductivity (or resistance) is only relevant to electricity. However there is nothing inherently different between unique and shared object properties.

Abstract properties are generally tied to only one particular implementation of a physics simulation, and are used to store per-object data directly with the object. It should be noted that it is also possible to store this kind of data within a hash-table (or some other collection), mapping objects to some arbitrary data directly within the resolver. This approach is useful if data is sparse, i.e. it does not apply to most objects in the game world. Conversely, using abstract properties is desirable if the data applies to most objects within the game world. As with shared and unique properties, there is nothing distinguishing real and abstract properties from each other, it is merely a matter of how they are used.

An object can have any number of real, abstract, shared and unique properties. The exact set of properties used at any time is implementation dependent and could potentially change as components are added to and removed from the system.

**Definition** `oProp()` is the set of all possible object properties within any given implementation of the methodology.

**Definition** An object property can be described using the following notation:

| *NAME* Property |
| --- |
| *INFORMAL DESCRIPTION OF THE PROPERTY.* |

## 5.4 Layer 2: resolvers

The *resolvers* represent the core part of each physics simulation. Each physics simulation, which affects the game world, must provide one, as a resolver is the only component that can manipulate object properties.

Each resolver provides two sets of object properties: one containing properties that will be read from, and one containing properties that will be modified. In the majority of cases these two sets are either the same or the write set is a subset of the read set, but this is not necessarily true in all cases.

A resolver might also provide *resolver properties* that are properties unique to the resolver and not related to any particular object. An example of a resolver property would be the global gravity vector of a linear gravity physics resolver (a simplified gravity model where gravity acts as a constant force on all objects in the game world).

The main part of a resolver is its *resolver algorithm*. It is unique to each resolver; this is where the simulation happens. In each time-step (also see Section 5.2) this algorithm is executed at most once. This process is called to '*advance*' a resolver. During this simulation step the resolver updates object properties as per its object property write set.

For example if the rigid body simulation resolver is advanced, it will resolve all forces on each point mass in the system and update the position, rotation and velocity properties of each object.

Depending on the simulation it is not strictly necessary to advance the resolver on every time-step, for example a buoyancy resolver will still yield believable results if it is only executed every 5th time-step; also see Section 6.12.5 (page 97) for a more detailed analysis of this.

Finally, a resolver provides *interaction functions*. They are the interface between the resolver and the rest of the system. The interaction layer of the methodology is the only part of the system that may call these interaction functions in order to feed new data to the resolvers. All interaction functions require a thread-safe implementation in order to allow more than one interaction to be called on the same resolver at the same time. Even if thread-safety is achieved by simply blocking until another call has finished, this is an essential property of interaction functions.

Each interaction function should execute in near constant time, and the order of calls should not matter. Each call to an interaction function will usually record its arguments in a queue within the resolver and upon the next time-step this stored workload is completed or resolved (hence the name).

Unfortunately, for certain simulations, order independence of the interaction functions cannot be guaranteed and in that case it would be the duty of the individual interactions to avoid calls in the wrong order. However, as it will be shown in Section 5.7.2, order independence can be guaranteed even for problematic resolvers (such as the electricity resolver) by modelling the game world in a particular way.

**Resolvers in summary**

- **Set of properties read**
  The properties that are required to run the simulation.
- **Set of properties written**
  The properties that are modified (but not necessarily read) by the simulation.
- **Set of resolver properties**
  Simulation and resolver-specific global variables.
- **Resolver algorithm**
  Implements a specific physics simulation, such as rigid body simulation or electricity. It is called at most once every time-step from the main loop. Takes data from the read set and modifies properties in the write set. Can both read from and write to the resolver's own properties. Most importantly, guarantees to resolve all interaction functions that have been called since the last time-step.
- **Interaction functions**
  Interface to the physics simulation, can be called by interactions. They have the following required properties:
  - Thread-safe.

  They have the following desirable properties:
  - $O(1)$ complexity.
  - Order independence within the same function: $f(x); f(y)$ should yield the same result as $f(y); f(x)$.
  - Order independence over different functions: $f(x); g(x)$ should yield the same result as $g(x); f(x)$.

**Definition** `prop`(Resolver **R**) is the set of resolver properties.

**Definition** `rProp()` is the set of all resolver properties within any given implementation of the methodology.

**Definition** `read`(Resolver **R**) returns a subset of `oProp()` containing the properties read during the resolver *advance*.

**Definition** `write(`Resolver **R**`)` returns a subset of `oProp()` containing the properties written to during the resolver *advance*.

**Definition** A resolver can be written using the following notation:

| *NAME* Resolver | |
|---|---|
| Resolver properties: | `prop(`*NAME* Resolver`)` |
| Properties read: | `read(`*NAME* Resolver`)` |
| Properties written: | `write(`*NAME* Resolver`)` |
| Work on *advance*(): | Algorithm description. |
| *Interaction functions:* | |
| `function(`arguments, ...`)` - *Description.* | |

The three property sets can be either given as above if they are identical to a previously defined set, or can be given literally.

### 5.4.1 Related concepts

Resolvers are related to rewrite rule systems. A resolver can be seen as a system that changes, upon each invocation, certain properties of the object in the world according to its rules. The union of all resolvers implement the mechanics and rules of the world and correspond to a more generic rewrite system.

The **St**anford **R**esearch **I**nstitute **P**roblem **S**olver (STRIPS) [FN71] is a well-known planning system in artificial intelligence. Given a world state and a set of operations that transform (i.e. rewrite) the world state into another, the task of the STRIPS solver is to find a sequence of operations to arrive at a given goal state. (A full STRIPS system will also contain some theorem prover as the world state is expressed in first order logic predicates, although implementations of STRIPS for games such as F.E.A.R. [Mon05] usually skip this step and choose a simpler representation.)

In STRIPS based system there exist a number of rules, each with preconditions (what must be true for the rule to apply) and postconditions (what will be true once the rule is applied). One could conceivably express a resolver or the union of all resolvers as a STRIPS instance, however due to the number of rules and predicates required this would be highly impractical. The important property however is that such a mapping is possible.

Conventionally, as STRIPS are used for AI problems that appear often in games; viewing the underlying physics also as a rewrite system is an interesting exercise in unification. More practically is the converse: it is also possible to embed the AI of a game in another resolver.

## 5.5 Layer 3: derived property sets

Derived property sets (abbreviated to DPS) provide a set of one or more related properties called *derived properties*. These properties are usually per-object properties but are not restricted to

a simple one-to-one mapping. For instance object properties could be per-pair (for penetration depth for two intersecting objects) or for a set of objects.

These derived properties are calculated by using object properties (and sometimes resolver properties). They are named derived properties because they are derived from a set of properties and are solely dependent on them, they change if the properties they are derived from change. Derived property sets are so named because they are simply a collection of such derived properties. Other ways to view a derived property are as an effect filter as found in image manipulation programs, or a function with no side-effects or a 'view'.

A typical example of a DPS is the *Collisions DPS*: it uses the position, shape and rotation of each object and computes collision information. It contains a number of derived properties:

- $S_C$, the set of objects that currently collide.
- The set of objects that have been added to $S_C$ in the last update step.
- The set of objects that have been removed from $S_C$ in the last update step.
- The set of contact joints intended to be used by the rigid body resolver.

A derived property set is prompted to *update* on most time-steps by the main loop. Like resolvers, derived property sets do not necessarily have to update on every time step.

Derived property sets can read from object properties but not write to them. They can also read data from resolvers, should they provide any. Derived property sets cannot read from other derived property sets, and thus all derived property sets can update in parallel.

**Derived property sets in summary**

- **Derived properties**
  A set of all derived properties provided by the DPS. These mappings can be of any conceivable type, such as:
  - all visible objects → texture or image (this would model the rasterisation of a scene)
  - one object → shadow volume
  - two objects → collision information
- **Update algorithm**
  Computes all derived properties.

**Definition** `prop(`DPS **X**`)` returns the set of derived properties.

**Definition** A derived property set can be written using the following notation:

| *NAME* DPS | |
|---|---|
| Properties read: | *SET OF OBJECT AND RESOLVER PROPERTIES READ* |
| Derived properties: | `prop(`*NAME* DPS`)` |
| Work on *advance*(): | Algorithm description. |

## 5.6 Layer 4: interactions

The purpose of interactions is to provide the feedback loop for the simulation. Interactions may read from all of the previous three layers; data can be taken directly from object properties, resolvers or (most commonly) derived property sets. An interaction will, with no or minimal processing, feed data from various sources back into the system. Each interaction operates on exactly one resolver, but it is possible for more than one interaction to operate on any one given resolver. The interaction provides data to this resolver by calling its interaction functions. As mentioned in Section 5.4, the order in that they are called should not matter, hence it is possible to run all interactions simultaneously, even if they operate on the same resolver.

Interactions are perhaps the most important parts of the methodology. Without them nothing interesting will happen, none of the physics simulations could communicate with each other in a meaningful way.

Interactions most commonly take data from the derived property set layer.

**Interactions in summary**

- **Resolver**
  The resolver used by the interaction. Only interaction functions from this resolver may be called.
- **Interaction algorithm**
  If an interaction is *applied* it will generally iterate over its input data and feed it back into the resolver with minimal processing.

**Definition** `prop`(Interaction **I**) returns the set of object properties, resolver properties and DPS properties read by the interaction.

**Definition** An interaction can be written using the following notation:

| (*RESOLVER* ← *SET OF DPS* or *SET OF PROPERTIES*) Interaction | |
|---|---|
| Properties read: | *SET OF OBJECT AND RESOLVER AND DPS PROPERTIES READ* |
| Work on *apply*(): | Algorithm description. |

## 5.7 Additional notes

The above fully defines the methodology; it is all that is necessary to construct the architecture of a program. In this section three incidental further points are discussed: how to take advantage of the different parts of the methodology and run them in parallel, some notes on order dependence of interaction functions and one shortcut that is relevant in particular to a dynamics simulation.

### 5.7.1 Scheduling and parallelism

The conditions under that a set of programs can be run in parallel safely is a well understood concept in computer science, and many solutions and mechanisms such as locking [Dij74,Lam67, Dij65, Knu66] and message passing [Sor73] have existed for a long time. In summary, any two programs may run in parallel safely as long as:

1. One program does not try to write to data that the other program tries to read from or write to at the same time (i.e. a race condition).

2. The order of execution of each statement within two programs running in parallel does not matter.

It is possible to run a number of components of the system in parallel by simply avoiding to violate the two conditions above.

Condition 1 can be fulfilled if read and write operations are atomic. This can be accomplished by two resolver components never writing to a property that is read by another.

Condition 2 seeks to avoid a particular kind of race condition. For example, it is broken if for a value $x = 1$ one program tries to add one to $x$ and another tries to multiply $x$ by three. If the first program runs first, $x = (1 + 1) * 3 = 6$ holds true; if the second program runs first then $x = (1 * 3) + 1 = 4$ holds true. The only way to resolve this problem is to always run these operations in the same order. Naturally, condition 2 is also trivially met if, for example, one program performs $x = x + 1$ and another performs $y = 3 * y$.

#### Resolvers in parallel

Given two resolvers $R_1$ and $R_2$, the set of read-properties $\texttt{read}(R_1)$ and $\texttt{read}(R_2)$, and the set of write-properties $\texttt{write}(R_1)$ and $\texttt{write}(R_2)$. Resolvers $R_1$, $R_2$ can be executed in parallel if and only if the following three properties hold:

- $\texttt{write}(R_1) \cap \texttt{write}(R_2) \equiv \emptyset$
- $\texttt{write}(R_1) \cap \texttt{read}(R_2) \equiv \emptyset$
- $\texttt{write}(R_2) \cap \texttt{read}(R_1) \equiv \emptyset$

If the properties hold, it is guaranteed that both conditions 1 and 2 are met.

#### DPS in parallel

All derived property sets can be updated in parallel as they are only allowed to read from object and resolver properties. The output they produce is stored within the DPS itself and hence this cannot not conflict with any other DPS.

It is impossible for condition 1 to fail, as object properties and resolver properties are the only common data between DPS and they are only read from. It is also impossible for condition 2 to fail as all derived properties are computed independently with respect to other DPS in the framework and DPS may not read data from each other.

#### Interactions in parallel

All interactions can be applied simultaneously if they are well formed. There are four sets of shared data: object properties, resolver properties, DPS and interaction functions. These can be separated into two distinct groups: read-only data (object properties, resolver properties and derived properties) and interaction functions.

The first group does not matter with respect to the two conditions, as the data is only read. It is thus impossible for condition 1 to fail as all data is read-only and interaction functions are thread-safe.

The second group should not matter, as interaction functions must be thread-safe and should be order independent, as their definition states in Section 5.4. Assuming that all resolvers concerned are order independent, then it is also impossible for condition 2 to fail.

If the interaction functions of resolvers are not order independent then the problem can usually be solved easily. For example the electricity resolver is not order independent: first trying to create a connection between A and B and then removing it has a different effect from first trying to remove the connection and then recreating it. In this case, a simple solution would be to adapt the electricity graph so that edges can be created more than once (a simple reference count should suffice) and only if all edges have been removed is the connection severed.

### 5.7.2 Order dependence within resolvers

During the proof of concept implementation work, which included rigid body simulation, electricity, magnetism and gravity, it was found that only the electricity and magnetics resolver provide interaction functions that are potentially order dependent. The two main interaction functions provided by the electricity resolver are `addConnection(`Object **o1**, Object **o2**`)` and `removeConnection(`Object **o1**, Object **o2**`)`; see the electricity implementation notes in Section 6.5 for more information.

For two given objects A and B, it matters whether `addConnection(`A, B`)` is called before or after `removeConnection(`A, B`)`. The result is either no connection or a connection. However, with some care in how the implementation is designed, this situation can be made to never occur: for a connection to be made or removed, there must be some physical occurrence that the connection is intended to reflect. For example two metal objects colliding would form a connection and it is the act of them colliding that triggers the (Electricity ← Collision) Interaction. With only this interaction acting on the electricity resolver it is impossible to both add and remove a connection as it is impossible for any two objects to both collide and not collide at any the same time.

However, if the implementation provides two or more independent ways for an electrical connection to be formed between two objects then it is possible for a connection to be both formed and removed in the same frame. For example, two alternative ways to provide an electrical connection between two objects are:

- An interaction that forms a connection between objects A and B if the air between them is sufficiently humid.

- An interaction that forms a connection between objects A and B if the air between them is extremely hot.

Then, if in the same frame the two objects A and B stop touching, but the air between them also reaches a certain threshold temperature, the (Electricity ← AirTemp) Interaction calls `addConnection(A, B)` before the (Electricity ← Collision) Interaction calls `removeConnection(A, B)` would lead to the wrong result.

There are two simple ways to approach the problem: inserting extra vertices into the electricity graph to represent each type of connection independently, or implementing 'edge counting' in the resolver.

**Extra vertices**

Each object will have more than one vertex in the electricity graph associated with it, these groups of vertices are connected inside them. In the above example, one vertex in a group would represent physical contact, another one current due to high humidity and another current due to high air temperature. Each interaction only connects its two relevant vertices in each vertex group, and thus the correct result is achieved as the same connection can not both be added and removed in the same frame anymore.


Figure 5-1: Solving order dependence of connections. The two nodes (2 and 3) from the left graph can 'doubled-up' and connected so that two different connections can be formed between the two nodes: one in the 'a-world' between 2a and 3a and one in the 'b-world' between 2b and 3b.

Figure 5-1 demonstrates this concept using the example mentioned above. Vertices 2 and 3 represent two objects that can form a connection by either touching each other or due to very hot air between them. Vertices 1 and 4 represent 'the rest' of the electricity graph. The left image illustrates the problem; the right image shows two additional vertices (2b and 3b). This method solves the problem of two different interactions calling `addConnection(2, 3)` and `removeConnection(2, 3)` in the same frame. The object is now represented by two vertices (2a, 2b) and (3a, 3b). For physical contacts the edge $2a - 3a$ is manipulated, and for 'hot-air'

contacts the edge $2b - 3b$ is manipulated instead. The electrical connection between the two objects is only broken if both $2a - 3a$ and $2b - 3b$ are removed.

**Edge counting**

This simple addition to the electricity graph counts how often a particular edge has been created. Only if this count reaches zero is the connection between two vertices destroyed.

This solution is better than the above in most cases as the resulting electricity graph will be much smaller and reference counting for each edge is not expensive. It will certainly not have a negative impact on the complexity of the algorithm operating on the electricity graph.

Its one downside would be that it is impossible to know *how* two objects are connected by just looking at the electricity graph.

### 5.7.3 A useful shortcut

Sometimes a back-end implementation of a physics simulation can be used to drive more than one component within an implementation. For instance the ODE plug-in provides both a rigid body simulation resolver and a collision detection derived property set. These two components can be tightly integrated with each other so that the collisions DPS makes use of a special ODE function, which can be used to pass all collision information to it at once, significantly reducing function call overhead.

The two components (rigid body resolver and collisions DPS) are separate and communicate via an interaction. Speed (a large number of function calls are eliminated) and simplicity of implementation (at least in terms of code size, as an entire interaction can be avoided) can be improved if the ODE collisions DPS can operate on the ODE rigid body resolver directly. ODE is used in this example, but this shortcut is relevant to any dynamics simulation with integrated collision detection.

A derived property set calling an interaction function on a resolver during its update step does not follow the definition of a derived property set (Section 5.5). The main ideas behind the definition of the methodology above are clarity, modularity and the ability to make statements about which components of the system can be executed in parallel. However, if speed is of significant importance in the user's requirements, it is possible to slightly relax the definition of the methodology and allow certain derived property sets to call the interaction functions of a resolver during their update step.

**Interactions inside a DPS** Any Interaction **I** can be applied out of order with other interactions (inside an update of a DPS **X**) if the Interaction **I** reads from only object properties, resolver properties and DPS properties provided by DPS **X**. I.e Interaction **I** can be executed directly after DPS **X** (or interleaved with it if the specific DPS update algorithm allows) if and only if `prop(I)` $\subset$ ( `oProp()` $\cup$ `rProp()` $\cup$ `prop(X)` ).

All interactions can be executed in parallel. When applying Interaction **I** out of order (earlier than usual in this case) the only thing that changes is the time at which it is applied.

The only setup that can lead to inconsistent behaviour is Interaction **I** violating either conditions 1 or 2. Since all interactions can be run at the same time, and interaction functions are by definition order independent, condition 2 does not apply.

The only way to violate condition 1 is if Interaction **I** reads data from a DPS that has not been updated yet or is currently updating. Hence if Interaction **I** only depends on one DPS (DPS **X**), it is valid to apply the Interaction **I** out of order directly after DPS **X** has updated.

## 5.8 Concluding remarks

This chapter provided the essential properties of each of the four components of the methodology. It also introduced a notation to describe an instance of one of those components.

The next chapter will describe an implementation of a number of instances of methodology components in the terms of this chapter. It will be demonstrated how they can work together with the aid of four experiments.

# Chapter 6

# The methodology: implementation, experiments and results

The previous chapter described each of the four components of the methodology. This chapter will discuss the implementation created during this work. Two very different implementations have been attempted. The first was not an implementation of the methodology as such, it was however what inspired and drove its initial conception.

For the purpose of this dissertation, the second implementation is more important. The second is an implementation for the methodology, it was used both to further develop and refine the methodology and to form the environment in which to conduct the experiments presented in this Chapter. The second implementation is illustrated and discussed in detail from three perspectives:

- The general architecture and how the physics simulations have been integrated into the implementation.
- Four experiments are presented in the form of scenarios with screen-shots and commentary, demonstrating how everything fits together.
- Finally, a discussion of how the physics simulations implemented *should* have been implemented, in retrospect, and why.

A brief discussion of the limitations of this second implementation is then presented. As the implementation grew with the concept of the methodology, some of the engineering choices made were not the best. Recommendations for a more 'pure' implementation are made in Chapter 7, "*Discussions and future work*".

## 6.1 Introduction

The very first experimentation work was exploratory, to try to reveal the problems needing attention. The program contained a basic 3D rendering engine, a lighting model, and the dynamics simulation package ODE. Most of the experimental work on electricity, up to, but excluding, the final two-wire approach given in Appendix A), was performed in it.

As it became more and more convoluted, it became apparent that to integrate a lot of physics simulations in a clean way it is necessary to have a clear concept of how they all fit together. This eventually gave rise to the development of the actual methodology.

The main visible difference between the second implementation and the first is that it only models 2D worlds. This decision was made for two reasons:

- Firstly, it became harder to create useful worlds in the first implementation because a lot of effort was spent on laying them out. It was also much harder to see what was going on, and interaction with the world was awkward at best. A 2D based approach was simpler to manipulate (simple, obvious drag and drop worked well), simpler to lay out and easier to visualise, both in terms of clarity and rendering. (This became more important as the main work had to be carried out on a low-powered computer.)

- Secondly, as the concept of the methodology was ironed out, it became clear that a new implementation would be better than trying to modify the original. The methodology was defined from the lessons learnt from both the first implementation and throughout the second implementation. In retrospect, a number of implementation decisions made should have been made differently and these form the basis of the recommendations in Chapter 7.

The remainder of this Chapter will exclusively discuss the second implementation.

## 6.2 Architecture Overview

This section presents a brief overview of the architecture of the implementation.

### 6.2.1 Implementation language and build system

The implementation language chosen for the second implementation was C++, for three main reasons:

- It is a language that has native bindings to nearly every physical simulation library and graphics API, as these are usually either written in C or C++. In fact, all physics engines mentioned in this dissertation are written in either C or C++.

- A powerful cross-platform GUI framework exists in the form of Qt [Tro94]. Although normally not a consideration for a game, the availability of an easy to use yet powerful GUI framework is important for an experimental implementation.

- The author was already familiar in it; furthermore most contemporary games involving 3D graphics are written in either C or C++.

A plug-in based architecture was chosen in order to be closer to the spirit of the methodology. This required a large number of shared libraries to be generated that in turn required a build system capable of dealing with this in a cross-platform way. The CMake [Kit00] build system was chosen as it can do this easily. In particular, it was chosen over the canonical GNU Autoconf [Mac91] based system as it appeared to be much easier learn and understand.

Although no effort was made to compile and run any of the experiments on any platform other than GNU/Linux running X11, each external library used (Qt, ODE, OpenGL, OpenAL, Boost Graph Library (BGL)) has been deliberately chosen due to their cross-platform nature and availability of source code under a Free Software license, such as the BSD license or the GNU General Public License (GPL) license. Hence the entire system depends on only Free Software and should be portable with minimal effort.

There are two reasons none of the readily available free-software graphics engines were used (such as id Tech 3 (the Quake3 engine) [id99], OGRE [StOCc] or Cube2 [dot]):

- The focus of the work is the methodology and not good graphics.
- Working with an existing engine could have constrained the work down a particular line due to arbitrary limitations with any chosen engine.

### 6.2.2 Architecture

Figure 6-1 shows an informal diagram of the architecture of the system and how the various libraries used fit together.

### 6.2.3 Design patterns

Since the focus of the work was to try out ideas, the overall design of the implementation could be more rigorous. As it stands, the only classic design decisions was to base everything on *plug-ins* and sub-classing often in order to bring some structure to everything. This also made it easy write new components and integrate them into existing framework.

In particular the main loop benefited from this, as all it did was iterate over the list of resolver components, DPS components and interaction components, and call the resolve, update and apply methods respectively on each. When adding new components to the system, it was not necessary to touch the main loop.

