Hindawi Publishing Corporation
International Journal of Computer Games Technology
Volume 2014, Article ID 623809, 12 pages
http://dx.doi.org/10.1155/2014/623809

Research Article

Desirable Elements for a Particle System Interface

Daniel Schroeder and Howard J. Hamilton

Department of Computer Science, University of Regina, Regina, SK, Canada S4S 0A2

Correspondence should be addressed to Howard J. Hamilton; howard.hamilton@uregina.ca

Received 11 September 2013; Accepted 29 October 2013; Published 5 January 2014

Academic Editor: Ali Arya

Particle systems have many applications, with the most popular being to produce special effects in video games and films. To permit particle systems to be created quickly and easily, Particle System Interfaces (PSIs) have been developed. A PSI is a piece of software designed to perform common tasks related to particle systems for clients, while providing them with a set of parameters whose values can be adjusted to create different particle systems. Most PSIs are inflexible, and when clients require functionality that is not supported by the PSI they are using, they are forced to either find another PSI that meets their requirements or, more commonly, create their own particle system or PSI from scratch. This paper presents three original contributions. First, it identifies 18 features that a PSI should provide in order to be capable of creating diverse effects. If these features are implemented in a PSI, clients will be more likely to be able to accomplish all desired effects related to particle systems with one PSI. Secondly, it introduces a novel use of events to determine, at run time, which particle system code to execute in each frame. Thirdly, it describes a software architecture called the Dynamic Particle System Framework (DPSF). Simulation results show that DPSF possesses all 18 desirable features.

1. Introduction

A *particle system* is a structure used to control the behavior of many elements called particles, where a *particle* is an object with some properties such as position, velocity, and size. Particle systems are typically implemented in software, and each particle is visualized on screen as a colored pixel, a texture (i.e., an image), or a polygon. Particle systems are widely used in video games and films to generate special effects and model fuzzy objects that do not have well-defined shapes, such as fire, smoke, flowing liquids, dust, clouds, fog, snow, rain, hair, fur, sparks, explosions, and abstract visual effects such as magic spells and glowing trails. These effects help immerse viewers in virtual environments by adding detail to them, as well as by making them more attractive. In addition to creating special effects, particle systems have many practical and research applications, such as visualizing and controlling implicit surfaces [1–6] and mesh deformation [5, 7].

When creating a particle system, a number of issues confront the client, where the *client* is the person creating the particle system, such as a designer or programmer. These issues include deciding how the particles should be drawn, how the particles should be managed in memory for efficient performance, which properties the particle system and its particles should have, and how algorithms should be coded for common particle system operations, such as updating the position of a particle according to its velocity.

To avoid dealing with these issues, every time a new particle system is required, Particle System Interfaces have been developed to assist clients with creating particle systems. A *Particle System Interface* (PSI) is a piece of software designed to perform common tasks related to particle systems for clients, while providing them with a set of parameters whose values can be adjusted to create different particle systems. PSIs allow clients to quickly create particle systems, which in turn produce new effects. By using a PSI, clients can also avoid having to design and implement the particle systems themselves. Instead, clients simply use the interface of the PSI to create particle systems and run simulations, which avoids issues such as how particles should be drawn and how they should be managed in memory. Additional terminology related to PSIs is described in Section 2.

A PSI is typically either interactive software with a Graphical User Interface (GUI), a set of related code that can be added to a client program, or a software library with an

Application Program Interface (API). Although many PSIs exist that can be used to create great visual effects, most PSIs are only useful for creating specific types of effects and are not easily extended. For example, the Particles [8] and Particle 3D [9] PSIs are capable of producing impressive fire and explosion effects, but they are not designed to handle collisions between particles and the virtual environment. Therefore, they are not suited to producing effects such as water cascading down over multiple surfaces or smoke from a fire accumulating near the ceiling in an enclosed area. If clients require some functionality that is not supported by a PSI, such as particle-environment interactions, they must either find another PSI that meets their requirements or create their own particle system or PSI. Unless the effect that the client is trying to create is simple or common, he or she will most likely have to create a particle system or PSI from scratch.

The goal of our research was to design, implement, and evaluate a software framework that is flexible enough to support creating particle systems for diverse applications while making the process of creating particle systems fast and easy for clients. This paper makes three original contributions to research. The first original contribution, as described in Section 3, is the identification of 18 features that are appropriate for a PSI capable of creating a wide variety of effects. A PSI with all these features should be applicable to the great majority of applications that require particle systems. Clients will likely be able to use such a PSI to meet their requirements while leveraging the functionality already present in the PSI, which can potentially save a significant amount of effort. Existing PSIs are evaluated with respect to these 18 desirable features in Section 4.

The second original contribution, as described in Section 5, is the idea of using events to determine when code for updating particles and particle systems should be executed. The event approach has three advantages over the traditional approaches of either using Boolean values to determine whether or not specific code should be executed or setting a parameter to a specific value, so that it does not affect the output. First, using events can increase performance since they avoid unnecessarily executing code. Secondly, using events can increase modularity by making it easier for clients to add or remove code. Thirdly, because events can be added or removed at run time, particle and particle system behavior can easily be changed at run time.

The third original contribution, as described in Section 6, is the Dynamic Particle System Framework (DPSF) software, which provides the 18 features mentioned above. DPSF was developed to help clients create PSIs. Rather than starting from scratch, clients can build upon DPSF to create PSIs, allowing them to avoid implementing common particle system tasks, such as managing particles in memory, while still being able to program desired functionality into the PSI. The experimental results and the Demo software, as described in Section 7, show that DPSF can be used for applications that other PSIs cannot. The results of experiments conducted with the Demo software, reported elsewhere [10], also show that DPSF runs fast enough to be used for interactive applications, such as video games. By integrating features from other existing PSIs into a single framework, DPSF enables the creation of visual effects that are qualitatively different from those possible with other PSIs. These features make using DPSF suitable for a wide variety of applications that require a particle system.

2. Terminology

This section defines terminology relevant to particles, particle systems, and PSIs. *Particle properties* are properties possessed by every particle in a particle system. Such properties are used to control a particle's behavior. Some typical particle properties are position, velocity, color, size, and *lifetime* (how long the particle should remain active). Particles that have an elapsed time less than their lifetime are referred to as *active particles*; all other particles are referred to as *inactive particles*. *Particle update functions* are functions that update the values of a particle's properties, such as by updating a particle's position according to its velocity. *Particle system properties* are properties of the particle system itself and are typically used to control the particle system as a whole. Examples of particle system properties include the maximum number of particles that can exist in the particle system, the rate at which particles are added to the particle system, and the magnitude of an external force that affects the trajectories of all particles in the particle system. *Particle system update functions* are functions that update the particle system's properties, such as by changing the rate at which particles are added. Particle and particle system properties are collectively referred to as *Particle System Interface properties (PSI properties)*, and particle and particle system update functions are collectively referred to as *Particle System Interface update functions (PSI update functions)*.

