An embedded language for simulating and visualizing particle systems

Marcus Lindblom

May 12, 2003

Abstract

English

Particle systems are becoming a common sight in today's video games as they are used for a variety of graphical effects, such as fire, smoke etc. Creating and maintaining them while retaining optimal performance has, however, gotten harder as the diversity and flexibility of the latest programmable graphics processors have increased tremendously. A single particle system may therefore need several implementations and these are often written for speed rather than clarity.

This report presents an embedded functional language that attempts to solve these problems, together with an optimizing compiler that generates C++-code (and is prepared to generate graphics processor code). The language unifies the particle system concepts of emitters and particles into *objects* which allow for more complex and interesting systems. The compiler outputs a C++-class which serves as a run-time system and controls object creation, simulation, visualization and removal. Objects can be created and affected by external input which is provided by the frameworks in which the system is run.

Svenska

Partikelsystem blir en allt vanligare syn i dagens videospel eftersom de används till en stor mängd grafiska effekter som eld, rök och mycket annat. Att skapa och använda dem har däremot blivit svårare, eftersom de senaste programmerbara grafikprocessorerna har ökat väldigt i flexibiltet och mångfald. Ett partikelsystem kan därför behöva flera implementationer och dessa är oftast skrivna med prestanda istället för tydlighet i åtanke.

Den här rapporten presenterar ett inbäddat funktionellt språk som försöker lösa dessa problem samt en optimerande kompilator som genererar C++-kod (och är förberedd för att generera grafikprocessorkod). Språket förenar koncepten emitterare och partikel till *objekt* som möjliggör mer komplexa och intressanta system. Kompilatorn ger som utdata en C++-klass vilken innehåller ett simuleringssystem för att skapa, simulera, visualisera och ta bort objekt. Objekt kan även skapas och påverkas externt via det ramverk som systemet används i.

Contents

Chapter 1

Introduction

In this chapter, we will give the background for the project, provide our definition of particle systems as they will be approached. We discuss our target language and architecture and state the problem we are adressing. We also discuss and motivate our choice of language and finally outline the structure of the report.

1.1 Background

The idea for developing a separate language and compiler came to life during the development of Reaper [1], a 3d action game developed in C++ and OpenGL (an open cross-platform graphics API). It features a number of different particle systems for explosions, missile trails, etc. Many of these are quite similar but not similiar enough to lend themselves to reuse without sacrificing performance. These small but numerous systems eventually became too hard to maintain, so a better solution was needed.

There was also a desire to use the programmable processing capabilities of emerging graphics cards and move as much of the calculations from the CPU, thus freeing it for more complex tasks such as AI, physics, etc.. Doing this while still maintaining backwards compability essentially required writing each system twice. Coupled with the diversity of ways for fast geometry transfer at that time, the potential combinatorial explosion of implementations was a sign that automation in some form was necessary. (Reaper was developed mostly during 2001, at which time there was no vendor-independent way of sending geometry to the card fast (i.e. via asyncrounous DMA over the AGP-bus). Recently, OpenGL has been extended with mechanism called *vertex buffer objects* [18] to facilitate this.)

A small compiler was written as the exam project for the course in Advanced Functional Programming at Chalmers and the results where quite promising. There where more than enough ideas for further development to allow the project to continue as a thesis work.

1.2 A brief description of particle systems

Particle systems are mainly associated with three different areas in computing; scientific calculations, movie special effects and real-time computer games. Our system targets the latter of these and strives to improve on ease of creation, maintainability and performance, all which are of equal importance.

The definition of particle systems varies from within computer graphics, but is usually assumed to be composed of a set of emitters that produce new particles and the created particles. We will only concern ourselves with particle systems for visual display, and specifically not those used for simulation complex physical behaviours were the particles can interact with each other. This is the norm for particle systems used in real-time computer graphics today.

A particle usually has the following properties:

- A set of initial values.
- (possibly) Some state that is updated iteratively.
- A behaviour that is shared with all other particles in the system and can be affected by external variables, such as wind direction.
- Some expression signifying when the particle is to be removed.
- A visualization, defining how the particle is to be displayed.

Similarly, an emitter:

- spawns new particles at a specified rate
- creates particles with different initial values
- (possibly) varies its behaviour over time

The independent and 'sameness' properties of particles are important since it allows us to process large number of particles in batches, exploiting the pipelining and parallellism commonly used in todays computing architectures. As we shall see later, our language extends this model to a hierachy of *objects*, that act both as particles and emitters.

1.3 Target platform & languages

Our main target is gaming platforms and especially those used by game developers. A large majority of games today are written in C or C++ and most of the platforms they run on have some kind of programmable graphics processor. C++ is a feasible target language, because it is human readable, which allows for faster compiler debugging, easy comparision of output with production code as well as hand-optimization after compilation. It is also high-level enough so that our compiler can defer common tasks such as register allocation or stack size counting to another compiler.

The programmable graphics processors (henceforth referred to as GPUs) are accessible either by assembly languages or more high-level shading languages

(HLSLs) that all basically are modified C. DirectX 9.0, the latest version of Microsoft's proprietary graphics API for games, also has a HLSL and the OpenGL ARB (Architecture Review Board, a commitee of hardware and software manufacturers that control the evolution of new versions of OpenGL) is at the time of this writing working on a version suited to that API.

However, there is one language that works both these APIs, namely Cg (C for Graphics [2]) developed by NVIDIA (a major manufacturer of high-performance graphics chipsets). It is also supported by some of the major content authoring packages and have existed in the market for some time now. The platform independence and the relative stability makes it our preffered choice. However, the similiarity between all these HLSLs will probably make retargeting a rather smooth procedure, since the hardware running beneath the abstraction layer does not change.

The available feature set naturally differs between different generations of GPUs (the Cg language offers different *profiles* for this), much more so than that of CPUs, so a specific target is necessary here too. At the start of this thesis project, the state-of-the-art GPU available for the PC was NVIDIA's GeForce4, which has the ability to execute small (128 instructions) custom programs for each vertex. It is not possible to branch, loop or call functions, and the output is always sent into the rasterization unit, yielding no way of regaining the full precision of the output (The writeable part of the frame buffer is four 8-bit integers). However, our chosen GPU is very similiar to the one found in the X-Box (A game console manufactured by Microsoft and powered by a modified version of NVIDIA's GeForce 3 GPU, which has a single programmable pipeline, vs. two on the GeForce 4) and thus our compiler target is not as restricted as it first may seem.

Since the start, new GPU's have emerged from multiple vendors that offers increased computational power and flexibility that lifts some restrictions that our compiler works with. More discussion on this can be found in chapter 7.

1.4 Problem statement

Language We want a language that makes it easy to design complex particle systems with a maximum of versatility and expressiveness. It should also allow as much reuse as possible. Ideally, a developer should be able to build a library of common particles, emitters and other objects that can be composed into new creations with very few lines of code.

The language should focus on clarity and ease-of-use, rather than potential performance and opportunities for the user to perform heavvy optimization. We will leave those tasks for the compiler and strive to make our design as user-friendly as possible.

Compiler The output from the compiler should be equal or better to that of a human programmer, but not necessarily one that is an expert in optimization. Since the output is in plain and readable C++/Cg, there is always an option for optimization by the developer. The biggest challenge for the compiler will probably be to take maximum advantage of the GPU. This means separating the expression depending on their semantic requirements.

Since we have full control of the generated code, we do not need to build any reusable structures in C++ and thus have the opportunity to produce code that is optimal in each case.

1.5 Language choice

Here we discuss a few approaches on how to design particle system languages, in order to motivate our final choice, which is a language embedded in Haskell. How well this approached worked is discussed in the concluding remarks, chapter 6.

To embed or not to embed

When designing a domain-specific language there are, strictly speaking, two different ways of solving the problem. The first is a self-contained language, with syntax and type-system. The second is to embed it in another language, meaning that we define our constructs in terms of functions or objects in the *host* language. Writing programs in our language essentially means writing a program in the host language that, in the end, uses only our constructs. This approach means that our language inherits what tools and features that language provide, e.g. syntax, parsers, static type-checking, debuggers, etc. Thus, one needs to consider each of these properties carefully.

Another thing to consider is the compilation target. Having the source language embedded in the target language essentially boils down to writing an API, which has a few advantages over a separate compiled language. Compilation is a one-step process and debugging of the source language is considerably easier as it can be inspected together with the running environment (in our case, the game engine).

Embedding in C++ One approach might be to embed in C++, since that is one of our target languages. In our case, as we are targeting both C++ and Cg, we need some form of compiler to bridge the gap to Cg, even if we did embed our language in C++. Since the basic operators are overloadable, it is possible to build expression trees directly in C++ (while still getting acceptable syntax) and thus write a separate compiler that transform these to partial C++/Cg and then compile again.

As C++ is a widely used language (and the C-inherited syntax even more so) specifically amongst game developers but also in the software community at large, using it means that our tool would be almost immediately applicable to many projects and useful for many developers. Still, the language has to be powerful and concise enough to be considered a valid choice.

Had we not wanted Cg-code, one approach could've been to build a set of smart, lazy-evaluated template functions that perform as much optimization as possible during compilation. This approach is called *template metaprogramming* [4] and would give good local optimization with a minimum of work, but we still to rely on the C++-compiler for global optimization, which is quite limited compared to what we can do with a separate compiler due to our better knowledge of the problem.

Embedding in Haskell A popular host language for domain-specific embedded languages is Haskell and there are numerous examples of successful languages that have used this approach (from music [5], animation [11], images [12] to hardware design [14] and robot control [6]. Details for some of these are given in chapter 5.2. Many of these are accompanied with compilers or interpreters also written in Haskell, and some of them match quite closely what we are trying to achieve.

Haskell's powerful features, such as type-inference, neat indent-sensitive syntax, first class functions and a rather powerful type-systems makes many of these specialized language appear very different and suprisingly well adapted to their problem, compared to their host language.

Self-contained language Designing a completely new language means that apart from just deciding semantics and implementing a compiler for that, one also has to decide on syntax and types, and implement the corresponding parser and type-checker. These are not trivial design decisions so one should be clear about what problem the language should attempt to solve and what sort of programs one wants to write in it. Also, more stages and transformations means that changes made to the language result in increased work spent updating the compiler.

