Workshop on Virtual Reality Interaction and Physical Simulation VRIPHYS (2010)
J. Bender, K. Erleben, and M. Teschner (Editors)

A Modular Physical-Simulation Methodology

Florian Schanda and Philip Willis

Media Technology Research Centre
Department of Computer Science
University of Bath
Bath BA2 7AY, UK
http://bath.ac.uk/comp-sci/

Abstract

Physical simulation is useful so that the behaviour of objects emerges from the actions performed on them. However, a simulation simulates only one thing: the mechanics of collision behaviour for example. Further physical properties require further simulators and the problems of making them work effectively together escalate. We offer a structured way of making multiple simulations cooperate. The methodology is reviewed, then demonstrated in use with examples of how users might construct novel objects, such as an electric motor, whose properties emerge from the combined effects of the simulations on its components. The approach has potentially wide uses, for example in interactive games, in a virtual teaching laboratory or in interactive virtual museum exhibits. Users can create new objects which behave in predictable ways, discover solutions other than those built in by a game designer or extend a virtual experiment in exploratory ways. For the designer of the game or experiment, our approach requires fewer scripts and gives more play value for the design effort.

Categories and Subject Descriptors (according to ACM CCS): I.6.8 [Simulation and Modeling]: Types of Simulation—Gaming I.6.7 [Simulation and Modeling]: Simulation Support Systems—Environments

1. Introduction

our primary concern is with managing multiple physical simulations. The motivation for this arose from computer games and we will occasionally refer to games in explaining what we have done. However it is equally applicable to other uses of physical simulation.

A typical 3D gaming world might include terrain and characters. The player character roams around and interacts with other characters and with specific objects. Implementation relies heavily on scripting for these interactions, though there may also be a physical simulation associated with the core script. Scripts are costly to develop, partly because they are necessarily specialised but also because of the large number needed in a large world. Moreover they usually define the “solutions” that a player can achieve. Certain objects can be collected, broken, filled with water etc, according to the scripts associated with them. This requires the game designer to anticipate the player’s actions sufficiently well that the game can reach a satisfactory conclusion and so that it is not possible for a player to reach an impasse.

A scripted world is constrained in ways which may not be apparent to the user. A simple example is a door, which turns out to be no more than a texture map and has no behaviour at all. More frustrating is the case where the player can solve a problem in principle but is prevented from deploying the imagined solution because the game designer did not think of it. If there is no script to permit this plausible solution the user’s imaginative effort is lost. Where the gaming world is realistic we mislead the player if they cannot perform realistic actions; we know how to operate the real world.

Physical simulation permits the behaviour of objects to emerge from the actions performed on them. Simulations can be costly though some run fast enough for real-time applications. A more compelling problem is that a simulation simulates one thing: the mechanics of collision behaviour for example. If we require an additional physical property, such as magnetism, then we need another simulation. Typical previous work on physical simulation offers comprehensive solutions for one kind of simulation [LLC10, DMYN08, LD09, VMTF09, NMK*06]. Our brief is to ensure multiple simulations (not even a fixed set) work together so that outcomes remain as the player expects.

We have written a harness which implements a structured way of making multiple simulations cooperate. Having reviewed some earlier work we will describe our methodology and demonstrate it in use. We show examples of how players might construct novel objects, such as an electric motor, whose properties emerge from the combined effects of the simulations. Importantly, such objects can be constructed by the game player from basic components found in the world, even if the game designer did not envisage them being combined. Our approach associates physical properties with components, as in the real world, and the simulation plug-ins for each property do the rest, just as the laws of physics combine in reality. The simulators can be chosen to have greater realism or to offer non-realistic effects such as magic. They can be swapped in and out, according to the immediate need. In combination this opens up a new approach to virtual environment design. It makes possible virtual worlds which are much more fluid and offer a richer range of behaviours.

2. Previous work

Rigid body simulation is perhaps the most advanced area of physical simulation. Its practical evolution can be seen in a series of published games. Doom featured mainly collision detection. Quake included vertical movement and basic effects such as players being pushed around by an explosion. Doom 3 finally applied this simulation to most smaller objects (such as canned drinks or small crates) so that a strong explosion in a store room would knock over most objects.

