The Magic of UI Physics

Author Name removed for blind review
Affiliation removed for blind review
Address removed for blind review
Address removed for blind review
Email removed for blind review

Author Name removed for blind review
Affiliation removed for blind review
Address removed for blind review
Address removed for blind review
Email removed for blind review

ABSTRACT
High-fidelity physics enhances user interfaces with realism that more closely engage users. However, indirect control of UI objects through physical forces interferes with direct control of the user experience, which limits the use of physics in productivity UIs. This paper demonstrates how UI physics should and can be enhanced with **magic** that violates physically consistent reasoning to allow for control over the user experience. We have found that magical forces such as teleportation, telekinesis, morphing, and lucky shots can be implemented in a straightforward way via a UI physics engine that supports direct constraints on positions. We demonstrate the compatibility of magic and high-fidelity physics by applying such a physics engine to a UI case study.

ACM Classification: H5.2 [Information interfaces and presentation]: User Interfaces. - Graphical user interfaces.

General terms: Design, Human Factors

Keywords: User interfaces, physics, naturalism.

1 Introduction

The fidelity of user interfaces are increasing rapidly with advances in graphics processors and input hardware such as multi-touch displays. Beyond eye candy and novelty, these resources can be used to build *natural user interfaces* that improve user experience with realistic metaphors. For example, a user can pull and leaf through a photo-realistic fan control rendered on a multi-touch screen with life-like motion and lighting. The user can then leverage their experience about the real world so that the user interface is both easier to use and aesthetically more pleasant.

Physics plays a crucial role in building natural UIs since physical reasoning is needed to both decode physical input such as touch or tilt and encode physically plausible output such as motion. For example, when a virtual object is flicked at another object on a touch screen, physics determines how touch causes velocity, how velocity moves the object and how friction slows the object down, whether the objects collide, and how the collision changes object velocity. Although the use of physics is well understood in games and simulations, *UI physics*–the application of physics to user interfaces–is a new topic. First and foremost, a user interface is not a game: physics must not make the user interface challenging and must enhance or at least preserve user productivity. For example, when flicking an object into a container, the user should not be penalized for bad aim!

In order for UI physics to be viable, it must coexist with **magic** that does not follow from consistent physical reasoning. Magic has two purposes. First, it can remove the challenge aspect of physics; e.g., a flicked object can be drawn to a container via a magical force even if the user's aim would result in a bad trajectory. Second, to paraphrase Randall B. Smith [21], magic breaks natural metaphors to allow users to do wonderful things that are far beyond the capabilities of these metaphors. The use of magic in natural UIs is well established in emerging touch-based platforms such as Apple's iPhone [3] and Microsoft's Surface [14]. This paper analyzes the need for magic in UI physics and describes techniques for incorporating magic into higher-fidelity UI physics.

Software support for physical reasoning exists in a continuum. At one end exists *pseudo physics* where timeline-based animation can be augmented with stateless computations that loosely approximate physical effects. For example, flicking an object can be "faked" by animating the object's position with a speed and destination related to only the start and end positions of the flick gesture. Given direct control, pseudo physics strongly supports magic but only allows for low-fidelity physical reasoning. At the other end of the software physics continuum are high-fidelity *physics engines* that are common in modern games. In our example, flicking an object could involve decoding touch into a force that is applied to the object over time, where the object's momentum is acted against by friction. In contrast to pseudo physics, a physics engine can often only provide weak support for magic, which conflicts with high-fidelity physical reasoning.

Can UI physics be supported with the high-fidelity of a modern game as well as productivity-enhancing magic? The problem with high-fidelity physics engines is that they monopolize control over physical properties so that magic cannot be encoded directly; e.g., moving an object directly will invalidate its velocity. Indirect encoding of magic through "force hacking" is unreliable and very expensive in terms of programming effort. However, we have found significant success in the form of Jakobsen's character physics [9]



Figure 1: A Solid Matte Formula Guide from Pantone as an example of a color-browsing fan.

technique that is based on Verlet integration [23]. Through Verlet, constraints can be expressed directly on object positions, which is allows magic to be encoded with reasonable programming effort. This paper describes our experience in using this technique to implement UI physics.

The rest of this paper is organized around a representative design-to-implementation case study of how physics can be applied to a UI. Section 2 introduces this case study with its physics requirements and describes how these requirements can be realized in software. Section 3 demonstrates through the case study why UI physics needs to be augmented with magic and the challenges of supporting this magic. Section 4 details a technique for supporting magic with UI physics, which is then applied to the case study and evaluated. Section 5 presents related work while Section 6 concludes.

2 Software Physics

The design subject of this paper's running case study is a hierarchical navigation fan widget based on the color-browsing fans that are often found in paint stores as shown in Figure 1. The fan metaphor is modified to support category browsing by associating each blade under the fan's cover with a category, which each contain multiple sub-blades that are associated with the category's sub-categories. Users can then "leaf" through the fan's categories by moving the blades with their fingers or by using the mouse. By using widgets based on the fan metaphor, a natural user interface can engage the user's experience with real fans, making the interface easier to use. The interface is also aesthetically pleasing through the fan blades' circular and fluttering motions.

The fan is a very physical metaphor that is significantly defined by its *kinematic behavior*, which is a description of its motion without regard for how this motion occurs. The fan's basic kinematic behavior is as follows:

- The fan's category blades are arranged around a ring. The blades can be moved by rotation around this ring, where blade endpoints are the center of this rotation.
- The fan can be opened or closed. When the fan is open, each blade is separated by a minimum rotation delta.
- The fan can be moved as a whole where its blades are pulled or pushed into different rotations by this movement.
- Sub-category blades have similar kinematic behavior around their category blades rather than the fan's ring.