Our choice

As we have previous experience in implementing compilers and some smaller embedded languages in Haskell, as well as developing larger C++ applications, C++ does not appear as suitable for rapid compiler development. Altough there are several compilers written largely in C or C++, such as the GNU Compiler Collection [7], they are rather large and unwieldy pieces of software, and often use macros or other type-unsafe means to make development feasible. On the other hand, Haskell has many nice features, especially pattern-matching, which makes it especially suited for compiler work, which is our choice as compiler development language.

The decision on embedding or not falls on the fact that we will try to write something which has little precedents, namely a domain-specific language for particle systems. Thus one can expect that the language will change during development as new opportunities or directions surface. Having an embedded language allows us to make changes without significant rework of the compiler. We also retain the option of developing a separate language should we feel that embedding hinders our design, once it is more clear.

1.6 Overview

Now that we know *what* we want to adress and the basic approach, we will spend the of the report on *how* we solved the problems. Our language is outlined in the tutorial in chapter 2, which explains the basic concepts of building expressions and defining particle systems.

Chapter 3 gives the syntatic constructs used in our language, followed by some more advanced examples and finally the formal semantics for the interesting (non-obvious) subset of the language. Following this, chapter 4 describes

our compiler, explaining our approach to compilation. There are also sections on the separate passes, how they relate and outlines on some of the algorithms used.

After describing our work, chapter 5 compares it to other attempts at defining and using particle systems, as well as looking at related embedded functional languages. We sum up our achievements in chapter 6 and the final chapter 7 looks at some possible directions for the future.

Chapter 2

Tutorial

This tutorial aims to introduce the user to the particle language and the basic concepts necessary to create simple particle systems. A more rigourous description of the language, together with more examples and semantic defintions is given in chapter 3. It is assumed that the reader is somewhat familiar with functional languages.

2.1 Introduction

Before we look at some concrete code examples, we need to clarify the basic building blocks of our particle systems. In our language, we differ between *objects*, *commands* and *expressions*, all of which are necessary to create a meaningful program.

expression An *expression* is a mathematical formula, such as $2x + \sin\alpha$. Our expressions can contain if-else conditionals, integration andderivation, local state, references to external data and non-determism through random values.

command *Commands* are statements that make things happen, this can either be to display a particle on the screen, create new objects or remove the current object from the system.

object We define an *object* to be a set of expression and commands that are used for simulation. An object usually computes some values such as position and color, and then act on these to perform certain commands.

A particle system can have any number of different objects, as well as many objects of the same type. An object can have expressions that depend on its parent, i.e the object that created it, (or any of its grand-parents), but not on its siblings or children.

2.2 Types and operators

In the following examples, we will give the types for the non-obvious new constructs used. Therefore, a few words about the type we use are in order.

Our main types are expressions and commands. Expressions can have four different types, *Boolean*, *Scalar*, *Vector* and *Color*. Vector and Color types are composites and have scalar (i.e. float point values) members, three and four respectively. Vector have the synonym Point, and these two are completely interchangable. Their use simply makes things easier to understand.

The type class *NumE* denotes numerical expressions, which is Scalar, Vector and Color. Also, both commands and expressions are of type `Exp *`, but with different types for `*`. (There are also another type class used to facilitate numerical operations between scalar, vector and color values, *NOp*, whose implementation is given in appendix A.3. We will give somewhat simplified types in this tutorial that do not use this type, as it would clutter the definitions too much.)

We have also created our own operators (since Haskell's own are typed in ways that does not fit our cause everywhere) but they are merely the standard operators with an extra character appended (where necessary, for some cases the usual operators work as expected).

- An asterix `*` as prefix to the comparision operators.
- A dot `.` to prefix numerical operators. Due to the type system, operations where one or two operands are literals (e.g. a float literal such as **3.5**) must use the standard Haskell operator, without prefix.
- A capital e `E` as postfix to standard functions (such as ifE).

2.3 Particles

For our first example, we want to a particle object that is affected only by gravity, starting at origo with a initial upwards velocity. A program which gives us this would then have the following look:

```
obj       :: (Scalar -> Cmd) -> Cmd
point     :: Point -> Color -> Scalar -> Cmd
kill      :: Boolean -> Cmd
integrate :: (NumE a) => Scalar -> Exp a -> Exp a
(<+>)     :: Cmd -> Cmd -> Cmd
vec3      :: (Scalar, Scalar, Scalar) -> Vector
vec4      :: (Scalar, Scalar, Scalar, Scalar) -> Color
white = vec4 (1,1,1,1)

--

myparticle =
 obj $ \t ->
  let pos = origo .+ integrate t vel
      vel = vec3 (0,10,0) .+ integrate t acc
      acc = vec3 (0,-9.82,0)
  in point pos white 1 <+> kill (t *> 10)
```

Here, the particle has a white color and a size of 1. The particle will be removed from the system after 10 seconds.

The function **obj** takes as argument a function from time to a command, so that we can evaluate the expressions a certain times, thus creating an animation over time. The execution of an object command results in the creation of a new object and a subsequent insertion into the system.

`<+>` is an operator that takes two commands and combines them. `kill` removes the object if the expression evaluates to true. `integrate` performs integration of an expression over time.

Had we want the pooint to fade from white to black over a period of ten seconds, we could have used the function **lerp** (type and example given below), which performs linear interpolation between two values.

```
lerp :: (NumE a) => Scalar -> Exp a -> Exp a -> Exp a

point pos (lerp (t/10) white black) 1
```

2.4 External variables

If we want to have our particles affected by some external force, which is controlled by the outside system (such as the wind direction in a game, something that probably depends on which world or where in a specific world we are), we can define variables to be from the environment:

```
envVector :: String -> Vector
-- similar for other types

myparticle =
  let wind = envVector "wind_direction"
      pos = origin .+ integrate vel
      vel = vec3 (0,10,0) .+ integrate acc
      acc = vec3 (0,-9.82,0) .+ wind .- vel
  in point pos red 1
```

`envVector` allows us to read an external variable and use it in our definition of the particle behaviour. However, this is not all. Our definition now has a cycle in it! Velocity depends on acceleration which depends on velocity. This is not a problem, since the cycle goes through an integration. As our compiler does not have a symbolic ODE-solver, we need to break this cycle in another way. We do this by switching to numeric integration and using the velocity at the previous update when calculating the acceleration for this frame. As we integrate from $t = 0$ and forwards, any evaluation at $t \leq 0$ yields zero, which is what we use as our initial state.

2.5 Randomization

Most of the time, we want each of our particles to look a bit different in order to give the system a more "natural" appearance, so what if we added a bit of randomness to the initial velocity?

```
rand     :: Rand a => (a -> Cmd) -> Cmd
```

```
nrand    :: Rand a => (a -> Cmd) -> Cmd
snapshot :: Exp a -> Exp a

myparticle =
 obj $ \t ->
  nrand $ \(x,z) ->
   rand $ \c ->
    let pos = origo .+ integrate t vel
        vel = snapshot $ vec3 (x,10,z)
        col = lerp c red blue
    in point pos col 1
```

`rand` and `nrand` both takes a function from a random-type to a command, and returns a command. A random type could be any of a float, a tuple of floats, a point, a vector or a color. The implementation of the `Rand` class is given in appendix A.2. The difference between the two functions is that `rand` gives random values in the [0..1] range while `nrand` uses the [−1..1] range. `snapshot` evalutes the expression once, when the particle is created. Now our particle has a initial random velocity and is flickering (rather wildly, one might add) in various shades between red and blue.

2.6 Emitters

To define an emitter that emits any object, at a rate proportional to $\sin t$, we would write is as a function which takes the object to emit as an argument, and returns an emitter object.

```
emitter :: Scalar -> Scalar -> Cmd -> Cmd

myemitter p =
 obj $ \t -> emitter (50 + 50 * sin t) 1 False p
```

The first argument to the emitter is the *rate*, i.e. how many emission events per second there should be. The second value is the number of objects per emission event[1]. This emitter will be creating new objects with a varying rate, one at a time but one hundred per second at maximum.

2.7 A complete system

Now we will link together our emitter with our previous particle, and make an object that emits two streams of particles.

```
rotatey :: Vector -> Scalar -> Vector

origin = vec3 (0,0,0)
```

[1] In the current version, only rate is used. The second value is completely ignored and treated as fixed to 1 (one). Future versions will use this and also make it possible to differ between particles created at one specific event.

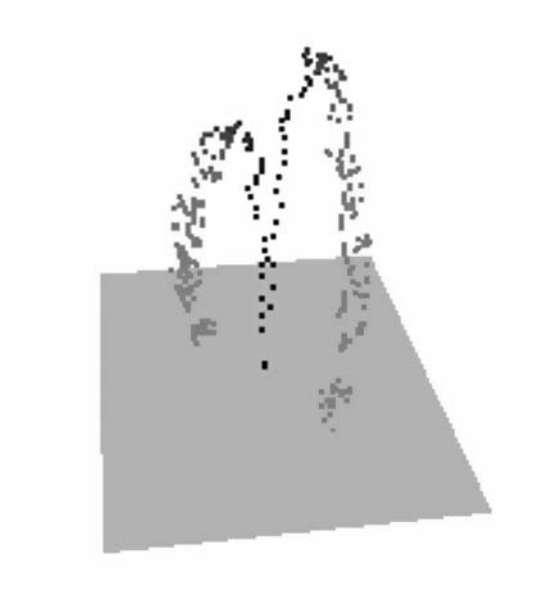

Figure 2.1: Rotating emitters

```
part v =
 obj $ \t ->
  let wind = envVector "wind_direction"
      pos = origin     .+ integrate t vel
      vel = snapshot v .+ integrate t acc
      acc = vec3 (0,-9.82,0) .+ wind .- vel
  in point pos red 1 <+> kill (y pos <* 0)

emit a =
 obj $ \t ->
  nrand $ \(x,z) ->
   let vel = rotatey (vec3 (x+3,10,z)) ((t+a*2*pi)/10)
   in emit 100 1 (part vel)

mysystem = emit 0 <+> emit pi
```

There, our very own particle system. Most of it we recognize from earlier examples. The two emitters will rotate with a speed of one revolution per 10 seconds and the particles will die once they fall below the ground level.