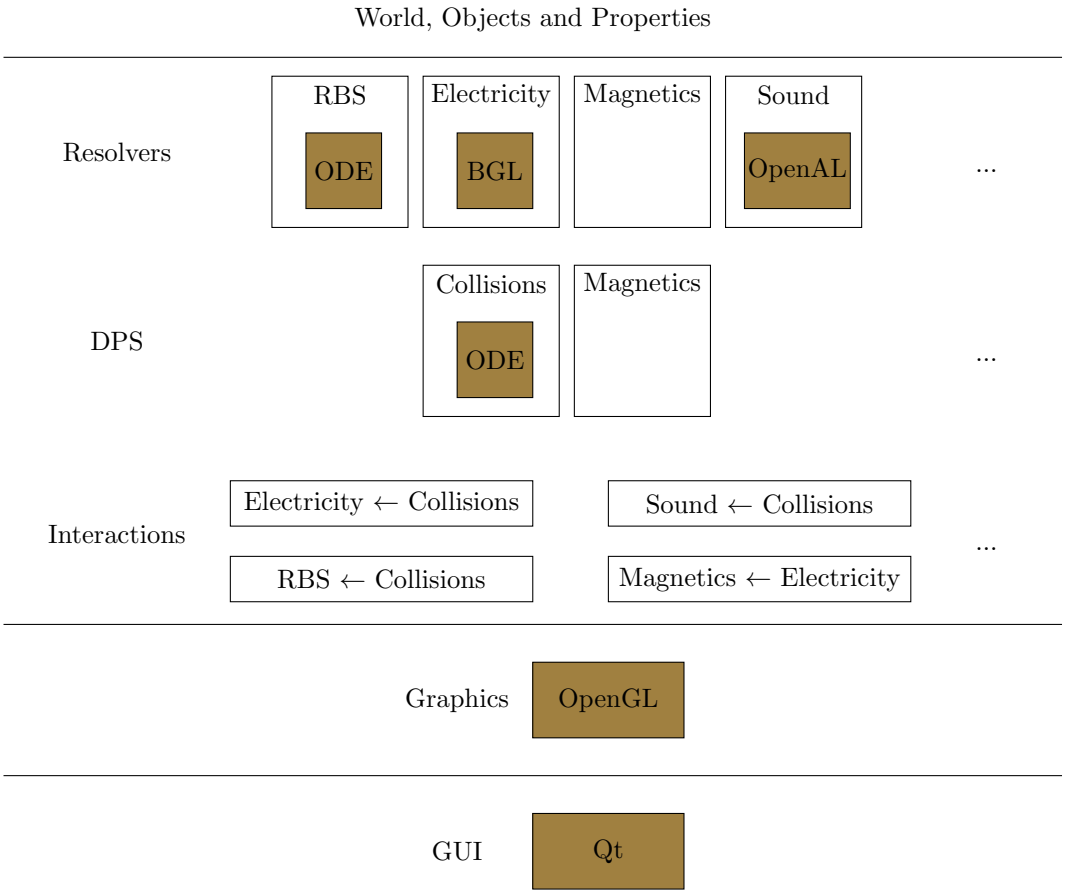


Figure 6-1: Architecture diagram of the implementation presented in this Chapter. This diagram is layered with the lowest level on top. Beige boxes represent other, third party, libraries used to help implement the functionality of a particular component. In the layer dedicated to the main functionality of the methodology the diagram tries to be generic as an implementation of the methodology is not limited to just these resolvers, DPS and interactions; this is what the ellipsis (...) at the end of each row represents. The specific examples of resolvers, DPS and interactions given represent a large subset of this implementation; absent are the components related to buoyancy and gravity due to space constraints.

In terms of implementation each box representing a resolver, DPS or interaction is implemented as a shared C++ class, sub-classed from each row. (Thus the Electricity resolver is sub-classed from a resolver class.) These three in turn are sub-classed from a generic physics component class.

Each property (not illustrated) is a separate class representing that type, thus there is a class for the position property, a class for the conductivity property, etc. Each is sub-classed from a generic property class.

The graphics layer is implemented as a separate class, and so is the GUI layer.

### 6.2.4 User interface

In the remainder of this Chapter many screen-shots are shown, but they only show the rendered scene and not the overall system for the sake of conciseness. Figure 6-2 shows the entire system.

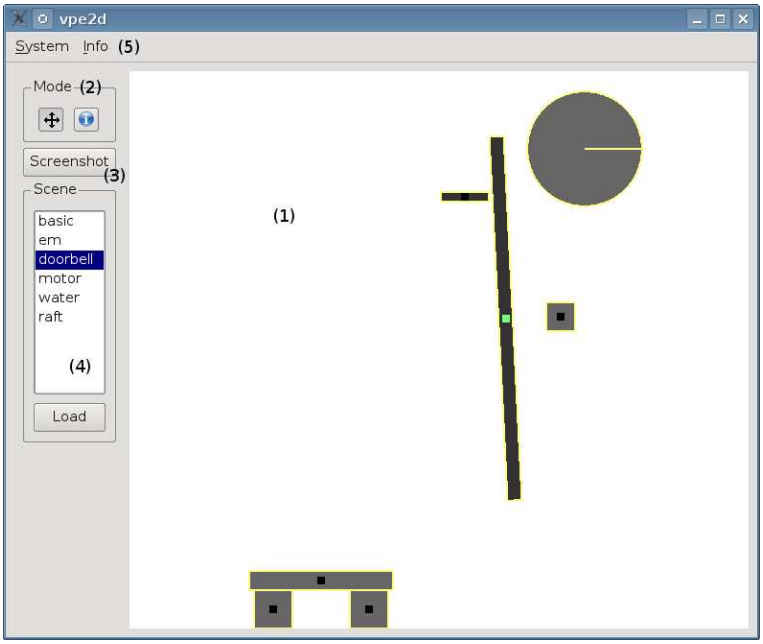


Figure 6-2: A screen-shot of the overall system. The labelled components are the following:

(1) The widget rendering the current experiment. Clicking and drag & drop will have different effects, depending on the current mode. A few keyboard shortcuts are also available, documented below.

(2) The mode selection box changes the behaviour of the system when interacting with the scene. Only two modes have been implemented so far: direct interaction (currently selected) allows objects to be dragged around in the scene. Information mode will dump statistics about the clicked object to standard output.

(3) This button produces a sequentially numbered screen-shot in the current directory. During the development it was found that screen-shot programs cannot necessarily grab the current OpenGL context, so this button was implemented as a workaround.

(4) The scene selector. Clicking 'Load' will either reset the current scene or load the specified scene.

(5) A simple menu system. The only option exported in the system menu so far is to quit and the information menu can be used to obtain information about the version of Qt and OpenGL used.

The following key-bindings are also available:

space Starts or stops the current simulation.

`f` Advances the current simulation by a single frame.

`r` Toggles recording. During recording a sequentially numbered screen-shot is produced at the end of every time-step.

### 6.2.5 Order of implementation

The rough order of implementation of the various components was:

1. The overall outside GUI was implemented first including a mechanism to render to a widget within the GUI.
2. The general skeleton of the methodology was implemented next, including a provisional main loop.
3. The first physical simulation implemented on top of this was dynamics using ODE, along with the simple object model (boxes and spheres) and a simple renderer.

The rest of the implementation was carried out in a less ordered fashion, as the above formed a simple but effective framework to carry out experiments in.

## 6.3 Implementing dynamics

Dynamics is a problem for that many self contained solutions already exist, all with different goals and benefits. For this section an existing solution, Open Dynamics Engine (ODE), will be examined and the process of integrating it into the proof-of-concept implementation will be described. Although this part of the implementation will be specific to ODE, many parts of this section will still apply to other dynamics engines.

Overall ODE consists of two tightly linked elements: a point mass rigid body simulator and a simple collision detection engine. Internally they operate on two different data structures that share some information and, if both the rigid body simulator and the collision detection engine are used, they also contain pointers to each other.

The integration of ODE is not as clean compared to the electricity simulation; this was necessary in order to achieve good run-time performance. One of the shortcuts taken was discussed in Section 5.7.3: this implementation for collisions effectively bypasses the interaction layer and communicates with the resolver directly. Another technique used was to use sub-classed properties to provide ODE-specific implementations of the position, rotation and velocity properties. However it should be stressed that the system could be adapted to use the generic properties that this implementation also provides. (All other simulations talk to the base-class, only the ODE-specific components know about the ODE-specific features.)

### 6.3.1 Properties

ODE itself keeps track of the position, rotation, velocity and angular velocity of each object. It was chosen to implement some of the basic properties using base classes (which other components will use) and provide ODE-specific implementations of them.

ODE also stores some ODE-specific properties, for instance whether an object is ‘enabled’ (i.e. to be considered during simulation). This state is exported in an abstract property, and is intended to be used by the other ODE driven components.

The following properties are provided and used:

| Position Property |
| --- |
| *This is the position of an object. It is also assumed to be the centre of mass and the origin of the model representing the object.* |

| Rotation Property |
| --- |
| *The rotation of the object.* |

| Velocity Property |
| --- |
| *The velocity of the object.* |

| Angular Velocity Property |
| --- |
| *The angular velocity of the object.* |

| Shape Property |
| --- |
| *The shape of the object. (In this example implementation only two different shapes are supported: spheres and boxes).* |

| ODE Properties Property |
| --- |
| *This abstract object property keeps track of two pointers to objects used by ODE: the dBody and dGeom objects.* |

Note that all the 'real' properties will be useful and relevant no matter which particular physics is used and that the abstract property is only relevant if ODE is used. (The dBody and dGeom data-structures are ODE-specific classes.)

This list is by no means exhaustive, the exact properties will depend on the base assumptions for the game world. For instance, if a more realistic dynamics simulation were required, properties like the coefficient of friction or (non-uniform) density and mass distribution of objects would also be useful.

### 6.3.2 Resolvers

The previous section has described the properties; this section will look at what the simulation does and which parts of it can be represented by a resolver. As previously indicated, there are two main parts to ODE: the rigid body simulator and a collision detection engine.

There are two possible options here: to treat the entire simulation as one unit and implement everything ODE offers in one resolver, or to split them up and handle them separately. The main advantage of the former would be a slightly simpler implementation. The advantage of the latter is modularity. For example, with this the collision detection engine can potentially be replaced with a different one or swapped out the rigid body simulation back-end. The second approach was chosen for this proof-of-concept implementation for the sake of simplicity of implementation.

The rigid body simulation in ODE operates on point objects, i.e. it does not care about geometry. The collision detection engine supplies the correct contact forces in the form of contact joints, which ensure that objects collide properly and do not intersect. At every frame in the simulation a certain function must be called that advances the state of the world by a specified time interval. This is known as the '*world step*'.

Note that any contact joints and other forces added to the world are only valid for the current world step. If a force needs to be applied over time, it must be applied every frame. The stepping function (dWorldStep inside ODE) will be the main part of the resolver and will form its 'algorithm'.

Thus resolver now mostly wraps the dWorldStep function in ODE. The next feature to add are the interaction functions. There are two types: normal ones and an ODE-specific one. The functions that can add arbitrary forces to any object in the world are an example of the former and addContacts is the latter.

*Note: this implementation assumes that ODE is responsible for both the collisions DPS and the rigid body resolver.*

This resolver will use all the properties listed in the previous section, resulting in the following resolver for ODE RBS:

| RBS [ODE] Resolver | |
|---|---|
| Properties read: | {Position, Rotation, Velocity, Angular Velocity, ODE Properties} |
| Properties written: | ≡ `read`(RBS Resolver) |
| Work on *advance*(): | Resolve all accumulated forces and joints in the system in one step. (In other words, perform rigid body simulation by calling dWorldStep.) |
| *Interaction functions:* | |
| `addForce`(object, force) - *Add an arbitrary force to the system acting on the centre of mass of a given object.* | |
| `addForceToPoint`(object, point, force) - *Add an arbitrary force to the system acting at the specified point on a given object.* | |
| `setLinearGravity`(*V*) - *Sets (or disables) the global gravity vector.* | |
| `addContacts`(ODE contact joints) - *Add all contact joints from the collision detection to the system.* | |

Most of the above resolver is implementation independent. Any dynamics simulation would need some kind of method for the user to add an arbitrary force to the system. Adding contact joints in bulk on the other hand is something that is specific to ODE. The overall work done would be the same for each resolver, but how it is achieved is of course particular to the implementation: using the ODE dWorldStep function, using a custom implementation or using special dedicated hardware all have the same purpose but function in radically different ways.

### 6.3.3 Derived property sets

The other main part of ODE is the collision detection engine. This could also theoretically be implemented as a separate resolver. However certain properties of a setup like this make it undesirable:

- There is no work to accumulate over a frame; collision detection has clearly defined goals and little room for interaction. Either objects collide or they do not.
- Which objects touch and how they touch is relevant for other simulations and can in fact completely drive other simulations, such as electricity. Thus, which objects touch and how they touch should be stored somewhere as a property. However, this information can be calculated using the position, rotation and shape properties; therefore collision information is derived from other properties.

Thus, it makes sense to implement collision detection as a derived property set. The only properties collision detection will need are the position, rotation and shape of an object. From these, it will derive a number of properties for each object and make them available to other simulations.

| Collisions [ODE] DPS | |
|---|---|
| Properties read: | {Position, rotation, shape, ODE properties} |
| Derived properties: | {Contact joints, $C_C$ (current colliders), $C_N$ (new colliders), $C_P$ (previous colliders).} |
| Work on *update*(): | Run dSpaceCollide to compute contact joints. Update the current colliders set with objects currently touching. Set new colliders to any new additions to that set and previous colliders to any removals from that set. $C_N \cap C_P \equiv \emptyset$ always holds. |

### 6.3.4 Interactions

The two main areas of ODE have now been covered; what is left to do is to tie the two together again. For this an interaction will be required.

This interaction would conceptually take all contact joints generated in the collisions derived property set and apply them to the rigid body resolver; although as indicated in the introduction of this section, this particular interaction will be interleaved with the ODE collisions DPS.

### 6.3.5 Remarks on dynamics

The implementations of the RBS Resolver and Collisions DPS in the proof-of-concept implementation are by far the messiest, one of the reasons being that they 'evolved' with the methodology. Dynamics was the first simulation to be implemented and it was never revisited. In a re-implementation many improvements could be made.

Although it would potentially lose some modularity, an implementation could gain a lot of clarity if it just provided a dynamics resolver, combining both RBS and collisions. What the collisions DPS would traditionally export would be provided as resolver properties.

Some modularity can still be accomplished by providing more than one version of this dynamics resolver, for example one named ODE+ODE and another ODE+GImpact.

Finally, the setLinearGravity interaction function was added to RBS resolvers as it is such a common scenario that the overhead of the interaction calling a function per object is not worth it. Other gravity resolvers will simply set this to zero and provide their own model.

## 6.4 Implementing linear gravity

Although the dynamics simulation described above has an interaction function to set a global gravity vector, this is really intended as a shortcut for the linear gravity resolver. A separate linear gravity resolver has been created because this is conceptually cleaner as the gravity model can then be changed by replacing the resolver. The linear gravity resolver has been implemented as follows:

| Linear Gravity Resolver | |
|---|---|
| Resolver properties: | {the linear gravity vector $V_g$} |
| Properties read: | $\emptyset$ |
| Properties written: | $\{V_g\}$ |
| Work on *advance*(): | Change the gravity vector if requested. |
| *Interaction functions:* | |
| `setGravity(`$V$`)` - *Sets the direction of gravity.* | |

The resolver for this solution is not much more than a neutral place to store the global gravity vector. An interaction is also required to tie dynamics and linear gravity together:

| (RBS ← Linear Gravity) Interaction | |
|---|---|
| Properties read: | {$V_g$ (Linear Gravity Resolver)} |
| Work on *apply*(): | Simply call setLinearGravity with the value of $V_g$, if the value has changed. An alternative implementation (if setLinearGravity is not available) is to simply call addForce for every object. |

If a more complete gravity model is desired (which could, for example, be capable of numerically solving the three body problem) it is likely that the following steps would be taken: remove the simple gravity resolver and replace it with one that calculates the gravity forces for each object, then exchange the interaction with one that uses the forces calculated and applies them to each object.

## 6.5 Implementing simplified electricity

This subsection describes how the two-wire electricity approach from Chapter A is integrated into this proof-of-concept implementation.

The BGL [Sie00] was chosen to search for block graphs, as it contains an algorithm of linear complexity for static graphs.

### 6.5.1 Properties

The simulation itself needs only two pieces of information: the electrical state of an object and the connectivity graph. Both of them map easily to properties in the methodology. Some additional properties will be essential for a useful simulation, for instance if an object is a conductor or not. For the simulation itself this is irrelevant, since it will assume the connections and disconnections to the electrical graph 'make sense', but for the interactions manipulating the electricity resolver this property is vital.

| Conductivity Property |
| --- |
| *This object property indicates if the object can be a member of the electricity graph. It is used by the (Electricity ← Collisions) Interaction to avoid adding non-conductors to the electricity graph.* |

| Electrical state Property |
| --- |
| *This object property can be 'on' or 'off'; it should be set only by the Electricity Resolver.* |

| $G_E$ (Electricity graph) Property |
| --- |
| *This object property is a simple set of other objects this object is connected to. Together these properties form the electricity graph.* |

### 6.5.2 Resolvers and derived property sets

As with most simulations, this one requires a resolver. Its interaction functions are easily defined: adding and removing a connection in the electricity graph.

It is however not entirely clear cut where the main 'work' of the simulation should happen. Depending on the implementation of the simulation there exist two distinct options:

- Implementation in the resolver. With this approach the electricity graph and electrical state of objects get updated directly as a result of the electricity graph changing.
- Implementation as a derived property set. Which nodes are 'on' or 'off' depends only on the connection graph. The resolver updates the connection graph and the derived property set provides on/off information for every node.

This is a similar dilemma to the one seen with dynamics, although in this instance there is not much difference between the two approaches in terms of the methodology. In practise, the first

approach was much easier to implement in this proof-of-concept work. This gave the following resolver:

| Electricity Resolver | |
|---|---|
| Properties read: | {Conductivity, $G_E$ (Electricity graph)} |
| Properties written: | {Electrical state, $G_E$} |
| Work on *advance*(): | Add and remove connections from the connectivity graph, updating the electrical state of nodes to reflect these changes. |
| *Interaction functions:* | |
| `addConnection`(object, object) - *Link two objects in the electricity graph.* | |
| `removeConnection`(object, object) - *Remove an existing link between two objects from the electricity graph.* | |

### 6.5.3 Interactions

So far, the electricity simulation will not do anything - it will have to be driven by some other means. Perhaps the most obvious way to interact with it is to make use of dynamics and add an edge in the electricity graph between two colliding conductors and remove it once they stop touching.

| (Electricity ← Collisions) Interaction | |
|---|---|
| Properties read: | {Conductivity, $C_N$, $C_P$} |
| Work on *apply*(): | For each new collision between two conductors, add a connection between them. For each separation of two conductors, remove the connection between them. |

## 6.6 Implementing simplified magnetism

Full magnetics simulation is a difficult problem, and most of the details of it will likely be irrelevant for games. This subsection describes an extremely simple approximation of magnetics. The examples contained in this chapter only model simple and intuitive behaviours like the attraction and repulsion of two magnets, a magnet attracting metal objects and electromagnets. The simulation will be restricted to the simple cases outlined; a complete simulation of magnetic fields is not attempted.

Many game scenarios involving magnets will not require the notion of magnetic poles, for instance using an electromagnet to lift a heavy metal object or using a magnet to move an otherwise unreachable metal object behind a solid surface such as a glass pane. This can be simulated simply by applying a certain amount of force between the 'magnet' and the metal object, pulling them towards each other. Since properly simulating magnetic fields is not the main priority, this force will be simplified down to a force proportional to the inverse square of the distance between the objects. In particular, this approach will not take into account the shape of a magnet; each object that is a magnet will have one or two points defined that

represent the magnetic poles. Incidentally this allows the use mono-poles if required as they are less expensive to simulate.

The case where a magnet will both attract and repel another will probably be less useful for games, but might still be interesting for clever puzzles. An application of this could be a compass, which is influenced by nearby strong magnetic fields.

### 6.6.1 Properties

Since all the properties of a real magnetic field are not important in this simulation, a single number will represent how 'magnetic' a magnet is. It will indicate both the strength of the magnetic field of the magnet and how much an object (such as a metal sphere) will be influenced by magnetism. This property will be called 'magnetic strength':

| Magnetic strength Property |
| --- |
| *This property records two things: whether the object is a magnet and the strength of the magnet. If the object is not a magnet but susceptible to magnetic forces it states how strongly it is affected.* |

The position of the two magnetic poles can be specified and whether they are relevant for the simulation. Usually, a simple electromagnet used for lifting heavy metal objects can adequately be modelled by just defining one of them: since the objective is simply the lifting of a heavy metal object, only concern is the magnet attracting the metal object not exactly *how* it is attracted.

| Magnetic poles Property |
| --- |
| *This property defines how many active magnetic poles an object has and their location relative to the object's position property. An object may have zero or one north poles and zero or one south poles.* |

Finally, a flag to enable or disable a magnet will be useful, in particular for electromagnets:

| Magnetic on/off flag Property |
| --- |
| *If this is 'on' the object is a magnet and will exert force on other magnets. If this is 'off' the object can be influenced by other magnets but will itself not attract or repel other objects.* |

It should be noted that such a flag is different from setting the magnet's strength to zero; this would have the effect of the magnet completely losing all its magnetic properties resulting in it no longer being attracted to other magnets.

### 6.6.2 Resolvers

The resolver for this simulation will be very simple, most of the real work will be done in a derived property set. The main purpose of the magnetics simulation is to create a number of forces for each object, which will be fed into the dynamics simulation.

The resolver itself will provide some interaction functions, which will cause simple changes to the properties used in the world step. The magnetics resolver can be summarised as:

| Magnetics Resolver | |
|---|---|
| Properties read: | $\emptyset$ |
| Properties written: | {Magnetic on/off flag} |
| Work on *advance*(): | Simply change the properties as requested by the interaction functions. |
| *Interaction functions:* | |
| `enableMagnet`(object) - | *Sets the on/off flag to 'on' for the specified object.* |
| `disableMagnet`(object) - | *Sets the on/off flag to 'off' for the specified object.* |

### 6.6.3 Derived property sets

The main work (the generation of forces for each object) is similar to the collisions derived property set. From the position, rotation and the magnetic properties of an object the resulting magnetic forces can be derived. Note that this would still be applicable even if implemented in a much more complicated and thorough magnetics simulation.

| Magnetic Forces DPS | |
|---|---|
| Properties read: | {Position, Rotation, Magnetic poles, Magnetic strength, Magnetic on/off flag} |
| Derived properties: | {$F_M$ (Magnetic forces for each object)} |
| Work on *update*(): | For each pair of objects, calculate the attracting and repelling forces and export the accumulated value. |

### 6.6.4 Interactions

The key interaction to make the entire simulation work feeds the forces calculated in the derived property set into the dynamics simulation. The interaction is implemented as follows:

| (RBS ← Magnetic Forces) Interaction | |
|---|---|
| Properties read: | {$F_M$} |
| Work on *apply*(): | Simply add all the forces calculated in the Magnetic Forces DPS to the RBS Resolver. |

In order to support electromagnets, another interaction and an addition to our magnetic properties is required:

| Electromagnet Property |
|---|
| *This property records whether a particular magnet is an electromagnet or not.* |

The following simple interaction is used to implement electromagnets:

| (Magnetics ← Electricity) Interaction | |
|---|---|
| Properties read: | {Electrical state, Electromagnet} |
| Work on *apply*(): | Simply set the on/off flag equal to the electrical state on any electromagnet object. |

## 6.7 Notes on the format of the experiments

The layout of each subsection dealing with an example is roughly the same.

- First, the experiment, why it was chosen and how it differs from the others is described.
- The implementation is then described, including an annotated screen-shot. It should be noted that all images are actual screen-shots; if they are manually modified for clarity then this is explicitly mentioned. This illustration is then followed by a table listing all significant objects in the scene and their properties; and a list of all resolvers, derived property sets and interactions used in this scene.
- This is then followed by a description of each important point in the simulation to illustrate what happens as each experiment is run. The format of this section is a frame-by-frame account of each simulation step, omitting any duplicate step or description (i.e. each subsequent step only describes new events).

  The frame-by-frame descriptions all share the same format. The frame numbers used are arbitrary, however their number with regards to the previous frame is meaningful: an increment of 1 over the previous frame indicates that this frame immediately follows the previous frame; any jump in numbering (such as a jump from frame 5 to 100) simply indicates that "more of the same" happens for a while, i.e. the scenario is fast-forwarded to the next interesting event that occurs.

  Each frame description is split into four sections, each of which is optional and is only included if something new and interesting happens. The four sections are: a general note and a section each on what happens with respect to resolvers, derived property sets and interactions.
- Finally, each experiment is compared to what would be a similar setup in a fictional, conventional script-driven game engine. In particular, the number of scripts required to achieve a similar setup is examined.

## 6.8 Experiment I: a switched electromagnet

### 6.8.1 Introduction

The first experiment is designed to demonstrate all the physics simulations discussed above working together. Two simple objects are modelled, both using electricity: a switch and an

electromagnet. In order to observe the electromagnet in action, a metal object is also introduced, which is intended to be lifted by the electromagnet.

The switch will need dynamics to work, the electromagnet will need magnetism, forces in the dynamics simulation and electricity to work and the object lifted by the electromagnet will need magnetism and dynamics to behave correctly.

The main goal of this experiment is to demonstrate how each of these separate simulations work together, via interactions, to form interesting emergent behaviour.

### 6.8.2 Implementation

This example will demonstrate all the physics simulations, including every single resolver, derived property set and interaction from the previous sections. Figure 6-3 shows an annotated screen-shot of how the scene has been set up.