The designer documents the fan's kinematic behavior and evaluates it through various paper prototypes. After being refined, the design is presented to developers for production. How can developers realize the physical aspects of the fan design?

An essential component of a natural UI is the ability to both decode input and encode output in a physically plausible way. For example, a touch-based push of an on-screen fan blade should cause the blade to move as if the touch had strength, the blade had mass, and the underlying surface had friction. An understanding of physics enables reasoning about how UI objects would behave if they were real; e.g., Newton's laws of motion can accurately predict how a puck flicks across a uniform surface. Incorporating physics into a UI utilizes a user's intuition about physics, which provides for a more immersive experience. However, effectively leveraging physics in software is not straightforward.

Pseudo physics is one option for software physics where physical principles are partially enforced through ad hoc heuristics that are often based on stateless computations. For example, the resulting velocity of a flicked UI object can be directly based on the flick's start position, end position, and duration. Pseudo-physic techniques will be noticeably inaccurate outside of their simplifying assumptions; e.g., when a flick gesture is performed with a variable velocity. However, pseudo physics still adds significantly to the realism of UIs as demonstrated in Apple's iPhone [3], which has many pseudo physical effects. For example, an iPhone contact list can be flicked with a scroll motion based on terminal velocity and flick length. The contact list can also be re-flicked with a simple gesture that causes a predictable amount of scroll motion. When flicking reaches the end of the list, the motion will bounce as if a simple collision occurred. Such behavior makes the iPhone feel more real even if the effects are heuristic-based.

Pseudo physics is often implemented through *timeline-based animations* where UI object properties, such as positions, are changed over fixed periods of time. Flow control for a timeline animation is limited: the animation can be stopped early as well as parameterized in its duration and *tweening* behavior, which defines the path via which destination values are reached. Various tweening techniques include specifying key frames or by using simple interpolation formulas in the form of Robert Penner's Easing Equations [19]. For example, a cubic ease-out equation, $p_0 + (p_1 - p_0) * t^3$, causes an animation to start slow at $t = 0$ but fast at $t = 1$. Through tweening, a timeline animation can produce motions that resemble physical interactions such as bouncing, collisions, and so on. Multiple pre-cooked parameterized animations can also provide some interactivity through branching. For example, a flick gesture can trigger an animation to the pre-determined destination of the flick with a bounce of this destination is the end of the list.

The disadvantages of pseudo physics include low fidelity, invariability, and poor support for interactivity. Applying

pseudo-physics to the fan case study, without stateful simulation, the basic motion needs can be met only with a lot of inaccuracies. For example, pulling a fan could cause its blade to rotate unnaturally if they rotate independently at all. Invariability arises from limited configurability and the inability to craft animations for infinitely many kind of interaction. For example, the pre-cooked open animation will more than likely not consider that the fan could be moved at the same time. Additionally, invariability in the fan's motion, such as the lack of jiggle when moving, can cause it to appear stilted and lifeless. Considering interactivity, a property cannot be influenced by more than one "force" at a time so an animation on a property has to be stopped while the user is manipulating it directly. For example, a fan blade cannot easily be under an attraction force while the user is pulling it.

Beyond pseudo physics, physics can be simulated much more accurately with a *physics engine* that continuously animates physical properties over time according to the various forces that affect them. Unlike timeline-based animations, properties that are animated by a physics engine can be influenced by multiple forces at the same time. For example, gravity can cause a box to fall, but while it is falling it can collide with a shelf where its trajectory would change accordingly. Besides gravity, other forces that influence position properties include friction, springs, magnets, and touch-based input. In addition to the dynamic simulation of physical properties, physics engines also detects collisions and resolves them in a way that is compatible with dynamic simulation.

Dynamic simulation of positions in a physics engine is often performed through integration techniques that are based on Newton's second law of motion, $F = ma$, and two first-order equations:

$$x_{n+1} = x_n + \Delta t * v_n$$
$$v_{n+1} = v_n + \Delta t * a$$

Where x is position, v is velocity, a is the current acceleration of the body (F/m, where F is the current force and m is the body's mass), and Δt is the time derivative. Analogous equations exist for rigid body torque and rotation. Given a discrete time step (Δt), the above two equations describe the explicit Euler method. In practice, explicit Euler is not very accurate or stable, and other more sophisticated integrators such as Implicit Euler, Backward Euler, or Runge-Kutta. However, knowing explicit Euler is sufficient to understand how a physics simulation is often configured: the programmer defines and updates F either directly or through high-level force constraints such as springs or joints.

Applying a physics engine to the fan case study, the motion would be much more real than pseudo physics. As the fan is pulled, its blades will rotate as expected with the motion. The integration is such that the motion is never repetitive, while multiple forces can act on the blades at the same time; e.g., the user can pull a blade away from the fan while it is being attracted back to the fan through another force. Unfortunately, as we discuss in Section 3, life-like motion comes at the cost of control: the control happens indirectly with the abstractions provided by the physics engine, which limits control over user experience in a UI.

Pseudo physics and physics engine are extreme points on a continuum of software physics solutions that differ in ease of programming, flexibility, efficiency, and fidelity. Pseudo physics is cheap, somewhat easy to program, highly controllable, but has very low fidelity. Achieving higher-fidelity involves more detailed animations that are expensive to build. In contrast, physics engines have higher fidelity but are very difficult to program when control is needed outside of its preferred model. Computer games have progressed from pseudo physics to physics engines such as Intel's Havok [8] and NVIDIA's PhysX engines [18], and are progressing to even higher-fidelity physics with the introduction of hardware accelerated physics computations. In contrast, modern UI toolkits such as Microsoft's WPF [13], Apple's Core Animation [2], and Sun's JavaFX [22] are designed to support pseudo physics through timeline animations. Because timeline animation is well supported in UI toolkits while existing physics engines are game-centric, pseudo physics is more common in existing natural UIs.