A picture of this system in action can be found in figure 2.1.

2.8 Output to C++ code

Once the system has been defined, compiling the system to a C++ class is straightforward.

```
cppout :: Cmd -> String -> IO ()

> cppout mysystem "particle_test"
```

This will write two files to the disk: particle_test.cpp and particle_test.h. The header file will look something like this:

```
#include <hspark.h> /* definition of vector/color/etc */

namespace hspark {

class particle_test
{
public:
  particle_test();

  void init();
  void update(float time);
  void draw();

  void set_wind_direction(vec3 &value);
  vec3& get_wind_direction() const;

  ...

private:
  ...
};

} // end namespace hspark
```

So, in our application we create a gl-window, create an instance of the class, set the wind-direction (intially zero), call `init()` to create the top-level objects and start updating/drawing to see your particles be born, animate and die.

There are also functions for inspect the particle system in a limited manner, to see how many particles are alive and dead, to ease debugging and tuning.

Chapter 3

The language

This chapter strives to explain our language. We begin by giving an overview of the syntax and the constructs we use to build particle systems. Some larger examples are also given and the chapter ends with formal semantics for a (conceptually equivalent) subset of our language.

This chapter assumes that the reader is familiar with functional languages and Haskell in particular. A detailed report on the entire Haskell language can be found here [3]. We use a few extensions to the Haskell language, mainly *multi-parameter type classes* and *functional dependencies*, both of which are implemented in GHC and Hugs, two of the most widely used Haskell compilers and interpreters.

3.1 Syntax

Here we describe the most important syntactic constructs in our language. This section is terse in some sections, as the basic concepts are outlined in the tutorial in chapter 2. Detailed description on the execution is given in the section on semantics 3.3. As with the tutorial (where the types are presented in more detail), the types of the functions are simplified versions of those used in the actual implementation, to avoid clutter.

3.1.1 Objects and commands

Object creation and destruction are handled with the following primitives:

```
obj       :: (Scalar -> Cmd) -> Cmd
kill      :: Boolean -> Cmd
(<+>)     :: Cmd -> Cmd -> Cmd
point     :: Point -> Color -> Scalar -> Cmd
emitter   :: Scalar -> Scalar -> Cmd -> Cmd
```

`obj` takes a function from time to command. Time roughly corresponds to the object's age (usually in seconds, but depends on the external framework's concept of time). Note that we cannot use a single time-variable in the entire system, since this would make it impossible to differ between

parent time and child time in certain expressions (e.g. integrations and derivations).

`kill` removes the current object from the system if the expression evaluates to true

`<+>` composes two commands. Their ordering is not significant and may be changed by the compiler.

`point` displays a point at the given position, with a color and a size[1].

`emitter` executes the commands at the given rate (first argument, event per second) with the given quantity (executions per event)[2].

3.1.2 Integration/derivation

We can integrate and derive values over time, using the following constructs:

```
integrate :: (NumE a) => Scalar -> Exp a -> Exp a
derive    :: (NumE a) => Scalar -> Exp a -> Exp a
```

The scalar must be a expression containing a time-variable for some object. It can be scaled, but results are undefined if a decreasing or negative value is entered.

3.1.3 State

In order to compute complex behaviours, we often want to track some state. This can be because we using a complex expression or simply because we want to know when some event has occured that we have no control over (such as randomization or external input).

```
state  :: (Exp s -> (Exp a, Exp s)) -> Exp s -> Exp a
state2 :: (Exp s -> Exp a) -> Exp a -> Exp a
prev   :: Exp a -> Exp a

prev e = state (\s -> (s,e)) e
```

`state` takes a function from state to output and new state, and an initial value.

`state2` is a convenience function when the output and the new state is equal. This is used to implement numerical integration and derivation, amongst other things.

`prev` function gives the value of the expression from the *previous* update.

Using `prev`, we can define a simple derivation approximation:

```
nderive e t = (e - prev e) / (t - prev t)
```

[1] Point primitives cannot be individually sized on current hardware. Currently, the compiler ignores the size value and uses a fixed size of two. This will be adressed in the future, allowing a set of points emitted by the same command to have the same size.

[2] Currently, the quantity value is ignored by the compiler

3.1.4 Random values

In order to provide random values into our systems, thus making them appear more realistic, we need to ensure that the same random value is used everywhere in the expression. Otherwise, a rotation matrix constructed from a random angle might turn out completely incorrect.

We solve this by forcing the user to provide a function that takes a value (which will be random) and produces an expression. We also provide two versions of our random construct, depending on the range of the random variable:

```
rand, nrand  :: Rand a => (a -> Cmd) -> Cmd
```

The first function works with any of our basic types and gives values of True/False or uniformly distributed in the range [0..1]. The second is not meaningful to booleans and gives values in the range $[0..-1]$. For vector types, each component is given a separate random value.

Several examples of random expressions are given in the tutorial, chapter 2.

3.1.5 Recursive lets

We do not have recursive lets (letrecs) as one would normally expect, since we do not have lazy evaluation in our targe language. Rather, we use it as syntactical sugar for state expressions were possible. One needs to be aware of the implications of our transformation here, as the semantics are not altogether obvious.

The reason for having letrecs is that they allow us to express differential equations in an easy manner. One would like to write something like to express the position in terms of starting position and velocity, where the latter depends on the position.

```
i = let p = 3 + integrate v
        v = 2 - p
    in p
```

This is transformed by the compiler into an expression using the previous value, which corresponds to Euler-step approximation for integration.

```
e = state (\p' ->
      let p = 3 + integrate v
          v = 2 - p'
      in (p,p)) 3
```

This tranformation is possible as long as the cycle passes through an expression that has a well defined value at $t = 0$, such as integration. If not, one can give any expression an explicit start value using the following predefined function:

```
start e i = state2 (\_ -> e) i
```

Note that no actual state will be held here, since the state variable does not occur in the result of the expression, the state-expression can safely be optimized into the original expression `e` once the initial value has been used in the transformation described above.

3.1.6 External variables

In order to access data provided by the outside world, the following constructs are used:

```
envBool   :: String -> Boolean
envFloat  :: String -> Scalar
envVector :: String -> Vector
envPoint  :: String -> Point
envColor  :: String -> Color
```

3.1.7 Trigger

If we want to execute commands from the outside, to allow the framework to create new objects inside our system, one can wrap the object command with the `trigger` function and provide a name. This will generate an accessible function in our resulting C++ class which will execute the wrapped command.

```
trigger :: String -> Cmd -> Cmd

mysys = trigger "medium_explosion" $
  obj $ \t -> ...
```

Triggers can be applied at any level and on any command, but if a object which depend on the parent is triggered, the external framework has to provide the missing information, which may not be trivial to compute.

3.1.8 Snapshot

Sometimes we want to freeze the value of some expression at the time of creation of an object. For instance, the position of a moving object that emits particles is a good initial value for each particle, but since the object is moving, we cannot simply evaluate the expression each frame. The `snapshot` function evaluates the expression once, at object creation, then it remains constant for the entire lifetime of the object.

```
snapshot :: Exp a -> Exp a
```

3.2 Examples

We give here a few larger examples on how particle systems are constructed. For a more basic explanation of our language, consult the tutorial in section 2.

3.2.1 Bouncing particles

To achieve bounce, we want to test for collision and alter position and velocity in that case. Our language is not powerful enough to do this properly, as we cannot hold two values in one state simultaneously. So, in this example, we do a compromise and only change the velocity. This works well under the assumption that particle velocities are low compared to the time-steps. A picture of this system is given in figure 3.1.

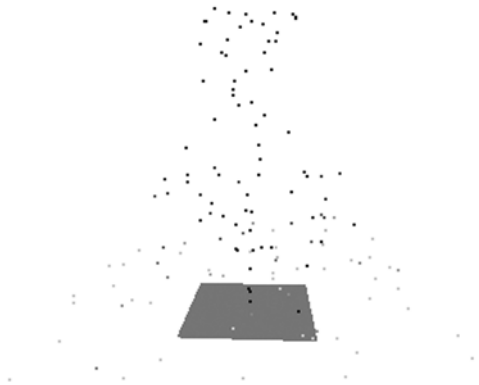

Figure 3.1: Bouncing particles, from example 3.2.1.

```
ifE     :: Boolean -> Exp a -> Exp a -> Exp a
changeY :: (Scalar -> Scalar) -> Vector -> Vector
gravity m = vec3 (0,-9.82 * m,0)


bounce =
 obj $ \t ->
  nrand $ \(x,z) ->
   emitter 20 1 (part $ vec3 (x*2,15,z*2))
    where
     part ivel =
      obj $ \t -> let
       pos = origin .+ integrate t vel
       vel = state2 (\v ->
      changeY (\yv -> ifE (y pos <* 0) (abs yv * 0.9) yv)
         (v .+ acc .* diff t )) ivel
       acc = gravity 0.3
       col = vec4 (1,1-t/5,1-t/10,1)
      in point pos col 2 <+> kill (t >* 7)
```

3.2.2 Fire simulation

Here, we define each particle to have a temperature that decreases exponentially over time. The temperature affects the particle's color and upward acceleration, as to create an effect of warm air surging upwards and then cooling. We also emit particles over an area of 2x2 units, with an random initial position, to simulate turbulence at the base of the fire. A picture is given in figure 3.2.

```
fire =
  obj $ \t ->
       nrand $ \(x,z) -> let ipos = vec3 (x,0,z) in
```

Figure 3.2: Fire simulation, from example 3.2.2.

```
      nrand $ \(ivel::Vector) ->
        emitter 500 1 (part ipos ivel)
  where
    part ipos ivel =
     obj $ \t ->
        let pos  = snapshot ipos .+ integrate t vel
            vel  = snapshot ivel * 3 .+ integrate t acc
            acc  = gravity 0.2 .- vel .+
                   vec3 (0,max 0 (temp/3-15), 0)
            temp = 100 * exp (-t * 0.3)
            ct   = temp/70
            col  = vec4 (1,ct/2,ct/4,ct)
        in point pos col 2 <+> kill (y pos <* 0)
```