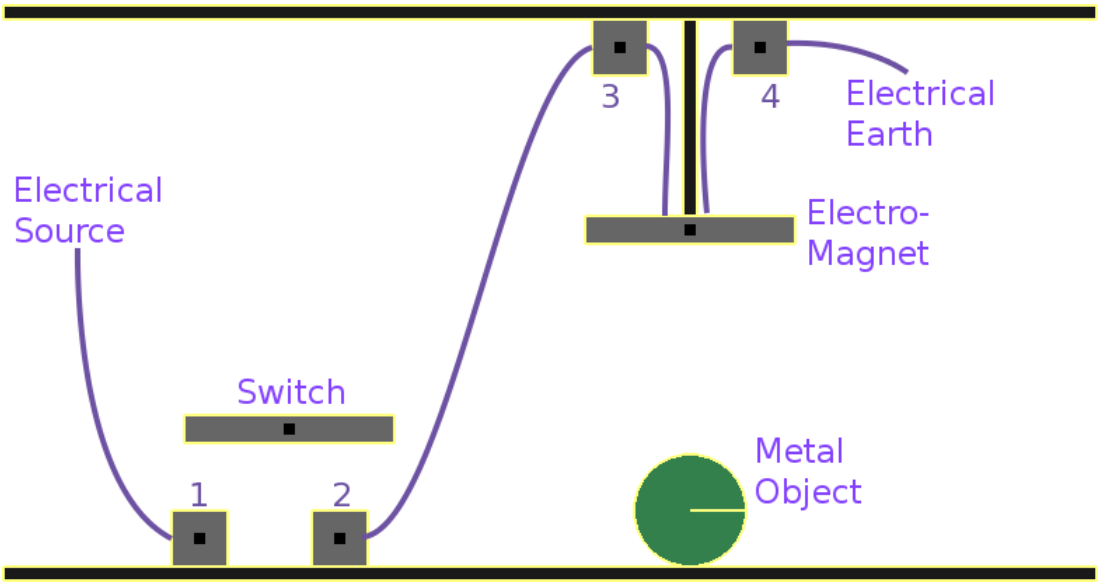


Figure 6-3: A screen-shot of the initial setup of the scene. The figure has been manually annotated, in purple, labelling each object with its name and to show the otherwise invisible connections in the electricity graph and the otherwise invisible source and earth objects.

Objects labelled 1, 2, 3 and 4 are electrical conductors. In the following 'Electrical Source' will be shortened to SOURCE and 'Electrical Earth' to EARTH. 'Electromagnet' will be abbreviated to EM. The purple lines have been drawn onto the image in order to visualise the otherwise invisible electricity graph between the different objects. Although these 'invisible' wires could be modelled by actual wire objects they would add nothing fundamentally different to this experiment.

Objects 3 and 4 are connected to the actual electromagnet itself and model its 'plug'. Applying a current between the two will cause the electromagnet to work. For the sake of simplicity the metal object is the only object in the scene that is affected by magnetism. The

'switch' object is a conductor that will be used to form a connection between objects 1 and 2. The following table shows the all objects involved in the physics simulation:

| Object | Dynamics | Electricity | Elec-Graph | Magnetics |
|---|---|---|---|---|
| Floor/Ceiling | Yes; Fixed | - | - | - |
| Electrical Source | - | Yes; Source | 1 | - |
| Electrical Earth | - | Yes; Earth | 4 | - |
| Object 1 | Yes; Fixed | Yes | Source | - |
| Object 2 | Yes; Fixed | Yes | 3 | - |
| Object 3 | Yes; Fixed | Yes | 2, EM | - |
| Object 4 | Yes; Fixed | Yes | EM, Earth | - |
| Electromagnet | Yes; Fixed | Yes; Device | 3, 4 | Electromagnet |
| Switch | Yes | Yes | - | - |
| Metal Object | Yes | - | - | Yes |

Table 6.1: Object behaviour summary

Fixed objects always maintain their position (are a static part of the world) but can influence other objects, for example via collisions.

As indicated above, this simulation will use all of the resolvers introduced in the previous sections, which are:

- Rigid Body Simulation
- Gravity
- Electricity
- Magnetics

The following two derived property sets will be used:

- Collisions
- Magnetic Forces

Finally, the interactions the system will use are:

- Rigid Body Simulation ← Collisions
- Rigid Body Simulation ← Gravity
- Rigid Body Simulation ← Magnetic Forces
- Electricity ← Collisions
- Magnetics ← Electricity

### 6.8.3 Important simulation steps

#### Frame 0 - Setup

Before the first world step, all the initial properties of the objects are set, such as their position and initial electrical connections. Global properties, such as the gravity vector in the gravity resolver, will also be set. The initial electricity graph will be as follows:

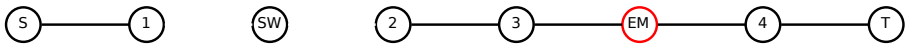


Figure 6-4: Initial electricity graph. The electromagnet is shown is red as it is a 'user' of electricity. The electricity graph is also visualised in the annotated scene setup figure above.

The switch starts out some distance above objects 1 and 2. Once the simulation starts, it begins to fall down because of gravity and eventually closes the electrical contact.

As a reminder, the general sequence of simulation is to first do any work for the resolvers, then update the derived property sets and finally perform the interactions.

#### Frame 1 - Gravity starts to act

Gravity has yet to be introduced; the force defined by the Linear Gravity Resolver must first be applied to every object in the scene, which will happen in the interaction layer. This effect will be resolved in the subsequent frame.

**Work for Resolvers** Since no forces are acting on any of the objects yet, the rigid body resolver has no work to do at this stage. None of the other resolvers have any work to do either.

**Work for Derived Property Sets** Currently there is only one collision (fixed objects do not generate collisions between each other as they cannot move) in the scene: the metal object is touching the floor. An entry is made into the new and current collisions lists, recording this collision.

Since there are no active magnets in the scene, there is no work for the magnetic forces derived property set.

**Work for Interactions** Since the only collision is a simple touching collision and the objects do not intersect, no contact forces (pushing the two objects apart again) have to be taken care of, thus there is no work for the RBS $\leftarrow$ Collisions interaction to do.

The global gravity vector is non-zero and therefore the RBS ← Gravity interaction will add this constant force to any non-fixed object (i.e. the switch and the metal object) in the scene. This will be done for every frame from now on and will not be mentioned again.

**Frame 2 - Metal object does not move**

**Work for Resolvers** The rigid body resolver has two forces to resolve: the gravitational force for both the switch and the metal object. Resolving these will move both objects slightly downwards, the metal object will intersect with the floor by a small distance. (This will get 'fixed' in the next frame via the contact forces generated in the collisions DPS; hence *reactive* collision detection.)

**Work for Derived Property Sets** Since the metal object is now intersecting slightly with the floor, the appropriate (non-zero) contact forces will be generated in the collisions derived property set. The collision of the floor and the metal object will be removed from the new collisions list, but will remain in the current colliders list.

**Work for Interactions** The RBS ← Collisions interaction will feed back all the contact forces generated into the rigid body resolver, forcing the metal object away from the floor in the next frame.

**Frame 50 - Switch closes contact**

Up until now the gravitational force on the metal sphere and the contact forces have kept it lying still on the floor, and the switch object has been falling (and accelerating) downwards. In this frame the switch will have just been moved into contact with objects 1 and 2. For simplicity's sake, it is assumed this collision will absorb all of the energy and the switch does not bounce, although the methodology would also cope with this.

**Work for Resolvers** The rigid body resolver has just moved the switch into a position that makes it intersect slightly with both objects 1 and 2.

**Work for Derived Property Sets** Two new collisions are entered into the new and current colliders lists: the switch colliding with object 1 and switch colliding with object 2.

**Work for Interactions** As usual the (RBS ← Collisions) Interaction will feed back the contact forces into the RBS Resolver.

Since all of object 1, object 2 and the switch are flagged as conductors in their properties, the (Electricity ← Collisions) Interaction now has work to do. Since object 1 and the switch create a new collision and both are conductors, the appropriate interaction function is called in the Electricity Resolver to add this connection to the electricity graph. Similarly for the collision between object 2 and the switch. The electricity graph is not modified at this point.

**Frame 51 - The electromagnet almost activates**

**Work for Resolvers** The rigid body resolver deals with the usual work, now also handling the two new collisions.

The electricity resolver has two pieces of work to do: add a connection between object 1 and the switch and a connection between object 2 and the switch. Doing so will cause the device node that represents the electromagnet to change to 'on'.

**Work for Derived Property Sets** The collisions derived property set will again clear the new colliders list, but the current colliders list will remain as it is, containing three collisions.

**Work for Interactions** Since the electrical state of the device node representing the electromagnet has changed from 'off' to 'on', and the object is flagged as an electromagnet, the appropriate interaction function is called in the magnetics resolver to activate the magnet.

**Frame 52 - The electromagnet activates**

By now the switch has already reached a stable state and is resting on objects 1 and 2. The electromagnet will be turned on in this frame and the resulting force acting on the metal object will be applied in the next frame.

**Work for Resolvers** The magnetics resolver has one magnet to activate because of the interaction function called in the last frame. Doing this is a simple operation, the main work is performed in the magnetic forces DPS.

**Work for Derived Property Sets** There now exists an active magnet in the scene and an object that can be influenced by magnetism is close enough to it to be affected. The magnetic forces derived property set will determine the strength of the force pulling the metal object upwards towards the electromagnet. The scene is set up so that this force is stronger than the gravitational force acting on the metal object.

**Work for Interactions** The RBS $\leftarrow$ Magnetics interaction will apply the magnetic force calculated in the magnetics DPS to the metal object by calling the appropriate interaction function in the rigid body resolver. This force, similar to the gravity force, will be applied every frame (although its direction and magnitude will change frame by frame), until the electromagnet is turned 'off' again.

**Frame 53 - The metal object is lifted**

**Work for Resolvers** The rigid body resolver will incorporate this new force acting on the metal object. As this force overpowers the gravitational force also acting on the metal object, the RBS Resolver will start to slowly move is upwards.

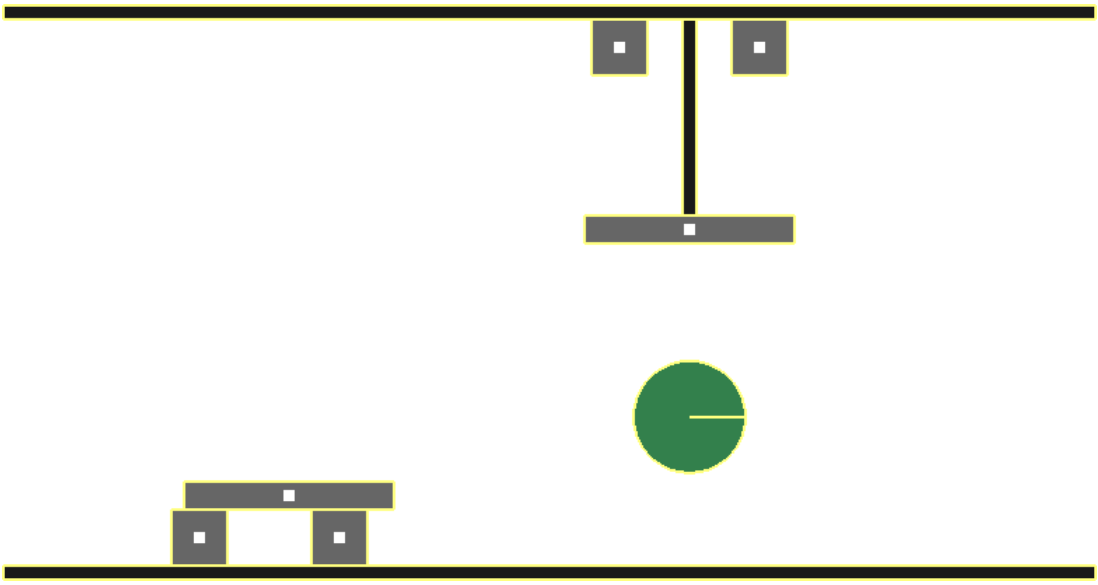

Figure 6-5: This figure shows the closed switch and thus active electromagnet pulling the metal object upwards.

**Work for Derived Property Sets** Since the metal object and the floor are no longer colliding, the corresponding collision is removed from the current colliders list; this change is also indicated on the previous colliders list. (This previous colliders list is cleared in the next frame, similarly to the new colliders list.)

Since the position of the metal object has changed, the magnetic force acting on it will be recalculated in the magnetic forces derived property set. This will happen every frame from now on, until the metal object no longer moves (i.e. is completely stuck to the electromagnet in this case).

### Frame 100 - The metal object collides with the electromagnet

By now the metal object has moved upwards enough to intersect with the electromagnet. Again, it is assumed that the collision absorbs all of the energy and the metal object does not bounce off the surface of the electromagnet (Figure 6-6).

**Work for Resolvers** The rigid body resolver has just changed the position of the metal object so that it intersects with the electromagnet.

**Work for Derived Property Sets** This new collision is handled in the usual way; contact forces are generated and the collision is added to both the new colliders list (for a single frame) and the current colliders list.

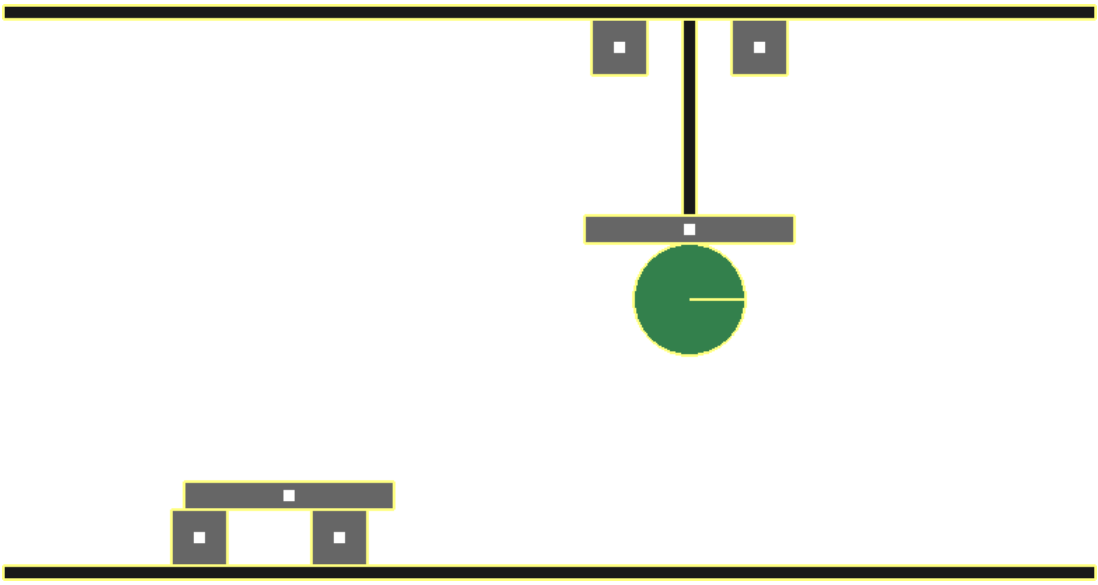

Figure 6-6: This figure shows the final, stable state at which the simulation will arrive in if nothing else changes. The electromagnet continues to exert a force on the metal object holding it in place.

### 6.8.4 Results and discussion

The previous section demonstrates that the basic setup and simulation framework works. Given the requirement from above "make a switch that turns on an electromagnet that lifts an object", this section examines the effort required to set this up in a more conventional scripted environment.

The modelling effort would be less, as most of the special properties such as 'conductivity' would not have to be assigned to each object. However a script is required to make the electromagnet work. Since the electromagnet is triggered by collision, it would make sense to attach a script to an 'onCollision' event of the switch object. For example:

```
event onCollision(self, other):
    if other not in (world.objects['1'], world.objects['2']):
        return

    if (world.objects['1'] in world.colliders[self] and
        world.objects['2'] in world.colliders[self]):
        world.dynamics.addConstantProportionalForce
          (world.objects['EM'],
           world.objects['METAL␣OBJECT'],
           DISTANCE_INVERSE_SQUARE,
           newton(-500.0))
```

Such a script is typical. Each such switch object will require a similar script, which has

encoded into it the logic, the relevant objects and their scripted behaviour. The script is also rather inflexible and hard to maintain. However, it is trivial (and thus quick) to implement. Furthermore, the game in this instance does not require a magnetics simulation as that behaviour is also encoded in the script.

However, its biggest problem is exactly what the methodology intends to resolve. The user could interact with the scene in a few ways that were not necessarily part of the specification of the original scene:

- The electromagnet can be turned off again by lifting the switch object. *This would require another script in our example, attached to an onSeparation event; or a modification to the current script assuming the game engine provides that if the collision event was a 'touch' or 'separation' event.*

- The switch could be disconnected from the electromagnet, causing the electromagnet to stop functioning, irrespective of the switch state. *This would require both a modification to the original q script(s) in order to do 'nothing' if a special designated wire object has been damaged or removed. It also requires another script connected to an onDestroy event of the wire to turn off the electromagnet.*

- The switch could be removed and a different conductor in the scene could be used to close the contact instead. *Perhaps the easiest solution in a conventional setup is to attach the switch script to every single viable switch. However this requires the level designer to manually identify each object that could be used as a switch. Thus no new scripting effort is required, but the process of manually working out which objects could be switches is tedious at best and the level designer is likely to miss some.*

- The electromagnet could be used to attract another metal object on the scene. *This requires the above script to be extended to list every object that can be attracted by the magnet.*

Using the methodology, all four of these will work without further effort in the scene as set up in this experiment as the overall behaviour is not scripted but emergent. It should be noted that the world must of course be designed to 'work'.

## 6.9 Experiment II: a doorbell

### 6.9.1 Introduction

The second experiment is based on the same simulations already used in the previous experiment, however this time the focus is to model a more complex compound 'object'. A doorbell was chosen for this, as it is a non-trivial object with complex overall behaviour made from more than one part. This overall behaviour - the bell rings if electricity is supplied - is not scripted but is purely an emergent behaviour.

### 6.9.2 Implementation

The scene as set up can be divided into two main parts: a switch, modelled as in the previous example by three objects, and the doorbell itself, modelled by four objects.

The doorbell consists of a bell and striker. The striker closes the electrical circuit if it is in its resting position. An electromagnet will pull the striker towards the bell causing them to collide and generating some noise. This action breaks the contact between the object the striker rests on (referred to as the 'contact' object) and the striker itself. This deactivates the electromagnet, reducing the force it exerts on the striker to zero (although not instantaneously; this time period is referred to as the 'cool-down'). Due to the way the striker is hinged, gravity will cause it to fall back into its resting position, closing the contact once again. The process of striking the bell will be repeated indefinitely. Figure 6-7 shows the initial setup of the scene.

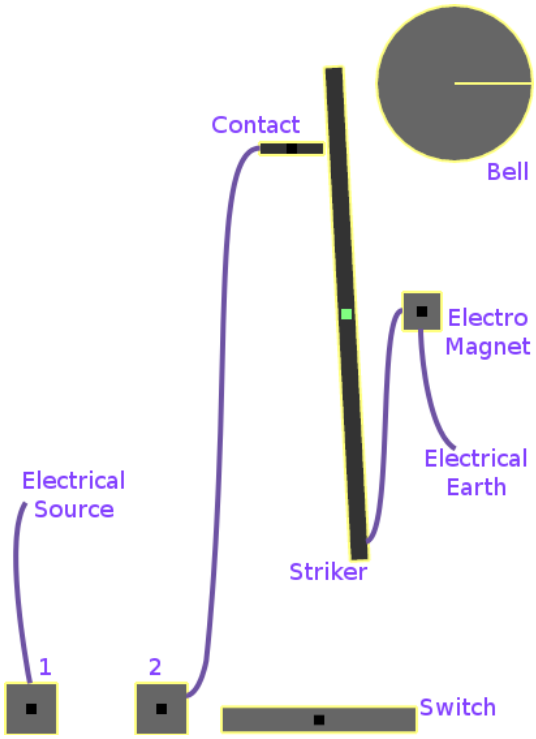


Figure 6-7: A screen-shot of the initial setup of the scene. The figure has been manually annotated, in purple, labelling each object with its name and to show the otherwise invisible connections in the electricity graph and the otherwise invisible source and earth objects.

The following table shows all objects involved in the physics simulation:

| Object | Dynamics | Electricity | Elec-Graph | Magnetics |
|---|---|---|---|---|
| Electrical Source | - | Yes; Source | 1 | - |
| Electrical Earth | - | Yes; Earth | EM | - |
| Object 1 | Yes; Fixed | Yes | Source | - |
| Object 2 | Yes; Fixed | Yes | Contact | - |
| Switch | Yes | Yes | - | - |
| Contact | Yes; Fixed | Yes | 2, Striker | - |
| Striker | Yes; Hinged | Yes | Contact, EM | Yes |
| Electromagnet | Yes; Fixed | Yes | Striker, Earth | EM |
| Bell | Yes; Fixed | - | - | - |

Table 6.2: Object summary

As in the previous example, fixed objects always maintain their position but can influence other objects, for example via collisions. This example will use the same set of resolvers, derived property sets and interactions as the previous example, plus the following interaction:

- Sound ← Collisions

This (Sound ← Collisions) Interaction is a trivial interaction (created for this example) that will play a pre-recorded ringing sample whenever an object collides with the bell.

### 6.9.3 Important simulation steps

These descriptions will not detail the basic dynamics, as they are fundamentally the same as in the previous example.

#### Frame 0 - Setup

As in the previous example, before the first world step all the initial properties of the objects are set as are global properties such as the gravity vector.

Since the main focus of this example is the operation of the doorbell and the initial closing of the switch will be identical to that in the previous example, these frames will be skipped. The first frame examined will be the one after the switch has been closed.

#### Frame 100 - The switch is closed

The switch made contact with both objects 1 and 2 in the previous frame and the (Electricity ← Collisions) Interaction has called the relevant interaction functions in the electricity resolver. The consequences of this will be resolved in the current frame.

**Work for Resolvers** The electricity resolver has two new connections to deal with: one between object 1 and the switch and the other one between the switch and object 2. Resolving them leads to the electricity graph shown in Figure 6-8.

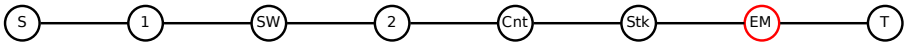


Figure 6-8: The electricity graph just after the switch has been closed and thus activating the electromagnet. Again, the electromagnet is shown in red as it is a 'user' of electricity.

**Work for Interactions** Since the electromagnet is now in a closed electrical circuit, the (Magnetics ← Electricity) Interaction will make this object a magnet by calling the appropriate interaction function within the magnetics resolver.

**Frame 101 - The EM starts to work**

This step is identical to when the electromagnet was activated in the previous example.

**Frame 102 - The striker starts to move**

The current state of the scene is shown in Figure 6-9.

**Work for Resolvers** The rigid body simulation will apply the supplied force to the striker, which will cause it to move towards the electromagnet (and ultimately the bell).

**Work for Derived Property Sets** Since the striker has moved away from the contact object, the collisions DPS will recognise that the striker is no longer colliding with the contact, remove this collision from the current colliders list and place it on the previous colliders list.

Since the electromagnet has not been deactivated yet and the position of the striker has changed, a new force between them is calculated.

**Work for Interactions** As in the previous frame, the (RBS ← Magnetics) Interaction will apply the force calculated in the magnetic forces DPS to the striker.

Since the contact and the striker are no longer colliding, the (Electricity ← Collisions) Interaction will call the disconnect interaction function within the electricity resolver to remove this connection.

**Frame 103 - The connection is severed**

**Work for Resolvers** The electricity resolver has been instructed to remove the connection between the contact and the striker from the electricity graph. Doing so will break the circuit and set the state of the electromagnet to 'off' once again. Figure 6-10 shows the electricity graph at this stage.

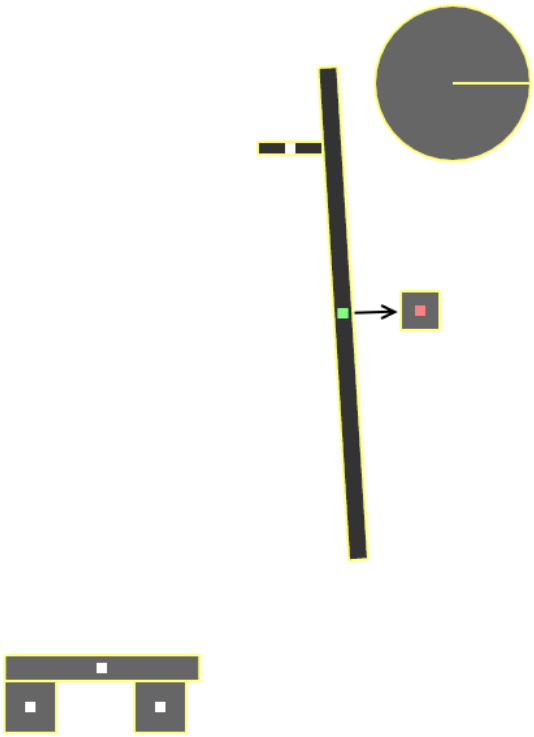

Figure 6-9: This annotated screen-shot shows the state of the simulation just after the electromagnet has been activated by closing the switch. The figure has been manually annotated with an arrow to visualise the force exerted on the striker by the electromagnet.