Other action games have also followed this trend. For example the title Half-Life 2 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.

Some puzzle games also use physics simulations, such as the Sierra classic 'The Incredible Machine'. This provided the player with basic building blocks ranging from familiar 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. The player was presented with a series of levels which had some objective, such as getting all the balls in the bucket, and was given a small number of these objects to place in the world. The player then started the simulation. If the goal was eventually fulfilled, the game progressed to the next level. 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 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 guard would notice the player. Items were included that could mitigate (moss to cover particularly noisy ground such as metal) or take advantage of this (sound emitters attached to arrows which could be shot into a useful location to draw the guards away to investigate allowing the player to move past them unnoticed).

The game Portal [Val07] demonstrates, in a unique way, that unexpected emergent behaviour is a good property. The original Source game engine developed for Half-Life [Val04], was found to have such a flexible physics engine (Havok) that 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. Entering one portal would cause one to emerge from the other, preserving momentum. The player had to find a way of escape, by creating portals. There was often more than one solution, not always the expected one.

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 API. For example OPAL [FRS] (Open Physics Abstraction Layer) 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.

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 is only available on Windows.

An early attempt to demonstrate interactive physical changes in a virtual world was the Virtual Manufacturing project [WBTB93, BTBW95]. Here a virtual workshop was used to cut and process metal in various ways, in order to make a virtual component. This required dynamic updates to a computational model of the object as a result of intersecting it with machine tools. A real component could then be made from the operations performed on the virtual milling machine, lathe and spray gun.

Later work at the same site showed the use of a virtual kit of standard parts to generate more than one new but physically-plausible behaviour [Wil04]. The kit consisted of gear wheels which could be plugged together in various ways and the correct rotations resulted. The gears consisted of two kinds of material, only one of which was conductive. Consequently electrical paths could be created or isolated at the same time, with the correct mechanical and electrical behaviour emerging automatically. However the construction area was a fixed grid and all gear wheels were the same size.

More recently, the STORM approach [MGA07, GGAT08] permits users to model the capabilities of an object and the ways it which it offers interactions to other objects. The authors describe these as “a public activity (the behavior and the object reaction) and an associated interface (a standard protocol of communication) which allows the object to communicate with other unknown STORM objects”. The method usefully abstracts these aspects from the way they are implemented. In common with other methods it requires a relatively heavy investment in creating the virtual object but benefits from getting better interaction between them.

3. Our approach

We now turn to our approach, which offers a generic but lightweight solution. 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.
- **Generic interactions** With multiple kinds of simulation acting on a given object, it is important that these combine plausibly to reach the overall goal of having an environment that ‘just works’.

To illustrate modularity, in dynamics simulation (rigid body simulation and collision detection) three well known examples are described on the web: ODE (a free software package), Havok (a commercial solution, originally provided by Havok and now by Intel) and PhysX (a hardware based solution, originally provided by AGEIA and now by NVIDIA). The modularity makes any existing solutions available to the game physics. They can easily be replaced as better simulators become available. They could even be replaced dynamically in different parts of the game. For example, a game set on a planet might represent gravity as a constant downwards vector. If the play moves to space we might substitute a more involved gravity model for orbiting bodies.

To illustrate generic interactions, consider a massive steel object which is attracted downwards by gravity and upwards by a nearby electromagnetic crane. Gravity and magnetism have to be combined in that object to determine whether the object lifts, whereas a similar object made of non-magnetic metal only responds to gravity. The electromagnet also requires properties to be combined: if the electricity supply is cut off it drops its load and gravity takes over.

To support this, each object in the scene has a list of physical properties. Mass, position and conductivity are all examples of properties. A practical implementation will define types for each of these properties, then instantiate a list of variables of corresponding types for each object. Thus a piece of brass will possess position, mass and conductivity properties while a piece of steel will additionally have magnetism.

In any simulation there will be a physics engine which simulates its particular kind of physics and resolves the forces applied to each object in the scene. As the term “physics engine” is often used more loosely than we intend, we will call these *resolvers*. An example of a resolver is the freely-available ODE rigid body simulator, which we use. Other examples include an electrical resolver, a magnetic resolver and a gravity resolver. Our task is to build a world with multiple resolvers and ensure that they work together to produce consistent outcomes.