3 The Need for Magic in UI Physics

Realism in a natural user interface enhances user intuition and experience by reusing metaphors from the real world. For example, a fan can be pushed, rotated, pulled apart, and leafed through just like a real fan. As the user is familiar with the metaphors, they can quickly use the interface effectively using their own intuition. However, the interface must go beyond realism to serve the user since, being software, it is not bound by reality. For example, a fan could be fully opened by pressing a button where such behavior would seem **magical** on a real fan. Without magic, using a computer offers no improvement over the reality-limited ways of doing things. A natural user interface must then effectively fuse reality with magic in a way that caters both to a user's intuitions and productivity needs.

The fan metaphor's core kinematic behavior alone cannot provide for a very good user experience; e.g., if the fan blade motions move too freely, they would be challenging to view and manipulate. To make the fan widget viable in a productive user interface, the following restrictions and behaviors are added to the design requirements that are considered magical with respect to consistent physical reasoning:

- Fan blades cannot be rotated out of order; e.g., the third blade is always between the second and fourth blades. Likewise, the last blade cannot pass the first or vice versa.
- A discrete tap on the cover will open (or close if already open) the fan as shown in Figure 2. Similarly, a discrete tap on a category blade will open/close its sub-category blades.
- When the fan is open, moving the fan's cover toward the first category blade will cause the fan to begin to close as shown in Figure 3. When the fan is closed, moving the fan's cover away from the first category blade will cause the fan to begin to open.
- When the cover is not being moved, the fan will tend to open or closed based on the rest angle between the cover and the first category blade.

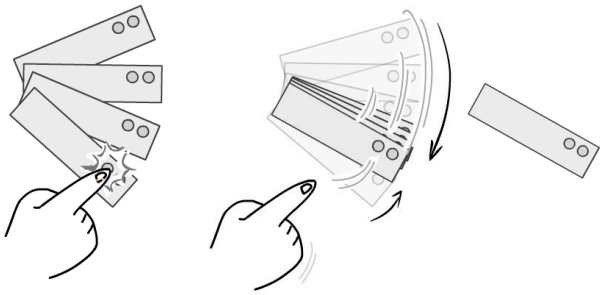


Figure 2: Closing the fan by tapping.

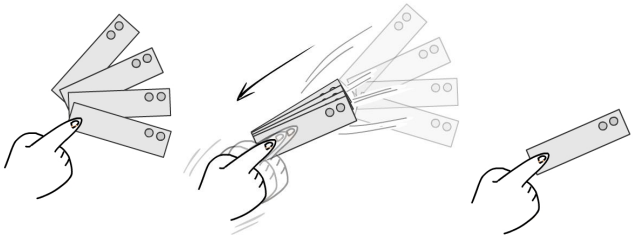


Figure 3: Closing the fan by pulling it.

- A category blade can be detached from the fan via some gesture/move combination; e.g., pulling the blade a certain distance away from the ring while also touching the cover. A user can re-attach a category blade by either moving it back to the fan or by flicking it in the general direction of the fan; the category blade will automatically dock with the fan with a property rotation. Dettachment and flick back are illustrated in Figure 4.

- Upon a specific gesture such as touching the cover with two fingers, fan blade rotation angles are locked so that the whole fan can be moved or rotated without changing the fan configuration. Locking is illustrated in Figure 5.

- Upon a specific gesture on a category blade, the fan will orient itself so that the blade is unrotated to the user as shown in Figure 6.

- Fan blades can be commanded to layout into a grid as shown in Figure 7.

- The entire fan moves more readily when the cover is manipulated as opposed to when its blades are manipulated.

These restrictions are meant to ease manipulation of the fan and enhance productivity. "Tap to open" provides a quick way of opening and closing the fan that is based on a conventional button metaphor, while moving the cover towards the first category blade is more natural. The fan will always drift open or close so that it will not continue to be in a partially opened state. Detachment allows for a user to explore a category blade independently of the rest of the fan. A mechanism for locking the fan is necessary otherwise the user will be afraid to move and rotate it and lose their place. Also, a gesture for reorienting a fan enhances productivity

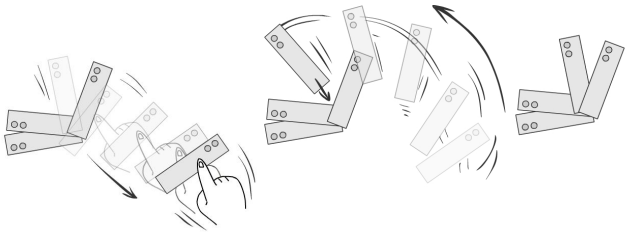


Figure 4: Detaching a blade from the fan and flicking it back.

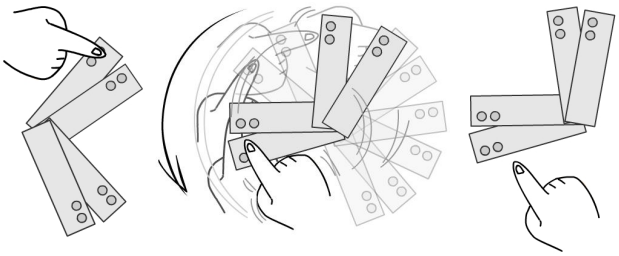


Figure 5: Locking the fan while moving and rotating it.

by making user-interested text unrotated to the user. Finally, allowing for substantial fan movement only by manipulating a "heavy" cover prevents the fan from jittering when the user is leafing through its category blades. However, allowing for a little movement while manipulating "light" category blades communicates to the user that the blades still have some influence on the fan.

