A Declarative API for Particle Systems

PADL 2011

Pavel Krajcevski (Disney Interactive Studios)
John Reppy (University of Chicago)

January 25, 2011

Background

Recent trends in Computer Graphics hardware (GPUs) and APIs (*e.g.* OpenGL) have significantly changed the way high-performance graphics applications are written.

- geometric data is communicated in bulk using buffers, instead of per-vertex.
- rendering behavior is controlled by shader programs running on the GPU, instead of by a state machine.
- shader programs are compiled at runtime in the GPU driver.

These trends create an opportunity for high-level languages and declarative approaches.

Background

Recent trends in Computer Graphics hardware (GPUs) and APIs (*e.g.* OpenGL) have significantly changed the way high-performance graphics applications are written.

- geometric data is communicated in bulk using buffers, instead of per-vertex.
- rendering behavior is controlled by shader programs running on the GPU, instead of by a state machine.
- shader programs are compiled at runtime in the GPU driver.

These trends create an opportunity for high-level languages and declarative approaches.

Background

Recent trends in Computer Graphics hardware (GPUs) and APIs (*e.g.* OpenGL) have significantly changed the way high-performance graphics applications are written.

- geometric data is communicated in bulk using buffers, instead of per-vertex.
- rendering behavior is controlled by shader programs running on the GPU, instead of by a state machine.
- shader programs are compiled at runtime in the GPU driver.

These trends create an opportunity for high-level languages and declarative approaches.

Background

Recent trends in Computer Graphics hardware (GPUs) and APIs (*e.g.* OpenGL) have significantly changed the way high-performance graphics applications are written.

- geometric data is communicated in bulk using buffers, instead of per-vertex.
- rendering behavior is controlled by shader programs running on the GPU, instead of by a state machine.
- shader programs are compiled at runtime in the GPU driver.

These trends create an opportunity for high-level languages and declarative approaches.

Background

Recent trends in Computer Graphics hardware (GPUs) and APIs (*e.g.* OpenGL) have significantly changed the way high-performance graphics applications are written.

- geometric data is communicated in bulk using buffers, instead of per-vertex.
- rendering behavior is controlled by shader programs running on the GPU, instead of by a state machine.
- shader programs are compiled at runtime in the GPU driver.

These trends create an opportunity for high-level languages and declarative approaches.

Particle systems

Particle systems are a Computer Graphics technique for modeling fuzzy phenomena [Reeves 1983], such as

- clouds, smoke, water, fire, explosions, *etc.* (dynamic)
- hair, fur, grass, *etc.* (static)

In this work, we address real-time dynamic particle systems.

Particle systems

Particle systems are a Computer Graphics technique for modeling fuzzy phenomena [Reeves 1983], such as

- clouds, smoke, water, fire, explosions, *etc.* (dynamic)
- hair, fur, grass, *etc.* (static)

In this work, we address real-time dynamic particle systems.

Particle systems

Particle systems are a Computer Graphics technique for modeling fuzzy phenomena [Reeves 1983], such as

- clouds, smoke, water, fire, explosions, *etc.* (dynamic)
- hair, fur, grass, *etc.* (static)

In this work, we address real-time dynamic particle systems.

Particle systems

Particle systems are a Computer Graphics technique for modeling fuzzy phenomena [Reeves 1983], such as

- clouds, smoke, water, fire, explosions, *etc.* (dynamic)
- hair, fur, grass, *etc.* (static)

In this work, we address real-time dynamic particle systems.

What are particle systems?

- Real-time graphics uses triangles to model objects, which does not work well for fuzzy objects that have irregular and dynamic shapes.
- Particle systems represent fuzzy objects as large collections of particles.
  - The set of particles is dynamic with new particles being born and old ones dying.
  - Particles have a position and other attributes that evolve over time according to a "physics" model.
  - Particle systems are stochastic.
- Particle systems substitute quantity for quality.
  - The physics model is iterative using Euler integration.
  - Particles are usually rendered using simple primitives (points, lines, or small polygons).

What are particle systems?