Each resolver only know about its kind of physics. However each object in the scene might be susceptible to more than one kind of physics. Some way has to be found to ensure the individual effects of several resolvers can be combined to produce a nett effect. Our approach is to use a lightweight process called an *interaction*. Each interaction takes whatever information it needs from wherever it is held and updates data associated with one specific resolver. Each resolver is associated with a list of zero or more interactions which can update it, so that several interactions can update a single resolver’s data. For example, magnetism, gravity and collisions can all result in forces needed by the dynamics resolver. When the dynamics resolver is next invoked these forces will be combined to move the object accordingly.

We also find it useful to group together some transient property-like features. For example magnetic force is derived from position, rotation, shape and magnetic strength of each object. It is practically useful to record this in one place so that any interaction can use it. Similarly it is useful to have the collision detection produce a list of colliding objects for the dynamics resolver and to generate the forces needed to prevent objects from overlapping in space. These pragmatic features we call *derived-property sets* (DPS) and they can be thought of as bundles which any of the interactions can read but not update. There can be any number of these, according to the needs of the world being constructed.

For simplicity of explanation it is easiest to consider clock-driven simulation, though our approach works just as well for event-driven simulation. The main loop will consist of the following steps:

1. For each resolver, call *advance*.
   This takes the simulation forward one step. Some optimisations are possible at this point: Certain resolvers may only be advanced every *n* time-steps, 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 addition that derived property sets may 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 step (though there may be no work to do for any given interaction).

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.

We now describe our four components in more detail.

3.1. Component 1: Properties and property types

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: the name of an object is one example. 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.

In our approach, properties are referred to by their type. For instance, the position property of an object could be implemented as a vector of floating point values or as a vector of fixed point values etc; however the rest of the system only cares about it being a ‘position’ property. In reality, a given 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 is interested in the position, rotation, velocity, angular velocity and mass of an object, whereas a simplified electricity model only considers conductivity, electrical state (perhaps just ‘on’ or ‘off’) and a list of connections to other objects.

Certain properties are common to 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 is an example for the former, and a pointer to a data structure is 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 3.3.

3.2. Component 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 move each of them to its new position.

For an electricity simulation the corresponding resolver updates the electrical state of any given object, reacting to changes to the electrical circuit, such as new connections and broken 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 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 in the next simulation step where all work is performed in a single self-contained step.

3.3. Component 3: Derived property sets (Re-factoring and abstraction)

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 quite complex in the work they do, sometimes even more complex than a resolver. Most of them update at each time step although if there is no change of the properties that the derived property set depends on, there is no need to update.

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.

Derived property sets provide a common place for 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 which is dependent only on 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 which are currently colliding.
- A list of objects which have just started to touch in this time-step.
- A list of objects which have collided in the previous time-step but no longer touch in the current one.

A very different example of a derived property set is the creation of a shadow volume for dynamic lighting. This can be represented by a derived property set calculated from four properties: Object geometry, position, orientation and the position of a light source. Generally shadow volumes are intended for rendering real time shadows but they can also be used for occlusion queries.

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 would be a rendered image.

3.4. Component 4: Interactions

The purpose of an interaction is to allow one simulation to influence (or interact with) another. For example if a new electrical connection is made which causes a light bulb to change its state to ‘on’, the mechanism which 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: an interaction is required to feed back the contact forces calculated by the “collisions” derived property set into the “rigid body” resolver.

The main purpose of interactions however is to connect different physics simulations together. This is the key to making a world which, from the user view, “just works”. For instance if two metal objects touch, an interaction calls the relevant interaction function in the electricity resolver to create an electrical connection; at the same time a different interaction would lead to the appropriate sound being made by the colliding bodies. 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 properties, resolver properties and/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).

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.

The components we have described form a harness which embraces the designer’s choice of simulations. The simulations can be physically precise, physically approximate or indeed unrealistic. The harness provides the channel for connecting the virtual world model to the required behaviour. There is no presumption in our approach about the dimensionality of the model or of the range of physical behaviours to be included.