3.2.3 Fireworks

Here we have three levels of objects. First, we have an emitter that fire rockets, ten each per second, at a random, but generally directed upwards, velocity. The rockets emit small particles as they fly away. Each of those particles are affected only by gravity, fading from green to black over time. A picture is given in figure 3.3.

```
fireworks =
 obj $ \t ->
 nrand $ \(x,z) -> emitter 10 1 (emit2 $ vec3 (x*2,15,z*2))
  where
   emit2 ivel =
```

Figure 3.3: Fireworks, from example 3.2.3.

```
 obj $ \t -> let
   pos = origin .+ integrate t vel
   vel = snapshot ivel .+ integrate t (gravity 0.3)
   in point pos white 2 <+> emitter 50 1 (part pos vel) <+>
      kill (y vel <* 0)

part ipos ivel = obj $ \t -> let
   pos = snapshot ipos .+ integrate t vel
   vel = origin .+ integrate t acc
   acc = gravity 0.4
   in point pos (lerp t green black) 2 <+> kill (t >* 1)
```

3.3 Formal semantics

In this section we give formal semantics for our language. The reason for this is that since the source and target languages have quite different semantics, we need to define this translation properly, so that there is no discrepancies between what the user specifies and what the compiler outputs. We also make different choices depending on if we want correct behaviour or efficient implementation, and these need to be documented.

Before describing the semantics in detail, we will look at our compilation target and how and where our program is executed. With this in mind, we present a complete, but restricted, abstract syntax that we will use to finally define our semantic relations.

3.3.1 Calculation model

We want to create, simulate and destroy objects (that is emitters and/or particles). Their behaviour depends on their age, but may also link to external data, such as wind direction or the position of an object or their parent object. These behaviours may also be quite complex, such as such as collision or physics simulation, which often requires both state and the ability to know how big the current time-step is (this is necessary in order to answer the question "Will we collide during this frame?").

The execution environment we target will not have lazy evaluation if we do not implement it ourselves and further, the graphics processor targeted in this thesis does not have sufficient semantics for efficient handling of expressions using any form of state, so we need to overcome this gap in some way.

As has been noted previously in this report, we will have a set of objects that exists within our systems and are simulated iteratively. Some will emit new particles, others will display some graphics, etc.. Assuming our system is running, for each update we need to perform a number of steps for each object in our system.

- Evaluate the expressions
- Execute any commands

Depending on the expressions and commands of an object, evaluation will take place at different places. Optimally, we want everything to be computed on the GPU, freeing the main processor for other, more complex work (typically AI, physics and similar tasks). However, the differing semantics force us to divide the work between the two execution units.

Evaluation of emitters must take place on the CPU, since an emitter will modify a state (the set of particles to be displayed). Particles, however, can be evaluated entirely on the graphics adapter, providing that they do not have any state in their expressions. In those cases, we can split the evaluation into stateful and stateless parts.

3.3.2 Calculation types

We have four types of calculations, with different complexity and semantics. We will use these properties to place their evaluation on different parts of the target platform.

E **expressions** Simple side-effect free expressions with non-recursive lets and strict (call-by-value) evaluation.

A **assignments** Assigns the value of an expression to a variable.

B **behaviours** This is what the user deals with when supplying values to commands. The more complex semantic constructs here are transformed into simpler expressions when the object is created.

C **commands** Commands are used at the highest level, where we actually get some visible results. They are used to create objects and display graphics.

Expressions and behaviours are polymorphic and four types of results are supported: Boolean, Scalar and Vectors (having three or four components). Only floating point arithmetic is supported, due to the architecture of our target platform, which means that booleans are represented with the numeric values 0 and 1.

We need to define two more things: our output, in form of graphic on the screen, and what data our objects should contain. For the latter, we must obviously include the command and the creation time of the object. We also need to hold some state (a mapping from variable name to value), and a list of assignments, which are used to update the state each frame.

Note that we have two states with different scope and mutability. One is per-object or local and mutable, the other is per-system or global, and is not mutable from the object's point of view. The latter data can only be modified from an external source (see section 3.1.6 for more details).

3.3.3 Abstract syntax

Figure 3.4 gives a restricted form of the abstract syntax of our language. We define *op* to be any of the usual operators over expressions (numerical, trigonometric or logical operations) and semicolon as the sequencing operator.

3.3.4 Semantic evaluation relations

Expressions

Expression evaluation will be denoted $\Downarrow_{\mathsf{e}}$, with the relation shown below. We will not provide detailed semantics for this relation, as they are rather obvious. Suffice to say is that they are strict, have non-recursive lets, a non-deterministic construct (**rnd**) and a set of operators commonly used with mathematical and logical expressions.

$$(Expression \times State) \times Value$$
$$(E, s) \Downarrow_{\mathsf{e}} v$$

Assignments

Assignments are denoted with $\Downarrow_{\mathsf{a}}$, a state-modifying relation. The semantics for this are, as with expressions, obvious and will not be discussed in detail.

$$(Assignment \times State) \times State$$
$$(E, s) \Downarrow_{\mathsf{a}} s'$$

Update

During each frame update, we process the objects together with the external state and the current time to produce new objects and output. This relation is denoted $\Downarrow_{\mathsf{u}}^{\alpha}$, where α indicates that output is being performed. This relation only has a single construct, *update*, which is used to make one iteration step for the system.

$$
\begin{array}{rcl}
C & ::= & \textbf{emitter } B_{rate}\ B_{qty}\ \mathrm{C} \mid \\
 & & \textbf{point } B_{pos}\ B_{color}\ B_{size} \mid \\
 & & C; C \mid \textbf{kill } B_{bool} \\
 & & \\
B & ::= & \textbf{snapshot } B \mid \\
 & & \textbf{state } (B' \rightarrow (B, B'))\ B' \mid \\
 & & \textbf{let } x = B' \textbf{ in } B \mid E \\
 & & \\
A & ::= & x\ := E \mid A; A \mid \textbf{nil} \\
 & & \\
E & ::= & \textbf{apply } op\ [E] \mid \\
 & & \textbf{let } x = E' \textbf{ in } E \mid \\
 & & \textbf{rnd} \mid x \mid n \\
 & & \\
Output & ::= & \tau \mid point(p, c, r) \\
 & & \\
State & ::= & Ident \rightarrow Value \\
 & & \\
Result & ::= & \textbf{ok} \mid \textbf{remove} \\
 & & \\
Object & ::= & (C, State_{local}, A, Time)
\end{array}
$$

Figure 3.4: The abstract syntax used in our semantic relations

$$
\begin{gathered}
(Time \times Objects \times State_{ext}) \times Actions \times Objects \\
\langle t, o, s_e \rangle \Downarrow_{\mathsf{u}}^{\alpha} \langle o' \rangle
\end{gathered}
$$

Commands

Command execution is denoted $\Downarrow_{\mathsf{c}}^{\alpha}$, whose result is a set of enew objects, some graphic output together and a status flag indiciating whether the object is to be removed or not.

$$
\begin{gathered}
(Command \times State) \times Actions \times (Result, Objects) \\
\langle C, s \rangle \Downarrow_{\mathsf{c}}^{\alpha} \langle r, o \rangle
\end{gathered}
$$

Behaviours

We define the interpretation of behaviours as a two stage process. At object creation, all contained behaviours in the commands are transformed by the initialization step, denoted $\Downarrow_{\mathsf{i}}$. This will reduce the command to containing only

expressions and also provide a possible initial state and a sequence of assignments used to update the state. Thus, at runtime, we only have expressions, assignments and commands to evaluate, which is rather straightforward.

$$(Behaviour \times State_{ext}) \times (Expression \times State_{local} \times Assignment)$$
$$\langle B, s_e \rangle \Downarrow_{\mathsf{i}} \langle E, s_l, A \rangle$$

To simplify our semantic rules, and since behaviours are always contained within commands, we use the same notation ($\Downarrow_{\mathsf{i}}$), for initialization on commands.

3.3.5 Semantics of update

Update's responsibility is to inspect each object and make sure that it is updated to the current time. For objects created during frame we have two choices. Assume that they are created at the end of the frame, thus we don't have to perform an update on them, or take into account the time difference between their creation and end-of-frame. Since the update frequencies usually varies between 20 and 60 hz, and one wants to use systems that emit more than a hundred particles each second, visual artifacts appear since particles do not appear to move in a continous stream. Therefore we take this extra step (which incur a one-time cost every time a particle is created, but is acceptable).