A *Particle System Interface parameter (PSI parameter)* is a parameter provided by a PSI whose value can be adjusted by clients to affect the behavior or visualization of the particle system and its particles. When a PSI parameter value is changed, it changes the value of one or more PSI properties, or changes the way that the PSI properties are updated internally by the particle system, causing a new particle system to be produced. PSI parameter values are typically specified by clients through code, a script, or a visual tool. Examples of common PSI parameters include the lifetime, initial position, and initial velocity of particles added to the particle system and the texture used to draw the particles. By allowing clients to set different values for the PSI parameters, many different particle systems can be created from a single PSI.

Figure 1 shows the relationship between a PSI, a particle system, and a visual effect. An example of a PSI is shown at the top of Figure 1. It provides functions to add, update, and draw particles; the particle properties called elapsed time, lifetime, position, and velocity which every particle will have; a particle system property called texture, which is used to draw the particles; and PSI parameters to set the lifetime, initial position, initial velocity, and texture of the particles. Below the PSIs in Figure 1 are examples of two particle systems created from the PSI. Each of the Fire and Smoke particle systems was created by specifying values for the PSI parameters. When particle system simulations run, they create effects, where an

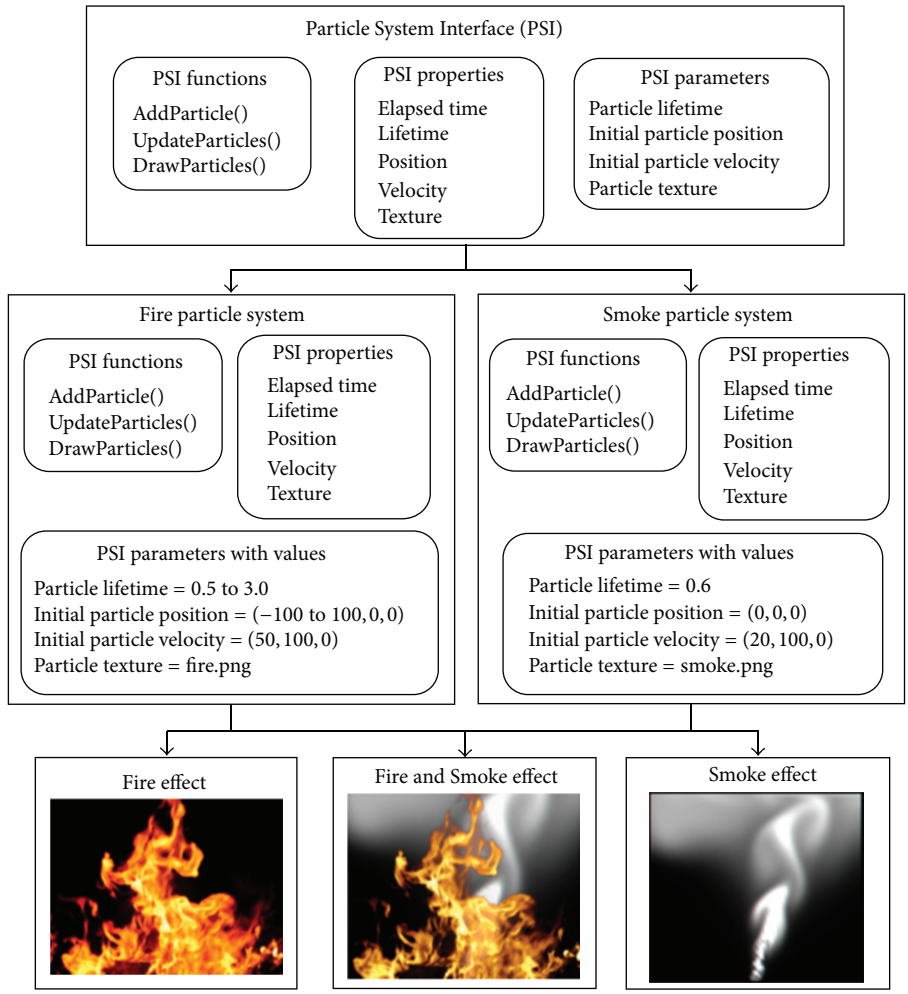


Figure 1: One Particle System Interface, creating two particle systems and creating three effects.

effect is the overall perception of the individual particles of the particle system. When an effect is visualized, such as by being shown on screen, it is referred to as a *visual effect*. Three visual effects created from the Fire and Smoke particle systems are shown at the bottom of Figure 1.

3. Desirable PSI Features

While many published works describe applications of particle systems, few describe the components that a PSI should possess in order to be suitable for many applications. Reeves [11] and van der Burg [12] identified the following desirable PSI properties: lifetime, velocity, and color for the particles, having a particle system texture, and external force. Identifying the desirable features of a PSI facilitates evaluation of PSIs. Arguably, a PSI is more likely to meet clients' requirements if it has a greater number of these desirable features.

A PSI flexible enough to support creating a wide range of diverse effects should offer the following features, which are discussed in more detail below.

(F1) Provide a method to add and update particles.

(F2) Provide some built-in PSI properties.

(F3) Provide PSI parameters to specify values for built-in PSI properties.

(F4) Provide a method to draw the particles.

(F5) Provide a method for inspecting the active particles in the particle system.

(F6) Allow clients to create PSI properties.

(F7) Allow clients to specify how the new PSI properties are updated each frame, and how they affect other PSI properties.

(F8) Allow clients to create PSI parameters.

(F9) Allow updates to be made when a particle condition is met.

(F10) Allow updates to be made when a particle system condition is met.

(F11) Allow particles to be affected by objects and forces in the virtual environment.

(F12) Allow particles to be affected by other particles in the same particle system.

(F13) Allow PSI parameters to be updated at run time by software events.

(F14) Allow PSI parameters to be updated at run time by user input, enabling real-time interaction with the particle system.

(F15) Allow PSI parameters to update the properties of active particles.

(F16) Allow PSI parameters to affect a subset of the active particles.

(F17) Provide support for animated particles.

(F18) Provide support to use multiple particle images in a particle system.

