A Declarative API for Particle Systems

Pavel Krajcevski[1] and John Reppy[2]

[1] Disney Interactive Studios Pavel.Krajcevski@disney.com
[2] University of Chicago jhr@cs.uchicago.edu

Abstract. Recent trends in computer-graphics APIs and hardware have made it practical to use high-level functional languages for real-time graphics applications. Thus we have the opportunity to develop new approaches to computer graphics that take advantage of the high-level features of functional languages. This paper describes one such project that uses the techniques of functional programming to define and implement a combinator library for *particle systems*. Particle systems are a popular technique for rendering fuzzy phenomena, such as fire, smoke, and explosions. Using our combinators, a programmer can provide a declarative specification of how a particle system behaves. This specification includes rules for how particles are created, how they evolve, and how they are rendered. Our library translates these declarative specifications into a low-level intermediate language that can be compiled to run on the GPU or interpreted by the CPU.
Keywords: Computer Graphics, Embedded DSL, Particle Systems

1 Introduction

In recent years, real-time computer graphics APIs, such as OpenGL, have shifted from working in terms of individual vertices to working with large batches of geometry. Furthermore, the computational load has shifted from the CPU to specialized GPUs that have supercomputer performance at commodity prices. These trends make it practical to use high-level functional languages to program real-time graphics and to take advantage of features like higher-order functions and polymorphic type systems [2]. This paper describes one such project: a declarative library for defining *particle systems*.

Particle systems are a technique for animating and rendering fuzzy phenomena such as fire, smoke, explosions, *etc.* in 3D graphics [8]. Because these phenomena have a fluid and dynamic appearance, traditional polygon-based rendering techniques are not well suited for rendering them. A particle system is a stochastic system that represents these phenomena as a cloud of simple *particles*. Each particle has state, *i.e.*, position, color, velocity, *etc.*, that is evolving over time according to some "physics" model [12].

Many animation systems today (*e.g.*, Blender [3] and Terminal Reality's Infernal Engine [4]) provide support for particle effects by requiring the user to specify a set of properties that are the interpreted by the underlying architecture. These methods for creating particle systems are designed to give artists much more creative control through a large variety of properties but discourage the creation of quick, dynamic particle systems that

[3] http://wiki.blender.org/index.php/Doc:Manual/Physics/Particles
[4] http://www.infernalengine.com/tech_particles.php

are easily simulated with many particles. In contrast, McAllister has described a C++ library for particle systems that provides a higher level of abstraction than hand-written effects and is more flexible than the canned systems found in animation tools [7]. In his system, one uses a sequence of *action* functions to specify how the state of the current particle system is modified. But his system has some limitations: it is not declarative, does not use runtime compilation techniques, and does not (yet) support running particle systems on the GPU.

This paper presents a declarative approach to implementing particle systems. Our library is implemented as part of the SML3d library [11], which is a library for OpenGL-based graphics for the MLton version of Standard ML. Our library has backends that support using either the CPU or GPU (via OpenCL [4]) as execution engines. Our library is inspired by McAllister's work, but it addresses the limitations of his library.

The remainder of the paper is organized as follows. In the next section, we provide a more detailed description of how particle systems work and give a simple example of such a system in pseudo code. In Section 3, we describe our combinator library for particle systems and illustrate its use with several examples. This section is followed by a description of the implementation, which includes a discussion of the optimizations that we perform. We then discuss related work in Section 6 and conclude in Section 7.

2 Particle systems

A **particle system** is a technique in computer graphics that is used to render *fuzzy phenomena*, such as rain, fire, explosions, *etc.* [8]. Particle Systems consist of a set of individual particles that each have their own **state**. The state of a particle is defined by a list of variables defining attributes that are used in either the physical simulation of the particle or in its rendering. A typical implementation of a particle system executes the following steps for each frame of animation:

1. New particles are generated by means of a stochastic process that determines the number of new particles and their initial states.
2. Particle states are updated.
3. Particles that have exceeded the qualifications for their existence are deleted.
4. Particles are rendered in the scene.