None of the mentioned design requirements focuses solely on aesthetic concerns. Instead, the designers started from the fan natural metaphor and fleshed it out in a way that would allow for productive use in a UI. In general, the natural metaphor dictates natural physically consistent behavior, while designers adapt the metaphor to be more usable via magical (physically inconsistent) behavior. Understandably, the designer should be focused on the user experience and unconcerned with what is physically consistent.

To generalize, magic in a UI can be described in terms of how the consistent laws that govern physical behavior are broken. Such laws are broken to improve user productivity and are a significant part of user experience. Magic that is useful in UIs is often analogous to supernatural behavior, consider:

- **Teleportation**: an object appears somewhere on a screen without moving there. Although less natural than motion, teleportation is useful when normal motion is too slow or disruptive.

- **Morphing**: an object can shrink, grow, or change shape. Morphing is useful when expanding and shrinking objects based on an event, activity, or perceived user interest. For example, we can expand the size of a blade when the user hovers over it, indicating activity.

- **Lucky shot**: an object can land in a specific spot without an initial trajectory to that spot. Lucky shots allow users

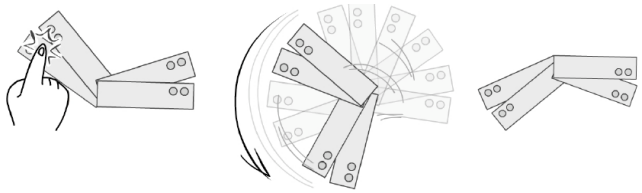


Figure 6: Re-orienting the fan through a tap gesture.

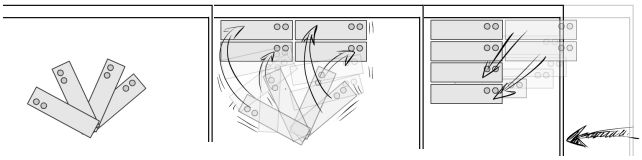


Figure 7: On command the fan moves into a grid layout that is adjusted as the fan's container is resized.

to interact physically with the UI while still using the interface efficiently. For example, flicking a detached blade back to the fan will automatically re-attach it regardless of aim.

- **Super strength**: a touch-based finger is just powerful enough to move an object so that the object remains under the finger at all times. A super strong finger provides the user with necessary control to use touch-based interfaces effectively.

- **Telekinesis**: objects can move seemingly by themselves into precise formations. For example, on command, fan blades can organize themselves into a grid by a specific time.

- **Force fields**: an object's movement is restricted by unnatural barriers. For example, a force field ensures that blades do not rotate out of order.

3.1 Existing Techniques

Magic is easily realized through pseudo physics. Because the details of physical effects are completely specified by timeline animations, physics and magic are equal: everything is specified so anything can be expressed. The drawback, of course, is the lack of fidelity.

Physics engines provide higher fidelity than pseudo physics but make magic much more difficult to encode. Most physics engines assume complete control over physical properties and maintain internal meta-properties about these properties. For example, position properties are often paired with velocity and acceleration properties. Directly changing a physical property causes related internal meta-properties to become inconsistent. For example, touch movement of an object can be implemented by directly updating the object's position, but the object's momentum in this case becomes undefined. With an undefined momentum, flicking the object or colliding it with another object is also undefined.

Pure physical simulation is often not feasible on objects that are externally controlled; e.g., by the player or AI. To handle characters, most physics engines provide support for some kind of *kinematic object* or actor. The position of a kinematic object is set directly by the programmer and can still influence physical objects by "pushing them away." Magic can then be encoded through kinematic objects, although control is limited to pushing. Beyond kinematic objects, another technique for adding magic to a physics simulation is through what we call *force hacking*: magic is encoded as a force that is designed to counteract other forces that go against the magic. Witkin [25] describes a technique called *constrained dynamics* for deriving counteracting forces, although this requires advanced math skills and a lot of programming effort.

Touch-based interaction resembles magic because a touch should have direct consequences to the object being manipulated, which goes against the indirect nature of physics engines. Wilson et al. [24] describe multiple strategies representing touch-based interaction on Surface [14]:

- Apply a precise direct force at the point of the touch based on size and velocity of touch to simulate direct control.

- Attach a virtual spring between a finger and the point where an object is touched.

- Create kinematically-controlled proxy objects, particles, or meshes that interacts with other objects via collision and friction.

The first two options are problematic. Computing a direct force to apply is extremely complicated and prone to inaccuracy as all other forces acting on the object must be counteracted. Using an elastic spring causes the object to bounce around the finger, which leads to a mushy and confusing user experience. Using an inelastic spring can cause the physical system to become over-constrained and unstable. These observation apply to magic in general as described in this section; e.g., using springs to implement telekinetic-like control would have similar results.

The last proxy object technique discussed in [24] can be successful because it works indirectly: the proxy object has no other forces applied to it so kinematic control is effective, while the proxy object interacts with physical objects through conventional friction and collision interactions. However, the proxy object is subject to the limitations of friction and collisions; e.g., the proxy object can "slip" off physical objects and collisions may be too strong or not strong enough. The problem is that friction and collision are being used without necessary inputs–how hard is the user pushing or pressing down? With respect to general magic, the proxy object approach is still problematic because it can only approximate control. Additionally, general magic, as opposed to touch input, can involve numerous custom constraints that could overwhelm a physics engine with proxy objects.

The use of kinematic objects and force hacking is essentially fighting against the physics engine. This might be okay in a game where physics are part of the game's ambiance: magic can manifest itself imprecisely through natural forces in a way that preserves the player's enjoyment. However, productivity-oriented UIs cannot tolerate such imprecision.