4. The Construction Kit Experiments

We implemented our methodology in code and, to illustrate emergent behaviour, we also coded a virtual construction kit for a user to play with. The user is allowed to take standard simple components and fix them together to construct new mechanisms of their choice. These kinds of kits are commonly sold in physical form for children, usually themed around mechanical, electrical or magnetic effects. Ours includes all three of these. There are no “preferred” solutions; the user is encouraged to use whatever knowledge they have to make something work. As we will see, unorthodox solutions were offered and still were made to work.

After some earlier 3D experimentation we went with 2D for this test because it is slower for the user to create useful 3D worlds. It is also less easy to see what is going on and interaction with the world is more awkward for novice users. A 2D world is simpler to manipulate (the obvious “drag and drop” works well), simpler to lay out and easier to visualise. There is nothing inherently 2D about our approach however. We make use of the publicly-available ODE software to handle the collisions and dynamics and this is a fully 3D package.

The components are simple metal bars and disks etc arranged on a flat “play area” to which they can optionally be pinned. This arrangement confines the mechanisms to a plane. There is otherwise no restriction on where the items are placed and they can be moved at any time to provoke different behaviour. The user is presented with the play area and a selection of basic components. They are invited to build a functioning object from the components by assembling them in any way that they choose.

We show here some actual results. We have however renumbered the frames in a consistent way because the actual frame numbers are not relevant. Sequentially-numbered frames correspond to real adjacent frames. A jump in the numbering means that some time elapsed with nothing significant happening in between.

4.1. Experiment 1: Making a door bell

A doorbell is a non-trivial object made from more than one part, showing complex overall behaviour – the bell rings if electricity is supplied causing a magnetic field to draw a metal striker to hit a gong. The user also built a switch to turn the electricity on or off. Figure 1 shows the setup after construction was complete.

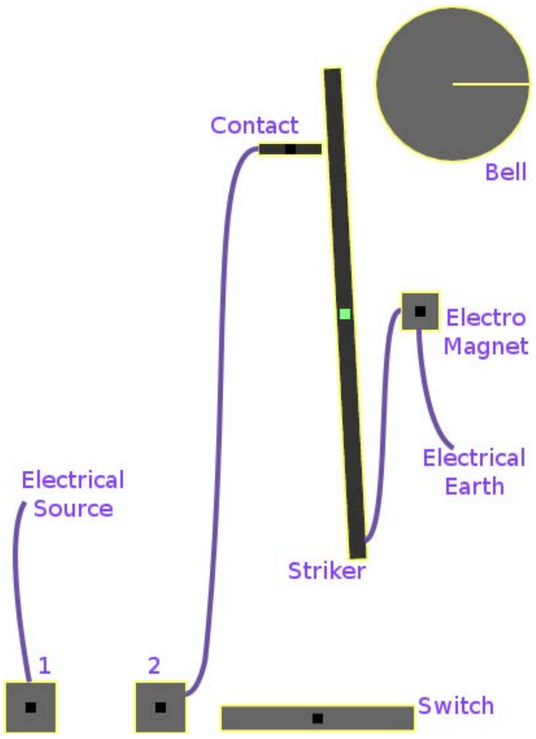

Figure 1: *A screen-shot of the initial setup of the scene. The figure has been annotated, in purple, labelling each object with its name and showing the electrical circuit (omitted later for clarity)*

Once the bell had been constructed the user closed the switch by moving the metal bar to bridge objects 1 and 2. The description below is based on the actual frames which resulted. We give a detailed explanation of how the bell shows the expected behaviour to illustrate how each component of our methodology contributes.

4.1.1. Frame 0 - Setup

Before the first simulation step all the initial properties of the objects are set, as are global properties such as the gravity vector.

The first few frames of simulation are when the user picks up the metal bar and drops it across the two contacts of the switch, so that the electricity will flow. We join the simulation at the frame after the switch has been closed.

4.1.2. 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.

4.1.2.1. 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.

4.1.2.2. 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.

4.1.3. Frame 101 - The electromagnet starts

4.1.3.1. 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 derived property set.

4.1.3.2. Work for Derived Property Sets There is an active electromagnet in the scene and so the striker, which has the property that it is susceptible to magnetism and is close enough for it to be affected, has to have a force applied to it. The magnetic forces DPS will determine the strength of the force pulling the striker. The scene is set up so that this force is stronger than the gravitational force acting on the metal object.