The most common state variables of a particle system are position, velocity, color, and age. A particle is considered *alive* if it exists within some predetermined boundary of the particle system and its age is below the maximum age of a particle.

The physical simulation of the particle system is defined by a set of rules that affect the particles. Some rules may be applied to only a subset of the particles while others affect all of the particles. An example of this is having particles bounce off of a platform versus gravity. In order to get the particle state for frame $i + 1$, we apply these rules to the state of the particles for frame i in relation to the time elapsed between rendering.

2.1 A simple particle system

As a simple example, let us consider a particle system that models a geyser or fountain.[5] In this system, particles represent water that is being ejected into the air from the ground. Particles will be emitted from a ground plane (the XZ plane) with some initial velocity vector pointing mostly up. As their state evolves, gravity slows their upward velocity until they eventually fall to earth and die when they reach the ground. For visual effect, we also want the color of the particle to evolve as it ages (*i.e.*, starting as white and then darkening to blue).

Simulating the basic physics of particle motion requires tracking the particle's velocity (**vel**) and position (**pos**) and checking to see if the particle has outlived its lifespan or hit the ground. The SML code for updating the physics state of a particle is as follows, where `NONE` is returned if the particle has died:

```
fun updateParticle ({pos, v, life} : particle, dt) =
      if (life = 0)
        then NONE
        else let
          val pos = Vec3f.add(pos, Vec3f.scale(dt, v))
          in
            if (#y pos <= 0.0)
              then NONE
              else SOME{
                  pos = pos,
                  v = Vec3f.sub(v, Vec3f.scale(dt, gravity)),
                  life = life-dt
                }
          end
```

Using our library, the physics of this particle system is specified as follows:

```
P.sequence [
    P.accelerate gravityVec,
    P.inside {
        d = groundPlane,
        thenStmt = P.move,
        elseStmt = P.die
      }
  ]
```

The physics is defined as a sequence (composition) of two actions (an action is just a function from particle states to particle states). The first action applies gravitational acceleration to the particle's state and the second tests to see if the particle is in the half-space defined by the ground plane. If it is, then the particle is moved (*i.e.*, its position is updated), otherwise it dies. The lifetime of the particle is set when it is created and is tracked automatically.

[5] SML source code for this example can be found in the SML3d source code.

2.2 Specifying particle systems

Specifying a particle system using our library is a staged process. First, the components for birth, simulation, and rendering are conglomerated into a single program. Each of these variables will be discussed in detail in Section 3

```
val create : {emit: emitter, physics : action, render : renderer}
      -> program
```

To run the program on a specific target (*e.g.*, CPU or GPU), we must first compile it to an executable. This involves breaking down the higher level combinators into a form representable on the target. In order to execute the program, it is instantiated with its own set of run-time variables. Each target supports a standard interface for compiling, instantiating, and running a program.

```
type exec  (* executable program *)
type psys  (* instance of an exec *)

val compile : Particles.program -> exec
val new : {exec : exec, maxParticles : int} -> psys
val step : {psys: psys, t : Time.time} -> unit
val render : psys -> unit
```

Lastly, we have a mechanism to delete a particle system when it is no longer needed.

```
val delete : psys -> unit
```

3 Particle-system combinators

An SML3d particle system is specified by three components: an *emitter*, which describes the generation of new particles, an *action*, which describes how the state of a particle evolves, and a *renderer*, which defines how a particle's state is visualized. This section describes how these components are specified and gives an example of a complete specification.

3.1 Variables

Particle systems are parameterized by *variables*, which have one of four possible types.

```
type 'a ty

val boolTy  : bool ty
val intTy   : int ty
val floatTy : float ty
val vec3fTy : Vec3f.vec3 ty
```

We use phantom types [6] to ensure type correctness for particle-system variables.

```
type 'a var

val new : string * 'a ty -> 'a var
```

Whenever a new variable is created, its type is specified. Subsequent uses of variables must match the specified type.