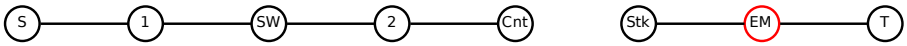


Figure 6-10: This figure shows the electricity graph of the simulation just after the striker has started to move away from the contact. The connection between the striker and the contact is broken as there is no more physical contact between the two.

**Work for Interactions** Since the electrical state of the electromagnet has changed, the (Magnetics ← Electricity) Interaction will instruct the Magnetics Resolver to rapidly (but not instantly) reduce the strength of the electromagnet to zero. This gradual 'cool-down' is both to make the simulation more stable and also to more accurately model real-life magnets, which also exhibit this behaviour.

**Frame 110 - The striker collides with the bell**

The force applied to the striker from the electromagnet during the first few frames was large enough for the striker to build up sufficient momentum to collide with the bell at a high velocity. Figure 6-11 shows the current state of the scene.

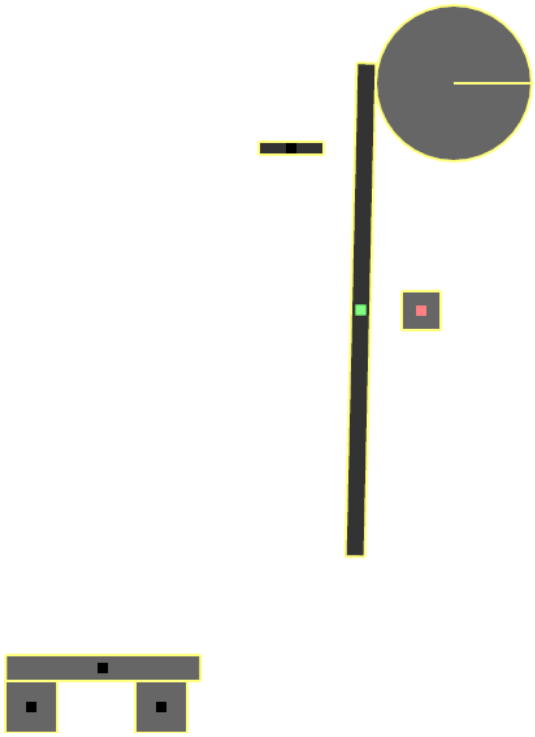

Figure 6-11: The state of the simulation just as the striker hits the bell. Since sound cannot be drawn, the reader is asked to imagine a 'ding' noise at this point.

**Work for Derived Property Sets** The collisions DPS will record the new collision between the bell and the striker and, as usual, will generate the appropriate contact forces.

**Work for Interactions** Since an object has collided with the bell, the (Sound ← Collisions) Interaction will cause a pre-recorded bell sample to play.

### 6.9.4 Results and discussion

As with the previous experiment, the scripting effort required to replicate the behaviour is now examined. If the original requirement is "pressing a switch will cause a bell to ring" then the simplest solution is a script such as this, attached to the 'activate' event of a switch; this activation is usually accomplished by the player facing the object and pressing a special key:

```
event onActivate(self):
    world.sound.emit('snd/doorbell.ogg',
                     world.object['BELL'].getPosition())
```

This would cause the 'doorbell' sample to be played at the position of the bell object. Because the script is written in the way it is, you cannot, for example, keep pressing the bell button and have continuous ringing. Should this be necessary then a pair of scripts is required, one to turn on the sound and one to turn it off.

As with the previous experiment, there are some interactions that may lie outside the initial set of requirements but are still possible with the scene as set up.

- Hitting the bell with the switch, which will also produce a ringing noise. *In the scripted equivalent, another onCollision script can be added to implement this.*
- Turning the doorbell on and then wedging an object between the bell and the striker. This will cause the striker to hit the other object instead, effectively silencing the bell. *To make this work reliably the bell would have to be scripted to operate in a similar manner as set up in this experiment. Instead of emitting a noise, the onActivate script will cause the striker object to undergo a periodic motion. A separate onCollision script will cause the bell to ring if struck.*
- As above, but placing a non-conducting object between the striker and the contact. This will also silence the bell by not allowing the electrical circuit to be closed. *To make this work reliable the entire setup needs to me made even closer into what the methodology naturally provides. The script activating the bell would toggle a flag on the striker object. If the striker is currently touching the contact a strong impulse is applied to it, causing it to hit the bell. A onCollision script would apply the same impulse every time time the striker touches the contact, as long as the above mentioned flag is set.*

As more and more features are added, the scripted example begins to emulate what happens in the methodology. However, in the latter as long as the world is constructed to be functional in the first place everything else will work due to emergent behaviour.

If most of the setup as described above were to be replicated in scripts, the following would be the result.

```
# Attached to the switch
event onActivate(self):
   # Set flag
   world.objects['STRIKER'].setUserFlag('active', True)
   # Get the striker moving
   if (world.objects['STRIKER'] in
         world.colliders[world.object['CONTACT']]):
      world.dynamics.addDirectedImpulse
        (world.objects['STRIKER'],
         world.objects['STRIKER'].anchor('tip').getPosition(),
         world.objects['BELL'].getPosition(),
         newton(100.0),
         seconds(0.2))

# Attached to the switch
event onDeactivate(self):
   # Clear flag
   world.objects['STRIKER'].setUserFlag('active', False)

# Attached to the striker
event onCollision(self, other):
   if (other is world.objects['CONTACT'] and
       self.getUserFlag('active')):
      world.dynamics.addDirectedImpulse
        (world.objects['STRIKER'],
         world.objects['STRIKER'].anchor('tip').getPosition(),
         world.objects['BELL'].getPosition(),
         newton(100.0),
         seconds(0.2))

# Attached to the bell
event onCollision(self, other):
   if (world.dynamics.getCollision(self, other).getForce() >= 20.0):
      world.sound.emit('snd/doorbell.ogg',
                       self.getPosition())
```

## 6.10 Experiment III: an electric motor

### 6.10.1 Introduction

The third experiment will also focus on a 'compound' object, similar to the previous experiment. The object chosen is an electric motor; another everyday object, which uses electricity and magnetism to operate. The setup in this experiment is different from the previous experiment in the following ways:

- The motion exhibited by an electric motor is more involved than that of the doorbell, as the goal is continuous smooth motion.
- The electrical wiring in an electric motor is more involved than the wiring present in the doorbell, as at least two alternating circuits have to be provided.
- The distinction between 'north' and 'south' poles of magnets is important. In the doorbell experiment either a mono-pole magnet of either polarity or a dipole magnet oriented in any way could have been used. In this experiment the polarity of the magnets is important.

### 6.10.2 Implementation

The electric motor will be modelled using the same basic ideas and physics used in the previous two examples. Due to the much more complex nature of this experiment, the scene setup is more involved as well since it has to work around some limitations in the physics model:

- It is not trivial to create good sliding contacts with ODE, thus a simple approximation of one is used.
- The electricity model does not handle AC or distinguish between positive and negative poles (as per its assumptions). This makes it impossible to create an electromagnet that changes polarity. Instead, two mono-pole electromagnets of opposite polarity are used and power is supplied to them in turn.

Figure 6-12 shows the initial setup of the scene. The sliding contact is hinged directly above the main motor and will rotate clockwise and fall into a vertical position once the simulation is started. It is connected directly to the electrical source.

The design of the motor used here differs somewhat from a conventional Direct Current (DC) motor. This illustrates that, even with a "non-obvious" solution, correct behaviour can emerge. The design is a sound one that would not necessarily have been anticipated by a game designer.

Around the main axis there are eight objects (called 'spokes' for the purpose of this simulation) that serve as both contacts and static (mono-pole) magnets. The spokes are joined to the main axis so that they rotate along with it. The main axis has a dampening force acting on it so that, should the main electrical circuit be broken, the rotation of the motor will slow down and eventually stop.

To the right of the motor there are two mono-pole electromagnets that model a single electromagnet of changing polarity. Depending on which of the spokes the sliding contact is

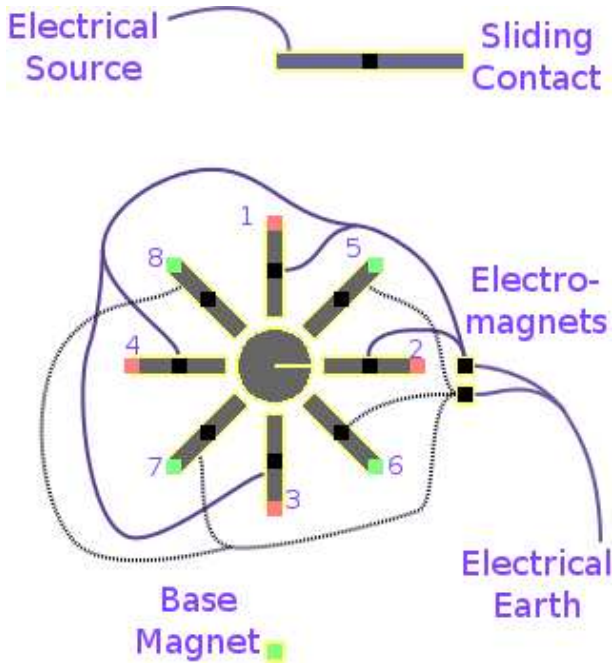


Figure 6-12: A screen-shot of the initial setup of the scene. The figure has been manually annotated, in purple, labelling each object with its name and to show the otherwise invisible connections in the electricity graph and the otherwise invisible source and earth objects.

touching, one of the two electrical circuits is closed, activating the electromagnet that in turn repels the spoke nearest to it and attracts the one below it.

Below the main motor a static magnet has been placed, called the 'base magnet'. The base magnet ensures that the motor will never enter a stable position, one from which it would never start turning. Figure 6-13 illustrates such a configuration.

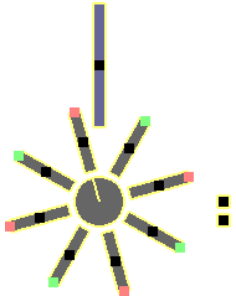

Figure 6-13: Illustration of a stable position arising from the lack of a 'base magnet'. This setup assumes all objects are at rest. The electric motor will never recover from this position unless there is some external stimulus as the sliding contact will never touch one of the spokes again.

In the absence of outside forces, the base magnet always turns the main motor into a position where one of the spokes will collide with the sliding contact and thus close one of the electrical

circuits containing one of the electromagnets at the side.

The following table shows all objects involved in the physics simulation:

| Object | Dynamics | Electricity | Elec-Graph | Magnetics |
|---|---|---|---|---|
| Electrical Source | - | Y; SRC | Sliding Contact | - |
| Electrical Earth | - | Y; ETH | EM-North, EM-South | - |
| Main Axis | Yes | - | - | - |
| Spokes 1 to 4 | Yes | Y | EM-North | N |
| Spokes 5 to 8 | Yes | Y | EM-South | S |
| Sliding Contact | Yes | Y | Source | - |
| EM-North | - | Y; Dev | Spokes 1 to 4, Earth | EM, N |
| EM-South | - | Y; Dev | Spokes 5 to 8, Earth | EM, S |

Table 6.3: Object summary

The main axis is also fixed to the static world with a joint allowing free rotation (but no other movement). As mentioned above, this rotation has dampening forces applied to it every frame, both causing the motor to come to a gradual stop should electricity no longer be supplied and also ensuring that the simulation is stable and the rate of rotation remains stable.

The sliding contact is also hinged to the static world at its left end directly above the main axis of the motor.

This example will use the same set of resolvers, derived property sets and interactions as the first example; the full list can be found in Section 6.8.2 (page 71).

### 6.10.3 Important simulation steps

Once electricity is connected to the motor (which it is by default in this setup) two distinct simulation phases can be identified: the 'start up', in which the motor begins to turn, and 'operation', in which the main axis rotates in a regular and smooth fashion.

Since the basic process of simulation is similar to the two previous examples, a much more high-level description is given in this example.

#### Frames 0 to 29

Initially the sliding contact will rotate around its hinged point due to gravity and will come into contact with the top spoke of the motor.

Once the contact collides with the top spoke the electrical circuit is closed and this activates the top electromagnet. Figure 6-15 shows the corresponding electricity graph.

#### Frames 30 to 59

Since there is some force transmitted from the collision of the sliding contact with the top spoke, the axis will turn slightly anticlockwise anyway. However since the 'north magnet' is now active, it will further push any 'north spokes' away from it, in particular the one closest to it, amplifying the anticlockwise rotation of the main axis.

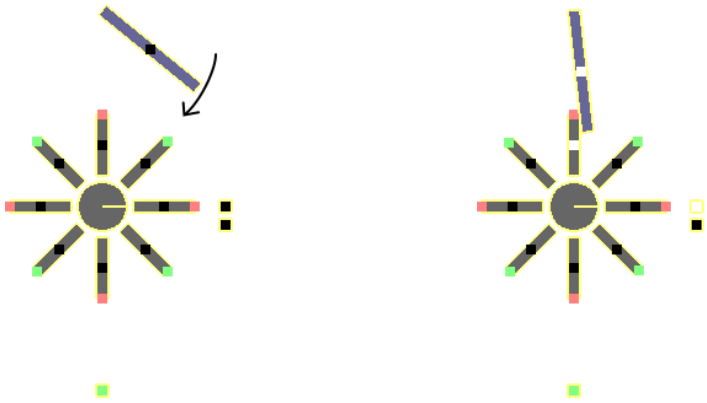

Figure 6-14: The left screen-shot shows the initial movement of the sliding contact due to gravity. It has been manually annotated with an arrow to show the movement exhibited by the sliding contact. The right screen-shot shows the moment the sliding contact collides with a spoke for the first time.

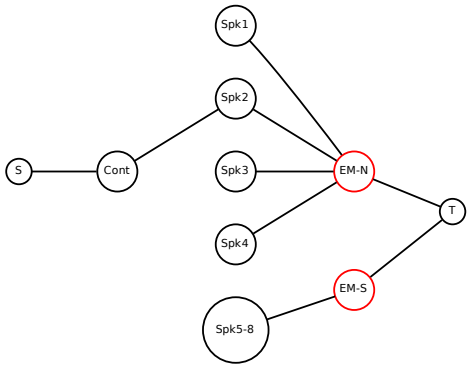


Figure 6-15: The slightly simplified electricity graph of the scene. Spokes 1 to 4 are all represented by an individual node, spokes 5 to 8 have been summarised into a single large node for layout reasons - in the actual graph each will have its own node. As with the previous example scenarios the 'users' of electricity are shown in red.

As the axis turns, the 'south spoke' below the one that was initially closest to the magnets at the right in Figure 6-14 will be the closest south pole to the active north pole of the electromagnet and thus there will be a large attracting force between them, again re-enforcing the anticlockwise motion. Figure 6-16 illustrates these forces. Although magnetic forces act on all the other spokes, due to the inverse square law (under which magnetic forces fall), they are not significant in comparison to the two main forces.

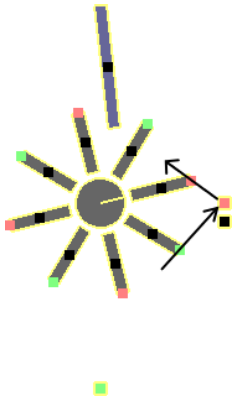

Figure 6-16: Illustration of the two main magnetic forces occurring in frames 30 to 59. Red dots represent active 'north' poles and green dots represent active 'south' poles. Previously the slider has been in contact with the 'north' spoke just to the left of it, causing the 'north' electromagnet to push the 'north' spoke above it away from itself and attract the 'south' spoke below it. This causes the entire construct to turn counter-clockwise. This figure has been manually annotated with two arrows visualising the two most important force vectors caused by the electromagnet.

#### Frames 60 to 89

This rotation will continue until the next spoke (this time a 'south' spoke) collides with the sliding contact, activating the other electromagnet (effectively reversing the polarity). This will causing the main axis to continue to turn anticlockwise.

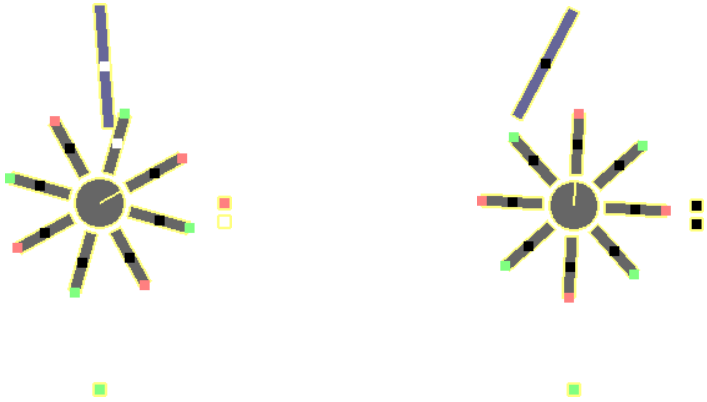

Figure 6-17: These two screen-shots capture what happens between frames 60 and 89. In the left image a spoke of the counter-clockwise rotating motor has just hit the sliding contact, pushing it out of the way. The state after a short time can be seen in the right image.

This basic process will repeat for a while (and the initial motion will be a bit jagged) until the simulation settles into a smooth rotation.

**Frame 200 onwards**

The simulation has now reached a state of constant and steady anticlockwise motion. Figure 6-18 shows snapshots of approximately one eighth of a full rotation of the main axis.

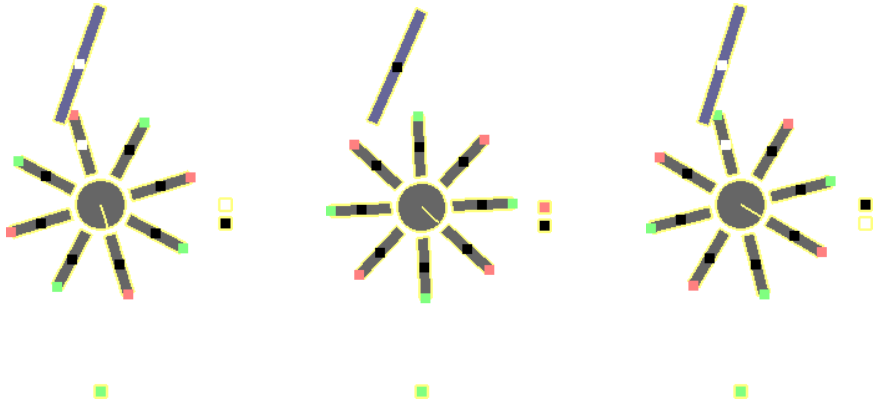

Figure 6-18: Further illustration of the stable rotation of the motor. In the left the sliding contact has just made contact causing the relevant electromagnet to activate, in the middle the rotation has advanced a bit and in the right the sliding contact has hit the next spoke.

### 6.10.4 Results and discussion

It is also possible to model the electric motor with just four spokes. However, the start up phase in such a configuration is a bit longer and the motion during the operation phase is far less smooth, since the impulse of the sliding contact colliding with the spokes is greater as the sliding contact has more space to build up momentum.

The implementation of this electric motor is more a proof of concept than anything else. In a simulation that required many robust electric motors (for instance a futuristic car racing game which exclusively used environmentally friendly electric cars) it would be far more beneficial to 'cheat' and give specific objects an 'electric motor' property. This property would, via an interaction between electricity and rigid body simulation, cause an object to rotate with a given torque around a specific axis if the object is part of a closed electrical circuit.

This would be much closer to the traditional scripted solution, with the same advantages and disadvantages. In particular it would not necessarily be possible to stop the electric motor by jamming a non-conductor between the sliding contacts. It would however still be possible to stop the motor by jamming the axis itself; as both a script and an interaction would apply a constant force each time-step and if a sufficiently large force opposes it, it would stop to rotate.

Even so, the methodology would allow more complex objects to be built from motors and other objects; or for more complex objects to be broken down and their components used in novel situations without additional scripts.

For scenarios where the inside workings of a motor are important or relevant to problem solving within the game, with some thought it is, as demonstrated, possible to design and model

a working electric motor. It follows that it is also possible to for the world designer to set out a puzzle that has the player searching for components for an electric motor and then assembling or repairing it, as opposed to just encountering assembled electric motors.

It should also be noted that due to the way electricity and magnetics are simulated in this example, the electric motor modelled in this experiment cannot be used as a generator as it is the case for normal electric motors. This is because there is no concept of 'induction' in the simulations implemented. The easiest way to model a generator with the set of physics simulations as described in this chapter is to create a special interaction that will make a particular object a conductor if it rotates at a certain velocity and then link it directly to the electrical source, causing it to 'emit' power for all intents and purposes.

Finally, during implementation, it was found that if an electromagnet is rapidly turned on and off a much smoother and more realistic simulation is achieved if its strength is quickly reduced to zero over a few frames, as opposed to instantly. This mimics the self induction in real coils causing the field to collapse more slowly. Applying this technique to the previous doorbell example also resulted in an overall smoother simulation. For the first experiment, the electro-magnet, there is, as expected, no visible difference as the electromagnet is not turned rapidly on and off.

## 6.11 Implementing simplified buoyancy

The last experiment presented in this chapter requires a buoyancy model to be added to the implementation. The generic requirements for the buoyancy model are:

- An ability to mark an object as a volume of liquid.
- The provision of a single force attached at a certain point (the centre of buoyancy) for each object suspended or submerged in the liquid representing the overall buoyancy force.

There are other useful features that take into account more properties, such as calculating the drag exerted on the immersed objects, but the overall output of a buoyancy simulation is always the same: a single force vector for each object partially or fully submersed.

The buoyancy force ($F_B$) for an object is directly proportional to the volume of liquid ($V_L$) it displaces. $d_L$ is the density of the liquid and $g$ the acceleration due to gravity.

$$F_B = V_L * d_L * g \tag{6.1}$$

An object is suspended in a liquid if it displaces exactly its own weight in liquid, as in that case the buoyancy force is identical to the gravitational force.

The buoyancy model chosen is a simple approximation that works well for any 3D (or 2D) model. It is based on a particle system that approximates the volume of any object with a number of small spheres [Fag]. It is easy to calculate the volume and centre of buoyancy of a sphere intersecting with a plane; the buoyancy force on the object itself is simply the sum of

all the buoyancy forces of each of the particles. Its centre of buoyancy is the mean of all the particles' centre of buoyancy, which again is easy to calculate for a sphere.

The main disadvantage of this approach is that it is not completely accurate and it requires a pre-processing step (and thus would not be ideal for soft bodies). It is however somewhat easier to implement than other solutions and works well with most static shapes.

There are slightly better approaches, for example using tetrahedrons to approximate the volume of the object which usually yields more accurate and stable results [Cat06].

Within this implementation, buoyancy has been implemented using a few properties, a resolver, an extension to the Collisions DPS and two interactions (one of which is implicit and implemented via a shortcut).

### 6.11.1 Properties

The implementation of this simulation requires three properties:

| Buoyancy particles Property |
|---|
| *This property models the volume of the object using a number of spherical particles. They can be fully submerged, fully out of the water or any fraction in between.* |

| Centre of buoyancy Property |
|---|
| *This is the centre of buoyancy, i.e. the centre of mass of the submerged volume. (For example if a box is exactly halfway submerged, then the centre of buoyancy is in the middle of that submerged half.)* |

| $F_B$ (Buoyancy force) Property |
|---|
| *This is the buoyancy force acting on each object. This force is intended to be applied to the centre of buoyancy of an object.* |

### 6.11.2 Resolvers

The following resolver (with a fixed assumption of $g = 9.81ms^{-2}$ and $d_L = 1.0$) was implemented:

| Buoyancy Resolver | |
|---|---|
| Properties read: | {Buoyancy particles} |
| Properties written: | {Buoyancy particles, Centre of buoyancy, $F_B$} |
| Work on *advance*(): | Calculate buoyancy force as described in 6.1. Interaction functions are used to modify the buoyancy property indicating how many of the particles are submerged. This resolver also calculates the centre of buoyancy. |
| *Interaction functions:* | |
| `setBuoyancyParticles`(object, data) - *Updates the Buoyancy particles property indicating which particles are submerged.* | |

### 6.11.3 Derived property sets

The Collisions DPS was augmented to also provide a per-object buoyancy property that records how many of the particles are suspended.

### 6.11.4 Interactions

The following interaction is used to communicate with the RBS Resolver:

| (RBS ← Buoyancy) Interaction | |
|---|---|
| Properties read: | {$F_B$, Centre of buoyancy} |
| Work on *apply*(): | Apply the forces at the centre of buoyancy using addForceToPoint. |

## 6.12 Experiment IV: a raft

### 6.12.1 Introduction

The fourth and last experiment presented in this dissertation was chosen for two reasons: to show that it is not difficult to add an entirely new simulation to the existing ones and to explore the concept of out-of-order updates.

This experiment makes use of two physical simulations: dynamics and the buoyancy simulation introduced in the previous section.

Conventionally, adding a feature such as buoyancy requires deep changes to an existing dynamics simulation; however using the framework provided by the methodology reduces the changes required.

### 6.12.2 Implementation

The following experiment will model a simple raft floating on a body of water. Figure 6-19 shows the initial setup of the scene. Two flotation devices will keep the raft floating and there are also a number of dense objects, which do not float, that can be used as cargo.