4.1.3.3. Work for Interactions The Rigid-Body-Solver ← Magnetics interaction will apply the magnetic force calculated in the magnetics DPS to the striker 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.

4.1.4. Frame 102 - The striker starts to move

The current state of the scene is shown in Figure 2.

4.1.4.1. 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).

4.1.4.2. 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.

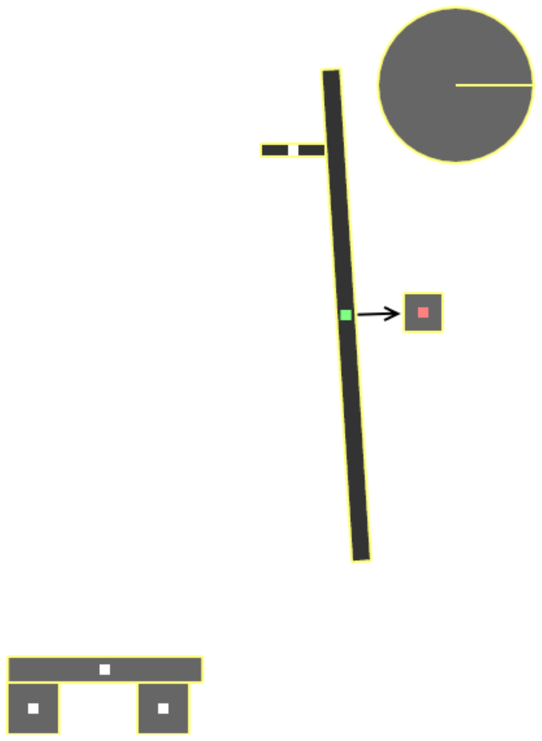

Figure 2: *This annotated screen-shot shows the state of the simulation just after the electromagnet has been activated by closing the switch. The figure has been manually annotated with an arrow to visualise the force exerted on the striker by the electromagnet.*

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

4.1.4.3. Work for Interactions As in the previous frame, the (Rigid-Body-Solver ⟵ Magnetics) Interaction will apply the force calculated in the magnetic forces DPS to the striker.

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

4.1.5. Frame 103 - The connection is broken

4.1.5.1. Work for Resolvers The electricity resolver has been instructed to remove the connection between the contact and the striker from the data structure representing the circuitry. Doing so will break the circuit and set the state of the electromagnet to 'off' once again.

4.1.5.2. 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 more accurately to model real-life electromagnets which also exhibit this behaviour due to self-induction.

4.1.6. Frame 110 - The striker hits 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 reasonably high velocity. Figure 3 shows the current state of the scene.

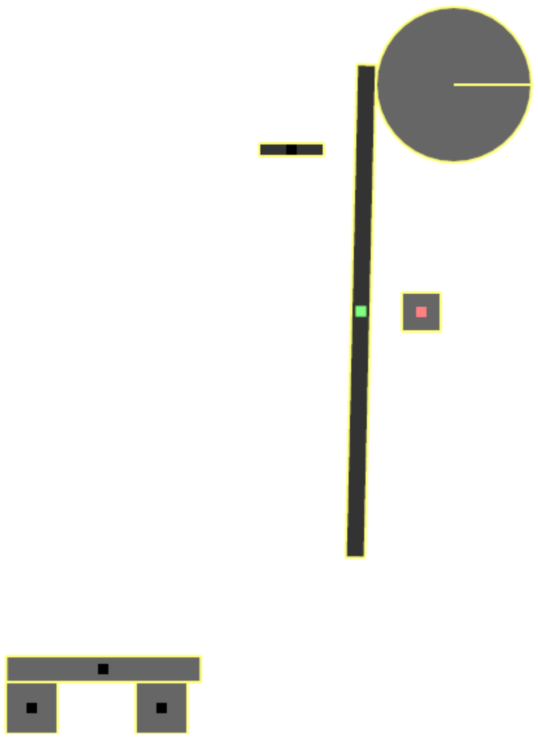

Figure 3: *The state of the simulation just as the striker hits the bell. Since sound cannot be drawn, the reader is asked to imagine a 'ding' noise at this point.*

4.1.6.1. 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.