Our system has two kinds of variables: constants and parameters. Constants are variables that are created with an initial value.

```
val constb  : bool -> bool var
val consti  : int -> int var
val constf  : float -> float var
val const3f : (float * float * float) -> Vec3f.vec3 var
```

Parameters are initially created without values associated to them. The act of associating a value with a variable is called *binding*. Variables that are bound before a program is compiled are treated as constants; unbound variables are bound on a per-instance basis.

The binding of variables can be done at any point during the execution of the particle system and must be done prior to execution. The program will abort at runtime if it encounters any unbound variables, and there are no static guarantees at runtime to detect unbound variables. Due to this freedom in variable definition, anything that is defined in terms of particle system variables can be made time-varying by re-binding the variable after given time intervals.

3.2 Domains

We use McAllister's notion of a *domain* to specify many of the properties of a particle system. In the abstract, a domain is a subset of $\Re^3$; examples of domains include line segments, discs, cylinders, and half-spaces (*i.e.*, planes). Domains are created by specifying a series of particle system variables as parameters to a constructor. Domains are used to specify three dimensional objects that the particles interact with and random vector generators by defining a space containing all possible values. Some examples of how domains are specified are

```
type domain
val point : vec3f var -> domain
val plane : {pt: vec3f var, n : vec3f var} -> domain
val sphere : {c : vec3f var, r : float var} -> domain
```

3.3 Emitters

In our particle system definition, we use the notion of an *emitter* to denote a collection of domains from which the particle state variables generate their values. Since the domains are specified using particle system variables, whose values are not fixed, the conditions under which particles are generated can change over the course of a particle system's lifetime. Along with the domains, the emitter also takes a particle system variable parameter that defines the rate at which the particles should be created.

One example in which these variables may change is a geyser that emits particles in bursts. Over the course of the animation, the domain specifying the initial velocity of the particles could be specified by a cylinder whose radius changes with the rate of particle creation. Both of the radius of the cylinder and rate of emission would be specified by particle system variables.

```
type action

val move : action
val die : action
val nop : action
val accelerate : Vec3f.vec PSV.var -> action

val bounce : {
      friction : float PSV.var,
      resilience : float PSV.var,
      cutoff : float PSV.var,
      d : domain
    } -> action
```

Fig. 1. Common actions used in SML3D particle systems

Particle state The state of a particle consists of a number of properties. The initial domain of each of the following properties is specified upon the creation of an emitter:

position a vector of three floats that specifies the current position of the particle
velocity a vector of three floats that specifies the current velocity of the particle
size a vector of three floats that specifies the size of the particle for each axis.
color a vector of four floats that specifies the color (RGB plus alpha) of the particle
lifetime a float which represents the number of seconds left for this particle to live. If this value is less than or equal to zero, then the particle does not get rendered.

3.4 Actions

The main part of a particle system is the specification of its physics. We define an abstract type of *actions* to represent functions that modify the state of a particle. We have three kinds of action combinators: basic actions, such as move a particle; sequences of actions; and predicate actions. The semantics of particle-physics simulation can be described by applying the physics action to the state of each particle to get a new state (or $\perp$ if it dies; we assume that for an action a, $a(\perp) = \perp$).

Basic actions Basic actions are operations that modify the state of the particle. Figure 1 gives a few examples of common actions that are used by particle systems. These include `move`, which updates the particle's position by adding its scaled velocity, `die`, which kills the particle, and `accelerate`, which adds a vector to the particle's velocity. A more complicated example is `bounce`, which reflects the particle's velocity when it reaches the edge of the specified domain.