In Figure 6-19, the long rectangular box represents the raft. It is attached by one fixed joint each to the two flotation devices. The long horizontal line represents the water level. All other boxes represent 'cargo boxes'. Each of them is very heavy, compared to their displacement volume, and thus the buoyancy forces acting on each of them are negligible.

As in the previous experiments, the user will be able to interact with this scenario. In particular, it is possible to pick up some of the cargo boxes, which are initially submerged, and place them on the raft, allowing the user to observe the effect on the raft's buoyancy. (This approach was chosen for simplicity, a more complete model could have provided a crane with an electro-magnet, which could be used to lift the cargo boxes and deposit them on the raft.)

The following table shows all objects involved in the physics simulation:

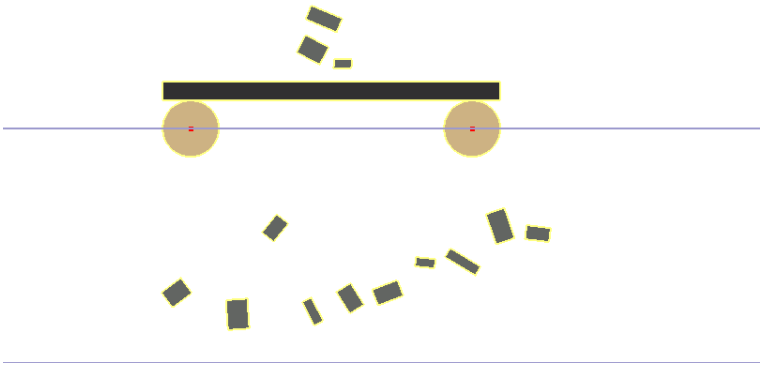

Figure 6-19: A screen-shot of the initial setup of the scene. Unlike the other initial setup screen-shots this has not been annotated as there is no electricity graph to visualise. The raft is modelled by composite object consisting out of a long rectangle with no special properties attached to two flotation devices. The other boxes represent 'cargo'. The dots inside the spheres represent the centres of buoyancy, although it should be noted that as this is the initial frame they have not been set to the correct position yet.

| Object | Dynamics | Buoyancy |
|---|---|---|
| Water Body | - | Fluid |
| Raft Body | Y | negligible |
| Flotation Device A and B | Y, fixed to Raft | Y |
| Cargo Objects 1 to 13 | Y | negligible |

Table 6.4: Object summary

### 6.12.3 Important simulation steps

Since the basic process of simulation is identical to the previous experiment above, this section only includes a high-level description of the simulation.

#### Frame 0

In the starting position seen in 6-19 the centre of buoyancy for the two flotation devices is in the wrong position; it should be in the centre of the submerged volume. This is because the resolver initialises them to the same position as the object's centre of mass.

All cargo objects start in a random position, in this instance three start above the raft and the rest underwater. Once the simulation starts, all cargo objects should fall (or sink) downwards; the ones above the raft will impact with it and cause the raft to be submerged further.

**Frame 1**

After the first simulation pass the collision detection has identified all submerged buoyancy particles. This information is made available to the buoyancy resolver in the next frame via an interaction.

The cargo boxes begin to fall and sink.

**Frame 2**

The buoyancy resolver calculates the buoyancy force for each flotation device and updates the centre of buoyancy for each.

This is fed into the rigid body simulation via an interaction, causing the raft to float.

**Frames up to 200**

In this time the simulation stabilises. Initially the raft bobs up and down but eventually the simulation will find an equilibrium and the raft floats suspended in the water as shown in Figure 6-20.

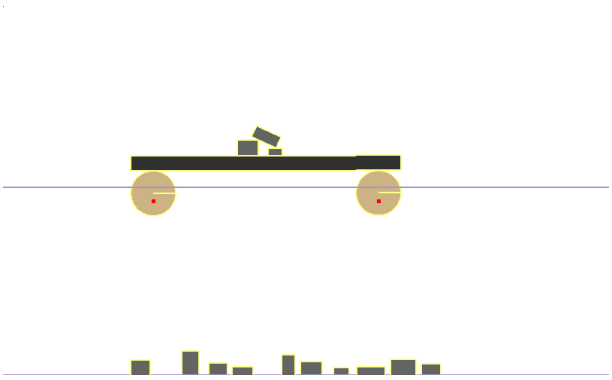

Figure 6-20: Screen-shot of the stable state that occurs after a short time. The centre of buoyancy is now set correctly for both flotation devices and they are buoyant enough to keep the raft and three cargo boxes afloat.

Note that if the buoyancy simulation uses asynchronous execution then this stabilisation phase generally takes a little longer; see Section 7.2.1 and the the end of this section for more information.

#### Applying a new force

This simulation can also cope with new forces added to the raft (no matter the source) in a generic way. Figure 6-21 shows the effect of moving some of the previously underwater cargo boxes and dropping them onto the left side of the raft, unbalancing it. If this is sufficient to cause the cargo to slide off, then the cargo sinks and the raft bounces up and eventually settles in a new stable position.

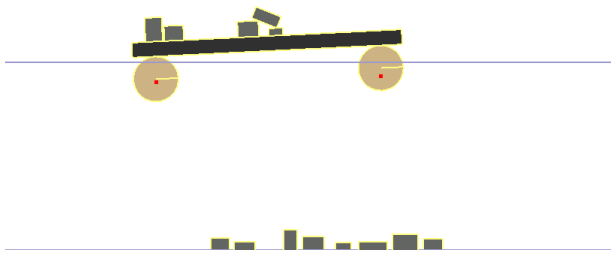

Figure 6-21: This image shows the effect after applying a force to the left side of the raft by means of stacking some extra cargo boxes on top. Since a grater downwards force is acting on the left flotation device the left side of the raft is more submerged than the right one. Were one to continue to stack boxes onto the raft, it would eventually sink or topple, depending on the placement of the cargo boxes.

### 6.12.4 Results and discussion

The implementation of buoyancy could be improved in the following ways:

- Using a more general implementation of buoyancy using tetrahedrons.
- Accounting for drag and convection properly without resorting to dampening.
- Supporting non-horizontal water surfaces.
- Moving intersection of volumes with the body of water into the buoyancy resolver itself.

The last point is an especially good idea because the information important to calculating buoyancy forces is independent of other collision detection. This would also make the implementation itself more independent and flexible. Collision detection is interested in keeping objects from intersecting with each other. Buoyancy only becomes interesting once objects partially or fully intersect with one another.

Conventional collision detection engines do not lend themselves easily to calculate this data. Another good consequence of moving the volume intersection into the buoyancy resolver is that the one frame lag, as shown in Section 6.12.3, is eliminated. However, with the implementation presented above, the behaviour is still plausible and responds appropriately to arbitrary user interaction.

### 6.12.5 Out of order execution

Buoyancy lends itself well to out-of-order execution as the approximation of buoyancy forces can be comparatively expensive. In practise, it does not matter at all if the forces are only updated every second frame. The movement will still appear to be smooth: forces are still applied every frame, however their magnitude changes only every second frame.

The number of frames skipped has then been increased systematically in order to find the points at which it is easy to observe that the simulation behaves differently and at which point the simulation is obviously wrong. A discernible difference first appears if it is executed every 6th frame. Below that it makes little difference. Above 6 frames the simulation still looks mostly believable but the difference in behaviour can be clearly seen. At around 10 frames the simulation has a hard time stabilising and thus appears 'wrong'.

Given how easy and cheap it is to implement out-of-order execution it is definitely something worth doing, in particular if the simulation itself is expensive. Unfortunately it was difficult to measure any difference in this setup, as the buoyancy simulation itself was too fast in this experiment for there to be any statistically significant impact on frame-time.

## 6.13 A better apportionment

As mentioned before, the proof-of-concept implementation is not optimal with respect to following all the ideas of the methodology. It does adhere to the rules, but it could be a lot cleaner. This section proposes a cleaner apportionment of dynamics, electricity, magnetism and buoyancy. It is completely implementation neutral, although it does assume a 2D world.

### 6.13.1 Properties

First, the properties that are primarily used by the dynamics simulation:

| Position Property |
| --- |
| *The X and Y coordinates of the object.* |

| Velocity Property |
| --- |
| *A vector indicating the current velocity of the object.* |

| Angular Velocity Property |
| --- |
| *A single number describing the angular velocity of the object.* |

| Rotation Property |
| --- |
| *An angle describing the rotation of the object.* |

| Mass Property |
| --- |
| *The mass of the object.* |

| Shape Property |
| --- |
| *The shape of the object.* |

The properties that are primarily used by the electricity simulation:

| Conductor Property |
| --- |
| *True if the object can conduct electricity, false otherwise.* |

| $G_E$ (Electricity graph) Property |
| --- |
| *The electricity graph.* |

The properties that are primarily used by the magnetics simulation:

| Magnetic strength Property |
| --- |
| *The magnetic strength of the object and its susceptibility to magnetic forces.* |

| Magnetic poles Property |
| --- |
| *The location of the magnetic poles. Any of the north or south definitions can be undefined. Zero defined poles means the object is simply an object that can be influenced by magnets. One pole models a mono-pole magnet. Two poles model a conventional magnet.* |

| Magnet on/off Property |
| --- |
| *If this is false, then the magnet will not exert any force. This is useful for turning electromagnets 'on' and 'off'.* |

| Is an electromagnet Property |
| --- |
| *This property indicates if an object should behave like an electromagnet.* |

Finally, the properties that are relevant to the buoyancy simulation:

| Density Property |
| --- |
| *The density of the object.* |

| Liquid Property |
| --- |
| *This will flag the object to be a body of liquid.* |

### 6.13.2 Resolvers

This apportionment combines rigid body simulation and collision detection into a single resolver:

| Dynamics Resolver | |
|---|---|
| Properties read: | {Position, Velocity, Angular Velocity, Rotation, Shape, Mass, Liquid} |
| Properties written: | {Position, Velocity, Angular Velocity, Rotation} |
| Resolver properties: | {$C_C$ (Currently colliding objects), $C_N$ (Objects added to $C_C$ this step), $C_P$ (Objects removed from $C_C$ this step)} |
| Work on *advance*(): | Performs both collision detection and rigid body simulation. The 'Liquid' property is read so that collision detection is not performed between liquid and solid objects. |
| *Interaction functions:* | |
| `addForceToPoint`(object, point, force) - *Adds the specified force to the object at the specified point.* | |

No gravity related resolvers are required as both gravity implementations are moved to a derived property set.

The electricity resolver is light-weight as the main work has also been moved to a derived property set:

| Electricity Resolver | |
|---|---|
| Properties read: | {$G_E$} |
| Properties written: | ≡ `read`(Electricity Resolver) |
| Work on *advance*(): | Manipulate the electricity graph as instructed. |
| *Interaction functions:* | |
| `addConnection`(object, object) - *Adds a connection between two conductors in the electricity graph.* | |
| `removeConnection`(object, object) - *Removes a connection between two connected conductors in the electricity graph.* | |

Magnetics simulation is identical to the current proof-of-concept implementation, consisting of a resolver, a derived property set (and an interaction, defined later):

| Magnetics Resolver | |
|---|---|
| Properties read: | ∅ |
| Properties written: | {Magnet on/off} |
| Work on *advance*(): | Change the properties as requested by the interaction functions. |
| *Interaction functions:* | |
| `enableMagnet`(object) - *Sets the on/off flag to 'on' for the specified object.* | |
| `disableMagnet`(object) - *Sets the on/off flag to 'off' for the specified object.* | |

Finally, buoyancy is implemented as a resolver. This implementation will only work with the linear gravity model and requires an interaction feeding it the current global gravity vector. property.

| Buoyancy Resolver | |
|---|---|
| Properties read: | {Position, Rotation, Shape, Density, Liquid} |
| Properties written: | $\emptyset$ |
| Resolver properties: | {Linear gravity vector, $F_B$ (per object buoyancy force), (Centre of Buoyancy (COB))} |
| Work on *advance*(): | Calculate the centre of buoyancy and $F_B$ and exports them as per-object properties. The linear gravity vector (which the model requires) is set via an interaction function. |
| *Interaction functions:* | |
| `setLinearGravity(vector)` - *Defines the linear gravity vector.* | |

### 6.13.3 Derived property sets

Whether a node is on or off can be derived only from the electricity graph:

| Electricity DPS | |
|---|---|
| Properties read: | $\{G_E\}$ |
| Derived properties: | {Electrical state (per object)} |
| Work on *update*(): | This can either be single-wire or two-wire as detailed in Chapter A. The electrical state of an object can be 'on' or 'off'. |

Again, magnetics is identical to the current implementation:

| Magnetic Forces DPS | |
|---|---|
| Properties read: | {Position, Rotation, Magnetic poles, Magnetic strength, Magnet on/off} |
| Derived properties: | {$F_M$ (Magnetic forces for each object)} |
| Work on *update*(): | For each pair of magnets, calculates the attracting and repelling forces and exports the accumulated value. |

Gravity (both linear and real) is implemented in a derived property set. Both DPS are presented:

| Linear Gravity DPS | |
|---|---|
| Properties read: | $\emptyset$ |
| Derived properties: | {$F_G$ (Force due to gravity)} |
| Work on *update*(): | Exports a force due to gravity for each object. The gravity vector is implementation defined and cannot be changed by the physical simulation itself. |

Note that the Real Gravity DPS exports exactly the same set of properties:

| Real Gravity DPS | |
|---|---|
| Properties read: | {Position, Mass} |
| Derived properties: | {$F_G$ (Force due to gravity)} |
| Work on *update*(): | Exports a force due to gravity derived from the position and mass of each object. |

### 6.13.4 Interactions

A RBS ← Collisions interaction is not required, as everything relating to dynamics is handled in the Dynamics Resolver. Both linear and real gravity are implemented with a single interaction:

| (Dynamics ← Gravity) Interaction | |
|---|---|
| Properties read: | {Position, $F_G$} |
| Work on *apply*(): | Applies the force due to gravity to each object at its centre of mass. |

Electricity and dynamics are tied together in the standard way:

| (Electricity ← Dynamics) Interaction | |
|---|---|
| Properties read: | {$C_N$, $C_P$, Conductor} |
| Work on *apply*(): | Connects any new colliding objects (if both are conductors) and disconnects any previous colliding objects (if both are conductors). |

The final two interactions are virtually identical to the ones defined in the proof-of-concept implementation. This interaction will apply the magnetic forces calculated to the dynamics simulation:

| (Dynamics ← Magnetic Forces) Interaction | |
|---|---|
| Properties read: | {Position, Magnetic poles, $F_M$} |
| Work on *apply*(): | Adds all the forces calculated in the Magnetic Forces DPS at the magnetic poles (or the centre of mass if none are specified). |

Finally, this interaction is required to make electromagnets work:

| (Magnetics ← Electricity DPS) Interaction | |
|---|---|
| Properties read: | {Electrical state, Is an electromagnet} |
| Work on *apply*(): | Set the Magnet on/off property equal to the electrical state of any electromagnet object. |

### 6.13.5 Scheduling and parallelism

A number of components in the apportionment proposed above can be run in parallel. All DPS and Interactions can be run in parallel by definition.

The Electricity Resolver and Magnetics Resolver are independent, both from one another and the other resolvers as no other resolver reads or writes any of the properties they write to or read from. The Dynamics Resolver and Buoyancy Resolver cannot be run in parallel because the latter reads a few object properties that the Dynamics Resolver writes to:

$$rProp(Dynamics) \cap rProp(Buoyancy) \equiv \{Position, Rotation, Shape\} \tag{6.2}$$

## 6.14 Concluding remarks

This chapter documented three distinct but closely related items:

- An implementation of the methodology.
- Experiments conducted in the implementation.
- Results showing that a wide range of non-scripted and non-trivial interactions is possible, using nothing but object properties and how the world is constructed.

There was never any question if an implementation of the methodology could be created, rather the question was how well it could be done. One important observation from the implementation presented here is that the correct design decisions are not obvious from the start, both on implementation details and how to structure physics within the methodology itself. A better apportionment has already been documented in Section 6.13 above. Although the "best" method of implementation is dependent upon the final goals of any given project; in the context of the work carried out for this dissertation a more 'true' implementation, i.e. one where any component can be swapped at run-time and that follows the definition of the methodology to the letter, could have been more useful. Section 7.2.1 contains some notes on how this could be accomplished in Python.

Each of the experiments shows that different kinds of physics can be brought to bear on a model, with no need for each kind to know how the others are simulated. The combined effects of the physical forces emerge naturally and in an intuitively obvious way. The work in this Chapter shows that our original aspirations have been achieved; complex objects can be assembled from smaller basic parts and their union gives rise to emergent, non-scripted behaviour.

Each of the experiments includes a comparison to how the initially observed behaviour can be implemented in a more classical, scripted environment along with a list of other potential interactions that are possible in the scenario modelled but not in the obvious scripted implementation. Although each of the items from these lists can be implemented by extending an existing script or adding yet another script; structuring the physic simulation according to the methodology will achieve everything in one step.

# Chapter 7

# Discussions and future work

This Chapter outlines possible future research and implementation work in three areas:

- Work related to the methodology itself and its limitations.
- Implementation work related to the methodology.
- Physical simulations.

This chapter draws primarily from the related work on physical simulations surveyed in Chapter 2 and the implementation described in Chapter 6.

## 7.1 The methodology

This Section will identify areas of the methodology that can be improved upon. This work would be independent of any implementation and may have significant impact on how the methodology works.

### 7.1.1 Scripts

Although one of the primary objectives was to abolish scripts, they still remain useful when applied at the interaction level. This scripting in the interaction layer is different to conventional scripts which hard-code a specific event, i.e. "*this* light switch activates *that* light".

Fundamentally, scripting an interaction is not that different from a 'normal' interaction. It is an implementation of an interaction except that it is interpreted and not machine code. For certain implementations, such as the Python implementation discussed in Section 7.2.1 below, this would not even be different from a standard interaction. Scripting within the methodology will require the following:

- A notation to express a scripted interaction.

- A way of attaching scripts to objects, either by special flags or by assigning the scripts themselves. This could be achieved by a 'Scripts' property that would be a set of interaction scripts that act on this object.
- Some analysis on which interactions it would be useful to implement as scripts.

It is assumed that scripting in the resolver or DPS layer would not be useful, but this assumption should also be explored.

### 7.1.2 Difficult object types

In theory, the methodology is not constrained to any type of object. However many simulations make assumptions as to the nature of objects, the most notable one being *rigid* body simulations such as ODE. Generally in such a simulation other types of objects are not accounted for at all. An example of this, which can be found in the implementation discussed in the previous chapter, is a static body of water; it never moves and its only purpose is to apply buoyancy forces to submerged bodies.

It would be useful if the methodology supported the following types of objects, but some thought is needed to seamlessly integrate them with the methodology:

- Open bodies of fluids and other deformable objects. Examples of fluids would be water in a glass, an ocean, a river or a blood splatter. Examples of other deformable objects include a bouncy and deforming rubber ball or a table cloth. These are traditionally hard problems and expensive to simulate.
- Particle systems. Although they could be modelled by one particle corresponding to one 'object', this would quickly become prohibitively expensive as particle systems generally consist of thousands of particles. They are all related however and usually share a substantial amount of data.

### 7.1.3 Object creation and removal

Certain physical simulations would benefit from the ability to create or destroy objects. For example imagine a simulation of a tap and an empty glass. The user can open the tap to fill the glass. At this point the 'Water in Pipes' simulation would ideally create a particle system representing the water. These particles then gather in the glass.

Currently there is no way to describe the creation or destruction of objects, however the best place to do this would probably be in the interaction layer; either expose the 'world' as a resolver or making available some global interaction functions which are not associated with any particular resolver.

### 7.1.4 Shortcuts

One shortcut has already been outlined and justified in Section 5.7.3; it potentially allows an implementation to interleave the work of an interaction with that of a DPS. Some physical

simulations (packages) are tightly integrated (for example ODE and ODE's collision detection library) and an efficient implementation should take advantage of this. Alternatively a physics simulation could provide a quick and convenient method to implement something related (for example ODE provides a simple shortcut to apply a gravity force to every object in the world). Again, an efficient implementation should ideally be able to use this without breaking the overall structure of the methodology.

It is worthwhile to investigate further possible optimisations and create a formal framework for them. Potential optimisations can be classified as being 'internal' or 'external'.

**Internal** In the instance where one simulation is the back-end for two or more components of the methodology, these components can know about each other and use shortcuts if possible. For example imagine an implementation where the RBS Resolver is backed by ODE, and the system also provides two components for the Collisions DPS: one backed by ODE and the other backed by GImpact. Each one of them should be written so that they can work independently of each other, but the ODE Collisions component could detect the presence of an ODE driven RBS Resolver and take advantage of the fact that they both are backed by the same library and use implementation-specific shortcuts.

This approach is *internal* to the components and takes advantage of a single library being able to provide more than one piece of functionality.

**External** Another approach is to apply some pattern matching and select the best suited component for each specific requirement or the best suited component for a specific system. For example, the user could specify that rigid body simulation and simple linear gravity is required. To fulfil the requirements of the rigid body simulation the implementation then examines the host system and detects the presence of a hardware based solution. The system selects the PhysX component to provide the RBS Resolver and Collisions DPS. On a different host system that lacks the PhysX hardware, ODE might be chosen by the system as a suitable fall-back.

A related example is if the system then tries to select an (RBS ← Linear Gravity) Interaction. The general fall-back interaction, which cycles through all objects in the game world and applies a force to each, is always available. However for the specific instance where the rigid body simulation is provided by ODE, a special shortcut interaction which knows about the ODE function `dWorldGetGravity()` is selected instead.

These patterns could either be encoded into all component implementations so that components "select themselves", or could be encoded into a separate priority list part of that particular implementation of the methodology.

### 7.1.5 LOSD

LOD is a very useful technique for reducing the complexity of a scene. Traditionally it is applied to rendered objects, most notably terrain. However, the concept of LOD is also applicable to

the methodology. Individual simulations may make use of LOD techniques anyway, but with respect to the methodology a different approach is interesting: the transition of one physical simulation to another, which can be more or less general. This concept will be called LOSD.

The methodology should try to formalise this. The following explores this idea with two examples, in order to show what would be useful to achieve.

The effect of rain drops hitting the surface of a largely static body of liquid such as a river or lake is seen frequently in games. A shader is applied to the surface and the illusion of ripples on the surface is created. This is usually fast, as it can be done entirely in the GPU; the geometry of the body of liquid is not actually deformed.

However, this model cannot handle larger objects, such as a heavy rock, hitting the surface of the liquid. A valid use of LOSD would be to fall back on a 2D or 3D fluid dynamics model in the case of larger objects hitting the surface and to only use the shader solution for small objects.

LOSD can also apply to rigid body simulation. For example, imagine a gear box containing three gears in the ratio of 10 to 8 to 15. To simulate this gearbox 'properly' three models of gears, the three axis and the gearbox are required and collision detection and rigid body simulation will take care of the rest. Modelling the gear box this way is the accurate thing to do, as collisions and forces are what make it work in the real world.

However it is equally valid to calculate the ratios of the gears and conclude that the third gear should turn at a ratio of 2/3 of the first gear. Using this calculation in place of rigid body simulation is a good example of LOSD; a more specific model is used to achieve a particular instance of physical simulation. Two non-trivial challenges derive from this:

- How to automatically determine when to fall back to the 'default' (general) simulation.
- How to automatically determine when to use a more specific simulation.

A simple first approach would be to construct larger objects and 'build in' the faster physics simulation and fall back to 'real' physics if necessary. This could be achieved by dropping to the more general solution if any of the individual objects are tampered with in an unexpected way. This only implements a solution to one of the two challenges, but it is a step in the right direction.

The other challenge, determining when to use a simpler physics simulation, is a lot harder and may not be feasible or possible in the general case. However, it may be possible to 'undo' the above if we can detect that the source of interference is gone.

Related to this is the task of working out how to represent both the challenges and their solutions in the methodology. It may be necessary to create a new component that describes the transition of one resolver to another.

### 7.1.6 An additional layer

All physical simulations considered during this work have fitted into the four layers provided by the methodology. It is worth considering whether an additional layer between the properties

and the resolver layer could be introduced. This layer would be identical to the DPS layer, except that resolvers can read from the properties derived in it as well.

This approach may make some implementations easier by potentially eliminating some interactions, but it will not achieve anything that was not already accomplished by this methodology.

## 7.2 Implementations of the methodology

This section will discuss two pieces of implementation work: a better implementation (for research) that will be as close as possible to the specification of the methodology as laid out in Chapter 5 and an implementation in the form of an example game.

### 7.2.1 An implementation in Python

The current implementation, as presented in Chapter 6, is not an unreasonable implementation of the methodology but it is closer to what would be done in a game. An implementation without shortcuts which follows the exact definition of the methodology could be useful when testing concepts that apply to the methodology itself.