Adding new particles to the particle system and updating existing particles (F1) are essential particle system tasks. Ways to perform these tasks are apparently always included in a PSI. Also, every PSI apparently provides some built-in particle and particle system properties (F2), but the number and types of PSI properties that they provide may differ. Having more PSI properties typically allows clients to define more detailed particle behaviors and create a wider range of effects.

A PSI property is only useful for creating diverse effects if the client is able to change its value. A PSI property's value can be changed by clients through the use of a PSI parameter (F3). For example, a particle system might emit particles at a rate corresponding to a ParticlesPerSecond particle system property. If the value of this property cannot be changed by clients, then they will not be able to use it to create a new effect. However, if clients are able to change the value of the ParticlesPerSecond property using a PSI parameter, then they will be able to create several different effects by adjusting that value. If a PSI provides more PSI parameters, clients will have more options available to them, which allows them to create more diverse effects.

Many particle system applications require visualizing the effect on screen, so providing a function to draw the particles (F4) can be very useful for allowing clients to visualize their effects with minimal effort. To allow clients to display their effects in a variety of ways, the PSI should also provide a function or property that allows clients to iterate through all of the active particles in the particle system and inspect their property values (F5). For example, if clients want to visualize their effect in a custom manner, such as by forming a mesh where particles correspond to vertices in the mesh and these vertices are connected by lines, they will likely be able to accomplish this by iterating over the particles but not by using the PSI's built-in drawing function. Providing feature F5 also has the benefit of allowing clients to collect information about the particles during the simulation, such as by tracking the positions of the particles over time or recording other information about them.

While there is a general set of PSI properties that are required to create many common effects, it is impossible to provide every property that a client may potentially require. Thus, it can be beneficial to allow clients to create their own PSI properties (F6). For example, if clients want their particles to each have an Orientation property, and it is not provided by the PSI, they could add it as a new particle property. This property would allow each particle to be oriented differently. Unless the values of the properties that clients create are intended to be constant and do not affect other properties, clients will also require the ability to define how the new PSI properties are updated and the effects, if any, that they have on the other PSI properties (F7). For example, clients may decide that they want the Orientation property of a particle to change based on a rotational velocity. In this case, they need to create a RotationalVelocity particle property and define how it affects the Orientation property (e.g., Orientation += RotationalVelocity $*$ ElapsedTime). Similarly, if clients want the velocities of particles to be affected by acceleration or friction, they need to define these new properties and specify how they affect the Velocity particle property. In addition to creating new PSI properties and defining how they are updated, clients will likely require the ability to specify values for these properties. This means that clients should also be able to create PSI parameters (F8) in order to specify values for the new PSI properties.

While particle systems are generally stochastic in nature, they are often intended to follow some general pattern of behavior. More complex patterns of behavior can be created by allowing updates to be triggered when specific particle conditions are met (F9). For example, clients may want each particle to randomly change its direction when half of its lifetime has elapsed or to change colors when it reaches a certain speed. Alternatively, they may want a counter to be incremented every time a particle collides with a specific object. Similarly, allowing updates to be triggered when specific particle system conditions are met (F10) can also allow more complex effects to be created. For example, clients might want the color of new particles to change after 1000 particles have been emitted from the particle system or to apply a gravitational force to all particles after the particle system has existed for 10 seconds. Being able to trigger updates according to internal particle and particle system conditions as well as specifying the updates that should be performed can allow clients to conveniently define deterministic aspects of particle system behavior and thus create more complex effects.

Providing a particle system with information about the surrounding virtual environment can allow particles to react to objects or forces in this environment (F11), which can enhance realism. For example, if a particle system in a video game is emitting sparks, it may be desirable to have the sparks bounce off the walls and floor in the virtual environment of the game instead of passing through them. A virtual environment is *dynamic* if it contains objects with properties that are updated at run time. Having particles that react to dynamic virtual environments can further increase the realism of an effect. Similarly, clients may want particles to be affected by other particles in the particle system (F12), such as by having particles collide instead of passing through one another. Because these operations can be computationally expensive to perform, a client is unlikely to use them frequently, but they should be available.

In addition to particle systems reacting to the environment, it can be beneficial to have them reacting to software events. Therefore, PSI parameters should be responsive to

Table 1: A list of PSIs, a PSI created using DPSF, and the features from Section 3 that they support.

Label	Particle System Interface	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17	F18
PSI0	Particle Chamber	X	X	X	X									X	X				
PSI1	Microsoft XNA Unleashed	X	X	X	X														
PSI2	Building a Particle Engine	X	X	X			X	X	X					X	X				
PSI3	Particle Systems API	X	X	X	X	X						X	X	X	X	X	X		
PSI4	Particles	X	X	X	X									X	X				
PSI5	Particle 3D	X	X	X	X									X	X				
PSI6	Balls	X	X	X	X							X		X	X				
PSI7	Declarative API	X	X	X	X		X	X	X	X		X		X	X	X			
PSI8	Autodesk 3ds Max	X	X	X	X	X	X	X	X	X	X	X	X	X		X	X	X	X
PSI9	Autodesk Maya	X	X	X	X	X	X	X	X	X	X	X	X	X		X	X	X	X
DPSF	PSI created using DPSF	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

software events at run time (F13), where a *software event* is triggered by something in the software that is external to the virtual environment. For example, clients may want the particle system to emit particles of a different color when the user accomplishes some goal, such as collecting 100 coins or achieving a mission objective. Also desirable is the ability to update PSI parameters at run time according to user input (F14). This feature allows for real-time interaction between users and the particle system and has many potential applications. For example, it could be used to allow users to control a fire-hose particle system, where they spray water at fires to put them out. Real-time interaction with particle systems at the particle level can also be achieved by allowing PSI parameters to affect the properties of active particles (F15), rather than only affecting the properties of particles added to the particle system after the PSI parameter value has been updated. Being able to change the properties of active particles at run time allows particle behavior to be updated at run time. For example, a particle system may initially emit particles that are attracted to some object. Clients may want the particles to change behavior and be repelled from the object when specific user input is received. By allowing the properties of active particles to be updated, this type of behavior can be achieved.

In addition to updating active particle properties, clients may want to change the behavior of only some of the active particles in the particle system. For example, when specific user input is received, clients may want to change the color of only 100 random particles or only the particles that meet some criteria, such as having a velocity greater than 100 or being within a certain distance of another object. For these cases, PSI parameters should be able to update a subset of the active particles (F16).

In order to create certain effects, clients may require the individual particles to be animated (F17). For example, clients may want to create a particle system representing hundreds of butterflies, where each butterfly is flapping its wings. Similarly, clients may want a particle system to display its particles using several different images (F18). For example, clients may want to simulate a tornado picking up many different objects, such as rocks, sticks, and clumps of dirt. In order for these different objects to be visualized properly, the particle system would need to support displaying its particles with multiple images.