Sequences Actions are typically combined using the `sequence` combinator

```
val sequence : action list -> action
```

which defines the composition of its arguments.

```
val inside : {
      d : domain, thenStmt : action, elseStmt : action
    } -> action

val outside : {
      d : domain, thenStmt : action, elseStmt : action
    } -> action
```

Fig. 2. Some predicate actions

Predicates Predicates are constructs that take a high level conditional statement and use it to determine which sub-action to apply. Figure 2 gives an example of two predicates that test to see if a particle's position is inside or outside a given domain. Each predicate has two subactions: the `thenStmt` is applied if the predicate's condition is true for the particle and the `elseStmt` is applied if the predicate's condition is false. If the particle's state is $\perp$, then the result is $\perp$.

The `die` action is often used in combination with predicates to mark particles as dead when they meet certain conditions such as their velocity gets too high or they enter (or exit) a specified domain.

3.5 Renderers

Renderers describe the mechanisms used to render the particle system. These are not complex in terms of particle systems and do not take any parameters (except perhaps textures). The main function of the renderer is to translate the particle state into pixels on the screen. The mechanisms required to do this vary based on the renderer, *e.g.*, the points renderer maps particles to points rendered at the particle's position with its color. Some examples of renderers are

```
type renderer
val points : renderer
val texQuads : Texture.texture_id list -> renderer
```

3.6 A complete example

To illustrate the use of these combinators in practice, we examine an example of a fountain implemented as a particle system. Figure 3 gives a screen shot of this system in action. The fountain consists of an emitter just above the XZ plane that launches particles into the air, a disc on the XZ plane that the particles bounce off of when they fall, and a cutoff plane three units below the XZ plane.

First, we define variables for the parts of the particle system that vary between instances.

```
val gravityVec = PSVar.new ("g", PSVar.vec3fTy)
val bounceFriction = PSVar.new ("bf", PSVar.floatTy)
val bounceRes = PSVar.new ("br", PSVar.floatTy)
val emitterRate = PSVar.new ("er", PSVar.intTy)
```

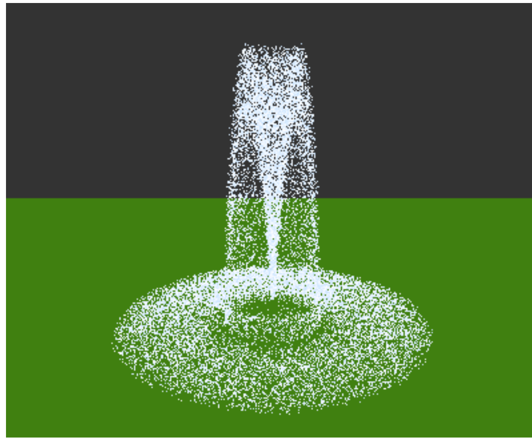

Fig. 3. The fountain particle system in action

Next, we specify the emitter. Each of the values of the emitter specifies a domain for vector properties and constants for the scalar variables. Note that these values do not all have to be constants. For example, the `emitterRate` variable, which specifies the number of particles to generate per-frame, is not a constant variable. Hence, we may change the variable to simulate higher or lower water pressure of the fountain.

```
val emitterFountain = P.newEmitter {
    maxNum = emitterRate, (* emission rate *)
    positionD = P.line(
      const3f (0.0, 0.01, 0.0), const3f (0.0, 0.4, 0.0)),
    velocityD = P.cylinder(
      const3f (0.0, 0.25, ~0.01), const3f (0.0, 0.27, ~0.01),
      constf 0.021, constf 0.019),
    colorD = P.line(
      const3f (0.8, 0.9, 1.0), const3f (1.0, 1.0, 1.0))
    sizeD = P.point(const3f (1.0, 1.0, 1.0 )),
    lifetime = PSVar.constf 100.0
  }
```