I believe that the choice of implementation language is an important one in this case, and a sufficiently flexible and true implementation will require a flexible language. Python [vRF] is such a language and it has many features that would make it a good choice:

- It is already a scripting language, which would make it easy to experiment with 'scripted' interactions as outlined in the previous section.
- It is easy to write bindings to C or C++ libraries (such as ODE) using Cython [BBSC], allowing a variety of physical simulations to be integrated. This also indirectly encourages a clearer separation between the implementation of the methodology and the simulations.
- Its dynamic nature and features, such as reflection and introspection, make it easy to have objects to which properties can be added and removed at run-time. Resolvers, DPS, Interactions and all object properties (as defined in Chapter 5) can be implemented precisely.
- Python is a good language to develop prototypes in and the resulting programs are usually highly maintainable and adaptable.

However, since Python is usually interpreted, its speed cannot be favourably compared to that of a compiled machine language, such as C++ or C; hence such an implementation will not be useful for writing a real game.

#### Scheduling and parallelism

Although Python itself is not multi-threaded as such, external operations (conventionally blocking I/O) can be concurrent [vRF08]. Thus, any resolver or DPS that is implemented externally

can potentially be run in parallel, allowing some limited exploration in that direction. However, the implementation in the form of a game proposed below is better suited to this kind of experimentation.

This proposed implementation can focus on exploring a different kind of optimisation: resolvers and DPS do not necessarily have to be executed every frame. If the results of a physical simulation are generally not required straight after an event has occurred, execution of the component can be scheduled to happen only every $N$th frame. Some initial experiments have been conducted with the buoyancy implementation as described in Section 6.12.5 (page 97); performing the adjustment of forces only every 5 frames still produces a believable simulation.

A variant on this is to make execution dependent on available CPU time (including a constraint so that the simulation is guaranteed to be executed periodically). In a 3D action game, this approach would see a resolver such as the Buoyancy Resolver only run when the rest of the game world is reasonably quiet as rendering large explosions and effects usually requires a substantial amount of CPU time. Furthermore, if large explosions do happen, the player won't be able to tell if the boat it not at rest because of the explosions or because the buoyancy simulation has been given low priority.

Out-of-order execution and on-demand execution may not be useful to every application and every physical simulation, but a strict framework would allow a detailed investigation of useful upper bounds, which in turn can be applied to specific implementations of the methodology as would be found in a complete game.

### 7.2.2 A complete game

A rather different implementation should be an application with a specific goal and a number of assumptions, comparable to those made when writing a real game.

Already mentioned in Chapter 2, in 1992 Sierra Entertainment released an innovative game called The Incredible Machine [Sie92], of which a screen-shot is show in Figure 7-1.

The game featured a simple deterministic rigid body simulation and objects such as ropes and pulleys, along with more esoteric items such as cats, goldfish and monkeys on bicycles. The objective for each level was stated and the user was provided with a number of additional objects that could be placed anywhere in the game world in order to achieve the objective.

A reimplementation of such a game would be an ideal scenario for the methodology. It would allow the overhead of a good from-scratch implementation of the methodology to be measured, and different approaches to implementing the methodology could be compared. Given the number of different physics simulations that would likely be involved, it also provides an ideal test environment to experiment with the various multi-threading approaches.

Another challenge this game in particular would pose is that the simulation must be completely repeatable; multiple runs of a scenario with identically placed objects must result in the same outcome. Note that physical accuracy is not necessary, but *repeatability* of the simulation is.

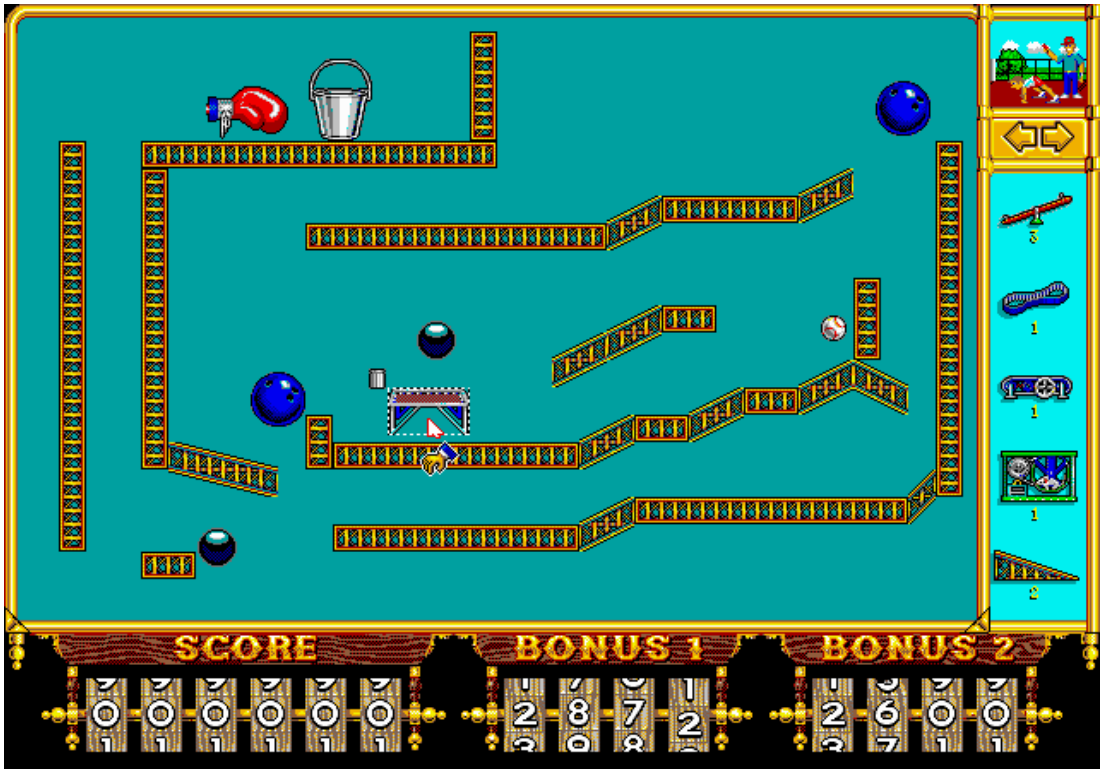


Figure 7-1: A screen-shot from "The Incredible Machine", © 1992, Sierra Entertainment. The objective of the level shown is to somehow move the bucket in the top left. The solution is left as an exercise to the reader.

## 7.3 Physical simulations

This final section will look at various physical simulations, with respect to how they could fit into an implementation of the methodology. Some of these simulations have already been implemented separately and it is only a matter of integrating them into the methodology, others are still rather experimental or not fast enough for interactive applications.

### 7.3.1 New components for existing simulations

As discussed in Chapter 2, there are many rigid body simulators and collision detection libraries available under various licenses, some free and others proprietary.

The basic idea of writing a common interface for a number of physics simulation libraries has been attempted before, for example with libraries such as Gangsta [Tea], PAL [Boe] and OPAL [FRS]. However, an implementation under the methodology would be attacking the problem from a slightly different angle since there are different concerns than just providing a generic API.

#### Rigid Body Simulation

The final goal would be to have a component for most rigid body physics simulations, however in the immediate future there are two particular engines that deserve special attention.

**PhysX** A start up company called AGEIA has implemented part of the Novodex physics simulation package on hardware. NVIDIA bought AGEIA in 2008 and is offering PhysX through their CUDA technology, which is available with all their modern graphics cards.

Writing a methodology component based on PhysX is interesting mainly because it is currently the only hardware based approach to physical simulation and promises to offload a significant chunk of processing time, leaving more compute time for other parts of the framework.

**Havok** Havok enjoys a good reputation among commercial physics engines and is used in a number of games. Writing a component based on Havok is interesting as it is a known good and robust solution for games and will provide a useful yardstick to compare other similar components to.

#### Collision Detection

There also exist a number of collision detection libraries that are independent from rigid body simulation libraries. In particular, GImpact [LtGC] is a popular open source collision detection engine with many useful features for games.

An implementation of the methodology also provides an ideal environment to test new and interesting collision detection algorithms. Of particular interest to larger game worlds are the stochastic collision detection methods mentioned in Chapter 2. They also fit in well with the rest of the methodology as they provide believable data, fast and in a fixed time frame, but not necessarily completely accurate.

### 7.3.2 Future work related to our electricity simulation

This section will examine future work related to the simplified electricity simulation as described in Appendix A.

#### Electrical discharges

Electrical discharges could be modelled with a new interaction. Two conductors placed close to each other will 'connect', producing an arc of lightning. In turn this could create a graphical effect visualising the arcs, light illuminating objects nearby and also some sound.

Moving the objects further apart would maintain the 'connection' until a cut-off point is reached. The electrical connection could also be toggled in some kind of pattern in order to produce a flicker effect for nodes that rely on this connection for power. An implementation would most likely require a simple interaction and some way to produce the visual effect:

| (Electricity ← Proximity) Interaction | |
|---|---|
| Properties read: | {Position, Rotation, Shape, Conductivity} |
| Work on *apply*(): | If the shape and proximity of the objects allow it, create a connection and a suitable special effect between the two conductors. A flag such as 'possible lightning emitter' would stop every random conductor producing such discharges. |

**Induction**

Induction is phenomenon of magnetics. This property can also be implemented with an interaction; concerned with generating electricity from induction. An interesting application of this would be the ability to model a metal detector or to construct a model of a generator.

| (Electricity ← Magnetic Forces) Interaction | |
|---|---|
| Properties read: | {Position, Velocity, Conductivity, 'Inductivity', Magnetic Properties} |
| Work on *apply*(): | Objects flagged as inductors will be connected to the electrical source if they move through a magnetic field. They will be disconnected if they stop moving. |

**Network-flow based approach**

A possible third model for simplified electricity is one based on network flow. Although computationally more expensive, a good implementation could provide a model that can account for some potential difference and some short circuits. It should not be hard to create a prototype, as BGL includes an implementation for static graphs with $O(ve^2)$ complexity [EK72]. Solutions for dynamic graphs also exist, which can be significantly cheaper with $O(ve \log(v^2/e))$ complexity [GT88].

A good implementation of this model would have other applications as well. Given an implementation for the network flow electricity model an accurate model for water in pipes can be constructed. A simplification would be to use the single cable approach that should give similar results, is easier to implement and easier for the level designer to use, but is slightly more expressive.

### 7.3.3 Other simulations

There are a number of physical simulations that have either been treated as a special case in an existing simulation or have not been attempted in game scenarios before. Roughly grouped by simplicity, both in terms of complexity and implementation, these are listed below.

### Easy

The following simulations should be easy to implement and generally are small variations on existing simulations or could be built on top of existing simulations with interactions.

**Simple applications of forces** A number of physical simulations can be represented by simple forces fed into the RBS Resolver. Examples of such small simulations are:

- Explosions, simulated by large forces acting from a point outwards for a short time. The implementation would likely consist out of a Explosions Resolver and an (RBS ← Explosions) Interaction.
- Simple airflow and wind, simulated by a constant force inside a volume (such as a cone). For wind, turbulence can be added to cause objects to tumble. A wind simulation could also make use of shadow volumes in order to determine if an object is affected by wind (which could be done in hardware, reusing existing shadow volumes), or fast hardware visibility algorithms.[1]
- Another application somewhat similar to a wind simulation would be a local anti-gravity field inside a volume, modelling an area without gravity (or even reversed gravity). This could be used to create a futuristic lift in a science fiction setting.

**Real gravity** A gravity simulator for use in space, calculating gravitational forces between large objects. This should not be challenging implement; the DPS and interactions required for this have been detailed in Section 6.13 (page 97).

**Buoyancy** A simple proof of concept model for buoyancy has been implemented, but there is some room for improvement, which has already been briefly outlined in Section 6.12.4 (page 96). The model implemented is based on particles and also assumes a level surface of the liquid and does not cater for other interesting properties such as drag, convection or the compression of objects at high pressures.

There is scope for development work to produce a more general buoyancy simulation, and to make it independent from the Collisions DPS.

Initial experiments suggest that buoyancy is a resolver that is well suited to not being updated every frame. The result from the previous frames (the location of the centre of buoyancy and a force) can be easily reused for a number of frames without noticeable loss in quality. The time gained from periodic execution means that the simulation can potentially be more involved and thus more accurate or complete.

[1] As an aside, shadow volumes could be used for many other things: for example rain in contemporary games appears everywhere, even if you are standing under a porch. A shadow volume, projected from somewhere above the player, could be used to work out which areas near to the player are affected by rain.

### Hard

The problems below are generally not attempted in game scenarios due to the complexity involved. There is ongoing research to develop real-time models for them; these simulations as they are now would be interesting to implement for the sake of completeness or for the sake of stress-testing the system.

**Open bodies of fluids** Simulation of fluids is a difficult problem and there exist several approaches. Initially 2D approaches have been used to simulate waves or splashes resulting from drops of liquid. They effectively create a simple height map, which is then applied to a mesh, but their main limitation the impossibility of modelling 'overhangs' such as the crest of a breaking wave or water pouring into a cup, as illustrated in Figure 7-2.

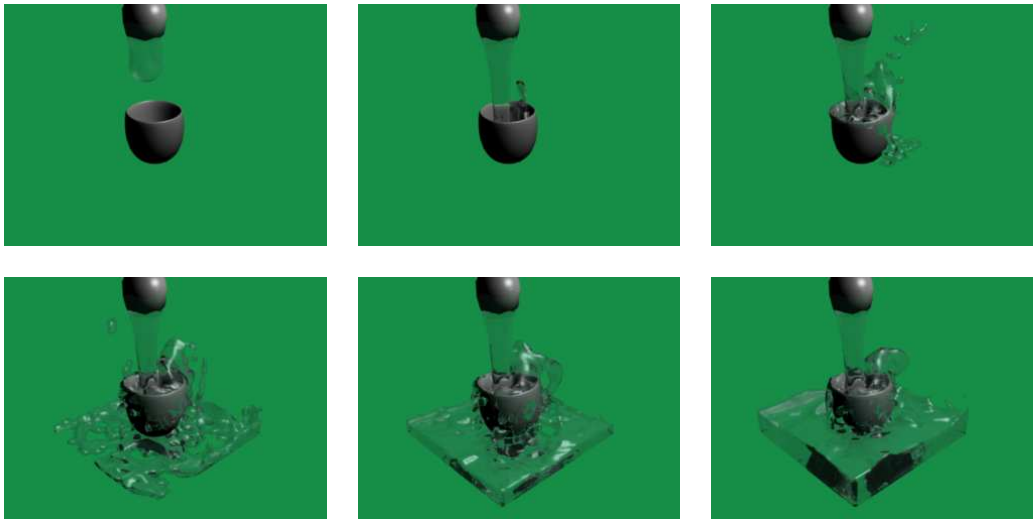

Figure 7-2: Six images from a short movie, modelled and rendered in Blender in approximately three hours, showing a simple LBM fluid simulation. From a 'water generator', shown as the sphere in the top of each image, water is poured into the cup. As it fills the cup, it overflows and splashes out into the environment. For simplicity and clarity an invisible box constrains the entire simulation, its dimensions can be clearly seen in the last three images.

For games a good compromise is 2D simulation; this can accurately model some waves and some splashes but can only be used in shallow water scenarios and cannot handle 'open' fluids. For example Figure 7-2 is impossible to model using only a 2D simulation, but a hybrid approach could give a good approximation: a particle system with a suitable fragment shader can easily represent water 'falling down', and if the particles collide with the inside of the glass, the volume of the liquid contained in the glass is increased. Should the particles miss the glass the liquid would disperse quickly and not leave any residues other than perhaps a decal.

This hybrid approach is inspired by the work already outlined in Section 2.6.3 (page 22). There are other special cases for water related simulation, for example Thuerey details a real-time model for modelling realistic bubbles and the formation of foam for liquids in a shallow water scenario [TSS+07]. This work also uses a LOSD approach and has separate models for

the movement of the bubbles and the vortexes they create and the resulting waves which form on the surface of the liquid.

Another example of such an approximation by the same author models waves using a conventional and efficient 2D method, but detects wave fronts in the resulting mesh and adds extra geometry and particles to visualise breaking and overturning waves [TMFSG07]. The objective of the work was to capture the key visual characteristics in an efficient manner and not to provide an accurate model.

Creating good 3D models of fluids that are fast enough to run in real time remains a hard problem. As a work-around, good approximations of various phenomena should be considered, but in the long term a general real-time simulation of liquids is desirable.

**Sound synthesis** The procedural generation of sounds based on physical properties of objects and interaction with these objects (such as hitting them, plucking a string, rolling a ball over a surface, etc...) is also an ongoing research topic. Broadly speaking there are two areas of interest with regards to the methodology:

- The procedural generation of sounds 'from nothing', i.e. generating believable sounds without pre-recording anything. Usually sounds are pre-recorded and the sample is played back when a specific event occurs.
- The procedural generation of the acoustic properties of an environment. For example an echo is usually generated by sampling an impulse (such as popping a balloon) in an acoustically lively environment (such as a church) and then convolving the sound with this environment sound.

As discussed in Chapter 2, there exists work to synthesise both of the above. However, only the former has really been synthesised in real-time. The potential to enrich a virtual environment with either is huge, especially if everything can be synthesised from object properties. This would save both storage space and money in game development, as fewer sounds would have to be recorded in a studio.

# Chapter 8

# Conclusion

## 8.1 Summary

This document laid out the overall goals of the methodology in Chapter 3. The first goal is to provide an environment that encourages different solutions by the player. The second, but equally important, goal was to find a way to describe otherwise disjoint physical simulations and how they can work together to become something greater than the sum of their parts.

The methodology was defined in Chapters 4 and 5. It was emphasised that the methodology is not a physical simulation as such; instead it is concerned with providing the programmer with a framework to approach the first goal.

The methodology draws upon the many and diverse physical simulations that are currently available and provides a way to describe each of these simulations with respect to how they interact with the rest of the system. Chapter 6 describes an implementation combining dynamics, electricity, magnetism, buoyancy and sound. In that Chapter the notation from Chapter 5 was successfully used to describe both the actual implementation used to carry out the experiments and how a refined, idealised implementation might be structured.

The experiments, which are described in Chapter 6, provide evidence of how the first goal can be achieved. Each of the experiments was put together using very simple parts; the overall behaviour of complex systems, such as the electric motor, emerged naturally. There was also scope in each of the experiment for the user to perform more than just the obvious interactions, for example silencing the doorbell by jamming an object between the bell and the striker.

The dynamics simulation used throughout this work was based on ODE, showing how existing physics simulations can be taken advantage of and integrated into an implementation of the methodology. Appendix A developed two models for simulating simplified electricity in the context of games. Furthermore, two simple simulations, magnetism and buoyancy, were added on top of this to provide more scope for interesting experiments and to demonstrate that the methodology can be used to describe something not originally included in either simulation: neither the electricity simulation nor the magnetics simulation dealt with the concept of electromagnets, this feature was implemented transparently using an interaction from the

methodology.

Many more interesting physical simulations exist and it would be exciting to see them appear in games, Chapter 2 and Chapter 7 provide a modest pool of ideas. In the latter, some areas in which the methodology itself can be improved upon are also outlined.

## 8.2 Conclusion

In conclusion, two important findings established during this endeavour must be highlighted.

The first of these is that producing a correct and well formed implementation of the methodology is not trivial. If one proceeds without a sound plan it is easy to come up with a framework that works for the original assumptions, but this can be difficult to change later. With a good plan and the important components of the methodology written down as to how they should appear (an example of this is the idealised implementation detailed in Section 6.13 (page 97)), it will be a lot easier. Of course this is obvious software engineering wisdom, but an implementation of the methodology especially will benefit from a good initial plan.

The second, and more important, item to highlight is that when considering whether to use the methodology for an application, it is vital to examine if the assumption from Section 3.3.1 (page 29) will hold. This assumption states that for some number of possible ways to interact with the game world, a roughly equal number of scripts are conventionally required and using the methodology and its interactions requires less overall effort after some point.

It is important to recognise that for some applications this assumption will not hold. Consider a game that only needs dynamics and contains only a couple of light switches. In this case the effort to lay out the world so that 'it works' will be greater than the effort to write the scripts. However, as the number of physical simulations emulated via scripts increases, there will come a point when it is easier to just provide a 'correct' world, although when precisely this point occurs can be difficult to measure.

Once the methodology is applied successfully, the number of possible interactions with the game world can be much greater. With every new physics simulation and interaction component added, this benefit is further amplified. Even if exposing a vast number of possible user interactions and allowing unconventional solutions to problems was not one of the programmer's initial goals, using the methodology will provide that 'for free', which can only be a good thing.

# Appendix A

# Electricity

This appendix proposes two approaches to model simplified electricity that are suitable for computer games and in particular for our methodology. As both approaches use parts of graph theory, the terms used are defined in Section A.2. The first approach is rather basic and was the core feature of the first experiments performed in the course of this work. This approach is described in Section A.4. A second, slightly more general, approach is then outlined; some of the approaches considered to get there in Section A.5 are first discussed and then a working algorithm is given in Section A.6. Finally, the theoretical complexity of both approaches is stated in Section A.7.

The implementation related to this appendix is discussed further in Section 6.5 (page 65) and some future work pertaining to this is outlined in Section 7.3.2 (page 110).

## A.1 Introduction

The work described in this appendix is self-contained with respect to the rest of this document; it was originally driven by the need for a simple physical simulation that is very different from dynamics.

Since the problem space will be a game, not every aspect of electricity is important, for example what the exact voltages are in a parallel circuit probably does not make for a dramatic set piece in a story.

In the initial phase of prototyping, physics simulations were essentially split into two basic types: simulations that would run frame-by-frame (such as dynamics) and simulations that are event driven and would only run when something changed, such as simplified electricity as discussed in this appendix. These event driven physics simulations were intended to be unified into a framework consisting of one or more connection graphs. Changes were propagated along it using generic messages. Electricity was chosen to be implemented as could be expressed naturally in terms of an event driven simulation. The very first implementation was message based, but subsequent implementations were based on graph theory.

The simplification of electricity involves looking at the overall behaviour of electricity and

expressing only a subset; one that only concerns itself with well known and obvious properties. These behaviours should be immediately intuitive to a potential user and should conceivably be useful in the context of a computer game.

The most basic property of electricity is that of an electrical circuit delivering power to a device. If the circuit is broken, the device can no longer function.

Other properties, such as voltages and the potential difference, will be ignored. Although to power one device, such as an Light Emitting Diode (LED), will require a different voltage to powering a washing machine, in the context of a game both of these circuits would function in the same way, but they would never need to interact with each other. Thus ignoring the entire problem set of voltages and potential difference is an example of a good simplification.

### A.1.1 Motivation for a new solution

The simulation of complex electrical circuits is a difficult problem, which generally involves numerical integration or similar iterative processes. A consequence of this is that run-time may not always be predictable or constant. As described in Section 2.6.2 the most widely used electrical circuit simulation SPICE3 [otUoCaB] and its descendants are all based on an iterative approach.

These iterative solutions are generally slower than event-driven simulations, at least in the context of action games where processor time is rather scarce.

Existing solutions also have a different aim: to simulate electrical circuits as accurately as possible. In such a simulation the user is usually interested in, and can query for, potential differences, resistances, voltages, feedback and minimal power consumption; none of which are relevant for games. The query "is any given device or component of a circuit 'on' or 'off"', is so basic that it is usually not worth running such a complex simulation in the first place.

Digital logic circuit simulations, on the other hand, are interested primarily in the 'on' or 'off' state of devices and are thus generally event based. The traditional digital circuit simulators XSPICE [oEEotGIoT] and NGSPICE [NtNd] are built on top of SPICE and are thus still backed by an iterative system. (XSPICE is built directly on SPICE; NGSPICE is a combination of XSPICE and two other SPICE modifications plus some additional features.) However their focus is the modelling of logic circuits, i.e. and gates, not gates, adders – all of which are far too complex for our purposes.

### A.1.2 Assumptions

This list of assumption applies to all approaches detailed below.

1. The circuit/electricity graph is a dynamic graph which is not necessarily fully connected[1]. A vertex in it usually represents an object in the game world that can be affected by electricity. For example: a wire, a light bulb, a socket or a TV. An edge in the graph

[1] All graph theory terms are defined in the next section.

represents a connection between two objects. For example an edge between the socket and the TV plug would indicate that the latter is plugged into the former.

2. An electrical circuit requires both a connection to a source and a return (or sink).
3. It is not necessary to identify or model short circuits.
4. There is no concept of potential difference or resistance.
5. There is no concept of positive or negative poles.
6. There is only one global power source and return per electricity graph. Other power sources are simply connected with an invisible, virtual connection to the source node; likewise other returns have such a connection to the return node.

### A.1.3 New approaches for games

In what follows two graph theory based approaches for simplified electricity will be detailed. They are of varying detail (in terms of how expressive the simulation is) and complexity and provide a model for simplified electricity for games. Neither of them is intended to be full circuit simulation such as SPICE3, but rather an approximation that still give useful results in the context of games.