The 18 features listed above are neither exhaustive nor independent. They are not exhaustive because someone might devise a new feature of general interest, but they correspond to all features of interest we observed in existing PSIs. They are not independent because several of them are interrelated. For example, F14 depends on F13, because the user input referred to in F14 is a specific type of software event, as referred to in F13. Nonetheless, we believe that it is worth distinguishing them, so we can better evaluate existing PSIs. For example, some systems react to some types of events (F13), but they do not react to input events (F14).

4. Evaluation of Current PSIs

Particle systems have become popular in recent years and many PSIs have been developed to make creating particle systems easier. Table 1 shows ten representative existing PSIs and the features from Section 3 that each supports. For simplicity, we refer to the PSIs as PSI0 to PSI9: PSI0-Particle Chamber [13], PSI1-Microsoft XNA Unleashed [14], PSI2-Building a Particle Engine [15], PSI3-Particle Systems API [16, 17], PSI4-Particles [8], PSI5-Particle 3D [9], PSI6-Balls [18], PSI8-Autodesk 3ds Max [19], and PSI9-Autodesk Maya [20]. For completeness, our DPSF software is also included in the table.

A PSI provides clients with a set of parameters (PSI parameters) whose values may be altered to produce different particle systems and, hence, different effects. Some PSIs, such as PSI0, allow clients to simply select the effect they want from a list of effects (e.g., fire, smoke, snow, etc.) and then the PSI sets all of the PSI parameters accordingly, allowing an effect to be created quickly and easily. If the PSI has the exact effect the client wants, this feature is ideal. Other PSIs allow clients to pick an effect and then accept default PSI parameter values or manually specify new ones. For example, a client might specify the permissible range for a particle's initial velocity and lifetime. Some PSIs, such as PSI0, provide a GUI that clients can use to adjust PSI parameter values and visualize

the particle systems in real time as the PSI parameter values are changed. Some PSIs, such as PSI8 and PSI9, allow PSI parameter values to be specified from a script, allowing new particle systems to be created by editing a simple text file. Several PSIs, such as PSI1, PSI4, and PSI5, are available as a set of related codes that can be added to a client program. In more detail, PSI1 provides Update and Draw functions and a Settings class to configure particle properties, and PSI4 and PSI5 both provide a base ParticleSystem class with Update and Draw functions that client classes can inherit. Finally, PS4 is available as a software library with a well-defined API. Recently, Krajcevski and Reppy have specified a declarative API for particle systems (PSI7) [21]. With their approach, a particle system is described in a declarative fashion by an emitter, which specifies how new particles are generated, an action, which specifies how particles are updated, and a renderer, which specifies how particles are drawn. A particle system can be targeted for either CPU or GPU execution.

Although space concerns prohibit a discussion of all features for all PSIs, a few observations are appropriate. As can be seen in Table 1, all ten existing PSIs support features F1, F2, and F3. Adding and updating particles (F1) and providing some built-in PSI properties (F2) and PSI parameters (F3) are essential PSI features. The PSIs differ in the PSI properties they contain and the PSI parameters that they provide to the client. The PSI parameters provided by most of the PSIs include setting the particles' lifetime, initial position, initial velocity, size, and an external force to affect the trajectory of all of the particles, as well as setting the texture used to draw the particles, the maximum number of particles that the particle system can contain, and the rate at which particles should be emitted. All PSIs except PSI2 provide a method to draw particles (F4). Only PSI3, PSI8, and PSI9 provide a method for inspecting the active particles in the particle system (F5).

The PSI parameters mentioned above are adequate for creating some simple visual effects, but these parameters alone are not capable of producing sophisticated effects, such as having the particles follow a path, react to environmental forces, or behave according to physically accurate rules, such as those used in fluid flow simulations. The main problem is that the PSIs are limited in the number of PSI parameters they provide to clients, and the PSI parameters they do provide typically do not allow clients to specify these sorts of behaviors.

By allowing clients to create PSI properties (F6), to specify how the new PSI properties are updated each frame and how they affect other PSI properties (F7), and to create PSI parameters (F8), PSI2, PSI7, PSI8, and PSI9 provide this type of flexibility. Two features that are not provided by most existing PSIs are allowing updates to be triggered when specific particle conditions are met (F9) and when specific particle system conditions are met (F10). Of the systems examined, only PSI8 and PSI9 provided both those features, although PSI7 provides F9.

Of the PSIs inspected, all but PSI1 support updating PSI parameters at run time according to software events (F13). PSI1 allows PSI parameters to be specified during initialization, but it does not allow them to be changed after the PSI has been initialized. The only three PSIs that do not support user interaction with the particle system at run time (F14) are PSI1, PSI8, and PSI9. This limitation is inherent in the design of PSI8 and PSI9 because they are both offline renderers. In other words, they do not perform the simulations in real time. Instead they precompute the simulation and record it as an animation (i.e., video). Clients can incorporate the animations in applications. However, as mentioned, since the users of the application are then simply watching videos, they are not able to interact with the particle systems. Although PSI8 and PSI9 provide more features than the other PSIs inspected, their applications are limited to recorded animation.

The only PSIs inspected that support animated particles (F17) and using multiple images for a particle system's particles (F18) are PSI8 and PSI9. These features are especially useful for films and video game cut scenes, where emphasis is put on the visualization of the particles. By allowing particles to be animated or using multiple images for the particles, more realistic and convincing visual effects can be created. However, because PSI8 and PSI9 are offline renderers, they cannot be used to generate particle systems for the interactive parts of video games.

No single previous PSI supports all 18 features, as shown in Table 1. When clients want to create an effect, they may need to investigate and test multiple PSIs, with no guarantee that the PSIs will support the effects that they require. This can be a time consuming process since clients will not only need to find the PSIs but also learn how to use each of them. To avoid this process, it is preferable to have a PSI that is capable of creating many diverse effects; since once clients learn to use the PSI, they will be able to create any effects they require.

5. Event-Based Updating

As mentioned in Section 2, when using a PSI, clients can define particle and particle system update functions to update the particle and particle system properties, respectively. Once these functions have been defined, clients can choose (1) which functions should be used to update the particles and particle system, (2) when the functions should be called, and (3) the order of their execution. The *event-based approach to particle and particle system updating* accomplishes this purpose through the use of events, where an *event* is a delegate (i.e., function pointer) with some extra data, such as the order of execution. Two classes of events are of interest. A *particle event* points to a particle update function and is used to update particle properties. A *particle system event* points to a particle system update function and is used to update the particle system properties. To cause an update function to be executed, an event pointing to that update function is added to the particle system. Events can be added and removed when initializing a particle system, as well as at run time. Because events can be added and removed at run time, real-time interaction with the particles can be achieved.