Each of the values that are specified by a domain uses the 3D primitive to randomly generate a point within the domain using a uniform distribution. Following the emitter, the action list describes the physics simulation of the particle system. For this system, we simulate a fountain that bounces off of the ground plane. Hence, since the emitter provides the initial upwards velocity, we specify gravity, the bounce off of the ground, and finally death.

```
val bounceDisc = P.disc(
    const3f (0.0, 0.0, 0.0), const3f (0.0, 1.0, 0.0),
    constf 5.0, constf 0.0)

val actionFountain = P.sequence [
```

```
    P.accelerate gravityVec,
    P.bounce {
        fiction = bounceFriction,
        resilience = bounceRes,
        cutoff = constf(0.0),
        d = bounceDisc
      },
    P.inside {
      d = P.plane(
          PSVar.const3f (0.0, ~3.0, 0.0),
          PSVar.const3f (0.0, 1.0, 0.0)),
      thenStmt = P.move,
      elseStmt = P.die
    }
  ]
```

Finally, we create the hooks that will incorporate the particle system into the desired runtime environment. First, we package all three components by calling the `P.create` function. Then, we compile the particle system for the desired environment. At this point, the system is converted first to an internal representation and then to a representation that caters to the chosen runtime. Finally, we create an instance of the particle system, specifying the total number of particles that we want to use for that particular instance.

```
val fountain = P.create {
    emitter = emitterFountain,
    physics = actionFountain,
    render = P.points
  }

val fountainExe = PsysCL.compile fountain

val fountainPsys = PsysCL.new {
    exec = fountainExe,
    maxParticles = 10000
  }
```

Before we are ready to enter the main loop of the program, some of the variables need to be initialized. First, we bind values to the particle system variables that were not initialized during compilation but will not be changing during the actual program execution.

```
PsysCL.bind3f (fountainPsys, gravityVec, (0.0, ~9.8, 0.0));
PsysCL.bindf (fountainPsys, bounceFriction, ~0.05);
PsysCL.bindf (fountainPsys, bounceRes, 0.35);
```

Then, we define a function that renders the system at the current time.

```
fun runsOnce (inst, emitRate) = (
      PsysCL.bindi (inst, emitterRate, emitRate);
      PsysCL.step {inst=inst, t = Time.now()};
      PsysCL.render inst)
```

In order to actually animate the particle system, all we need to do is update it with our desired emitter rate. Since the emitter rate was not bound, we are free to bind the value dynamically, allowing us to increase and decrease the rate of particle creation at runtime.

```
runsniqOnce(fountainPsys, 100 + (100 * sin(timeElapsed)));
```

4 Implementation

Once the particle system has been compiled into an internal representation (IR), we perform a number of optimizations on the IR itself before we go to code generation. During code generation, we allow the user to specify different backends on which the particle system will run: OpenCL, GLSL, or the CPU.

4.1 Internal Representation

Execution of the particle system is handled in a number of steps. First, the emitter and physics components are compiled down into an internal representation. The IR represents programs as a DAG of *blocks*, where each block is a list of *statements*. Variables in the IR are single assignments and we use explicit parameter passing (instead of ϕ-nodes) to represent live variables in control-flow between blocks. Each block contains a list of *statements*, which are procedures used to manipulate *IR variables*. Then, if applicable, the IR is compiled down onto the host runtime environment, *e.g.* OpenCL on the GPU.

Due to the nature of the variables in the particle state, all of the operations performed by the particle system's emitter and physics can be described by primitive vector and scalar operations. This is another reason why this representation of a particle system lends itself to the GPU. But more importantly, it means that all of our higher level constructs involving domains can be represented by a relatively minor set of vector operations.

Variables The IR has its own class of variables to parameterize its operations. Some of these variables correspond to the user-defined variables described in Section 3.1 and others are internal. Similar to the particle system variables, the IR variables have a name, type, and scope associated with them. The scope of the IR variables is restricted to the following:

Constant — a variable that represents a constant value. These variables have global scope and correspond to either user-defined constants and internal constants.

Global — a variable whose value is defined outside the IR and has global scope. These include the unbound user variables and the particle state variables.

Parameter — a parameter to a block. The scope of the variable is its block.