- Real-time graphics uses triangles to model objects, which does not work well for fuzzy objects that have irregular and dynamic shapes.
- Particle systems represent fuzzy objects as large collections of particles.
  - The set of particles is dynamic with new particles being born and old ones dying.
  - Particles have a position and other attributes that evolve over time according to a "physics" model.
  - Particle systems are stochastic.
- Particle systems substitute quantity for quality.
  - The physics model is iterative using Euler integration.
  - Particles are usually rendered using simple primitives (points, lines, or small polygons).

What are particle systems?

- Real-time graphics uses triangles to model objects, which does not work well for fuzzy objects that have irregular and dynamic shapes.
- Particle systems represent fuzzy objects as large collections of particles.
  - The set of particles is dynamic with new particles being born and old ones dying.
  - Particles have a position and other attributes that evolve over time according to a "physics" model.
  - Particle systems are stochastic.
- Particle systems substitute quantity for quality.
  - The physics model is iterative using Euler integration.
  - Particles are usually rendered using simple primitives (points, lines, or small polygons).

What are particle systems?

- Real-time graphics uses triangles to model objects, which does not work well for fuzzy objects that have irregular and dynamic shapes.
- Particle systems represent fuzzy objects as large collections of particles.
    - The set of particles is dynamic with new particles being born and old ones dying.
    - Particles have a position and other attributes that evolve over time according to a "physics" model.
    - Particle systems are stochastic.
- Particle systems substitute quantity for quality.
    - The physics model is iterative using Euler integration.
    - Particles are usually rendered using simple primitives (points, lines, or small polygons).

What are particle systems?

- Real-time graphics uses triangles to model objects, which does not work well for fuzzy objects that have irregular and dynamic shapes.
- Particle systems represent fuzzy objects as large collections of particles.
  - The set of particles is dynamic with new particles being born and old ones dying.
  - Particles have a position and other attributes that evolve over time according to a “physics” model.
  - Particle systems are stochastic.
- Particle systems substitute quantity for quality.
  - The physics model is iterative using Euler integration.
  - Particles are usually rendered using simple primitives (points, lines, or small polygons).

What are particle systems?

- Real-time graphics uses triangles to model objects, which does not work well for fuzzy objects that have irregular and dynamic shapes.
- Particle systems represent fuzzy objects as large collections of particles.
  - The set of particles is dynamic with new particles being born and old ones dying.
  - Particles have a position and other attributes that evolve over time according to a "physics" model.
  - Particle systems are stochastic.
- Particle systems substitute quantity for quality.
  - The physics model is iterative using Euler integration.
  - Particles are usually rendered using simple primitives (points, lines, or small polygons).

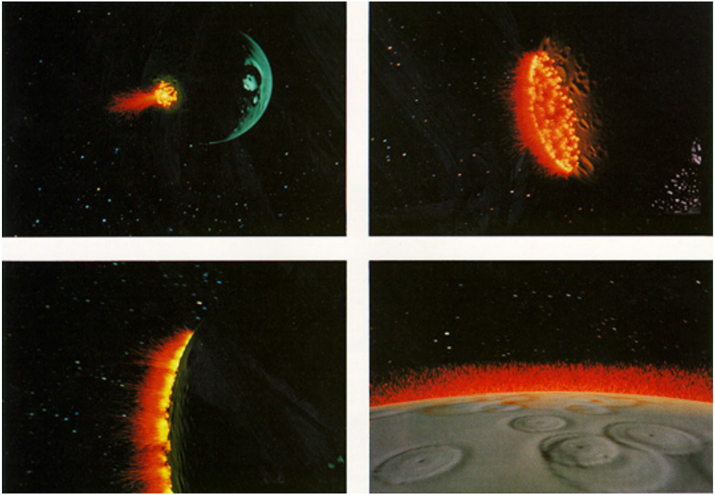
What are particle systems?

- Real-time graphics uses triangles to model objects, which does not work well for fuzzy objects that have irregular and dynamic shapes.
- Particle systems represent fuzzy objects as large collections of particles.
  - The set of particles is dynamic with new particles being born and old ones dying.
  - Particles have a position and other attributes that evolve over time according to a "physics" model.
  - Particle systems are stochastic.
- Particle systems substitute quantity for quality.
  - The physics model is iterative using Euler integration.
  - Particles are usually rendered using simple primitives (points, lines, or small polygons).

What are particle systems?