When an event *fires*, it executes the particle or particle system update function to which it points, updating the particles or particles system, respectively. When and how

often an event fires depends on the type of event being used. Four types of events are distinguished.

(1) *Every Time Events:* these events fire every time the particle system is updated and are applied to every active particle in the particle system. Every Time Events are used for updates that should be performed for every frame, such as updating a particle's position according to a velocity or interpolating a particle's color between two color values.

(2) *One-Time Events*: these events fire the next time the particle system is updated and are then removed. Thus, each event only fires once. Like Every Time Events, One-Time Events are applied to every active particle in the particle system. One-Time Events are useful for interactively controlling particles at run time. For example, clients could add a One-Time Event to the particle system each time a specific button is pressed, allowing an effect such as changing the color or size of all particles to occur each time the button is pressed.

(3) *Timed Events*: these events fire at a specific moment in a particle's lifetime and are only applied to that particle. For example, if a particle is set to live for 4 seconds and clients want it to change colors half way through its lifetime, a Timed Event could be specified to fire after 2 seconds (absolute time) or 0.5 of the lifetime (relative time) to change the color of the particle. Timed Events are useful for deterministically controlling a property of a group of particles, such as changing the velocities of all particles at specific times to have them move in a planned pattern.

As mentioned above, Every Time Events and One-Time Events are applied to every active particle in the particle system. If clients only want specific particles to be updated, they can code a condition into the particle update function. For example, if clients want some smoke particles to be moved around when an object passes by them, they can code the particle update function to take into account the particle's distance from the object when updating the particle's velocity. This allows particles close to the object to be greatly affected by it and particles far from it to not be affected by it. By allowing clients to write the particle update functions, triggered by events, the event-based approach allows update functions to affect only some of the active particles (F16).

When a Timed Event fires, it executes the particle update function on only the particles that have reached a specified age. Timed Events typically only fire once, but if clients explicitly change the age or lifetime of a particle, then these events could fire several times. For example, a Timed Event could be set to fire when the particle reaches half of its lifetime. If some update resets the lifetime of a particle to 0.25 when it reaches 0.75 of its lifetime, the event would fire a second time.

Using events allows clients to choose which update functions should be used to update the particle system and its particles. By selecting which PSI update functions to use and by being able to modify the selection at run time, many diverse effects can be created with a single general-purpose PSI. Specific particle systems can be created from such a PSI by specifying the events and update functions that the particle system should use. For example, a PSI might provide three different particle update functions for updating the transparency of a particle as it ages. The first function might linearly interpolate the transparency to have the particle fade evenly during its lifetime. The second function might leave the particle opaque for the first half of its life then fade it out during the last half, and the third function might do the same except restrict the fade out to the last 20% of the lifetime. Clients could then choose which fade out method to use by adding an Every Time Particle Event that points to the appropriate particle update function.

Similar functionality could be offered by traditional PSIs by allowing clients to set a fade out variable with a value from an enumeration listing the possible fade out methods and then using a series of if/else statements to select one. However, using this approach would require that the value of the fade out variable be checked once for every update for every particle in the particle system in order to determine which fade out method to use. If the particle system is being updated 60 times per second and the particle system contains 5000 particles, this method would require 300,000 extra operations every second in the best case scenario, which would be when the fade out variable's value matched the first if statement's conditions. It is expected that this method would decrease performance, especially if the PSI offered similar variables for controlling how the particle's position, velocity, acceleration, color, size, orientation, and so forth were updated, and each of these variables had many possible values. By using events, this potential performance loss is avoided, since the functions to perform the necessary operations are called directly without requiring any additional operations to check variable values. Also, using the method just described, if clients wanted to add additional methods to fade out particles, they would need to add a new value to the fade out enumeration, add an if statement to the logic controlling which fade out method is used, and write the actual operations to perform the new fade out method. By using events, clients only need to define a single new function and specify that it should be called by adding to the particle system a new event that points to the function.

Events can be used to further increase speed over traditional PSIs. Because clients can choose which particle update functions they want to use, they can avoid wasting CPU cycles updating particle properties that are not being utilized. For example, if the particles contain an acceleration property, but the clients do not want to use acceleration to update the velocity of the particles, they simply need not to add an event to update the particle velocity according to the acceleration. In a traditional PSI, clients would set the acceleration of each particle to zero, and during each update the new velocity of every particle would be calculated. Since the acceleration is zero, these operations would be pointless. With an event-based approach, these calculations are avoided altogether by simply not specifying events to update the velocity.

Using events along with PSI update functions provides support for several of the features listed in Section 3. Because

events can be added and removed at run time, real-time interaction with the particles is possible (F14). By giving clients the ability to create PSI update functions, which have access to the particle and particle system properties, updates can be triggered by specific particle conditions (F9) and by specific particle system conditions (F10). Similarly, this ability also allows clients to update PSI properties according to the forces in the virtual environment (F11) and other particles in the particle system (F12), as well as to specify which, if any, of the active particles should be updated (F16).

6. DPSF

The Dynamic Particle System Framework (DPSF) is a framework written in C# and XNA that provides the 18 features mentioned in Section 3. It consists of two base classes (the base particle class and the base particle system class) and two default classes (a default particle class and a default particle system class). To use DPSF, clients incorporate it into their code, add variables to represent PSI properties, and add functions for initializing and updating. The base particle and particle system classes included in DPSF provide features F1 to F4. Clients can have the list of active particles returned to the external software by using the Active Particles property (F5), allowing them to explicitly draw the particles themselves or to perform other operations such as analyzing the particles and collecting data. Since these base classes can easily be extended, clients can define new PSI properties (F6), new PSI update functions (F7), and new PSI parameters (F8). The *base particle class* contains only the essential particle properties, which are specifically the particle's lifetime, elapsed time, and visibility. Clients can create particle classes that inherit from this base particle class, allowing them to specify any additional particle properties that they may require, such as position, velocity, acceleration, size, color, or mass. The *base particle system class* contains functions to perform common particle system functions, such as adding, updating, and removing particles in the simulation. As with particle classes, clients can then create particle system classes that inherit from the base particle system class, allowing them to add new PSI properties, PSI update functions (to update the new PSI properties), and PSI parameters (to specify values for the PSI properties). Also, clients must specify a *particle initialization function*, which is a function used to specify the initial values of particle's properties when it is added to the simulation. Once clients have created their particle and particle system classes, they have effectively created a PSI; DPSF can then run the simulation using the particle and particle system classes specified by the clients.