**Single Cable** This approach models complete power cords[2] with both a positive and negative/neutral lead. This is sufficient to implement simple scenarios such as light switches or plugged-in devices that can be deactivated by unplugging them.

The single cable approach will represent both source and return leads in the same connection and thus does not require a return node as it is the same as the source node.

**Two-wire** This approach models individual wires and can be used to model more complicated problems. In this approach, the positive wire and the negative wire will each be modelled explicitly.

## A.2 Background graph theory terminology

This section will introduce and define various graph theory related terms which apply to all the methods outlined in this appendix. Terms related only to specific approaches are described alongside the individual approaches. The graph in Figure A-1 will serve as an illustration for each term defined in this section.

A circle with a number (or some other text label) represents a vertex in the graph. A line between two vertices represents an edge . It should be noted that the terminology in graph theory is unfortunately not completely consistent and a few concepts are known under more than one term. For instance, the term *vertex* is used interchangeably with *node*. Sometimes

[2]Such as standard International Electrotechnical Commission (IEC) cables

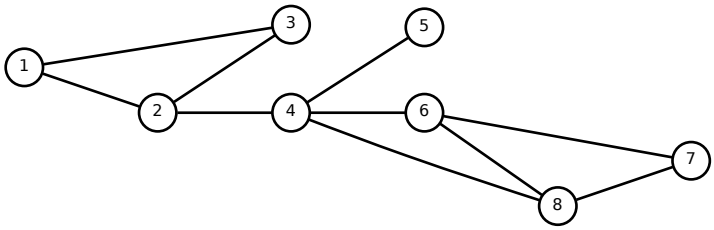


Figure A-1: Example graph which will be used to illustrate the import graph theory concepts.

one term can be more intuitive than another, equivalent, term depending on the context in which they are being used. For instance, if the graph theory aspect of a particular method is examined, the term *edge* is used. However when talking about an electrical network, the term *connection* is more useful.

### A.2.1 Dynamic graph

Conventional graph theory is usually only interested in static graphs: given a particular graph, certain statements can be made about it and its properties. Algorithms give a particular answer to a particular problem and assume the graph will not change.

A dynamic graph is a graph where vertices and edges can be added or removed at any point in time. A dynamic graph algorithm operates on such a graph and generally maintains some property of the graph (such as connectivity or minimal spanning tree [HdLT01] or transitivity [PL88]), updating it as edges and vertices are added and removed from the graph. A good dynamic graph algorithm can maintain such a property (for example connectivity) over a number of arbitrary graph updates without the need to fully calculate it from scratch on each modification.

Under the broad heading of dynamic graph theory there are roughly two different classes of algorithms:

- Can deal with additional edges and vertices (constructive updates), but cannot deal with the removal of edges or vertices (destructive updates). These are also known as incremental algorithms and a number of conventional graph theory algorithms can be adapted to be incremental.
- Can deal with both constructive and destructive updates to the graph.

### A.2.2 Subgraph

Informally a *subgraph*, or sometimes *component*, of a graph is a small part of it. A subgraph of graph $G$ is defined by the vertices it contains which must be a subset of the vertices of $G$. It also contains all edges between any of the vertices in the subgraph. For example, a subgraph of the graph in Figure A-1 is $(2, 3, 4, 5)$, this is illustrated in Figure A-2.

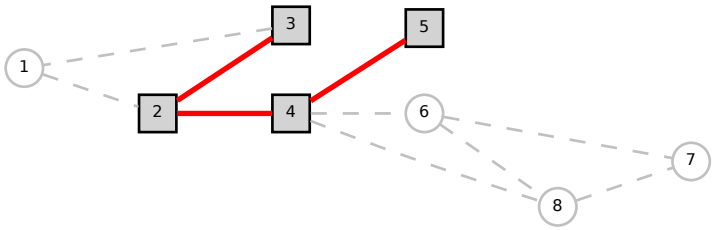


Figure A-2: The example graph with the subgraph $(2, 3, 4, 5)$ highlighted.

Every individual vertex of $G$ is also a trivial subgraph of $G$. A graph (or subgraph) is *connected* if there exists a route between any two vertices $v_1$ and $v_2$ that are part of the graph (or subgraph). The graph in Figure A-1 is a connected graph.

A subgraph does not necessarily have to be connected, for example $(1, 8)$ is also a valid subgraph of the graph in Figure A-1.

### A.2.3 Articulation vertex

Also called a *cut-vertex*. An articulation vertex of a graph is a vertex that, if removed, will disconnect the graph. The graph in Figure A-3 contains exactly two articulation vertices: 2 and 4.

### A.2.4 2-Connectivity

Also called *bi-connectivity*. A 2-connected graph or subgraph contains no articulation vertices, i.e. it cannot be disconnected by the removal of any one vertex.

The graph in Figure A-1 is not 2-connected; however the following subgraphs are examples of 2-connected subgraphs: $(1, 2, 3)$, $(6, 7, 8)$, $(4, 6, 8)$ and $(4, 6, 7, 8)$. Technically the trivial subgraph $(5)$ is also 2-connected. Note that this list of 2-connected subgraphs is not exhaustive and the example graph in Figure A-1 contains 22 2-connected subgraphs in total:

- Eight in trivial subgraphs (vertices 1 through 8).

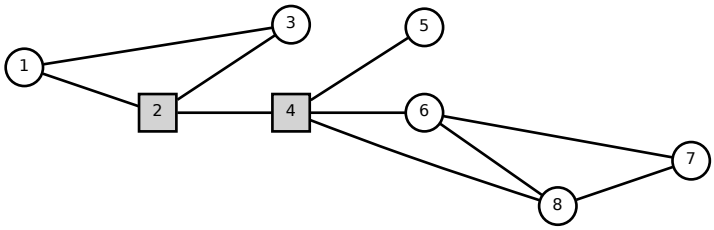


Figure A-3: The example graph with its two articulation vertices, 2 and 4, highlighted. If one were to remove any one of them, the resulting graph would be disconnected. In particular 5 is not an articulation vertex, because if it were to be removed the rest of the graph will still be fully connected.

- Ten in trivial subgraphs (all subgraphs containing only 2 vertices connected by a single edge).
- One in the subgraph $(1, 2, 3)$.
- Three in the subgraph $(4, 6, 7, 8)$. (One of them is shown in Figure A-4.)

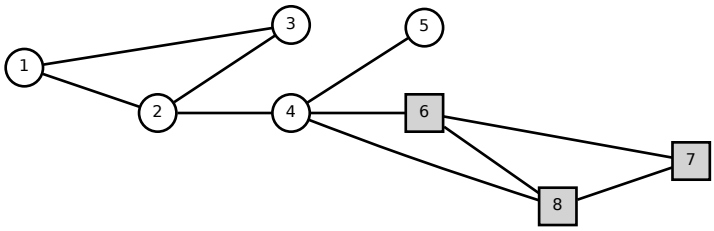


Figure A-4: The example graph with one of its 22 2-connected subgraphs highlighted: $(6, 7, 8)$. Note that this is not a block graph as vertex 4 can be added to it and the result is still 2-connected. The subgraph $(4, 6, 7, 8)$ is a block graph however.

### A.2.5 Block

A block of a graph is a *maximal* 2-connected subgraph. The graph in Figure A-1 has four different blocks, they are $(1, 2, 3)$, $(2, 4)$, $(4, 5)$ and $(4, 6, 7, 8)$.

In particular the subgraphs $(4, 6, 8)$ and $(6, 7, 8)$ are not blocks; although they are 2-connected they are not *maximally* 2-connected: to the subgraph $(4, 6, 8)$ one can add vertex (7) and the resulting subgraph is still 2-connected. However $(4, 6, 7, 8)$ is maximal as neither vertex (2), nor (5), or any other vertex or subgraph can be added without making vertex (4) an articulation vertex.

### A.2.6 Other terms

The *degree* of a vertex is equal to the number of edges it is a part of. The degree of vertex 1 in Figure A-1 is two, the degree of vertex 4 is four.

A graph is *cyclic* if it contains one or more cycles, i.e. a path from one vertex through any number of vertices and back again without using any edge more than once. A 2-connected graph is always cyclic, but the converse is not true. The subgraph $(1, 2, 3, 4)$ in Figure A-1 is an example of a cyclic graph, but it is not 2-connected.

In this appendix only *undirected* graphs are of concern. An undirected graph contains edges that can always be traversed in both directions.

A *tree* is a connected graph containing no cycles. Trees also contain a *root* vertex.

A *spanning tree* of graph $G$ is a tree containing every vertex of $G$. Since only unweighted edges are important with the work in this appendix, any such spanning tree is also the *minimum spanning tree*.

A disconnected graph can have a *spanning forest*.

### A.2.7 Source node and return node

Unrelated to standard graph theory but used in this work: the source and return nodes are two vertices labelled as such. They have no other special properties.

## A.3 Basic model for both approaches

For the two electricity models in this appendix, the following basic properties hold true:

- Objects in a scene can be conductors or non-conductors. Some objects might do special things if they are 'on', these effects are handled by other physics simulations via interactions. For example, if turned on, a lamp might emit light or an electromagnet will begin to exert a magnetic field.
- Objects are represented by vertices in a graph called the *electricity graph*. This graph can be disconnected. An edge between two vertices symbolises that they are connected in some way - but this does not necessarily mirror the 'real' world and implies a tangible physical link. For instance, two conductors physically touching would be represented by a connection in the electricity graph, but an electrical arc between two non-touching conductors would also be represented by a connection in the electricity graph.

Another example of a non-tangible connection could be shortcuts on the game world designer's part, for example if it is found that a certain wire will never be accessible to the player since it is embedded in an indestructible wall, it will not have to be modelled at all and can be represented by simply adding an edge between the relevant vertices in the electricity graph.

- The simulation is event driven. The information stored within the electricity graph (for example if any particular object is 'on' or 'off') is only updated when necessary, i.e. when the electricity graph is modified.

## A.4 Approach I: single-cable

Our first attempt at modelling electricity for games modelled complete power cords (containing both a live and a neutral lead) between vertices in the electricity graph. Essentially a device was considered to be 'on' if a connectivity query between the power source (designated by the source node) and the vertex representing the device in question returned true.

### A.4.1 Motivation

For the first experiments this single-cable model was devised as it was exceptionally simple in both concept and implementation. As it was largely proof of concept experimental work, the main goal at the time was simply to be able to observe the on/off state of devices; simplicity of the algorithm and implementation were the primary considerations.

### A.4.2 Algorithm

The single-cable electricity model is very simple in terms of its expressiveness. It can be fully defined as such:

- Any one vertex can be the 'source node'.
- Any object in a connected subgraph containing the 'source node' is 'on'.
- All other objects are 'off'.

This approach, and all others outlined in this section, distinguishes between two events: connection and disconnection.

#### Connection

For this particular approach the connection case is trivial. All vertices in any connected subgraph are always in the same state: 'on' or 'off'. If two 'off' or two 'on' subgraphs are connected, no work needs to be done. If an 'off' subgraph is connected to an 'on' subgraph then the resulting subgraph is 'on'. Adding a new vertex is the trivial case of connecting a trivial subgraph (that vertex) to another subgraph.

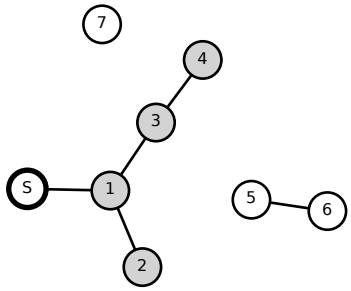


Figure A-5: A simple graph illustrating the single-wire approach. Objects 1-4 are on as they are connected to the source; objects 5-7 are off as they are not.

For example if a connection between objects 2 and 3 in Figure A-5 is made, no work would need to be done. This is also the case for a connection between objects 5 and 7. However if objects 2 and 5 were to be connected, the entire subgraph containing object 5 (i.e. objects 5 and 6) would now be in state 'on'. (Complexity is at worst $O(v + e)$ where $v$ is the number of vertices in the graph and $e$ is the number of edges in the graph.)

#### Disconnection

The disconnection case is more complicated. There are two naïve approaches to it: either rebuild the graph from scratch or issue a connectivity query from both vertices concerned to the source node. If a query does not succeed, change the state of all the vertices in the relevant subgraph to 'off'. If a minimum spanning tree is maintained, this can be improved: disconnecting any edge is trivial unless it is part of the minimum spanning tree.

Complexity would be the same as for connection for the first approach. It can vary for the second approach depending on the connectivity algorithm used. For example [Tho00] gives an $O(\log n / \log\log\log n)$ solution to the dynamic connectivity problem (updates are $O(\log n(\log\log n)^3)$), and [HdLT01] gives a more general $O(\log n / \log\log n)$ solution (updates are $O(\log^2 n)$).

### A.4.3 Discussion

#### Generality

Some physics simulations share a lot of properties. Electricity and water in pipes are an example of this. For instance, consider the graph shown in Figure A-6.

The source node represents the main source of water, for example a water tank. Vertex 1

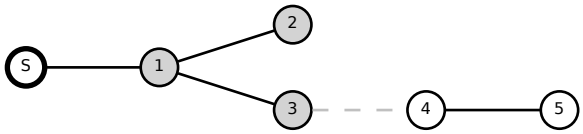


Figure A-6: Example of a water network, using the single-wire approach. If object 5 is a tap and objects 3 and 4 together represent a valve then the tap would only be able to emit water if the valve is 'open', i.e. a connection exists between 3 and 4.

represents a pipe with a T-junction, routing water to both objects 2 and 3. Vertex 2 could represent a simple dead-end pipe. Vertices 3 and 4 could represent a simple valve (i.e. a switch). Vertex 5 could be a tap that, if 'on', causes water to flow into a sink. Single-cable electricity is flexible enough to model such a network and is a sensible simplification for part of the water problem. For open water (i.e. oceans, coffee in a mug or the water flowing from a tap) a different model is required and also some means of interfacing or 'converting' between them. (Also see Section 7.1.5 (page 105) for more discussion on such hybrid simulations.)

If the sole use of pipes is to transport water to some destination, then this single-cable approach is sufficient to model all scenarios. If on the other hand 'devices' such as a turbine are driven by water then the method outlined above is not enough and a more powerful approach is necessary (such as the two-wire approach outlined below).

**Limitations**

While this approach adequately models most simple things (such as a light switch set into a wall or perhaps forcibly turning off a TV by unplugging it) it does exhibit some obvious limitations, the first two of which are addressed directly by the two-wire approach:

1. By definition, there is no separate source and return. Sometimes it might make for a more interesting scenario if both the live, or positive lead and the neutral, or negative, lead of an object have to be connected properly to make it work. Simply adding a return node and also testing for a connection to it is not sufficient: for example, in the graph in Figure A-8 (page 128) nodes 4, 5 and 6 would also be 'on'.

   For some scenarios it might be better to allow the disruption of an electrical circuit in two ways, severing either the live wire (to the 'source') or severing the neutral wire (to the 'sink').

2. It is impossible to break chains of devices. When connecting a number of devices in a simple chain with sequential wiring, breaking any link should disrupt the entire circuit, not just all devices *after* the broken link.

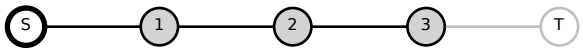


Figure A-7: Example of a sequential device chain. T is a return node.

In Figure A-7, using the single-cable approach, removing the connection between vertices 2 and 3 will only turn off the object represented by vertex 3. If however both the live and neutral lead are modelled separately as shown in the graph, then disconnecting 2 and 3 should result in breaking the entire circuit. Such a behaviour is not possible with just one wire and will require a different model.

3. As per our assumptions, there is simply no way to express the concept of a short circuit within this model. Since not only does it lack the concept of an earth or neutral lead, it also does not consider potential difference or resistances.

## A.5 Towards a usable two-wire approach

Developing a working two-wire approach was significantly more complicated than creating a good single-cable solution. This section is included for the purpose of recording the approaches considered. It will describe the original steps made that were eventually replaced by the working solution as described in Section A.6.

### A.5.1 Initial two-wire approach

This approach uses both a source and a return node; depending on what kind of electrical circuit is being modelled this can represent the + and - poles, or alternatively, a live and a neutral lead. A connection to both the source and return nodes is required to form a working electrical circuit. The aim of this approach is to allow the modelling and expression of more complicated problems, whilst still being at least as powerful as the single-cable approach.

#### Motivation

This approach was considered because many potentially interesting game scenarios can easily be imagined that are based on problem 1 or 2 listed above for the single-cable approach.

For example, offering more than one 'natural' way to disable a device and accurately modelling a chain of light bulbs where the game allows the destruction or removal of wires are not possible to model by using the single-cable approach.

#### Overview of approach

The new concept in this model is the introduction of a return node. To form an electrical circuit a device needs both a connection to the source node and the return node. The connection to the source node and the connection to the return node must not overlap. Hence in Figure A-8 nodes 4, 5 and 6 are 'off'. Although each of them have both a connection to the source node and the return node, every single route to either source or return node must pass through node 2.

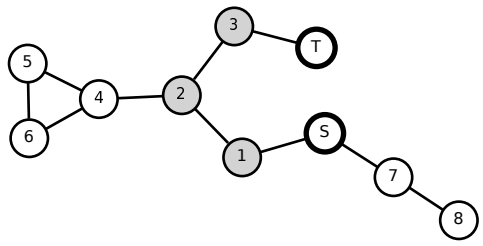


Figure A-8: Goal of a working two-wire approach. Objects 1-3 should be 'on' as they form a circuit between S and T; object 4-8 should be 'off' as they are either in 'dead-ends' or connected to only one of S or T.

Examples of routes from $S$ to $T$ are: $(S-7-S-1-2-3-T)$ or $(S-1-2-4-6-5-4-2-3-T)$. However there is - in this case - only a single route that does not use any traversed node more than once: $(S-1-2-3-T)$. The set of 'on' nodes is all nodes used by all such routes; in this case $\{S,1,2,3,T\}$.

#### Problems highlighted and other notes

Although dealing with incremental connections is still easy (with the exception of recognising constructs such as $(4,5,6)$ in Figure A-8), incremental disconnect is a lot harder. This, and the fact that the final solution to this problem as described in Section A.6 (page 132) was not discovered until much later, led to various refinements and abstractions of the approach in order to reduce complexity. Although most of them will be outdone by the final approach, some of them could still be useful in niche scenarios.

### A.5.2 Initial tree based approach

#### Motivation

Since the difficulties in finding a simple solution for the disconnect case persisted throughout the search for an efficient simplified electricity algorithm, the classic method of divide and

conquer was considered. It was recognised that electricity networks tend to be hierarchical in nature, on both the source and the return side. For example, consider the following model of a city:

- A power plant delivers power to various main transformers inside city sectors.
- Each main transformer powers a block of buildings.
- Each building has its own distribution network, distributing power to each floor.
- Each floor distributes power to each room.
- Each room contains power sockets, providing electricity for a number of devices.

Similarly, on the return side of the graph:

- Each room has power sockets with returns and possibly other earthed objects such as metal radiators.
- Each building is earthed.

The main motivation behind this approach is to automatically recognise such a structure and split the electricity graph into easier to handle smaller parts. Given the city scenario above, the electricity graph would be large, but the parts the user is likely to interact with will be small. A lot of time spent to figure out if a particular node is on or off will be wasted on performing the same searches many times. The resulting computational cost can be reduced by structuring the problem, so that not everything will have to be solved every time.

#### Overview of approach

The main idea behind the tree based approach is to take advantage of the hierarchical nature of electricity graphs such as the one outlined above. This should significantly reduce the size and complexity of the graphs which are examined to determine if the state of any one vertex is 'on' or 'off'. This is achieved by partitioning the entire electricity graph into three types of subgraphs:

- A source tree - this acyclic subgraph is a tree and is rooted at the source node.
- A return tree - as above but rooted in the return node.
- Zero or more device graphs.

Since the source tree subgraph and return tree subgraph are both trees, any connectivity query on them is trivial and fully dynamic. Device graphs are what is left over 'in-between' the two; these are potentially cyclic graphs that have at most one connection to the source tree and at most one connection to the return tree.
In the example shown in Figure A-9 the source tree is formed by vertices 1 to 5 and the return tree by vertices 11 to 14. These trees are rooted in vertex S and vertex T respectively. Vertices 6 to 10 form a smaller electricity graph, a device graph, connected to the main electricity network. This subgraph is connected to both the source (via vertex 3) and the return (via vertex 11).

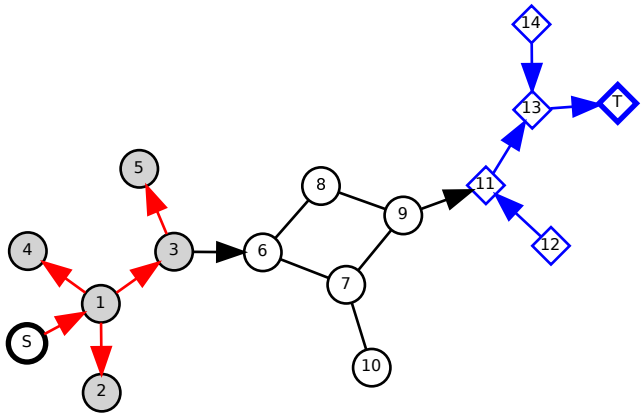


Figure A-9: Example electricity graph with source and return trees shown in red and blue respectively. Nodes 6 to 10 are part of neither.

### Problem highlighted

Unfortunately it is not necessarily as clear cut as it is in the above example where the source tree stops and the return tree begins. For example consider the previous example with the edge between vertices 8 and 9 removed, as shown in both graphs in Figure A-10. All vertices being equal, there exist many possible divisions between the source and return tree. Both graphs show a perfectly valid division of source tree, device graphs and return tree.

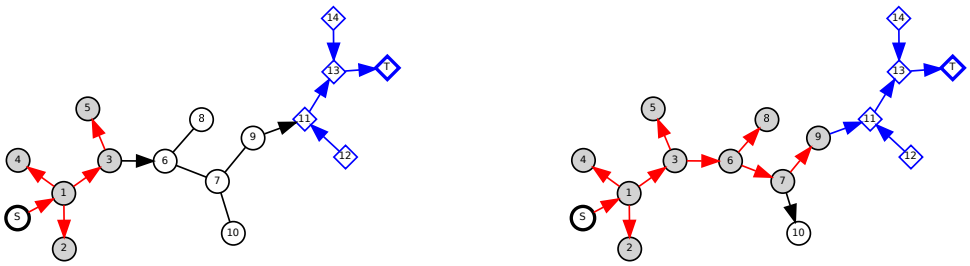

Figure A-10: Two graphs illustrating two of many more possible divisions between the source and return trees. Having more than possible valid representation of a problem is clearly not ideal.

### A.5.3 Device node and device graph refinement

#### Motivation

In order to tackle the problem of not having a unique division between source and return graphs the concept of a device node (each represented by a single vertex in the electricity graph) and a device graph can be introduced. This addition no longer assumes that all vertices are equal: device nodes are vertices that represent objects that draw or use power in some observable fashion, such as a light bulb. A wire would only serve to get power from one point to another, but in itself does not 'consume' power (ignoring resistance); such a wire is not a device node.

#### Overview of approach

Any vertex in the graph can be declared to be a device node. A device node indicates that electricity is used in some fashion at the object represented by that vertex; for instance the filament of a light bulb would be a device node, its winding would not be. A device node's degree is at most two.

A graph is a device graph if it has at least one of the following two properties: it is cyclic or it contain at least one device node. The device graph can have up to one edge connecting it to the source tree and up to one edge connecting it to the return tree. The definition of a device graph is as follows:

- A device node is always part of a (potentially trivial) device graph.
- A vertex contained in a cycle is always part of a device graph.
- It is not possible to create a connection between two different device graphs; instead adding such an edge will merge the two device graphs into a larger one.
- Source and return trees must always be maximal. If any given vertex could be part of either a device graph or a tree, it should be part of a tree. For example, vertex 2 in Figure A-11 could, without this criterion, be part of both a device graph and the source tree; thus it is part of the source tree.

Figure A-11 gives an example of an electricity graph containing four device graphs. They are $(4, 5, 6, 7)$, $(8)$, $(11)$, $(12, 13, 14)$.

#### Problems highlighted and other notes

As with the previous approach, the state of each individual vertex (although only device nodes are relevant) inside a device graph must be solved by a separate algorithm. The expensive brute force method is to calculate all possible routes, eliminate overlapping routes and turn all vertices in one or more of the remaining routes on.