The evaluation basically follows the step outlined in the beginning of this chapter. The external state is merged with the local state during the evaluation of each object.

$$\frac{\begin{array}{c}\langle C, s \rangle \Downarrow_{\mathsf{c}}^{\alpha} \langle \mathbf{ok}, o' \rangle \\ \langle A, s \rangle \Downarrow_{\mathsf{a}} \langle s_l' \rangle \\ \langle \mathbf{update}, o : o', s_e \rangle \Downarrow_{\mathsf{u}}^{\alpha} \langle o'' \rangle\end{array}}{\langle \mathbf{update}\ t\ (C, s_l, A, t_{create}) : o\ s_e \rangle \Downarrow_{\mathsf{u}}^{\alpha} \langle (C, s_l', A, t_{create}) : o'' \rangle}$$

$$\frac{\begin{array}{c}\langle C, s \rangle \Downarrow_{\mathsf{c}}^{\alpha} \langle \mathbf{remove}, o' \rangle \\ \langle \mathbf{update}\ t\ o\ s_e \rangle \Downarrow_{\mathsf{u}}^{\alpha} \langle o' \rangle\end{array}}{\langle \mathbf{update}\ t\ (C, s_l, A, t_{create}) : o\ s_e \rangle \Downarrow_{\mathsf{u}}^{\alpha} \langle o' \rangle}$$

where $s = [s_e, s_l, \mathbf{time} \mapsto t - t_{create}]$

3.3.6 Semantics of commands

kill

The kill command evaluates the boolean expression and signals if the result is true, which aborts execution of other commands.

$$\frac{\langle E, s \rangle \Downarrow_{\mathsf{e}} \langle \mathbf{true} \rangle}{\langle \mathbf{kill}\ E, s \rangle \Downarrow_{\mathsf{c}}^{\tau} \langle \mathbf{remove}, - \rangle}$$

$$\frac{\langle E, s \rangle \Downarrow_{\mathsf{e}} \langle \mathbf{false} \rangle}{\langle \mathbf{kill}\ E, s \rangle \Downarrow_{\mathsf{c}}^{\tau} \langle \mathbf{ok}, - \rangle}$$

sequencing

The sequencing execution only ensures that any **remove** status aborts evaluation of further commands.

$$\frac{\langle C_1, s\rangle \Downarrow_{\mathsf{c}}^{\alpha} \langle \mathbf{remove}, o\rangle}{\langle C_1; C_2, s\rangle \Downarrow_{\mathsf{c}}^{\alpha} \langle \mathbf{remove}, o\rangle}$$

$$\frac{\begin{array}{c}\langle C_1, s\rangle \Downarrow_{\mathsf{c}}^{\alpha} \langle \mathbf{ok}, o\rangle \\ \langle C_2, s\rangle \Downarrow_{\mathsf{c}}^{\alpha} \langle r, o\rangle\end{array}}{\langle C_1; C_2, s\rangle \Downarrow_{\mathsf{c}}^{\alpha} \langle r, o\rangle}$$

point

$$\frac{\begin{array}{c}\langle E_{pos}, s\rangle \Downarrow_{\mathsf{e}} \langle p\rangle \\ \langle E_{color}, s\rangle \Downarrow_{\mathsf{e}} \langle c\rangle \\ \langle E_{size}, s\rangle \Downarrow_{\mathsf{e}} \langle r\rangle\end{array}}{\langle \mathbf{point}\ E_{pos}\ E_{col}\ E_{size}\ E_k, s\rangle \Downarrow_{\mathsf{c}}^{point(p,c,r)} \langle \mathbf{ok}, -\rangle}$$

We simply evaluate the expressions and use them as parameters in our output.

emit

$$\frac{\begin{array}{c}\langle E_{rate}, s\rangle \Downarrow_{\mathsf{e}} \langle r\rangle \\ \langle E_{num}, s\rangle \Downarrow_{\mathsf{e}} \langle n\rangle \\ \langle C_i, s\rangle \Downarrow_{\mathsf{i}} \langle C_i', s_{i,local}', A_i\rangle \text{ for each } i\end{array}}{\langle \mathbf{emit}\ E_{rate}\ E_{num}\ E_k\ C, s\rangle \Downarrow_{\mathsf{c}}^{\tau} \langle \mathbf{ok}, o\rangle}$$

$$\text{where} \quad o = \forall i. \cup (C_i', s_{i,local}', A_i, ctime(t, i, r, n))$$

E_{rate} determines the number of emits events per second, E_{num} the number of particles to be emitted at each such event. $ctime(t, dt, i, r, n)$ calculates creation times for the new objects, given the current time t, time passed since last update dt, index of the current object i, rate r and number n as defined above. The function also makes sure that expected behaviour is achieved if either r och n is below 1. I.e. for constant rate and number of 0.5, a particle will be emitted every 4 seconds.

3.3.7 Semantics of behaviours

The section on syntax overview describes briefly what each of these behaviours do. Here, we formalize that description.

snapshot

Snapshot's intent is to capture a behaviours value at the creation time of the object. Therefore, we take care of snapshotting during the initialization stage.

$$\frac{\begin{array}{c}\langle B, s_e\rangle \Downarrow_{\mathsf{i}} \langle E, s_l, A\rangle \\ \langle E, [s_e, s_l]\rangle \Downarrow_{\mathsf{e}} \langle v\rangle\end{array}}{\langle \mathbf{snapshot}\ B, s_e\rangle \Downarrow_{\mathsf{i}} \langle v, -, -\rangle}$$

Note that we discard any assignments that result from intializing the "snapshotted" behaviour, as they are not relevant to the value of the expression.

state

State fits our iterative execution model perfectly, so by creating a new variable we can transform it into a new expression, an assignment and add the variable with an initial value to the state.

$$\frac{\begin{array}{c}\langle B_{init}, s_e\rangle \Downarrow_{\mathsf{i}} \langle E_{init}, s_{init}, A_{init}\rangle \\ \langle E_{init}, [s_e, s_{init}]\rangle \Downarrow_{\mathsf{e}} \langle v\rangle \\ \langle B, s_e[x \mapsto v]\rangle \Downarrow_{\mathsf{i}} \langle E, s_l, A\rangle \\ \langle B_s, s_e[x \mapsto v]\rangle \Downarrow_{\mathsf{i}} \langle E_s, s_{sl}, A_s\rangle\end{array}}{\langle \textbf{state } f\ B_{init}, s_e\rangle \Downarrow_{\mathsf{i}} \langle E, [s_l, s_{sl}, x \mapsto v], A; A_s; x := E_s\rangle} \text{ if } \begin{cases} x \text{ not } \in dom(s) \\ f(x) = (B', B_s) \end{cases}$$

Note that we, as with snapshot, discard the assignments A_{init} caused by the initial state behaviour, as we are not interested in any further evaluations. Also, the state s_{init} is used only when evaluating the intial expression.

It's worth noting that:

$$\textbf{snapshot } B = \textbf{state } \lambda x.(x, x)\ B$$

Chapter 4

Compiler

Here we describe our implementation of the compiler. First, we will do an overview of our target language and describe our approach to the compilation. Then, we explain the abstract syntax and types used inside our compiler. We also go through the passes briefly while explaining how they relate and their ordering (if necessary), then each pass is discussed in more detail. Finally, we describe our C++-backend.

4.1 Overview

The central item in our language is the *object*. As was noted in the section 3.3 on semantics, an object contains a number of commands and a state. The state is initialized at object creation and updated with each object, together with the execution of any commands that exist. We will therefore attempt to compile our program into a set of low-level operations, such as assignments and expressions, which are supported in our rather high-level target language. This allows us to focus on even higher-level optimizations and avoid many previously solved problems, such as register allocation or stack-usage calculations.

Our compiler will do as much simplification and optimization as possible, making the iterative update explicit, while still retaining the original hierarchial layout of our syntax tree. The backend is then responsible for transforming the tree into a structure more suitable for the target language.

4.2 Syntax

In this section, we will explain the abstract syntax and the types, using a restricted version for brevity. The full types are given in appendix B for completeness.

Throughout the compiler, we always work with the same type of syntax tree, altough different passes may expect the tree to have different properties. This makes it easier to reorder, insert or remove passes in order to find the combination which yields the best output. Also, we can use a common set of operations on the syntax tree (such as free variable extraction), almost regardless of which pass we are in.

4.2.1 The expression type

Here we explain the expression type and the different constructors.

First, a word about recursive expressions. We have our expression stored in so called Normal Form, i.e. *Apply Op [Expr]*, rather than as separate applications for unary, binary or n-ary operations. The advantages of this is that we only have one constructor which is hold other expressions and this makes it easy to write recursive operations over the syntax tree. The disadvantages is that we may create malformed expressions, i.e we cannot be sure that an operation only has the exact number of arguments that it requires. This has not shown itself to be a problem while developing the compiler and the ease of which recursived operations can be written has definitely improved code readability.

Our expression type have the following definition:

```
data (Indir e) => E e =
  A Op [e] |
  F (Func e) |
  Var Scope Type Id |
  LitB Bool |
  LitF Double |
  RandFloat |
  Void
```

The `Indir` type class provides a level of indirection in the recursive step, which we use both to annotate the expression with more information, as well as wrapping it in other types that provide extra functionality. It is explained in more detail in the next section.

Apply (A) This is our basic recursive step, where several expressions are composed into one, using operations such as addition, assignment, etc.

Function (F) We allow the user to build expressions that contain functions, so we must have functions in our type as well. However, our first transformation is to introduce variables and replace the functions by actual, inspectable expression.

Variables (Var) Our variables have, beside an identifier, also type and scope. Scope defines if the variable is global (external), local (stored with an object) or temporary (only used during computation of an expression).

Random (Randfloat) This expression is replaces by a pseudo-random number each time it is evaluated.

Literals (LitB,LitF) We use doubles to retain as much precision in the compiler as possible.

Void This is used during optimization to denote the empty expression.

It is worth noting that we use a restriced set of numerical operators, which still allows all possible computations, to make optimization easier. More on this in section 4.3.8.

4.2.2 Wrapping the expression

In order to add information about our expressions, our expression type is parametric over a wrapper (or indirection) type. This type can either just add information about an expression (e.g. a list of subexpressions occurìng in it, see section 4.3.8) or store a pointer to the expression, so that cyclic expression can be found, as described in section 4.3.3.

The definition for our indirection type class is as follows:

```
class (Eq e, Show e) => Indir e where
  unwrap :: e -> E e
  wrap   :: E e -> e
```

These wrapping functions are used whenever one wants to inspect an expression or (re)build expressions. For the main part of the compiler, no annotation is added so the wrapping type has the following definition:

```
newtype Expr = MkExpr (E Expr) deriving (Eq,Ord)
```

The downside of this is that it is cumbersome to use pattern matching on composite expressions, as the `MkExpr` type constructor clutters the code. This can be eased by applying `unwrap` on the list of subexpressions but is not always feasible.

4.3 Passes

Here, we first give an overview of the passes and a short description of their individual purpose, as well as relating them to other passes. Then, each (non-trivial) pass is discussed in detail.