Local — a variable defined by an IR binding in a block. The scope of the variable is the remainder of the block (*e.g.*, similar to a let-bound variable in a lexically-scoped language).

The translation to the IR generates a mapping from user-defined variables to IR globals. This mapping is used to supply instance-specific values for these variables when the particle system is run.

Blocks The IR representation of an emitter or physics component is a DAG of *blocks*. Each block has a list of parameter variables and a body, which consists of a tree of statements. The five basic types of statements are:

`PRIM(y, p, xs, s)`
: is a binding of `y` to the result of applying primitive operator `p` to the argument variables `xs`. The scope of `y` is the statement `s`.

`IF(x, s1, s2)`
: is a conditional where `x` is a boolean variable that is tested, `s1` is the *then* branch, and `s2` is the *else* branch.

`GOTO(b, xs)`
: is an unconditional control transfer to block `b` with arguments `xs`.

`RETURN(xs)`
: is a statement that marks the completion of the component's execution. The variables `xs` represent the results of the computation and correspond to the particle state variables.

`DISCARD`
: is a statement that terminates the particle and discards its state.

State variables State variables are a specific set of IR variables used by the implementation to track the state of the particle. These variables are special because they are passed to the root block as parameters. Then, after the processing is completed, a `RETURN` statement is called with parameters matching those that were passed into the root block of the program.

The state variables are those specified by the emitter (position, velocity, size, color, and life) plus an extra one, called *secondary position*, which contains the value of the particle's position on the previous frame. As described below, some of these variables may be eliminated by optimzation.

4.2 Optimizations

We perform a number of optimizations on the IR to eliminate unused computations and reduce the memory requirements of representing particles. These optimizations are performed in light of the *implicit* data and control-flow between the components. Figure 4 illustrates these dependencies for a simple example. The dashed edge from the return node of the emitter code to the entry of the physics code represents the fact that the particle state created by the emitter is used by the physics. Likewise, the backedge from the return node in the physics code to its own entry represents the fact that the resulting state from one physics step is the input to the next. Lastly, there is an edge from the physics return node to the renderer, which represents the fact that the renderer renders the particle state.

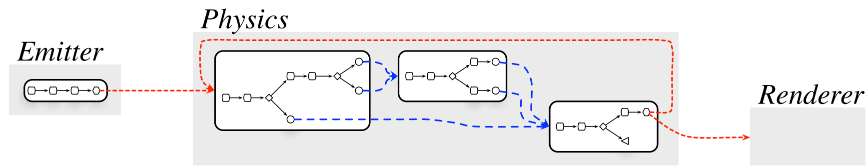

Fig. 4. The IR graph for a simple particle system

Contraction Our optimization performs a number of standard *contractive* optimizations that serve to simplify and reduce the code. These include eliminating unused local variables, merging blocks that have only one predecessor into their predecessor, and constant folding.

Useless variable elimination One of our most important optimizations is *useless variable elimination* (UVE). Many particle systems only use a subset of the state variables, but the translation to the IR must be conservative and include code to support all of them. We use UVE to prune out state variables that cannot possibly affect the rendering of the system.

We use a simplified version of Shivers' UVE algorithm [10], that starts by marking the inputs to the renderer as *useful* (*e.g.*, if we are rendering the system as points, then the position and color are marked as useful). This information is then propagated back through the physics code's control flow, with the right hand side variables of a `PRIM` node being marked as useful if the left hand side is useful, arguments to conditionals are marked as useful, and the useful parameters of a block cause the corresponding arguments to `GOTO`s to be marked useful. Because the physics code is implicitly in a loop, we must propagate usefulness from the root block's parameters back to the corresponding arguments of the `RETURN` statements. Once a fixed point is reached, we propagate the useful variable information into the `RETURN` statements of the emitter and apply the analysis to the emitter code. Since the emitter is run only once per particle, we do not need to iterate to a fixed point. Once the analysis is complete, we rewrite the code to remove any variable that is not marked as useful.