4 Magic-friendly High-fidelity Physics

The force-based approach to software physics is based on active entities that indirectly constrain the physical system. For example, a spring specified by a programmer will act to constrain a body to be at or around some point. Although the force-based approach eases the programming of natural relationships, it makes programming magic difficult as we cannot manipulate first-order properties such as position or second-order properties such as velocity. An alternative to the force-based approach is through *impulses* that allow us to manipulate second-order properties such as velocity directly. However, it is still very difficult to derive the impulses that can effectively encode specific magic.

To ease physics programming in general, Jakobsen [9] proposes using Verlet integration [23] that does not explicitly trace velocity. Instead, the last position (x_{n-1}) is saved and used in the computation of the new position (x_{n+1}) according to the current position (x_n):

$$x_{n+1} = 2x_n - x_{n-1} + a * \Delta t^2$$

However, rather than compute x_{n+1} all at once with one acceleration (a), the force computation can be spread out over multiple constraints expressed directly on x_{n+1}, which is exactly what we need to express magic! Magic occurs by *projecting* x_{n+1} to the desired arbitrary location, where the system will automatically compute a viable velocity for the resulting movement. For example, touch dragging an object can be implemented directly specifying the object's desired position, expressed as $x_{n+1} = p_{mouse}$, where flick and collision will just work as desired according to the mass of the body. Expressing super strength magic is then not a problem since displacement is expressed directly.

As with magic, natural forces are also encoded as position constraints; e.g., a spring can be expressed as:

$$x_{n+1} = \texttt{lerp}(x_{n+1}, p_s + \texttt{spring}(x_{n+1} - p_s, l_s), .5)$$
$$\texttt{spring}(v, l) = v * l/|v|$$
$$\texttt{lerp}(x_{from}, x_{to}, t) = x_{from} + t * (x_{to} - x_{from})$$

where the spring's anchor is at p_s, its rest length is l_s, and the spring's "stiffness" is such that the constrained position travels halfway toward the spring's goal point each time the constraint is solved. Because natural forces are expressed directly as constraints, we can modify their behavior as needed to encode magic. Consider that linear interpolation (`lerp`) allows us to adjust the strength (t) of a constraint dynamically. A spring with a rest length (l_s) of 0 is disabled when its strength is zero. However, we can directly animate this strength from zero to one in one second, causing the position to travel to the spring's anchor (p_s) in one second while still being influenced by other constraints such as collisions until the second is over. A generalization of this technique is referred to as *inverse kinematics* [5] and allows physics to be used as tweens for otherwise standard animations. It is also through inverse kinematics that telekinesis magic can be encoded: increasingly strong springs can draw bodies to desired locations by desired times.

Multiple constraints on the same property are supported via *relaxation*: constraints are successively applied in a loop some number of times so that they converge to being satisfied. Since constraints are not solved at the same time, solving one constraint can overwrite a solution for a previous constraint. However, as long as the system is not over-constrained and all constraints are solved with respect to the property's current value, relaxation converges to a solution for all constraints or some kind of compromise if a solution does not exist. **Absolute magic** completely ignores a property's current value, such as $x_{n+1} = p_{mouse}$, and is not desirable as it ignores the effects of momentum and other constraints. However, constraint strength is adjustable through interpolation; e.g.,
$x_{n+1} = \texttt{lerp}(x_{n+1}, p_{mouse}, .9)$
causes the mouse position goal to be satisfied by 90% so that the magic is very strong but still allows for the influence of other constraints. Additionally, many constraints are at least related to the current value of the property being constrained even if they are at full strength; e.g., a spring still depends on position's value to compute the goal point.

Control is enhanced with Verlet by not modeling rigid bodies directly. Instead, the corners of a rigid body can be modeled as a set of particles with constraints that pull them into the rigid body's shape. Such constraints can either be modeled as very stiff springs or via *shape matching* [16] where the orientation and rotation of the body is computed from particle positions. By modeling constrained particles instead of rigid bodies, rotation and torque are both implicit in the configuration of the particles. As a result, rotation will automatically occur as positions are modified, or, if shape matching is used, can be manipulated directly according to convenience. Alternatively, constraints that keep the body in shape can be weakened to allow for deformation, creating *soft bodies*. Expanding on this technique, the body constraints can be modified dynamically to change the body's size or geometry, allowing us to easily encode morphing magic: morphing is simply encoded as body particle movements.

Because velocity is computed according to position displacement, Verlet is very stable. As a result, the physical simulation has less chance of going berserk, even in the presence of magic that drastically changes property values. However, velocity-dependent forces and impulses, such as friction and accurate collision response, are more difficult to express. Collisions are handled through simple projection: move the particles involved out of each body. However, projection does not create the full "bounce" that a collision between two bodies would otherwise incur; i.e., momentum and energy are not conserved correctly. Likewise, without velocity friction cannot be computed. Velocity can be manipulated by updating the previous position (x_n when x_{n+1} is being computed), but accuracy suffers because the final constrained x_{n+1} is unknown. The previous position is sometimes manipulated to express magic that manipulates momentum. For example, teleportation magic must set both the current and previous positions to the target position to prevent the particles from carrying on as if they covered the displacement by super-fast motion.