This extensibility gives clients freedom to create many types of effects with DPSF. For example, if clients want to create an effect where hot particles ascend and cool particles descend, they could first create a particle class with position and temperature particle properties, by copying the base particle class and inserting two instance variables, one for position and one for temperature. Then they would create a particle system class that includes the following: (1) a particle system property for ambient temperature; (2) a particle update function that updates the particle's temperature to slowly approach the ambient temperature; (3) a particle update function to update the particle's vertical position according to its temperature; (4) a particle initialization function to specify a particle's initial position and temperature; and (5) PSI parameters to specify the ambient temperature and the minimum and maximum values for a particle's initial temperature. They would then run the simulation simply by specifying values for the PSI parameters and calling the Update() and Draw() functions provided by the base particle system class. By extending DPSF, clients do not have to decide how to manage particles in memory or draw them to the screen and can instead focus on the remaining components required to create the desired effect.

DPSF provides default particle and particle system classes. They are useful because many effects require similar sets of PSI properties, PSI update functions, and PSI parameters. The default classes include the components required for particle position, velocity, acceleration, rotation, friction, color, size, and so forth. By using default classes, clients can create common effects quickly, as with existing PSIs. They can also easily extend them to create uncommon effects, such as those required in research applications.

Because DPSF allows clients to create PSI properties (F6), write PSI update functions (F7), and create PSI parameters (F8), they are able to quickly integrate their code into a PSI, allowing custom behaviors to be created. This feature removes many of the restrictions that other PSIs place on clients. Clients can be more productive, because they can incorporate their behaviors and constraints into an existing PSI instead of having to create one from scratch. Because DPSF allows clients to inject their own code into a DPSF PSI, the PSIs described in Section 4 could be implemented using DPSF. Also, because DPSF supports creating a wide variety of effects, clients will likely be able to use DPSF to create all of the particle systems that their applications require, instead of having to use multiple PSIs. Thus, clients will not need to learn to use multiple PSIs and integrate them into a single project.

7. Results

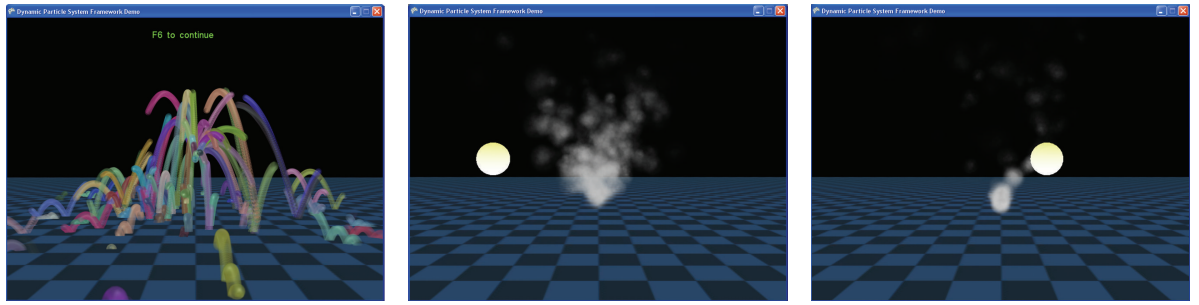
To evaluate the capabilities of DPSF, software called *Demo* was implemented that uses DPSF to create 35 different effects. Table 2 lists 8 of the effects and the features from F1 to F18 that a PSI must possess in order to create them. The table shows that DPSF supports creating PSIs with all 18 of the features listed in Section 3. As shown previously in Table 1, none of the existing PSIs analyzed in Section 4 provides all 18 features.

For example, Demo demonstrates that PSIs can interact with the virtual environment (F11). In Demo's *Fountain* effect, balls are emitted in an upward direction and are affected by gravity, which pulls them downward. When the balls hit the floor, they bounce off it. This bouncing is shown in Figure 2(a), which displays the particle trajectories over a span of three seconds.

Demo's *Smoke* effect demonstrates interaction with objects in the virtual environment, where smoke particles are attracted to an orb that moves past them from left to right, as shown in Figures 2(b) and 2(c).

Table 2: Effects created in Demo, the features they demonstrate, and the number of lines of code used to create them.

Feature	Brief feature description	Fire and Smoke	Fountain	Smoke	Square Pattern	Figure-eight	Image	Sprite force	Animated butterflies
F1	Add and update particles	X	X	X	X	X	X	X	X
F2	Built-in PSI properties	X	X	X	X	X	X	X	X
F3	Built-in PSI Parameters	X	X	X	X	X	X	X	X
F4	Method to draw particles	X	X	X	X	X	X	X	X
F5	Method to return active particles						X	X	X
F6	Create new PSI properties	X	X	X			X	X	X
F7	Create new update functions	X	X	X	X	X	X	X	X
F8	Create new PSI Parameters	X		X			X	X	X
F9	Trigger updates on particle conditions	X			X		X		
F10	Trigger updates on PS conditions						X		
F11	Particles affected by environment		X	X					X
F12	Particles affected by other particles							X	
F13	Update PSI parameters from events	X	X	X	X	X	X	X	
F14	Update PSI parameters from user input	X	X	X	X	X	X	X	
F15	Update properties of active particles		X	X	X		X	X	
F16	Update a subset of active particles			X			X	X	
F17	Animated particles								X
F18	Multiple particle images						X		
Lines of code	N/A	187	76	107	89	61	322	413	275



(a) Fountain effect interacting with the floor (shown over a span of three seconds)

(b) Smoke effect before orb attracts particles

(c) Smoke effect while orb attracts particles

Figure 2: Two effects showing interaction with the virtual environment.

Deterministic particle systems can also be created using DPSF. For example, Demo's *Square Pattern* effect shows a simple deterministic effect where each particle follows a square path. This effect is created by using three timed events (F9), which change a particle's velocity direction at 25%, 50%, and 75% of the particle's lifetime. Figure 3 shows the particle trajectories over a span of five seconds.

The *Figure-Eight* effect shown in Figure 4 also displays deterministic behavior, where the particles follow the path of a figure-eight. This effect was achieved by writing a function with code to make a particle travel in two circles and then adding it as a Timed Event (F7). The three screen shots in Figure 4 show the particles as they travel in a figure-eight pattern.