4.1.6.2. Work for Interactions Since an object has collided with the bell, the (Sound ⟵ Collisions) Interaction will cause a pre-recorded bell sample to play. The rebound, aided by gravity, will cause the striker to return to its start position.

4.2. Experiment 2: Making an electric motor

An electric motor rotates because a magnetic field around the armature repels a similar field generated by the armature. We will use this to illustrate how our approach causes the behaviour (such as rotation) of a complex object (the motor) to emerge from basic properties (electricity induces magnetism; magnetic repulsion). In short, we see a motor built and work as a consequence of its construction.

As it turned out our experimental subject built a non-standard DC motor, with the fixed magnets on the armature and the field generated by an electromagnet. This should still work and indeed it did. The subject had to use an even number of poles in consequence (a traditional DC motor has three or five poles to ensure repulsion) to ensure field alternation and adopted a bias magnet to prevent dead spots. This all worked.

The example is related to that of the doorbell but extends it in the following important ways:

- We have rotary rather than oscillatory motion.
- The motion exhibited by an electric motor is more involved than simple repulsion: the goal is continuous smooth rotary motion.
- For the electrical wiring we have an arrangement in which two circuits have to alternately make and break.
- The distinction between the 'north' and 'south' poles of magnets is important.

Figure 4 shows the motor as built. The sliding contact is connected directly to the electrical source and is hinged at the left end, directly above the motor. It has mass and will rotate clockwise under gravity, falling towards the vertical.

Around the axle is a rotor constructed from eight permanent magnets (the 'spokes') that serve also as conducting contacts. The exposed poles of the magnets alternate North-South as we go round the circle and so an even number are needed. These are are joined to the axle so that they rotate along with it.

To the right of the motor there are the two electromagnets which model a single electromagnet of changing polarity. These provide the field. Depending on which of the spokes the sliding contact is touching, one of the two electrical circuits is closed, activating the electromagnet to repel the spoke nearest to it (and to attract the one below it). The switching of the circuit thus contrives to ensure that the field reverses to oppose the nearest rotor pole and this leads to rotary motion.

Below the main motor a static magnet has been placed. This magnet ensures that the motor will always settle in a position from which it can start turning, just as a real armature "clicks" to a preferred magnetic position. The axle is pinned to the static world with a joint allowing free rotation in the plane (but no other movement). This rotation has frictional forces applied to it, so the motor will come to a

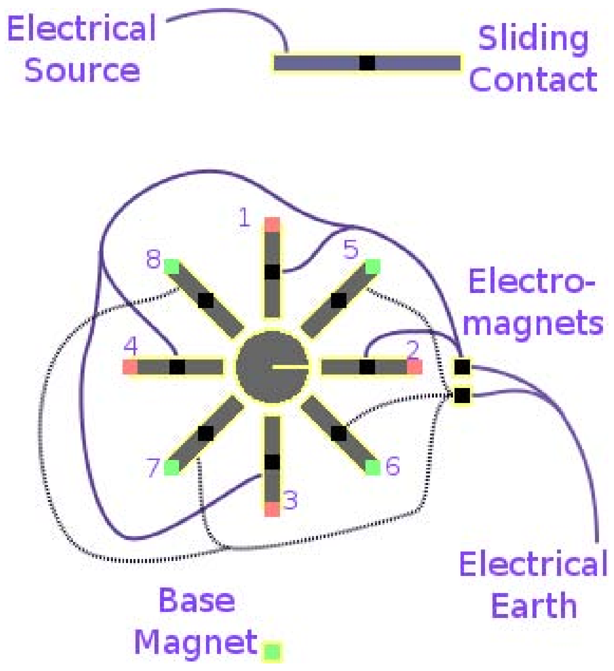

Figure 4: *A screen-shot of the initial setup of the scene. The figure has been annotated, in purple, labelling each object with its name and showing the electrical circuit (omitted later for clarity)*

gradual stop when the electrical circuit is broken. This also ensures that the simulation is stable and the rate of rotation remains steady.