The main motivation for attempting to decompose the original problem into three parts was that a more efficient method to determine which nodes are 'on' and 'off' had not been

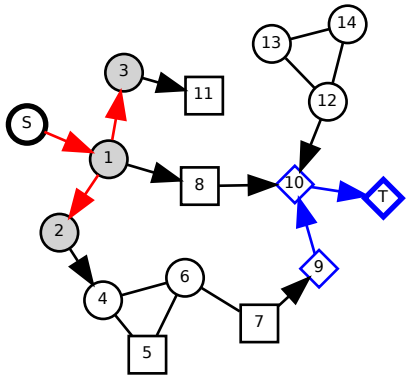


Figure A-11: Example of four different device graphs. Vertices shown as a box are device nodes. The source and return trees are shown in red and blue as usual. The four device graphs are: $(4, 5, 6, 7)$, $(8)$, $(9)$ and $(12, 13, 14)$.

found yet. It was hoped that this approach would reduce the problem space to something more manageable where brute force was a valid solution.

Despite the efforts to simplify the algorithm by finding a unique partitioning of the electricity graph, the benefits of this are easily outweighed by the effort it takes to find device graphs. Furthermore it is easy to inadvertently collapse the entire tree/graph structure into a single device graph, making partitioning impossible due to one unfortunate connection. For example, should the user form a short circuit connection between vertices 1 and 10 in the graph shown in Figure A-11, the entire graph (other than vertices S and T) will collapse to become a single device graph.

## A.6 Approach II: two-wire

This section will describe a working and efficient solution to the two-wire model as initially outlined in Section A.5.1 (page 128). This approach could also be used to solve the device graph problem of the previous section, but there is little point in most cases as this method is much simpler and more efficient.

### A.6.1 Algorithm

In the following example the only two valid non-intersecting paths from S to T are: $(S{-}1{-}2{-}3{-}4{-}5{-}T)$ and $(S{-}1{-}6{-}3{-}4{-}5{-}T)$. In particular a path such as $(S{-}1{-}10{-}1{-}2{-}3{-}4{-}5{-}T)$ is not valid, as the vertex 1 is used twice.

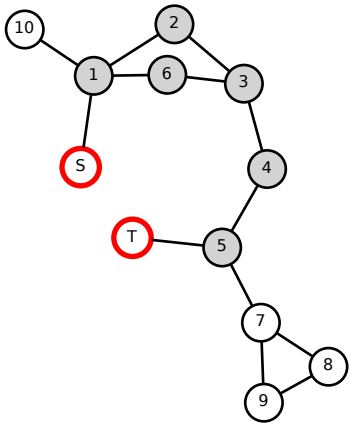


Figure A-12: An example circuit demonstrating the result after applying the two-wire algorithm. The shaded vertices (1 to 6) are 'on'.

Unfortunately there is no simple graph theoretical construct that describes the set of 'on' (or 'off') nodes[3]. However, if a simple 'virtual' edge between the source and return node is added to the electricity graph (as shown in Figure A-13), the set of 'on' vertices becomes identical to the block of the electricity graph that contains both the source node and the return node.

The following Figure A-14 shows the same example with edge $3-4$ removed. Note that the block containing the source and return node has become very small indeed, containing just $(S,T)$. The list of all block graphs in Figure A-14 is: $(S,T)$, $(S,1)$, $(1,10)$, $(1,2,3,6)$, $(T,5)$, $(4,5)$, $(5,7)$ and $(7,8,9)$.

Holm and Lichtenberg [HdLT01] give an $O(log^5 v)$ (where $v$ is the number of vertices) algorithm to identify articulation vertices for dynamic graphs. An implementation for static graphs with complexity $O(v+e)$ (where $v$ is the number of vertices and $e$ the number of edges) can be found in BGL [Sie00], which is based on Tarjan's work [Tar72].

It should be pointed out that for smaller graphs (containing less than $2^{19}$ elements), the static methods operating in $O(v+e)$ are faster.

### A.6.2 Drawbacks

Aside from the obvious drawbacks of not being able to model short circuits (which could be interesting for certain games; bypass a security device by causing a short circuit around it) or

[3]There is however an algorithm: find all cycles in the graph and collapse them into single nodes. Then find the single remaining path from S to T; all nodes in it are 'on'. However, this approach is more expensive than the solution described in this section.

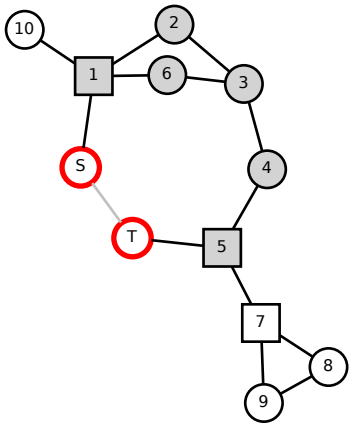


Figure A-13: The same circuit with the virtual edge $S - T$ shown; vertices 1 to 6 are 'on', vertices 7 to 10 are 'off'. Box shaped vertices indicate articulation vertices; shaded vertices are in the source/return block.

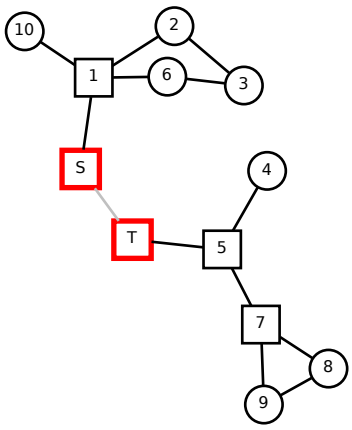


Figure A-14: The previous example with the edge from 3 to 4 removed, showing the state of the graph after breaking the connection. As before, boxes represent articulation vertices.

potential differences and resistances, the drawback of being restricted to only a single global power source is much more subtle.

An easy way to get around this restriction is to create several power nodes and link them all to the main source node. This approach works if all power nodes represent roughly equal power sources, but will fail in some corner cases.

For instance, assume the goal is to model both a nuclear power plant (which will power a gigantic laser) and a small AA battery (which will power a flashlight).

In Figure A-15 vertex 1 represents the nuclear power plant, vertex 2 the battery, vertex 3 the laser and vertex 4 the flashlight. This scenario is believable as long as there is no interaction between the subgraphs $(S, 1, 3, T)$ and $(S, 2, 4, T)$.

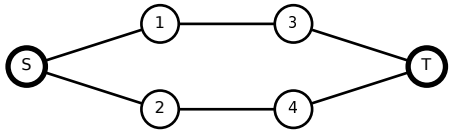


Figure A-15: A simple graph showing how different power sources can be modelled naïvely. To make objects 1 and 2 power sources, they are connected directly to the source node. Objects 3 and 4 consume power, they are each connected to their relevant power source.

If the user now disconnects the laser from the power grid by removing edge $1 - 3$ from the graph, vertex 3 is 'off' as expected. However it is still possible to reactivate our laser simply by forming a connection between one pole of the flashlight battery and the power socket of the laser as shown in Figure A-16.

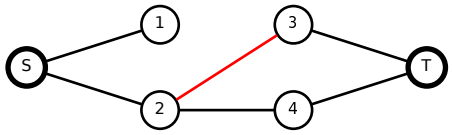


Figure A-16: Same as the previous image, but demonstrating the problems of 'unrealistic' connections. Object 3 is meant to draw power from the power source represented by object 1, but clearly a valid connection can be formed by just connecting it to object 2 instead.

This does not make much sense, so it is advisable to only group power sources of similar

output into any electricity graph. Creating two graphs, low (batteries) and normal (standard national grid), will likely be enough for most scenarios. The amount of time taken to update the entire simulation increases with the number of separate electricity graphs present since the block graph discovery algorithm will have to be run over every single one of them if any vertices or edges are inserted or deleted (unless the graphs are kept completely separate).

### A.6.3 Proof

The original requirement from Section A.5.1 (page 128) states "a device needs both a connection to the source node $s$ and the return node $t$ [and they] must not overlap". This requirement will be known as $R$. The electricity graph will be known as $G$. There exists at least one block graph $G'$ for that $\{s, t\} \subset G'$. To summarise, $R$ states two things:

1. An 'on' device requires both a connection to the source and the return node.
2. These two routes combined do not use the same vertex more than once (other than the starting point of course).

This subsection outlines an informal proof (due to the use of some English, in particular when describing $R$) that $R$ being true for a vertex $v$ in $G$ is equivalent to the vertex $v$ being contained in the block graph $G'$. The proof justifies the use of 2-connectivity and proves equivalence by proving implication in both directions:

1. It shows that $\forall v$ such that $v \in G'$ then $R$ must hold for $v$.
2. It shows that if $R$ holds for a vertex $v \in G$ then $v \in G'$.

Hence $R$ holds for $v$ if and only if $v \in G'$.

**Part 1** Assume $v \in G'$. The block graph $G'$ contains by definition both the source and return node so this part of $R$ is trivially true as $v \in G'$ and $G'$ (a 2-connected graph) is by definition also a connected graph.

For the second part of $R$, proof by contradiction is used. Assume the second part of $R$ does not hold and a vertex must be used more than once. The only way a single vertex *must* be traversed by both the route to the source node $s$ and the route to the return node $t$ is if there is simply no other possible route. If all routes from $v$ to $s$ and $t$ must pass through that vertex, than removing that vertex would disconnect $v$ from both $s$ and $t$. This means that vertex $v$ is an articulation point.

However as $G'$ is a 2-connected graph and $v \in G'$, removing a single vertex must still leave a fully connected graph, which in turn means there must be another route; this contradicts our assumption.

**Part 2** Assume $R$ is true for $v$, that $v \in G$, $v$ and $s$ are connected, $v$ and $t$ are connected and these routes are simple (i.e. do not need to use one vertex more than once). The route from $v$ towards $s$ (but excluding the part of the route that is $\subset G'$) shall be known as $r_s$ and $r_t$ is

the equivalent for $t$. It is known that $v$ must be connected to $s$ and $s \in G'$ and $v$ also must be connected to $t$ and $t \in G'$. Thus $\{v\} \cup r_s \cup r_t \cup G'$ is at least connected.

$v \in G'$ is proven by contradiction: assume that $v \notin G'$. From $R$ it is known that it is possible to reach $s$ and $t$ without using the same edge twice. Thus $r_s$ and $r_t$ must be entirely different as per definition of $R$ (i.e. $r_s \cap r_t \equiv \{v\}$). In particular, their endpoints where they lead in to $G'$ are different. These endpoints must be connected, as they are both part of $G'$.

As $\{v\} \cup r_s \cup r_t$ is a 'line', having a connection at the endpoints of $r_s$ and $r_t$ in the form of $G'$ will form a circle, which per definition, is 2-connected. Hence $\{v\} \cup r_s \cup r_t \cup G'$ must be 2-connected. As $G'$ is a block graph, $v \in G'$, which is a contradiction.

## A.7 Complexity Overview

| Approach | Static Graphs | Theoretical Complexity for Dynamic Graphs |
|---|---|---|
| Single Cable | $O(v+e)$ (BGL) | $O(\log^2 v)$ [HdLT01, Tho00] |
| Two Wires | $O(v+e)$ (BGL, [Tar72]) | $O(\log^5 v)$ [HdLT01] |

The static graph implementations used in the proof of concept implementation described in Section 6.5 (page 65) use the BGL [Sie00]. Boost also supports incremental graph algorithms for single connectivity, which can give performance gains in certain scenarios, but they do not support edge or vertex removal from the graph and hence are not full dynamic graph algorithms.

# Bibliography


[3D 96] 3D Realms. Duke nukem 3d. Multiple Platforms, 1996.

[AB99] Chris Avellone and Black Isle Studios. Planescape torment. PC Game, 1999.

[AK08] Apple Inc and Khronos Compute Working Group. Open computing language. http://www.khronos.org/opencl, 2008.

[AN] AGEIA and NVIDIA. Physx. http://www.nvidia.com/object/nvidia_physx.html.

[Ata72] Atari Inc. Pong. Arcade Game, 1972.

[BBSC] Stefan Behnel, Robert Bradshaw, Dag Sverre Seljebotn, and Others Contributors. Cython. http://cython.org.

[Beg94] Durand R. Begault. *3-D sound for virtual reality and multimedia*. Academic Press Professional, Inc., San Diego, CA, USA, 1994.

[Boe] Adrian Boeing. Physics abstraction layer. http://www.adrianboeing.com/pal/index.html.

[BTBW95] A. Bowyer, R. Taylor, G. Bayliss, and P. Willis. A virtual workshop for design by manufacture. *15th ASME International Computer in Engineering Conferenc*, 1995.

[Cat] Erin Catto. Box 2d. http://www.box2d.org.

[Cat06] Erin Catto. Buoyancy. In *Game Programming Gems 6*, 2006.

[Cor08] NVIDIA Corporation. Realtime stable fluids on cuda enabled gpus. http://www.nvidia.com/object/cuda_sample_simulation.html#fluidsGL, 2008.

[Cou] Erwin Coumans. Bullet. http://www.bulletphysics.com.

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.

[Dij74] Edsger W. Dijkstra. Over seinpalen. circulated privately, 1974.

[dot] dot3 labs. Cube2. http://sauerbraten.org.

[ea89] Mike Stephenson et al. Nethack. Multiple Platforms, 1989.

[ED98] Epic Games and Digital Extremes. Unreal. Multiple Platforms, 1998.

[EK72] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, 1972.

[Ele98] Electronic Arts. Tiger woods pga tour. Computer Game / Multiple Platforms, 1998.

[Ent04] Blizzard Entertainment. World of warcraft. PC Game, 2004.

[Ern] Emil Ernerfeldt. Phun. http://phun.cs.umu.se.

[Fag] Mattias Fagerlund. Buoyancy particles. http://www.hypeskeptic.com/Mattias/DelphiODE/BuoyancyParticles.asp.

[FF01] Nick Foster and Ronald Fedkiw. Practical animation of liquids. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 23–30, New York, NY, USA, 2001. ACM.

[FN71] Richard E. Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189 – 208, 1971.

[FRS] Alan Fischer, Andres Reinot, and Tyler Streeter. Physics abstraction layer. http://opal.sourceforge.net/index.html.

[GT88] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, 1988.

[Hav00] Havok and Intel. Havok physics. http://www.havok.com/, 2000.

[HdLT01] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, 2001.

[HHL+05] Trond Runar Hagen, J. M. Hjelmervik, Knut-Andreas Lie, Jostein R. Natvig, and M. Ofstad Henriksen. Visual simulation of shallow-water waves. *Simulation Modelling Practice and Theory*, 13(8):716–726, 2005.

[id93] id. Doom. Multiple Platforms, 1993.

[id96] id. Quake. Multiple Platforms, 1996.

[id99] id. Quake3. ftp://ftp.idsoftware.com/idstuff/source/quake3-1.32b-source.zip, 1999.

[id04] id. Doom 3. Multiple Platforms, 2004.

[Ima82] Hiroyuki Imabayashi. Sokoban. Multiple Platforms, 1982.

[Ion00] Ion Storm Inc. Deus ex. PC Game, 2000.

[Kit00] Kitware. CMake. http://cmake.org, 2000.

[KM90] Michael Kass and Gavin Miller. Rapid, stable fluid dynamics for computer graphics. *SIGGRAPH Comput. Graph.*, 24(4):49–57, 1990.

[KM07] Bill Kapralos and Nathan Mekuz. Application of dimensionality reduction techniques to hrtfs for interactive virtual environments. In *ACE '07: Proceedings of the international conference on Advances in computer entertainment technology*, pages 256–257, New York, NY, USA, 2007. ACM.

[KMKK08] Bill Kapralos, Nathan Mekuz, Agnieszka Kopinska, and Saad Khattak. Dimensionality reduced hrtfs: a comparative study. In *ACE '08: Proceedings of the 2008 International Conference on Advances in Computer Entertainment Technology*, pages 59–62, New York, NY, USA, 2008. ACM.

[Knu66] Donald E. Knuth. Additional comments on a problem in concurrent programming control. *Commun. ACM*, 9(5):321–322, 1966.

[KZ03a] Jan Klein and Gabriel Zachmann. Adb-trees: Controlling the error of time-critical collision detection. In *8th International Fall Workshop Vision, Modeling, and Visualization (VMV)*, pages 19–21, University München, Germany, nov 2003.

[KZ03b] Jan Klein and Gabriel Zachmann. Time-critical collision detection using an average-case approach. In *Proc. ACM Symp. on Virtual Reality Software and Technology (VRST)*, pages 1–3, Osaka, Japan, October 2003.

[Lam] David Lam. Tokamak. http://www.tokamakphysics.com.

[Lam67] Butler W. Lampson. A scheduling philosophy for multi-processing systems. In *SOSP '67: Proceedings of the first ACM symposium on Operating System Principles*, pages 8.1–8.24, New York, NY, USA, 1967. ACM.

[Lem] Scott Lembcke. Chipmunk. http://wiki.slembcke.net/main/published/Chipmunk.

[LI99] Looking Glass Studios and Irrational Games. System shock 2. Multiple Platforms, 1999.

[Loo94] Looking Glass Studios. System shock. PC Game, 1994.

[Loo98] Looking Glass Studios. Thief: The dark project. PC Game, 1998.

[LtGC] Francisco Leon and the GImpact Community. Gimpact. http://gimpact.sourceforge.net/.

[Luc89] Lucasfilm Games. Indiana jones and the last crusade. Multiple Platforms, 1989.

[Luc90] Lucasfilm Games. The secret of monkey island. Multiple Platforms, 1990.

[LVF+09] Vincenzo Lombardo, Andrea Valle, John Fitch, Kees Tazelaar, Stefan Weinzierl, and Wojciech Borczyk. A Virtual-Reality Reconstruction of the Poème Électronique Based on Philological Research. *Computer Music Journal*, 33(2):24–47, 2009.

[Mac91] David MacKenzie. GNU Autoconf. http://www.gnu.org/software/autoconf/, 1991.

[Men] Dylan Menzies. Phya. http://www.zenprobe.com/phya/.

[Mic91] MicroProse. Sid meyer's civilisations. Multiple Platforms, 1991.

[Mol08] Media Molecule. Little big planet. PlayStation 3, 2008.

[Mon05] Monolith Productions. F.e.a.r. Multiple Platforms, 2005.

[Nin86] Nintendo. Metroid series. Various Nintendo Consoles, 1986.

[Nin07] Nintendo. Wii fit. Wii Game, 2007.

[NtNd] Paolo Nenzi and the NGSPICE developers. Ngspice. http://ngspice.sourceforge.net/.

[NVI07] NVIDIA Corporation. Compute unified device architecture. Proprietary Library, Multiple Platforms, 2007.

[oCSotUoC] Department of Computer Science of the University of Copenhagen. Opentissue. http://www.opentissue.org.

[oEEotGIoT] School of Electrical Engineering of the Georgia Institute of Technology. Xspice. http://users.ece.gatech.edu/~mrichard/Xspice/.

[otUoCaB] EECS Department of the University of California at Berkeley. Spice. http://bwrc.eecs.berkeley.edu/Classes/IcBook/SPICE/.

[Per10] Markus Persson. Minecraft alpha. Multiple Platforms, 2010.

[PL88] Johannes A. La Poutré and Jan van Leeuwen. Maintenance of transitive closures and transitive reductions of graphs. In *WG '87: Proceedings of the International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 106–120, London, UK, 1988. Springer-Verlag.

[Rav08] Arun Ravindran. Reduced dimensional hrtf processing for gaming environments. In *ACE '08: Proceedings of the 2008 International Conference on Advances in Computer Entertainment Technology*, pages 413–413, New York, NY, USA, 2008. ACM.

[RL06] Nikunj Raghuvanshi and Ming C. Lin. Interactive sound synthesis for large scale environments. In *I3D '06: Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, pages 101–108, New York, NY, USA, 2006. ACM.

[Sau04] Scott Saulters. *Visibility Determination of Dynamic Models for Application in Realtime Computer Games*. PhD thesis, The Queen's University of Belfast, June 2004.

[Sch83] Ron Schnell. Dunnet. GNU Emacs, 1983.

[Sie92] Sierra Entertainment. The incredible machine. Abandonware / DosBox, 1992.

[Sie00] Jeremy Siek. Boost graph library. http://www.boost.org, 2000.

[Sim93] Simtex. Master of orion. PC/Mac Game, 1993.

[Sor73] P. G. Sorenson. Interprocess communication in real-time systems. *SIGOPS Oper. Syst. Rev.*, 7(4):1–7, 1973.

[Spi89] J. M. Spivey. *The Z notation: a reference manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[SS00] Mel Slater and Anthony Steed. A virtual presence counter. *Presence: Teleoper. Virtual Environ.*, 9(5):413–434, 2000.

[Sta99] Jos Stam. Stable fluids. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 121–128, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[StOCa] Russell Smith and the ODE Community. Open dynamics engine. http://www.ode.org.

[StOcb] Loki Software and the OpenAL community. Open audio library. http://connect.creativelabs.com/openal.

[StOCc] Steve Streeting and the OGRE Community. Ogre. http://www.ogre3d.org/.

[SW10] Florian Schanda and Philip Willis. A modular physical-simulation methodology. *Workshop on Virtual Reality Interaction and Physical Simulation VRIPHYS*, 2010.

[TA74] Auke Tellegen and Gilbert Atkinson. Openness to absorbing and self-altering experiences ("absorption"), a trait related to hypnotic susceptibility. *Journal of Abnormal Psychology*, 83(3):268 – 277, 1974.

[Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[TB98] Ton Roosendaal and Blender Foundation. Blender. Multiple Platforms, 1998.

[Tea] The Gangsta Wrapper Team. The gangsta wrapper. http://sourceforge.net/projects/gangsta.

[Tho00] Mikkel Thorup. Near-optimal fully-dynamic graph connectivity. In *STOC '00: Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 343–350, New York, NY, USA, 2000. ACM Press.

[TMFSG07] Nils Thürey, Matthias Müller-Fischer, Simon Schirm, and Markus Gross. Real-time breaking waves for shallow water simulations. *Proceedings of the Pacific Conference on Computer Graphics and Applications 2007*, page 8, October 2007.

[Tro94] Trolltech. Qt. http://qt.nokia.com, 1994.

[TRS06] N. Thürey, U. Rüde, and M. Stamminger. Animation of open water phenomena with coupled shallow water and free surface simulation. *Proceedings of the 2006 Eurographics/ACM SIGGRAPH Symposium on Computer Animation*, pages 157–166, Jun 2006.

[TSS+07] Nils Thürey, Filip Sadlo, Simon Schirm, Matthias Müller-Fischer, and Markus Gross. Real-time simulations of bubbles and foam within a shallow water framework. *SCA '07: Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer animation*, pages 191–198, July 2007.

[Val04] Valve Corporation. Half-life 2. MS Windows, Mac OSX, Playstation 3 XBox and XBox360, 2004.

[Val07] Valve Corporation. Portal. MS Windows, Mac OSX, Playstation 3 and XBox360, 2007.

[vdD] Kees van den Doel. Jass. http://www.cs.ubc.ca/spider/kvdoel/jass/jass.html.

[vdDKP01] Kees van den Doel, Paul G. Kry, and Dinesh K. Pai. Foleyautomatic: physically-based sound effects for interactive simulation and animation. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 537–544, New York, NY, USA, 2001. ACM.

[vdDP98] Kees van den Doel and Dinesh K. Pai. The sounds of physical shapes. *Presence: Teleoperators and Virtual Environments 7:4*, pages 382–395, 1998.

[vRF] Guido van Rossum and The Python Software Foundation. Python. http://www.python.org.

[vRF08] Guido van Rossum and The Python Software Foundation. Thread state and the global interpreter lock. *Python/C API Reference Manual*, 2008.

[WBTB93] P. Willis, A. Bowyer, R. Taylor, and G. Bayliss. Virtual manufacturing. *International Workshop on Graphics and Robotics*, April 1993.

[Wes92] Westwood. Dune ii. Multiple Platforms, 1992.

[Wil04] Philip Willis. Virtual physics for virtual reality. pages 42–49, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[WM89] Will Wright and Maxis. Simcity. Multiple Platforms, 1989.

# Glossary

**API** Application Programming Interface. 18, 31, 109

**BGL** Boost Graph Library. 57, 65, 111, 133, 137

**BIP** break in presence. 26

**BSP** Binary Space Partition. 23, 31

**CFD** Computational Fluid Dynamics. 22, 24

**CUDA** Compute Unified Device Architecture. 17, 18, 110

**DC** Direct Current. 85

**GPL** General Public License. 57

**GPU** Graphics Processing Unit. 17, 23, 106

**HRTF** Head Related Transfer Function. 19

**IEC** International Electrotechnical Commission. 119

**LBM** Lattice-Boltzmann Method. 22, 113

**LED** Light Emitting Diode. 118

**LOD** Level of Detail. 24, 105, 106

**LOSD** Level of Simulation Detail. 24, 106, 113

**NS** Navier-Stokes. 22

**ODE** Open Dynamics Engine. 16–18, 57

**OPAL** Open Physics Abstraction Layer. 18, 109

**PCI** Peripheral Component Interconnect. 17

**RBS** Rigid Body Simulation. 15, 62–64, 68, 72–74, 80, 93, 101, 105, 112

**RPG** Role-Playing Game. 12

**RTS** Real-Time Strategy. 13

**SPH** Smoothed Particle Hydrodynamics. 17, 22, 25, 27, 37