- Real-time graphics uses triangles to model objects, which does not work well for fuzzy objects that have irregular and dynamic shapes.
- Particle systems represent fuzzy objects as large collections of particles.
  - The set of particles is dynamic with new particles being born and old ones dying.
  - Particles have a position and other attributes that evolve over time according to a "physics" model.
  - Particle systems are stochastic.
- Particle systems substitute quantity for quality.
  - The physics model is iterative using Euler integration.
  - Particles are usually rendered using simple primitives (points, lines, or small polygons).

In the beginning ...



Fountain demo

DEMO

Smoke demo

DEMO

Defining particle systems

Particle systems can be specified in three parts:

1. The emitter, which specifies rules for generating new particles.
2. The physics, which specifies how the state of a particle evolves.
3. The renderer, which specifies how to render a particle.

Particles have a state, which includes attributes like position, velocity, color, *etc.*.

Defining particle systems

Particle systems can be specified in three parts:

1. The emitter, which specifies rules for generating new particles.
2. The physics, which specifies how the state of a particle evolves.
3. The renderer, which specifies how to render a particle.

Particles have a state, which includes attributes like position, velocity, color, *etc.*.

Defining particle systems

Particle systems can be specified in three parts:

1. The emitter, which specifies rules for generating new particles.
2. The physics, which specifies how the state of a particle evolves.
3. The renderer, which specifies how to render a particle.

Particles have a state, which includes attributes like position, velocity, color, *etc.*.

Defining particle systems

Particle systems can be specified in three parts:

1. The emitter, which specifies rules for generating new particles.
2. The physics, which specifies how the state of a particle evolves.
3. The renderer, which specifies how to render a particle.

Particles have a state, which includes attributes like position, velocity, color, *etc.*.

Defining particle systems

Particle systems can be specified in three parts:

1. The emitter, which specifies rules for generating new particles.
2. The physics, which specifies how the state of a particle evolves.
3. The renderer, which specifies how to render a particle.

Particles have a state, which includes attributes like position, velocity, color, *etc.*.

Particle physics

The physics can be captured in an update function.

```
val update : state * float -> state option
```

Here is a simple example of particle-system physics code that would be suitable for water droplets.

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

But writing this code is tedious and it is not portable to other compute devices (*e.g.*, GPUs).

Specifying a particle system

In this talk, we present a declarative approach to specifying particle systems that uses combinators to define particle system behavior.

The specification is split into two steps.

The first step allows one to specify a device independent program consisting of the emitter, physics, and renderer.

```
type program    (* particle-system specification *)

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

Specifying a particle system *(continued ...)*

The second step is device dependent.

```
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
```

The application must choose a device-specific implementation of this interface (*e.g.*, CPU, GLSL, *etc.*).

Variables

Particle systems are parameterized by variables, which can be bound to values at three different times:

1. specification time (these are called constants)
2. per-instance
3. per-frame

We use phantom types to enforce type correctness.