Once electricity is connected to the motor two distinct simulation phases can be identified: The start up, in which the motor begins to turn, and the steady state in which the axle rotates in a regular and smooth fashion. The following descriptions are based on an actual run of the complete system.

4.2.1. 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 that activates the upper electromagnet.

4.2.2. Frames 30 to 59

Since there is some force transmitted from the initial collision of the sliding contact with the top spoke – the components have mass and this is correctly simulated – the axle will turn slightly anticlockwise anyway. However since the 'north electromagnet' is now active, it will further push any 'north spokes' away from it, in particular the one closest to it, increasing the anticlockwise rotation of the rotor

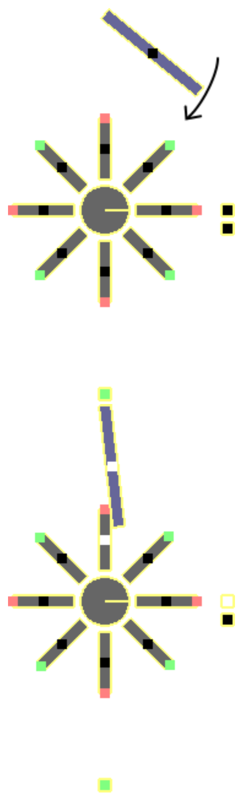

Figure 5: *The left screen-shot shows the initial movement of the sliding contact due to gravity. It has been manually annotated with an arrow to show the movement exhibited by the sliding contact. The right screen-shot shows the moment the sliding contact collides with a spoke for the first time.*

As the axle turns, the 'south' spoke below the one that was initially closest to the magnets at the right in Figure 5 will be the closest south pole to the active north pole of the electromagnet and thus there will be a large attractive force between them, again reinforcing the anticlockwise motion. Figure 6 illustrates these forces. Although magnetic forces act on all the other spokes, they are not significant in comparison to the two main forces due to the inverse square law.

4.2.3. 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 cause the axle to continue to turn anticlockwise.

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

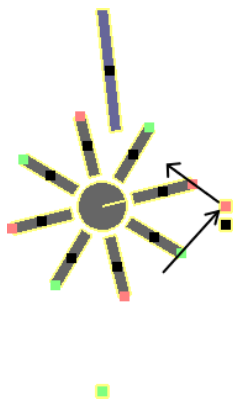

Figure 6: *Illustration of the two main magnetic forces occurring in frames 30 to 59. Red dots represent north poles and green dots represent south poles. Previously the slider has been in contact with the 'north' spoke just to the left of it, causing the 'north' electromagnet to push the 'north' spoke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

4.2.4. Frame 200 onwards

The simulation has now reached a state of constant and steady anticlockwise motion. Figure 7 shows snapshots of approximately one eighth of a full rotation of the axle.

5. Discussion

Our examples illustrate atomic objects being freely combined under user control to give broader behaviours. Our approach does not require objects to be modelled at the most primitive level, even though we have done so. In most real-life situations an electric motor is treated as an atomic object with convenient properties. We would expect this to be the same in a corresponding virtual world. What we additionally offer however is that the motor could be made to drive a buggy or a hoist or be turned into a kitchen mixer. All of these can be achieved by the user whether the designer intended or not, as long as the appropriate components are around. Indeed the designer can turn this to advantage by providing objects which can be disassembled to help solve problems. For an electrical item, the user can extract the motor and use it in other equipment, confident that the "electrically-powered rotation" property will transfer.

What we offer is a harness to permit simulations to co-operate, avoiding the cost of writing many scripts. Our approach trades the high development cost of scripting for the computational cost of simulations. This trade is increasingly