Domain-specific optimizations Many optimizations can be performed on geometric operations to simplify the actual execution of the code. For example, if we take the dot product of a vector v and the unit vector pointing along the y-axis, this is identical to just extracting the y-coordinate from v. We can avoid many superfluous operations in this manner. This optimization is mostly useful in conjunction with constant folding, since many of the available optimizations do not become apparent until one or the other happens.

5 Targetting GPUs

Our system is designed to support multiple backend targets for executing particle systems. These include a CPU target that interprets the IR, a planned backend that translates the IR to GLSL [9] to run on the GPU, and a backend that generates OpenCL [4] code that can run on either the CPU or GPU. These backends implement the standard interface that was described in Section 2.2. In this section, we discuss some of the issues in generating code for the OpenCL target.

Code Generation Generating OpenCL code from the IR is complicated by the fact that the IR is a control-flow graph, while the OpenCL is a block-structured language. Fortunately, however, our combinators do not produce cyclic graphs, so the process is possible. We do a prepass that matches blocks with the `IF` statements (if any) that they are a join continuation for. We then use this information to translate the IR into an AST representation that can be pretty printed as OpenCL code. The code generation also deals with mapping IR variables to OpenCL variables, *etc.*.

Representing particles Particles are represented as an array of OpenCL `structs`, where each field corresponds to a particle state variable. We also create arrays of OpenGL attributes to hold those particle state variables that are used by the renderer. While using separate arrays for the renderer results in some redundancy, it is necessary because some rendering methods require multiple vertices, each with their own set of attributes, per particle. For example, line segments require two vertices per particle, which are defined by the position and secondary position state variables.

Random numbers To support random-number generation, each particle has a random seed as part of its state. The OpenCL translation uses a functional random-number generator that takes a seed and produces both a random number and a new seed. We then thread the seed state through the generated code and save it back in the particle state at the end of the execution.

Particle birth and death Another tricky issue is managing particle birth and death, since the size of the total population affects the number of new particles generated each iteration. Because we want to avoid moving data from the GPU back to the CPU, we have to use a parallel-scan algorithm to compute for each particle the number of live particles with lower IDs [3]. This information can then be used to manage births as follows. Assume that we want to add k new particles, then a dead particle with ID i is reborn if $i - j \leq k$, where j is the number of live particles with IDs lower than i.

6 Related work

Reeves was the first researcher to suggest the combination of stochastic processes and particles to render fuzzy phenomena. His seminal paper describes the basic ideas that

underly all modern particle systems [8]. He describes several applications, including the use of particle systems to render the wall of fire caused by the "Genesis bomb" in the movie *Star Trek II: The Wrath of Kahn* (Paramount 1982).

Our work was inspired by McAllister's C++ library for building particle systems [7]. In this library, McAllister introduced the notion of domains as a type associated with particle systems. From this, we were able to abstract a functional approach to creating particle systems at a high level. Another contribution was Kipfer *et al.*'s method of simulating particles on the GPU [5]. In their paper, they used the GPU to handle both sorting and particle interaction, motivating the design for how we should present our data in order to streamline it on the GPU. Finally, Yi and Froemke's Ticker Tape library provided an example for how particle systems can be created using more intuitive methods [13]. In this library, each operation on a particle system was defined by having a user-defined creation, physics, and rendering operation, which was later composted into one system.

In practice, animation tools define particle systems by manipulating their properties directly. Many of these tools specify particle systems in similar ways. They require the user to define methods for creation and rendering, and then have mechanisms by which the appearance of the particle system is specified. For example, Blender's particle system animation allows for splines and other user-defined particle paths. This method is useful for artists, but it also is limited in that the animations generally target non-real time rendering. As a result, performance and ease of creation are not measures by which we judge the tool's effectiveness.

7 Conclusion

In this paper we have presented a declarative approach to defining particle systems. Our implementation provides a set of combinators for specifying the physics of a particle system; these combinators are then compiled into an internal representation that can either be interpreted on the CPU or translated to code that can run on the GPU. This approach takes advantage of the high-level features of functional languages and demonstrates a way that high-level languages can provide a better programming model for computer graphics.