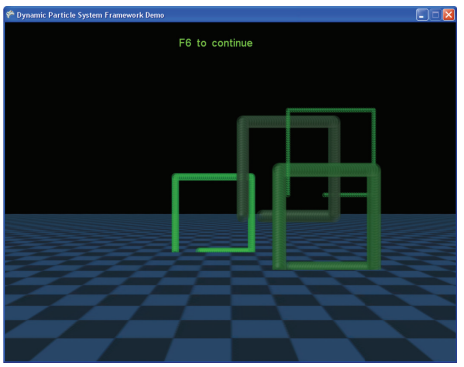
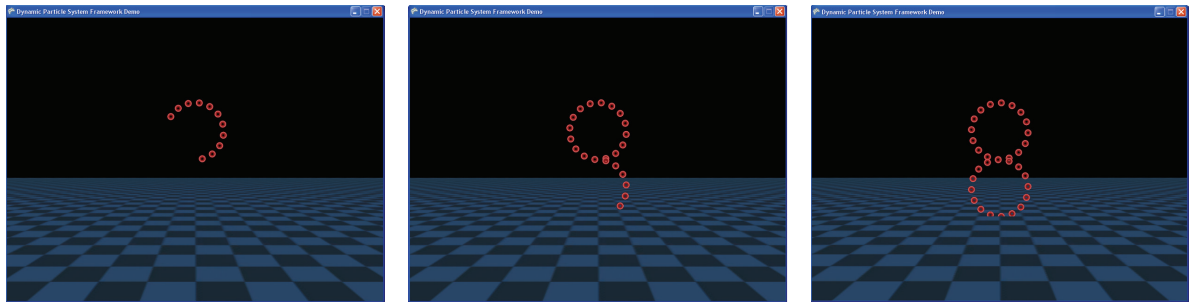
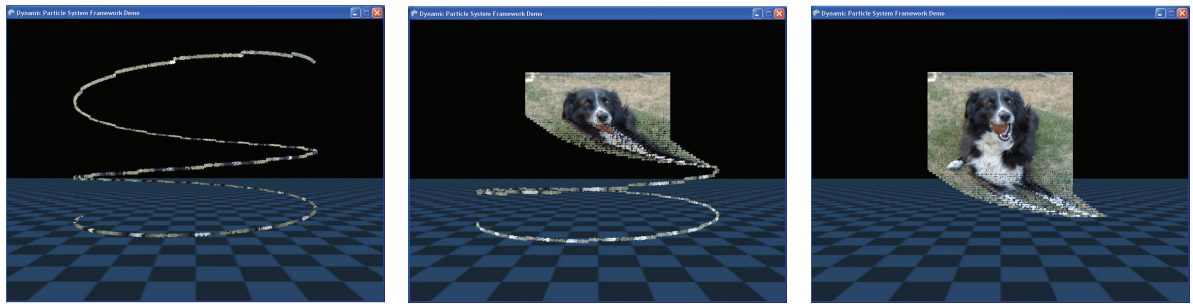


Figure 3: Particles traveling in square patterns.



(a) (b) (c)

Figure 4: Particles traveling in a figure-eight pattern.



(a) (b) (c)

Figure 5: Particles traveling in a spiral before coming to rest at their final positions, which forms an image.

The *Image* effect provided by DPSF Demo also shows that complex deterministic effects can be created. In this effect, particles travel in a spiral before coming to rest at their final positions, as shown in Figure 5. The combination of all particles in their final positions forms a single image (F18).

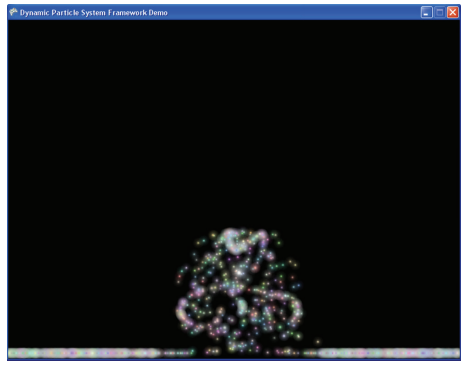
Almost all of the effects created in Demo allow for interaction between the user and the particle system (F14). Most effects allow the user to move and change the orientation of the emitter at run time, affecting where particles are emitted and the direction they travel. Many effects offer other types of user interactions as well. For example, with the Fountain effect, shown in Figure 2(a), users can turn collisions between the balls and the floor on and off. Figures 2(c) and 6(a) show particles being attracted to a user-controlled particle, and Figure 6(b) shows particles being repelled from a user-controlled particle. These effects also demonstrate interactions between particles (F12) as well as control at the individual particle level (F16), because the particle that the user controls is one of the particles in the particle system.

The *Animated Butterflies* effect, shown in Figure 7, demonstrates animated particles (F17) by creating animated butterfly particles that flap their wings. The animation class allows the animations for the individual butterflies to be played at different speeds simultaneously and to have these speeds updated at run time. By increasing the animation speed as particles travel upward, the butterflies appear to flap their wings faster when ascending and slower when descending, producing a more convincing effect.

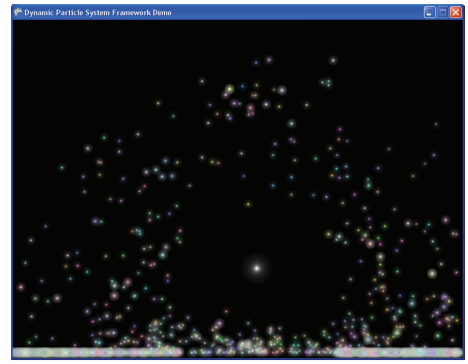
By looking at the number of lines of source code used to create the effects in Demo, which were counted using Code Line Counter Pro [22] and are listed in Table 2, one can see that few lines of code are required on the clients' part to create a PSI. In contrast, many more lines of code would be needed to create a PSI from scratch or to modify an existing one to meet the clients' requirements. The small size of the source code provides evidence that many effects can be created quickly and easily by using the given templates and default classes.

8. Conclusion

The goal of this research was to design, implement, and evaluate a software framework that is flexible enough to support creating particle systems for diverse applications while making the process of creating particle systems fast and easy for clients. The DPSF software was developed to accomplish this goal. To simplify learning how to use DPSF and to make creating PSIs faster and easier, templates are provided that clients can use when creating PSIs. In addition, DPSF provides default classes that include particle properties and update functions that are required by many common effects. The default classes make it faster and easier for clients to



(a) Particles attracted to a client-controlled particle



(b) Particles repelled by a client-controlled particle

Figure 6: Particles being attracted to and repelled away from a client-controlled particle.