4.1 Prototype

We do not claim that Jakobsen's Verlet-based method is the best solution for encoding UI physics. Instead, we use it as an example of how magic can be effectively encoded through a physics engine that allows constraints to be expressed directly. To evaluate this claim, we have implemented Jakobsen's method in a prototype UI physics engine and applied to the fan case study. Our UI physics engine is programmed through constraints so that behavior specific to the fan can be implemented in less than five hundred lines of code. Beyond particle positions, we can also express constraints over the rotation of a body or its center point. Custom properties are defined to store intermediate state and constraints. For example, a fan's "open state" is a custom percent property that is used to constrain blade rotation so that delta angles between decrease as the fan is closed. The open state state property is then itself constrained according to the delta angle between the fan cover and first blade, when the cover is being manipulated, or to increase or decreasing depending on whether it is below or over 50% closed. All of these constraints are expressed directly as assignments:

```
body.Rotation.Relax[.5, OpenState < 1] =
 prev.Rotation + AngleDelta * OpenState;
body.Rotation.Relax[.5, OpenState < 1] =
 next.Rotation - AngleDelta * OpenState;
OpenState.Relax[1, !bodies[0].IsTouched]
 += (OpenState >= .5) ? (+.01) : (-.01);
OpenState.Relax[1, bodies[0].IsTouched] =
 ((bodies[1].Rotation - bodies[0].Rotation)
 / AngleDelta);
```

The above code expresses four constraints that use and constraint an `OpenState` property. The first two pair of constraints set the angle between two blades based on the open state if the open state is not at 100%, in which case the blades can move more independently, and since they are a pair the constraints must be applied with a 50% strength. Because blades are interdependent, this restriction is split into two constraints on both the next and previous blades. The last two constraints then set the open state based on its current value if the cover (`body[0]`) is not being manipulated (`IsTouched`), or based on its difference with the first category blade if it is being manipulated.

4.2 Experience

The most obvious way to judge the result of our approach is to observe the behavior of the produced prototype. Therefore, we describe this experience as a walkthrough and narration of a demonstration in one plus five scenes.

Scene 0: first, we demonstrate what the fan is like without magic. The fan blades rotate naturally, but are otherwise difficult to manipulate without restrictions on freedom of movement.

Scene 1: the fan is closed at the beginning of this scene. The user opens the fan by tapping on its cover, or by dragging the cover up away from the first category blade. Either action feels mechanical to the user: either as an immediate spring loaded action that is triggered by tapping, or a synchronous uniform action where each blade is in synch with cover movement through multiple invisible cogs. Magic is often rationalized in the user's mind through complex machinery even if none is involved in the implementation. This mirrors how people rationalize real and fictional artificial motion; e.g., a car moves by its engine, a robot walks by motors, and Captain Kirk teleports by a transporter.

Scene 2: the user leafs through the category blades. She can group the blades around the fan ring, but cannot move the blades out of order. The user taps on a category blade and the sub-category blades "bloom" out. The user can leaf through these blades and open another category as desired. Finally, once the user gets the fan in a desired configuration of blade rotations, they can move and rotate the fan without disturbing this configuration through a gesture on the cover blade. The stiffness of this lock is configurable via a design-time slider: a weak lock will cause the blades to jiggle a bit more during movement before settling on the locked rotation once movement stops, while a strong lock will cause the fan to move stiffly as one fixed unit. There are design advantages and disadvantages of each choice: a weak lock seems flexible but toyish, while a strong lock is rigid but lifeless. The best choice lies between these two extremes.

Scene 3: through a gesture on a category blade, the user can cause the fan to reorient as a unit, making the category blade easier to read. As with lock, the stiffness of this re-orientation can be tweaked so that the re-orientation is bouncy (weak) or jerky (strong). The bouncy case is similar to exaggerated motion in cartoon animation [6], although this effect is an artifact of a weak constraint that will not cancel velocity out.

Scene 4: the cover blade has a lot more mass than the category blades, so moving a category blade does not cause the fan to move very much. As a result, the user can leaf through the category blades without accidentally moving the fan. The user can detach a category blade by pulling it away from the fan, which will cause a small jerk in the fan before the blade is free and can be rotated independently. The blade is re-attached by moving it near the fan. Alternatively, lucky shot magic can be used: the user can toss the blade in the general vicinity of the fan and the blade will automatically re-attach itself to the fan.

Scene 5: the user can instruct the fan blades to detach and orient themselves in a grid, which is physically realized via inverse kinematics. Again, we can adjust the stiffness of this grid force to control how bouncy or jerky movement into the grid is. The number of columns is determined by the size of the application and can change dynamically: when the user resizes the window to be more narrow, the blades automatically move so that there are more rows and fewer columns. Finally, the user can undo the grid so that the blades fly back together to form a fan.

4.3 Comparisons

We compare our physics technique to hypothetical implementations through pseudo physics or using a game-oriented physics engine. First, a game-oriented physics engine approach requires either encoding the restriction as a natural force or barrier or using a kinematic object. The basic kine-

matic behavior of the fan can be implemented through sliding joints around the ring, springs that pull the blades into valid rotations, and colliding obstructions that prevent the blades from sliding past each other. An approach that is analogous to controlling a string/spring puppet can then be used to implement most of the described magic: springs and joints pull the blades to a certain angular configuration while the fan is being opened via cover movement; while springs can pull the blades back to a locked configuration. The drawback of this approach is that force-based physics engines cannot handle multiple kinematically moving joints or stiff springs very well, while elastic springs can create too much jiggle in the fan to be very useable. This is not to say that the physics engine approach is unworkable: a programmer can spend time tuning the physics to increase control and reduce jiggle with, for example, additional opposing springs, using more damping, getting rid of springs when over-constraining occurs, and so on. Alternatively, the programmer can modify the engine directly with custom fan constraints. However, either approach is expensive with respect to programming effort.

The pseudo physics approach emphasizes control: the programmer can simply move the fan blades to wherever he or she wants. As a result, all design requirements can be implemented directly. However, motion behavior must be added manually to the fan, probably as a heuristic. For example, when the fan is moved by its cover, the rotation difference of each blade should either decrease, if the fan ring is moving away from the blade, or increase, if the fan ring is moving toward the blade. Alternatively, the user could move the ring in a direction perpendicular to the blade, causing it to rotate appropriately. This behavior constitutes a mini-motion model that the programmer must encode herself–components such as the Microsoft's Inertia Processor [12] are of little help because they are designed to deal with non-pivoted rigid bodies. Additionally, if the cover blade reaches a certain threshold angle with the first category blade, opening or closing behavior must also come into effect; completely overriding the motion model.