In practical environments, this method of defining particle systems could be used to create tools that provide a much more visual approach to creating particle systems to be used in the field. Visual programming languages have been used to create procedural animations before, such as Apple's Quartz Composer [1]. Such a programming tool would increase the ease of creating particle systems in both video game and movie production.

7.1 Future Work

There are many features of particle systems that are not implemented in the programming model that we have introduced in this paper. Most notably, there is no way to specify the sorting of particles in our programming model. Also, allowing the user to pass state variables into the particle system run-time environment would introduce many new opportunities for more dynamic definitions of particle systems.

User-defined state variables Actions, such as `accelerate` and `move`, are specific cases of the vector operation $\mathbf{x}_{i+1} = \mathbf{x}_i + t\mathbf{y}_i$, where the $\mathbf{x}$ and $\mathbf{y}$ are bound to specific state variables. By exposing state variables to the user, we can use a smaller set of actions to support our current behaviors. Furthermore, we can allow user-defined state variables and renderers to increase the flexibility of the system. This generalization only requires changes to the user API, since the IR and backends already deal with the general case.

Sorting When rendering a group of translucent polygons, it is generally assumed that the polygons are sorted by their distance from the camera. Kipfer *et al.* introduced a nice way to sort particles on the GPU using a bitonic sort [5]. This sorting algorithm lends itself fairly well to the GPU and would not require an extensive overhaul of our current system. Their implementation, however, leverages the technical aspects of GLSL and does not provide a platform independent way of representing the sorting of particles.

Particle-particle interaction Finally, one last feature of particle systems that should be supported is the idea of particle-particle interaction. Such interactions are a challenge to GPUs, but there are techniques for spatial sorting that can be applied to handle them, such as Kipfer *et al.*s Uberflow system [5]. Using the particles' sorting to determine proximity would allow for many other effects as well, such as flocking.

References

1. Apple Inc.: Quartz Composer Programming Guide (Oct 2008), available from `http://developer.apple.com`
2. Elliott, C.: Programming graphics processors functionally. In: Proceedings of the 2004 Haskell Workshop. ACM Press (Sep 2004)
3. Harris, M.: Parallel prefix sum (scan) with CUDA. In: GPU Gems 3. Addison-Wesley, Reading, MA (2007)
4. Khronos OpenCL Working Group: The OpenCL Specification (Version 1.1) (2010), available from `http://www.khronos.org/opencl`.
5. Kipfer, P., Segal, M., Westermann, R.: Uberflow: a gpu-based particle engine. In: HWWS '04). pp. 115–122. ACM, New York, NY, USA (Aug 2004)
6. Leijen, D., Meijer, E.: Domain specific embedded compilers. In: DSL '99. pp. 109–122. ACM, New York, NY, USA (Jan 2000)
7. McAllister, D.K.: The design of an API for particle systems. Tech. rep., University of North Carolina (Jan 2000), available from `www.particlesystems.org`.
8. Reeves, W.T.: Particle systems—a technique for modeling a class of fuzzy objects. ACM Trans. Graph. 2(2), 91–108 (1983)
9. Rost, R.J., Licea-Kane, B.: OpenGL Shading Language. Addison-Wesley, Reading, MA, 3rd edn. (2010)
10. Shivers, O.: Useless-variable elimination. In: WSA'91 (Oct 1991)
11. The SML3d library, available from `http://sml3d.cs.uchicago.edu`.
12. Witkin, A.: An introduction to physically based modeling: Particle system dynamics (1997), available from `http://www.cs.cmu.edu/~baraff/pbm/constraints.pdf`.
13. Yi, M., Froemke, Q.: Ticker tape: A scalable 3d particle system with wind and air resistance (May 2010), available from `http://software.intel.com/en-us/articles/tickertape`.