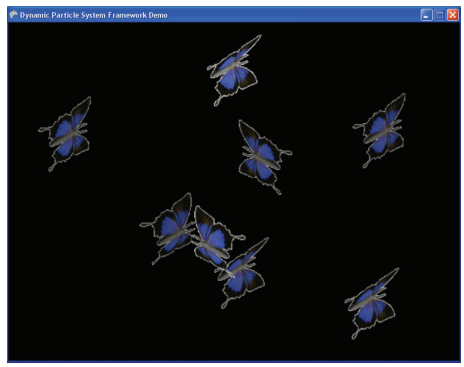


Figure 7: Animated butterfly particles.

create common particle system effects. DPSF is available for download at http://www.xnaparticles.com/Download.php.

The Demo software demonstrates that DPSF provides the 18 features listed in Section 3 and shows that DPSF can be used to create many diverse effects. Since DPSF supports these features, it can be used for traditional particle system applications as well as new ones. Because DPSF allows clients to create their own rules and constraints, it is a good candidate to be used in research applications. Also, performance tests reported elsewhere [10] show that the speed of DPSF is comparable to existing PSIs, except GPU-based systems give significantly faster performance for 10,000 or more particles.

Up to the time of writing, DPSF has been used in Holophone 3D [23] (a holographic phone app for Windows-Phone), an XNA-based game engine called PloobsEngine [24] and 18 independent video game development projects [25], including AvaGlide [26], Cannon number 12 [27], Defy Gravity [28], Orbitron: Revolution [29], and Perkuna's Dragon [30]. DPSF could be improved in several ways. More features could be added to the default classes, such as automatic collision detection and reaction between particles and the virtual environment, nearest neighbor detection, and a bounding box containing the particle system. As well, a Graphical User Interface (GUI) could be provided to allow clients who lack programming knowledge to create particle systems using DPSF.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgment

This work was supported by the Natural Sciences and Engineering Research Council of Canada via a Discovery Grant to H. J. Hamilton.

References

[1] E. Galin, R. Allègre, and S. Akkouche, "A fast particle system framework for interactive implicit modeling," in *Proceedings of the IEEE International Conference on Shape Modeling and Applications (SMI '06)*, p. 32, June 2006.

[2] J. C. Hart, E. Bachta, W. Jarosz, and T. Fleury, "Using particles to sample and control more complex implicit surfaces," in *Proceedings of the ACM International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '05)*, p. 269, ACM, Los Angeles, Calif, USA, 2005.

[3] M. Meyer, B. Nelson, R. M. Kirby, and R. Whitaker, "Particle systems for efficient and accurate high-order finite element visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 5, pp. 1015–1026, 2007.

[4] W. Y. Su and J. C. Hart, "A programmable particle system framework for shape modeling," in *Proceedings of the International Conference on Shape Modeling and Applications (SMI '05)*, pp. 114–123, IEEE Computer Society, Cambridge, Mass, USA, June 2005.

[5] R. Szeliski and D. Tonnesen, "Surface modeling with oriented particle systems," in *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '92)*, pp. 185–194, ACM, Chicago, Ill, USA, 1992.

[6] A. P. Witkin and P. S. Heckbert, "Using particles to sample and control implicit surfaces," in *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '94)*, pp. 269–277, ACM, Orlando, Fla, USA, 1994.

[7] P. Volino and N. Magnenat-Thalmann, "Simple linear bending stiffness in particle systems," in *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*

(SCA '06), pp. 101–105, Eurographics Association, Aire-la-Ville, Switzerland, 2006.

[8] Microsoft, Particles, April 2007, http://creators.xna.com/en-us/sample/particle.

[9] Microsoft, Particle 3d, May 2007, http://creators.xna.com/en-us/sample/particle3d.

[10] D. Schroeder, *Dynamic particle system framework: a framework for building custom particle system interfaces [M.S. thesis]*, Department of Computer Science, University of Regina, 2009.

[11] W. T. Reeves, "Particle systems—a technique for modeling a class of fuzzy objects," *ACM Transactions on Graphics*, vol. 2, no. 2, pp. 91–108, 1983.

[12] J. van der Burg, "Building an advanced particle system," *Game Developer*, vol. 3, pp. 44–50, 2000.

[13] R. Benson, Particle chamber, June 2000, http://archive.gamedev.net/archive/reference/listce83.html?categoryid=225.

[14] C. Carter, *Microsoft XNA Unleashed: Graphics and Game Programming for Xbox 360 and Windows*, Sams, 2007.

[15] Michael Fotsch, Building a Direct3D Particle Engine, December 2000, http://realmike.org/blog/articles/building-a-direct3d-particle-engine/.

[16] D. K. McAllister, "The design of an API for particle systems," Tech. Rep., Department of Computer Science, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 2000.

[17] D. K. McAllister, Particle Systems API, December 2008, http://archive.gamedev.net/archive/reference/listce83.html?categoryid=225.

[18] bjackdl, Balls, March 2008, http://en.pudn.com/downloads102/sourcecode/windows/csharp/detail416495_en.html.

[19] Autodesk, Autodesk 3ds Max, September 2013, http://usa.autodesk.com/3ds-max/.

[20] Autodesk, Autodesk Maya, September 2013, http://usa.autodesk.com/maya/.

[21] P. Krajcevski and J. Reppy, "A declarative API for particle systems," in *Practical Aspects of Declarative Languages*, vol. 6539 of *Lecture Notes in Computer Science*, pp. 130–144, Springer, Berlin, Germany, 2011.

[22] BistoneSoft, Code Line Counter Pro—C# Version 3.8, May 2009, http://www.softplatz.net/Downloads/Development/Compilers-Interpreters/Code-Line-Counter-Pro-C-Version.html.

[23] Holophone3D, Experience 3D holograms on Windows Phone, September 2013, http://dpsf.freeforums.org/holophone3d-experience-3d-holograms-on-windows-phone-t110.html.

[24] tpastor, PloobsEngine, September 2013, http://dpsf.freeforums.org/ploobsengine-t94.html.

[25] DPSF Forums, Projects that use DPSF, September 2013, http://dpsf.freeforums.org/.

[26] Haiku Interactive, AvaGlide, September 2013, http://haikuinteractive.com/.

[27] SquigglyFrog Studios/Microsoft Games Studios, Cannon #12, September 2013, http://cannon12.squigglyfrog.com/.

[28] Fish Factory Games, Defy Gravity, September 2013, http://dpsf.freeforums.org/defy-gravity-t81.html.

[29] Firebase Industries, Orbitron: Revolution, September 2013, http://dpsf.freeforums.org/orbitron-revolution-t106.html.

[30] Middle Lands Studios, Perkuna's Dragon, September 2013, http://www.wp7connect.com/tag/middle-lands-studios/.