4.3.1 Overview

Each pass attempts to deal with a single problem only. The first passes work mainly to make the expression more applicable to further processing, such as optimization of different kinds. Here follows a brief description of each pass, ordered from first to last.

functions and cycles First, we need to introduce variables to get results from any functions we have. We also need to find cycles and make them explicit to avoid infinite structures. As the cycles easily run through our functions, we need to perform both of these in the same pass.

resolve cycles We transform cyclic expressions into expressions with state, using the previous value where possible (otherwise we fail).

scoping Expressions in child objects that refer to their parent are moved to the parent object.

integrate/derive Integration/Derivation expressions are turned into iterative approximations using state.

snapshot Snapshot expressions are changed into the equivalent state expressions (see section 3.3.7) that never update their state.

state State expressions are split into initialization, value calculation and update parts, as in our semantic model in section 3.3.7. This means that we partially intialize our behaviours in the compiler.

if-floating Computation is pushed down the branches of conditional statement to open up more opportunities for optimization later in the compilation stage

let-floating Let-expressions are also transformed to form as "dense" expressions as possible.

scalar-conversion Computations performed on the CPU need to be scalar, thus we split all vector expressions into their components.

local optimization Algebraic optimization, constant folding, etc.

common subexpression elimination Finds shared expressions and replaces them with a single computation.

output Transforms the compiled and optimized program into C++-code.

After the snapshot pass, our expression mostly contains different let and state-expressions. Many of these are equal and could be unified by common subexpression elimination, had we been able to test not only for exact equality but also for alpha equality.

The passes from state to scalar-conversion attempts to produce as uncluttered expressions as possible, in order to get the maximum benefit out of the optimization stages.

4.3.2 Introducing variables

As we permit expressions with functions and functions are black-box data in Haskell, we need to transform them into concrete expressions before we do any further processing. We do this by generating fresh variables and applying them to the functions. Also, we need to combine this with the search for cycles, as there is no guarantee that a cycles will not involve a function, or vice versa.

We have three types of function expressions, whose definition is as follows.

```
data (Indir e) => Func e =
  RandVar  (e -> e) |
  StateVar (e -> (e,e)) e |
  TimeVar  (e -> e)
```

RandVar As noted in the syntax section 3.1.4, we need to ensure that introduced random variables are different. By wrapping this in a function it is possible to introduce separate variables for each random expression during compilation, thus avoiding any confusion.

StateVar State is represented as a function from the previous state to a tuple of the value and the new state. As we have a primitive `StateE` in our syntax, this application is very straight-forward.

TimeVar Objects are, as noted in section 3.1.1, functions from time to commands and the hierarchial structure of objects makes it possible for childs to contain expression referring to their parent's time.

4.3.3 Finding cycles

The cycle in the following expression is easy to see, but evaluating this in Haskell yields an infinite loop structure.

```
> let p = (3 + p) ::  Expr
> p
3 + 3 + 3 + 3 + ...  <forever>
```

In order to find cycles, we want *observable sharing*, a technique introduced in Lava [14], an embedded language for defining, testing and compiling hardware descriptions. In that report, a more in-depth discussion on this problem is given. The solution is a type called `Ref`, which allows the user to determine whether two `Ref`'s were created from the same variable.

Ref basically allows the following operations:

```
data Ref = ...  -- :: * -> *; Eq, Ord, Show

ref   :: a -> Ref a
deref :: Ref a -> a
```

This is exactly what is necessary to find cycles, as it is indeed the same expression used again. In order to apply this to our expressions, we make sure that the expression type is wrapped with `Ref` at every use. This is accomplished by using the following indirection type (outlined in section 4.2.2) for expressions:

```
newtype ERef = MkERef { eref :: Ref (E ERef) } deriving (Eq, Show)

instance Indir ERef where
  unwrap (MkERef e) = deref e
  wrap e = MkERef (ref e)

> let x = ref 2
> x == x
True

> ref 2 == ref 2
False
```

When traversing the syntax tree, we keep track of all references used previously (in expressions further up in the tree) and search the current expression's reference in order to find cycles. If a cycle is found, a new variable is created and we get an explicit recursive let-expression (the `LetRec` operator, see section B). These are then resolved (if possible) in later pass.

So, running the previous example again, but with our `ERef` type instead, we see that we have found the cycle successfully.

```
> let p = (3 + p) ::  ERef
> p
letrec rec_1 = 3 + rec_1
```

We could also have used this to find sharing between subtrees (horizontal sharing), but have chosen not to. The main reason is that this is also done during common subexpression elimination (CSE), where we also find sharing between expressions not having the same origin (see example below). Further, the compiler might choose to inline these expressions and altough they appear shared in the source, they are not so after compilation.

```
-- sharing possibly found with Refs
y = let z = sin x
    in z + z

-- sharing impossible to find with Refs
-- (might be the result of inlining the previous example)
y = sin x + sin x
```

4.3.4 Resolving cycles

The previous section described how to find and make explicit any cyclic structures. There is still a problem in evaluating these, as we do not have lazy evaluation in our target language. Our strategy at solving this involves noting that we are evaluating our expressions iteratively and thus can use an iteration approximation instead.

Consider the following expression, which is a differential equation of the first order. Our compiler finds the cycle and returns an expression with a recursive let, something our target language lacks semantics to evaluate properly.

```
let e = let p = 3 + integrate t v;
            v = 2 - p
        in (10 + p) :: Scalar

> e
e = 10 + (letrec rec_1 = 3 + integrate t (2 - rec_1))
```

As this is a differential equation, we could apply a symbolic ODE-solver and get the resulting expression directly. However, the possibility of creating complex expressions means that such a solver might not always be successful. Thus, we need a numerical solution, and a a common approach to solve differential equations numerically is to transform them into a state model. We have here chosen the simplest solver, namely Euler integration. By finding the initial value of this expression and introducing a state, we get the following expression:

```
e = 10 + state (\p' ->
          let p = 3 + integrate (2 - p')
          in (p,p)) 3
```

Actually, we can do this tranformation automatically as long as the cycle passes through an expression that has a well defined value at $t = 0$. For expressions involving integration, who essentially are differential equations, this works well. For other expressions (e.g. boolean expressions), this approach might not be feasible at all, so the user must be aware of this when writing cyclic expressions.

4.3.5 Scoping

Since we allow child objects to depend on their parent, i.e. the child contain an expression with variables referring to the parent's time and/or random values, it is necessary that these expressions are evaluated in the parent object, not in each child. Naturally, this also is an optimization in most cases. Therefore, we need to move expressions into their correct scope and replace them with links to the parent.

One problem with this is that when a parent is removed from the system, we cannot immediately replace it with a new object if it still has children. Some sort of reference counting or garbage collection is necessary to support this scheme. Of course, one could decide to kill all child objects when a parent dies, but this is often not desirable. Our current protocol also permits children to detect and act on the case when their parent is dead, as the same expression to kill the parent could be use in a conditional statement to alter the behaviour of the children.

One optimization that we currently do not employ is to detect whether parent expressions are used only in snapshots, and thus need not be stored in the parent's state, but only computed when the parent creates a new child.

4.3.6 Integration/Derivation

Integration and derivation can be done either symbolically or numerically. Currently, we have chosen the most general path and always use numerical integration (although we have prepared for the future extension to attempt symbolic integration), employing Euler-integration, as there are some expressions that are hard or impossible to perform symbolic integration or derivation on.

Numerical integration works well for most expressions, but as is the case with Euler steps, the result is dependent on the step-length. With particle system, one is normally only concerned with visual appearance, so as the resulting system appears correct, there is no problem. However, in order to achieve this, one expects every particle to behave like every other of the same type. This means that the step-length should be the same for each step during each particle's life time, i.e. for all updates if particles are created continously. This may or may not be the case, depending on what framework the system is inserted in.

Since it is often enough to fix the time-step at a certain value to get reproducible results, one could use a rather low time step and interpolate between the computed values. By tweaking the constants of the equations the desired apperance is then achieved, which may be different from the physical, correct behaviour, but as noted above, correctness is not as important as appearance.

4.3.7 Floating

If-floating

If-floating attempts to push computation down the branches of conditional expressions, to open up possibilities for optimization. This technique is used the Pan [12]-compiler (Pan is an embedded language for describing images over time). An example of how if-floating works can be seen in figure 4.1.

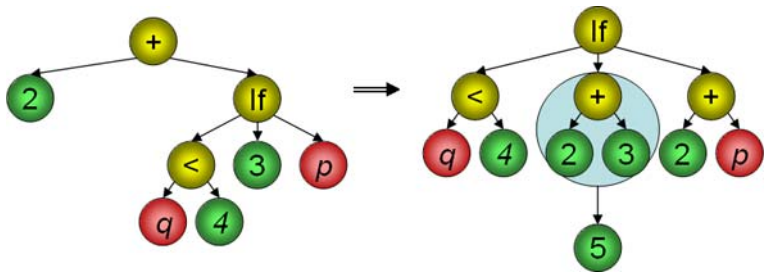

Figure 4.1: If-floating facilitates optimization.

```
-- start
x = (let y = sin t in 2*y*y) / 2

-- lift y
x = let y = sin t in (2*y*y/2)

-- apply partial evaluation
x = let y = sin t in y*y
```

Figure 4.2: Let-floating

On the CPU, this does not create extra work, only more code in the program. However, as the GPU does not have branching[1], we should probably be more careful about using this. However, we currenlty assume that our later common subexpression elimination will take care of any expressions occuring in both branches, eliminating duplicate work.

Let-floating

Let-floating works in much the same way as if-floating, lifting let-expressions as high as possible in the tree and allowing it to be inlined therefore producing a "tighter" expression with more optimization capabilities. An example of this is found in figure 4.2.

4.3.8 Optimization

We optimize our expressions in two steps. First we perfrom *local optimization* using algrebraic rewrite rules to minimize each expression as much as possible. Secondly, we perform *common subexpression elimination* to avoid duplicate work.

In order to simplify constant folding and the number of cases in the local optimizer, as well as reducing the forms an expression can take (to ease equality comparision when doing global optimization), we have removed some operators that can be expressed in terms of others. For instance, we do not have subtraction, only negation and similarly only reciprocals, not division. This means

[1]Our target, the GeForce3/4 does not. Newer chips have emerged since then which support branching and looping in the vertex processor. They do not have it for the fragment processor yet, though.