The primary drawback of the pseudo physics approach is fidelity; i.e., the fan is functional but lacks "life." For interactions that are initiated with discrete gestures, such as tapping the cover to open or close the fan, time line animations can be tweened with acceleration so that the behavior seems more real. However, our options are much more constrained for continuous interactions, such as moving the cover of the fan. Expected momentum is missing from motion; e.g., the fan blades do not flutter as they are pulled by the cover, they just move very mechanically. Ironically, whereas too much jiggle can be a problem in the pure physics approach, the complete lack of jiggle in the pseudo physics approach is unsettling. To improve the aesthetics of the fan, we could add artificial jiggle to fan blade rotations through tweened timeline animations, which is similar to adding an artificial bounce to a flick that hits the end of a list. Although artificial jiggle can improve the aesthetics of the fan somewhat, the lack of fidelity means that it does not correspond very well to motion initiated by the user.

Ease of programming not only concerns how much code is written but also the technical skills required of the persons writing this code. In our approach, force hacking, which requires advanced calculus skills, is not involved because constraints are expressed directly as modifications to properties. However, expressing the constraints still involves substantial mathematical reasoning, especially in the areas of geometry and trigonometry. For example, writing constraints to prevent blades from passing themselves is non-trivial since angles loop; e.g., 10 degrees is often greater than 350 degrees! Because of this, writing constraint code requires some domain knowledge in math or physics, while the potential for math-free design tooling support appears low. This is in contrast to timeline animation which is commonly expressed directly by designers using high-level design tools. If a physics engine is being configured using existing constraints, this also can be done in a design tool: force-based constraints such as springs are expressed easily as simple connections. However, at the high-end of physics engine programming, a strong skills in physics and advanced calculus is required. In contrast, our approach requires basic skills in physics and math that could be expected from a typical UI programmer.

We can draw two conclusions from this case study: the fan's magical features are feasible and can be implemented with an acceptable amount of effort at a high fidelity. We argue that the fan case study is of a sufficient complexity that we can draw a broader conclusion for general natural UIs. Jakobsen's approach is promising as an enabler for UI physics given that it allows magic to be expressed as direct constraints on physics properties. On the other hand, we have by no means shown that Jakobsen's approach is the only way: our inability to implement the fan with a game-centric physics engine is not evidence that such engines are inappropriate. Today, a programmer with deep physics knowledge, force-hacking skills, and spare time could probably implement the fan on a game-oriented physics engine. Regardless, even if game-oriented physics engines are inappropriate for user interfaces today, our experience with Jakobsen's approach shows that it is possible in the future to construct physics engines that are appropriate for user interfaces.

5 Related Work

The inclusion of magic in user interfaces was explored by Smith [21] in his description of the Alternate Reality Kit (ARK), which is a tool for creating interactive simulations. He concludes that metaphors should be broken to benefit users to take advantage of the virtual environment, and more so that magic does not necessarily make the user interface harder to learn. For very similar reasons, we believe that magic in UI physics is both useful and acceptability from a usability perspective, although work presented in this paper focuses on how to deal with magic in implementation.

Animation in user interfaces have evolved from providing continuity [26, 4] to copying realistic physical behavior. Techniques for animation through retained graphics were first explored in Amulet [17], which are now standard in modern UI toolkits such as Microsoft's WPF [13] or Sun's JavaFX [22]. Chen and Ungar [6] expand on how user interface animation can be augmented with the pseudo-realism of cartoon physics

such as through the use of motion blur and exaggerated slow in/out movement. User interface toolkits designed around timeline animation support the expression of such pseudo-physics either directly or through additional techniques such as by using Robert Penner's Easing equations [19]. This paper works toward bridging the gap between timeline animation and pseudo physics to the high-fidelity physics of physics engines.

Various projects have applied higher-fidelity physics to user interfaces. Mander et al [11] explores how physical piles can be added to a user interface. BumpTop [1] goes further by using a physics engine to add physical properties to a pile-oriented desktop. BumpTop identifies the tension between pure physics and usability through a design goal of *polite physics*: physical behavior disabled to avoid usability problems such as messy or jittery piles. Our work focuses on the techniques of making polite physics possible, namely through the support of magic in physics engines.

With the advent of touch, multi-touch, and touch-enabled tables and walls, much work has recently focused on physics-like interactions with respect to touch-based environments. Kruger et al. [10] explores rotation while Reetz et al. [20] explore flicking using physical rules, both of which involve substantial tweaking of pure physics for reasons of usability. Wilson et al. [24] explores how touch in general can be represented in physics. Wilson's work, in particular, attempts to remain pure to physics concepts that are supported by existing game-oriented physics engines. Our work focuses on the more general problem of UI physics magic and not on specific issues related touch. However, we would classify many techniques described in this work as magical and best expressed through a magic-friendly physics engine: touch-based control could be tweaked through arbitrary position-based constraints that allow for unnatural but user-satisfying manipulations.

This paper focuses on how physics can be applied to user interfaces and does not innovate on the techniques and algorithms of physics-based simulations. Erleben et al. [7] overview the techniques of cutting edge physics engines such as PhysX [18] and Havok [8]. However, for reasons of programmability and control, we forgo many more advanced physics techniques and use Jakobsen's [9] relatively straightforward technique based on Verlet Integration [23]. Whereas most physics engines are based on forces that update body acceleration or impulses that update body velocity, Jakobsen technique allows us to update positions directly and hence encode magic in a very direct way. Jakobsen's technique has been generalized and enhanced by Müller et al. [15] into what they refer to as *position-based dynamics*, which they then apply to cloth simulation. Alternatively, Witkin [25] describes how positional constraints can be defined in force-based simulations using a technique called *constrained dynamics* to counteract the forces with Lagrange multipliers, with the added benefit of energy conservation. UI physics can probably benefit from either position-based dynamics or constrained dynamics along with other more advanced simulation techniques.