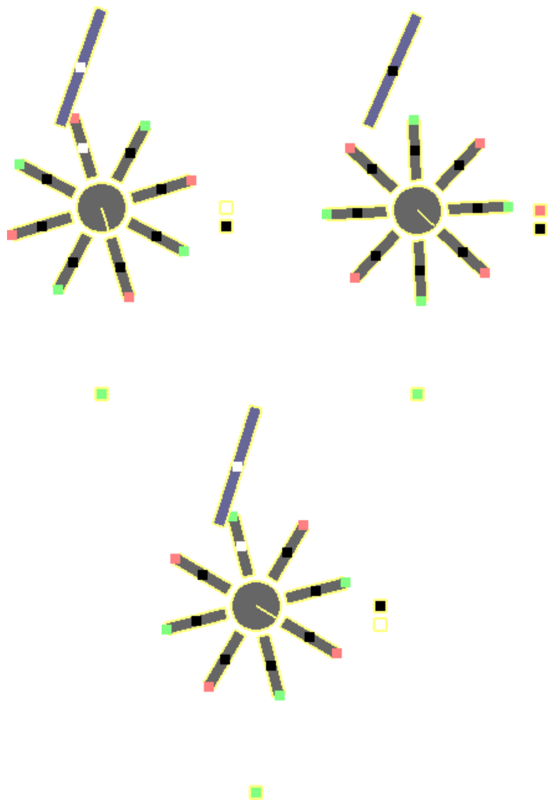

Figure 7: *Illustration of the stable rotation of the motor. On the left the sliding contact has just made contact causing the relevant electromagnet to activate, in the middle the anticlockwise rotation has advanced slightly and on the right the sliding contact has hit the next spoke.*

attractive as computers improve and the cost of skilled programmers rises.

Simulations can be tuned to the problem in hand, being simple and computationally light where precise results are not essential, or heavier where full accuracy is required. They can be dynamically swapped from one environment to another, ensuring that a simulation is only as complex as need be. The same simulations can be used in other games or virtual environments.

In our approach, objects are given physical properties only, not scripts; their behaviour emerges from the simulation associated with each property. There is a direct analogy with what we do and with the real physical world. When objects interact, our software invokes the necessary simulators and applies the relevant forces to produce the emergent action. In the real world, objects have physical properties and the relevant laws of physics act on them in a similar way to generate the emergent behaviour. It is this which gives our approach its strength and generality.

Acknowledgements

We are grateful to the anonymous referees who provided many useful suggestions.

References

[Boe] BOEING A.: Physics abstraction layer. http://www.adrianboeing.com/pal/.

[BTBW95] BOWYER A., TAYLOR R., BAYLISS G., WILLIS P.: A virtual workshop for design by manufacture. *15th ASME International Computer in Engineering Conference* (1995).

[DMYN08] DOBASHI Y., MATSUDA Y., YAMAMOTO T., NISHITA T.: A fast simulation method using overlapping grids for interactions between smoke and rigid objects. *Computer Graphics Forum 27*, 2 (2008), 477–486.

[FRS] FISCHER A., REINOT A., STREETER T.: Physics abstraction layer. http://opal.sourceforge.net/index.html.

[GGAT08] GERBAUD S., GANIER P., ARNALDO B., TISSEUA J.: Gvt: a platform to create virtual environments for procedural training. *Virtual Reality* (2008), 225–232.

[LD09] LENAERTS T., DUTRÉ P.: Mixing fluids and granular materials. *Computer Graphics Forum 28*, 2 (2009), 213–218.

[LLC10] LI H., LEOW W. K., CHIU I.-S.: Elastic tubes: Modeling elastic deformation of hollow tubes. *Computer Graphics Forum 29*, 6 (2010), 1170–1782.

[MGA07] MOLLET N., GERBAUD S., ARNALDI B.: Storm: a generic interaction and behavioral model for 3d objects and humanoids in a virtual environment. *Eurographics Symposium on Virtual Environments (Short papers and posters)* (2007), 95–100.

[NMK*06] NEALEN A., MÜLLER M., KEISER R., BOXERMAN E., CARLSON M.: Physically based deformable models in computer graphics. *Computer Graphics Forum 25*, 4 (2006), 809–836.

[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.

[VMTF09] VOLINO P., MAGNENAT-THALMANN N., FAURE F.: A simple approach to nonlinear tensile stiffness for accurate cloth simulation. *ACM Transactions on Graphics 28*, 4 (2009).

[WBTB93] WILLIS P., BOWYER A., TAYLOR R., BAYLISS G.: Virtual manufacturing. *International Workshop on Graphics and Robotics* (April 19–22nd, 1993).

[Wil04] WILLIS P.: Virtual physics for virtual reality. *Theory and Practice of Computer Graphics* (2004), 42–49.