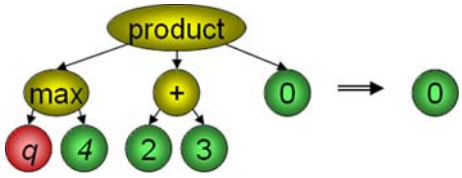

Figure 4.3: Optimization of products

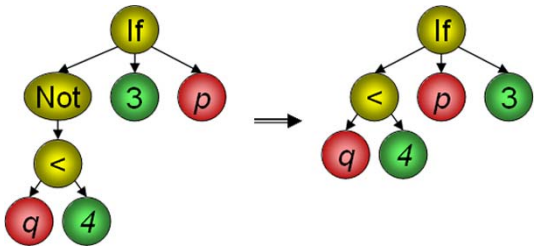

Figure 4.4: Optimization of if-expressions

that we have less cases to deal with during local (algebraic) optimization, and less variations of the same expression, so that we can identify equal expressions easier during global optimization.

Local optimization

Local optimization essentially amounts to using common algebra rules together with partial evaluation to minimize the expressions. In making sure that each transformation results in a smaller expression, we can use fixpoint iteration to reach the smallest possible expression.

Examples of how local optimization work can be found in figures 4.3 and 4.4, where the product in the first figure is simply reduced to the zero as it is the identity element for multiplication. In the latter, the negation of the boolean expression can be removed if we interchange the two branches, yielding an equivalent, but smaller, expression.

Common subexpression elimination

In order to identify equal expressions, we keep track of all expressions encountered and label each with an identifier. We also annotate the expressions (using an indirection type described in section 4.2.2) with its identifier and the identifiers of all its subexpressions. This is useful in determining where to insert the computation of the common subexpression, as we shall see later.

Our common subexpression elimination (CSE) algorithm works in three steps which requires a total of two traversals of the syntax tree. Figures 4.5 and 4.6 outline the steps taken by the algorithm, which works as follows:

- First, we walk down the tree and label all expressions, ensuring that equal expressions get identical identifiers.
- Secondly, as we back up, we gather all identifiers from the expression's

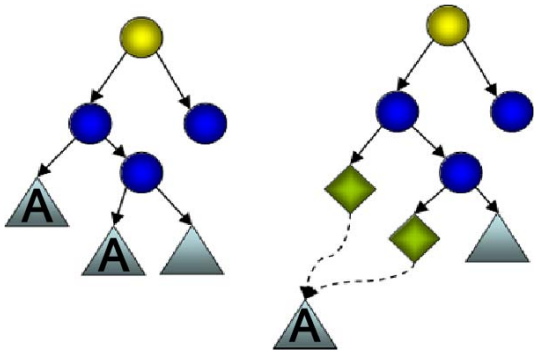

Figure 4.5: CSE algorithm part 1: find shared trees and replace by identifiers.

subexpressions. This means that the top expression, it will hold a list of all identifiers ever.

- Last, we go down the tree again and examine each expression and the identifiers of its subexpressions. If an identifier occurs in two or more subexpressions, we insert a let-expression and replace the common expression with a variable in all subexpressions.

As with local optimization, we apply this repeatedly until no further change occurs (this might be possible to avoid by rewriting the algorithm slightly). We must take care not to unify only side-effect free expressions, that is no random expressions or commands (assignments, etc.). We also avoid optimizing some (basically) zero-cost operations, such as array-indexing.

We do not identify subexpressions of n-ary operators as common subexpressions, i.e $a + b$ is not identified in $(a + b + c) \cdot (a + b + d)$. One way of attacking this is to analyse what the typical expressions look like and use this heuristic to break them into composite expressions (i.e. sum of sums), in order to get better output. Also, we do not perform alpha-equivalence tests in order to compare expressions with bound variables.

Apart from this, the algorithm produces good output, but is a bit slow (ca 5 seconds on each of the examples in section 3.2) and produces a lot of "garbage" due to the reapplication, i.e. let-expressions such as `let x = y in x`, that has to be removed later (by another pass of local optimization, for instance).

4.3.9 Output

The final stage is to turn our compiled and optimized program into compilable C++-code. This done by first breaking the hierarchial structure into a set of objects and extracting the necessary information. This allows us to build a run-time system that creates, updates and removes objects during simulation. The final step is to create concrete, readable code, which is done with the aid of a pretty-printing library which is now available as a part of the GHC Haskell compiler's standard library.

The backend does not support GPU-execution at this moment, but the compiled program is structured such that the state-less expressions are easily identifyable.

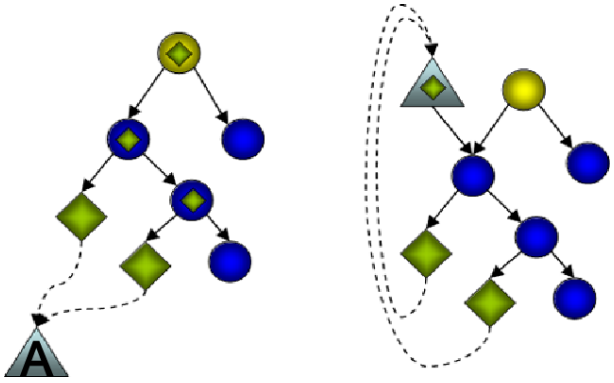

Figure 4.6: CSE algorithm 2: propagate indentifiers and reinsert tree at parent.

Separating objects

After compilation and optimization, the syntax tree is still hierarchial, but now only contains basic operations, such as assignments, if-statements and our commands (emit, point, etc). The first step is to separate each object into state variables, create and update procedures. We also gather external variables (to generate access functions) and trigged commands.

Processing objects

As the different types of objects are fixed (by the way we define our systems) but the number of objects may vary, objects of the same type are stored in separate, resizable containers. During update, we then iterate through each container and apply the same set of commands to each object.

Each object has a separate create-functions, which takes the current time and a reference to the parent object, should such exist. The create function initializes an object and updates it so that it matches the current time. The reason for this is, that our emitters create several objects during a frame and we want correct behaviour for this. This is also noted in section 3.3.6.

There is only one update-function, which in turn processes each type of objects. This allows us to compute expressions common to several objects once. This function also makes sure to process parent objects before child objects, so that childs who reference their parents always get the correct value.

Aging objects

Since the created system stores time internally and only updates it with deltas, it can be defined to start at zero. Then, by storing the creation time of each object, we can compute the age without updating any state for the object. This is desired as floating point computations are faster than memory accesses on modern memory-cached computing architectures. It is also essential if we want to move the computations to the CPU in its entirety.

By marking dead objects with an impossible creation time (such as -1) and storing the locations of these in a separate contaniner we can efficiently skip them while updating and quickly find free locations for new objects.

Chapter 5

Related work

Here we look at other approaches to particle systems and embedded languages that relate to our own and compare their approaches with ours, where appropriate.

5.1 Particle systems

Particle system tutorials Tutorials that introduce particle systems to beginner programmers (e.g. a tutorial on Gamasutra [8], the website of a game developer maganize and NeHe [9], a OpenGL tutorial website) both state that the usual way to build system is to write a single particle structure that holds all properties of each particle (position, velocity, color, mass, etc.) and then define emitters and simulations over these.

Clearly not all particles require all these properties, certainly not as a state, and there may be other properties one desires that does not fit this model. Also, moving this to Cg essentially boils down to writing and maintaing two code bases. This may be feasible for small applications, but as was noted in the introduction, does not scale well to larger systems.

Our approach ensures that each system only does the required computations and also eases reusability and maintainability due to the declarative approach.

GPU-only particle systems In order to show what the new GPUs can do, manufacturers present demos where the particle is computed (almost) entirely on the GPU (creation and destruction are still CPU based).

However, as one cannot hold any state for the particles, the simulations becomes rather restricted. Also, on systems without programmable graphics adapters, one again has to duplicate the development effort. Our approach is better for the same reasons given the previous section.

The Particle Systems API The Particle Systems API [10] is an OpenGL-like API in which the user can design particle systems by defining properties and forces on particles and how they should be created and destroyed. It defines a notion of *particle groups* which are a set of particles with identical behaviour. Each particle group can then be affected by predefined *actions*, such as gravity, wind, etc.

Once a system is specified, the list of actions can be compiled similarly to a displaylist in OpenGL. These list are then executed on the current particle group, either on the main CPU or on the custom geometry processors in the SGI PixelFlow architecture, a workstation predecessor to today's user-level programmable GPUs.

This system also suffers from the restriction on predefined properties and behaviours, making customization difficult. The author mentions this in the design report and expresses the need for a more general way to handle properties and actions.

5.2 Languages & compilers

There are a few systems that also work with compiling a embedded visualization and simulation language. The author have not found any programming languages or compilers dealing directly with particle systems, but there are a number of closely related products.

All of these are embedded languages in Haskell and deal either with animation or interesting compilation problems.

Fran Fran (Functional Reactive Animation [11]) is used to model 2D and 3D animations and sound. It has a continuous representation of time and uses behaviours and events. This requires complex interval analysis to acertain as when events occur, value updating and event handling/detection. The events occur at certain, fixed points in time so that it is possible to shift behaviours at mouse presses or when a boolean expression becomes true (e.g. when time is equal to 10 seconds). The behaviours are evaluated lazily which makes it possible to use cyclic structures.

The analysis required to compute this performed in an asynchrounous thread evaluating a Haskell program, running in parallell a high-performance rendering thread. The rendering thread samples values and performs interpolation to generate values at an interactive rate. Events and behaviours allow Fran's output to change, appear, disappear and vary in many interesting ways, reacting to external input.

Fran is more expressive than our langugage, allowing behaviours to change completely and even link them in cycles. This is due to its use of lazy evaluation in the interpreter, which is written in Haskell, something that we cannot reapply in C++ without considerable effort in our compiler. It also uses multithreading, something we do not wish to impose upon the user of our system.

As there is only one time in the system, expressions for integration and derivation are unambigous, and therefore there is no problem due with referential transparency, something that we had to take into account.

Pan Pan [12] allows the user to specify how images vary over position and time. Images can be transformed, stretched, interpolated or modified in almost any way possible, since a whole programming language is available to the user.

Images are simply represented as functions from time and position to color. The expression for an image is compiled down to optimized C or machine code which is then used to render each frame. The time is increased in discrete steps and interval analysis (such as done in Fran) is performed.