As an application of physics to animation, our work is closely related to work on physical animation, ragdoll physics, and inverse kinematics [5], which all use high-fidelity physics to tween animations. Inverse kinematics is especially useful in character animation where the low-level details of activities such as walking can be computed automatically via constraint solving. With respect to UI, these techniques are forms of magic that pulls objects into a fixed configuration. Given support of Jakobsen's technique for direct position manipulation, it is ideally suited to techniques such as inverse kinematics as applied in Jakobsen's Hitman [9]. Our work shows how user interfaces are similar to animation in that control is needed through techniques like inverse kinematics, and generalize this need through support for magic.

6 Conclusion

This paper has shown how magic is needed in natural user interfaces and how this magic can be economically realized in a UI physics engine. Given cheap and controllable UI physics, designers can improve user experience by adding high- fidelity physics to their natural UI designs with confidence that developers will be able to realize these designs in code. In the future, physics engines tailored specifically to the strong magic-needs of UIs could be incorporated into mainstream UI toolkits similar to how timeline animation are now incorporated into WPF [13], JavaFX [22], and Core Animation [2]. Future work includes exploring how game-centric physics engine technology can be adapted to fit the needs of productivity user interfaces, and exploring the application to UIs of other physics-like technologies such as lighting and sound.

ACKNOWLEDGMENTS

This section should be left blank for blind review

REFERENCES

1. Anand Agarawala and Ravin Balakrishnan. Keepin' it real: pushing the desktop metaphor with physics, piles and the pen. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 1283–1292, New York, NY, USA, 2006. ACM Press.
2. Apple Inc. *Core Animation*. http://www.apple.com/macosx/technology/coreanimation.html.
3. Apple Inc. *iPhone*, 2007. http://www.apple.com/iphone.
4. Ronald M. Baecker and Ian Small. Animation at the interface, 1990. A chapter (pp. 251–267) in Brenda Laurel, editor, The Art of Human-Computer Interface Design, Addison-Wesley.
5. Ronen Barzel and Alan H. Barr. A modeling system based on dynamic constraints. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 179–188, New York, NY, USA, 1988. ACM.
6. B. Chang and D. Unger. From cartoons to the user interface. In *Proc. of UIST*, pages 45–55, 1993.
7. Kenny Erleben, Jon Sporring, Knud Henriksen, and Kenrik Dohlman. *Physics-based Animation (Graphics Series)*. Charles River Media, Inc., Rockland, MA, USA, 2005.
8. Intel. *The Havok Physics Engine*.
9. T. Jakobsen. Advanced character physics. In *Proc. of Game Developer Conference*, 2001.

10. Russell Kruger, Sheelagh Carpendale, Stacey D. Scott, and Anthony Tang. Fluid integration of rotation and translation. In *CHI '05: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 601–610, New York, NY, USA, 2005. ACM.

11. Richard Mander, Gitta Salomon, and Yin Yin Wong. A "pile" metaphor for supporting casual organization of information. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 627–634, New York, NY, USA, 1992. ACM.

12. Microsoft. *Manipulations and Inertia*. http://msdn.microsoft.com/en-us/library/dd317309(VS.85).aspx.

13. Microsoft. *Windows Presentations Foundation*. http://msdn.microsoft.com/en-us/netframework/aa663326.aspx.

14. Microsoft. *Microsoft Surface*, 2007. http://www.surface.com.

15. Matthias Müller, Bruno Heidelberger, Marcus Hennix, and John Ratcliff. Position based dynamics. *J. Vis. Comun. Image Represent.*, 18(2):109–118, April 2007.

16. Matthias Müller, Bruno Heidelberger, Matthias Teschner, and Markus Gross. Meshless deformations based on shape matching. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 471–478, New York, NY, USA, 2005. ACM.

17. Brad A. Myers, Robert C. Miller, Rich Mcdaniel, and Alan Ferrency. Easily adding animations to interfaces using constraints. In *In Proceedings of the ACM Symposium on User Interface Software and Technology (UIST '96*, pages 119–128, 1996.

18. NVIDIA. *NVIDIA PhysX*.

19. Robert Penner. *Easing Equations*. http://www.robertpenner.com/easing/.

20. Adrian Reetz, Carl Gutwin, Tadeusz Stach, Miguel Nacenta, and Sriram Subramanian. Superflick: a natural and efficient technique for long-distance object placement on digital tables. In *Graphics Interface 2006*, pages 163–170, June 2006.

21. R. B. Smith. Experiences with the Alternate Reality Kit: an example of the tension between literalism and magic. In *Proc. of CHI+GI*, pages 61–67, 1987.

22. Sun Microsystems. *JavaFX*. http://www.sun.com/software/javafx/.

23. L. Verlet. Computer experiments on classical fluids. i. thermodynamical properties of Lennard-Jones molecules. In *Proc. of Phys. Rev.*, 1967.

24. A. D. Wilson, S. Izadi, O. Hilliges, A. Garcia-Mendoza, and D. Kirk. Bringing physics to the surface. In *Proc. of UIST*, 2008.

25. Andrew Witkin. Constrained dynamics. 1997. http://www.cs.cmu.edu/ baraff/pbm/pbm.html.

26. David D. Woods. Visual momentum: a concept to improve the cognitive coupling of person and computer. *International Journal of Man-Machine Studies*, 21:229–244, 1984.