```
val constf : Float.float -> Float.float var
val bindf  : Float.float var * Float.float -> unit
```

Domains

Domains [McAllister 2000] are an abstraction of a region in $\Re^n$.

We use domains to specify the distribution of random points and vectors in emitters (*e.g.*, to specify initial velocity), and to specify effects and boundaries of a particle system.

For example,

- a spherical velocity domain for specifying fireworks, and
- a plane to specify the ground.

Domains are parameterized by variables.

Domains

Domains [McAllister 2000] are an abstraction of a region in $\Re^n$.
We use domains to specify the distribution of random points and vectors in emitters (*e.g.*, to specify initial velocity), and to specify effects and boundaries of a particle system.
For example,

- a spherical velocity domain for specifying fireworks, and
- a plane to specify the ground.

Domains are parameterized by variables.

Domains

Domains [McAllister 2000] are an abstraction of a region in $\Re^n$.
We use domains to specify the distribution of random points and vectors in emitters (*e.g.*, to specify initial velocity), and to specify effects and boundaries of a particle system.
For example,

- a spherical velocity domain for specifying fireworks, and
- a plane to specify the ground.

Domains are parameterized by variables.

Emitters

The emitter controls the creation of new particles according to several parameters:

- the rate of new particle creation (range and distribution),
- the initial position, velocity, and color domains

Actions

An action is an abstraction of a particle-state to particle-state function.

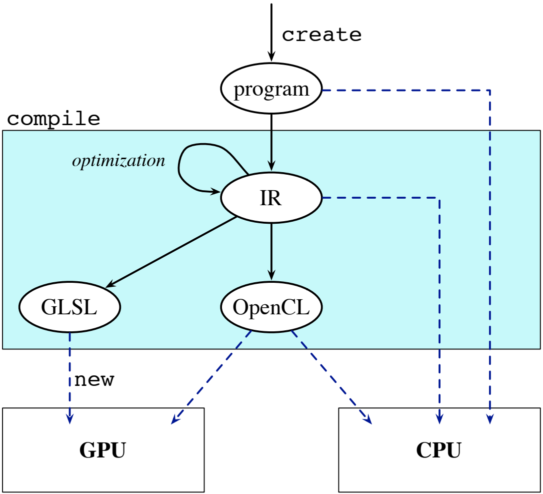
We compose actions to specify the physics of a particle system.

Actions include sequencing, conditionals, and state transformers.

For example, here is a specification of the simple physics for water droplets.

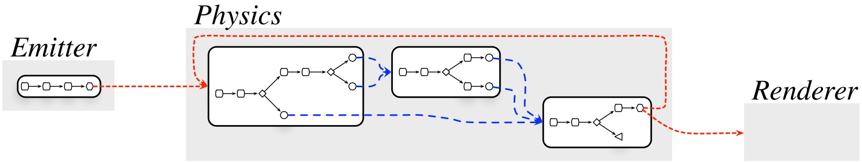
```
P.inside {
    d = groundPlane,
    thenStmt = P.sequence [P.accelerate gravityVec, P.move],
    elseStmt = P.die
  }
```

Implementation overview



Optimizations

We perform a number of optimizations on the IR.

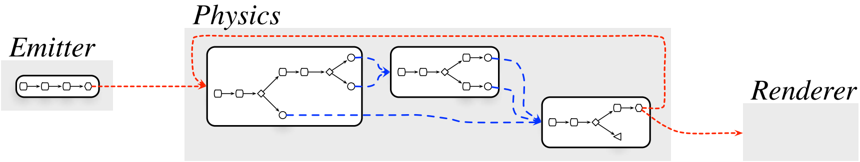


These include:

- useless-variable elimination
- constant folding
- jump elimination
- dead-code elimination

Optimizations

We perform a number of optimizations on the IR.



These include:

- useless-variable elimination
- constant folding
- jump elimination
- dead-code elimination

Status

- System is available as part of the SML3d library.
- Combinators and IR optimizations are implemented.
- CPU-based IR interpreter is working (but is too slow for real-time).
- OpenCL implementation should be available soon.

Status

- System is available as part of the SML3d library.
- Combinators and IR optimizations are implemented.
- CPU-based IR interpreter is working (but is too slow for real-time).
- OpenCL implementation should be available soon.

Status

- System is available as part of the SML3d library.
- Combinators and IR optimizations are implemented.
- CPU-based IR interpreter is working (but is too slow for real-time).
- OpenCL implementation should be available soon.

Status

- System is available as part of the SML3d library.
- Combinators and IR optimizations are implemented.
- CPU-based IR interpreter is working (but is too slow for real-time).
- OpenCL implementation should be available soon.

Future work

- Allow user-defined state variables and generalize actions.
- Multiple emitters for a particle system.
- Particle-particle interactions (*e.g.*, flocking, collisions, *etc.*).
- Apply this approach to other problems: *e.g.*, shading and skeletal animation.

Future work

- Allow user-defined state variables and generalize actions.
- Multiple emitters for a particle system.
- Particle-particle interactions (*e.g.*, flocking, collisions, *etc.*).
- Apply this approach to other problems: *e.g.*, shading and skeletal animation.

Future work

- Allow user-defined state variables and generalize actions.
- Multiple emitters for a particle system.
- Particle-particle interactions (*e.g.*, flocking, collisions, *etc.*).
- Apply this approach to other problems: *e.g.*, shading and skeletal animation.

Future work

- Allow user-defined state variables and generalize actions.
- Multiple emitters for a particle system.
- Particle-particle interactions (*e.g.*, flocking, collisions, *etc.*).
- Apply this approach to other problems: *e.g.*, shading and skeletal animation.

Questions?

`http://sml3d.cs.uchicago.edu`