Pan does not have any notion of events but allows the user to provide externally modifyable variables which can be integrated into the expressions, such as the amount of rotation that is to be applied to an image, etc. Also, as with Fran, there is only one time defined for the whole system.

Vertigo Vertigo [13] is essentially Pan for GPUs, although the compiler does more on optimization and is specifically targetted towards GPUs. It produces C#-code and DirectX-GPU assembly. Worth noting is that it handles both position and color, so that one both make complex geometry transformations and do custom lighting calculations.

Using Vertigo to compile our GPU-expressions directly is an option that we would like to have explored. It is not suitable for CPU-compilation as it is strictly functional (side-effect free). Otherwise, it shares much of the same pro:s and con:s (with respect to particle systems) with its predecessors, Pan and Fran.

Lava Lava [14] is a tool for describing and verifying hardware. The circuit description can be compiled to VHDL for construction of hardware.

The interesting technique used here that applies to our compiler is the one of detecting cyclic structures. Haskell does not normally allow this, which makes it difficult to express such constructs in an intuitive manner. How this is used syntactically is described in section 3.1.5 and how the compiler resolves this for our target is discussed in section 4.3.3.

Chapter 6

Conclusions

The traditional way of developing particle systems for games written in C++ (or similar languages) usually consists of encapsulating different behaviours and visualizations in different classes, and using parameters or inheritance to alter properties of these systems. This can lead to inefficiencies (in the case of an overly general particle class and a user only wanting simple properties) or low reuse (systems with similiar behaviour often ends up as products of copy-paste coding).

To remedy this, we have developed a language specifically targeted for particle systems and an accompanying compiler which takes system descriptions and outputs them in the form of a complete C++-class.

6.1 Language

We have described our language in the form of a tutorial and given formal semantics on the less obvious subset of the language. The language, like many declarative languages, focuses on describing *what* should happen, not *how* it should happen. This allows for remarkably concise descriptions of systems and could be further improved once a library of common behaviours and concepts is developed.

After a few iterations over the design, we ended up with a declarative language similar to those of Fran[11] or Pan[12], which all differ rather much from the usual imperative programming approach. At this stage, it became obvious that it was possible to allow the user to build hierarchies of systems, emitters and particles, something that is rarely seen in real-time particle systems.

6.2 Embedding in Haskell

Our language is embedded in Haskell, which provides us with indent-dependent parsing, type-inference and first-order functions, among many other powerful features. (See [3] for a complete description of the Haskell language.) Since our language is defined as a set of Haskell objects (functions), it has been easy to update, change and extend the syntax during development.

The use of Haskell means that we get some syntactic overhead that one might have been able to avoid with a proprietary language, but this is mostly

minor issues. More major issues are those of referential tranparency, which we did not anticipate. The ability to define and resolving cyclic expressions is nice, but the fact that expressions defined using let completely lacks any context makes it necessary to ensure that each expression only depends on its input (as true functions do). If one comes from a background in imperative languages, one expects statements and expressions to depend on where they were written. This is to be expected, since natural languages also has this property, so humans are prepared to deal with this.

However, this has forced us to carefully consider what information is necessary to evaluate certain expressions. This has led to the current language having the good fundamentals on which to do further work.

6.3 Compiler

The compiler is capable of taking a description and compile it to a C++-class that visualizes the particle system as having the expected properties. It deals with referential transparency and both detects and resolves cyclic expressions where possible and ensures that expressions are evaluated in the correct context. The compiler currently handles integration and derivation numerically only and is thus able to convert a mathematical expression to an iterative state-based computation, which we can evaluate in our target language. It does local optimization based on simple algebraic rewrite rules and also limited global optimization. The backend outputs the system as human readable C++ code.

As has been proven in many previous projects, Haskell is well suited for compilation tasks, and did not fail here either. However, the choice of having a functional language as a source language made the compiler rather hard to write, as the semantic gap is bigger compared with to the imperative target language.

Chapter 7

Future Work

7.1 Language

The language is rich enough to describe most particle systems whose basic primitive is a point. Expanding to allow different primitives is an easy task and does not affect the language much. We will now look at some of the parts of our language that does need extension.

Tuples Our language does not support tuples as a native data type, something that is necessary in order to have a composite state. Composite state is required whenever one wishes to simultaneously update several values, e.g. when handling collisions both position and velocity needs to be changed for proper behaviour. However, this gives us some problems with integration, as we want to change the accumulated value. An example using such a language extension would perhaps look something like figure 7.1.

Monads The language might gain some clarity and become less error prone if the systems were written using monads. Adding such a layer on top of the existing primitive operations would probably be a rather minor effort, as the mechanisms are already there. An example might be something like figure 7.2.

If all our computations where in the monad, the context would determine which time would be used where and snapshots (see section 3.1.8) could be taken automatically, depending on what the user desires. To define cyclic equation a resursive monad [15] would probably have to be used.

Library We have only defined the basic building blocks that is necessary to produce particle systems. Many properties reappear in different systems and they should be factored out to a separate library. Also, the need for a comprehensive geometric and linear algebra package is apparent.

Particle dependencies The property of particles being independent in our system is not entirely correct, as singular particles can alter their behaviour depending on their parents, but particles of the same type cannot affect each other, i,e. it is currently not possible to simulate flocking or galaxies.

```
obj $ \time ->
 state (\(pos,vel) ->
  let colTest p v = ... -- determine collision time and normal
      (hasCol,colT,colNormal) = colTest pos vel
      dt = time - prev time
      dt1 = colT - prev time
      dt2 = colT - time
      (p1,v1) = simulate dt1 pos vel
      vc      = collide v1 colNormal -- new velocity
      (p2,v2) = simulate dt2 p1 vc
      (p',v') = simulate dt pos vel
      simulate dt p v = p .+ v .* dt
  in ifE hasCol (p2,v2) (p',v')
 ) (initpos, initvel)
```

Figure 7.1: Correct collision handling using tuple in the state

```
mysystem = do
  ivel <- rand
  pos <- origin .+ integrate ivel
  emit 100 (part pos)

part ipos = do
  pos <- ipos + integrate vel
  point pos white 2
  kill (time >* 3)
```

Figure 7.2: Using a monadic approach

In order for this to work, we need to be able to iterate over all *other* particles of the same kind and accumulate an expression that we calculate once for each particle. An example with particles that gravitate toward each other is given in figure 7.3.

If the language is extended in this direction and add a feedback mechanism to allow an application to inspect values in the particles, it should be possible to apply it to scientific simulations. Coupled with execution on an efficient, parallelized graphics card, this seems like a promising area for future research.

7.2 Compiler

Currently, the compiler does not produce code whose efficiency is even close to that of a human programmer. The reason for this is mainly that the language design took more effort than expected, and consequently the time left to implement optimizations and other features in the compiler was not enough. However, many of the planned items (listed below) do not relate to our language as such, and could rather be implemented in any particle system.

```
foldOther :: Exp a -> Exp b -> (\Exp a -> Exp b -> Exp b) -> Exp b

myparticle =
   let pos = ipos + integrate vel
       vel = ivel + integrate acc
       acc = foldOther pos 0
              (\opos sum -> sum .+ 1 ./ lengthSq (pos-opos))
   in point pos white
```

Figure 7.3: Gravity example where particles depend on each other

Reference tracking Currently, child objects with references to their parent are not tracked, so the output is incorrect if a parent dies and some of its children still references it. This could easily be implemented with a smart pointer, using either reference counting or some garbage collection scheme.

Optimization Adding symbolic integration and adapting global optimization to test for alpha-equivalence[1] should do a lot for the optimization, reducing the calculations of typical systems to about 20-40% of their unoptimized counterparts, which would be well within the realm of human-produced code.

Adding a symbolic ODE solver that would be capable of resolving most of the differential equaitions that occur would also improve performance a lot, since each particle would only depend on its age and intial values. This would reduce storage and memory bandwidth and also be ideal for GPU-execution.

GPU computation In the outset of writing the compiler, we wanted to move stateless expressions to the GPU, in order to free the CPU for other work. We were not able to complete that part, but our compiler already separates stateful and stateless expressions. The remaining task is to output code in some shader-language and ensure that data is fed correctly to the graphics card.

It is also desirable to move expressions containing state to the GPU as well, as it is possible that the CPU has to do a major part of the calculations for some particle systems. Clearly, we would like the GPU to do as much as possible.

The hardware used for development in our case (NVIDIA GeForce4) actually has the ability to reroute output from the rasterizer back into the geometry stage, and thus allowing on-board per-vertex state, which is what we want. This is achieved by storing our object data as pixels in a texture and using that to produce a new texture, whcih then is fed back as vertex data. This is achieved by the `NV_pixel_data_range`-extension [17] in OpenGL. However, the rasterizer only outputs 8-bit integer precision per component and a maximum of four components (red, green, blue, alpha) as output, which obviously is inadequate both in precision and quantity. It also lacks any form of advanced fragment[2]-processing capabilities, which means that the hardware simply lacks the semantics necessary to evaluate most expressions.

[1] Two expressions are said to be alpha equivalent if they are the equal except that their bound variables may have different names.

[2] Normally equivalent to a pixel. The difference between fragments and pixels appear when doing anti-aliasing.

Newer hardware that support floating-point precision in the rasterizer stage, such as NVIDIA's GeForceFX or similar cards, makes this technique much more appealing. These cards also have programmable fragment processing, similar to the vertex processing found in our card. In fact, this technique has already been used for cloth simulation [16].

SIMD Since we already deal with the vectorized execution environment of GPUs, we could just as well use some of the SIMD-instructions available on todays modern CPUs and further enhance the performance and applicability of our system. Since there are certain requirements on how the data is laid out in memory for these instructions to be efficient, using the compiler to automate such task would be ideal.

Scene-graph inclusion In order to include the systems in a scene graph, there will be a need for some communication between the framework and our systems, to allow for updates, renderings, viewport/occlusion culling and the access of external variables. Appendix C completes the tutorial in the language section with the basic concepts for insertion into in external frameworks, and thus gives an outline of what to expect from future versions of this system.

Application output Currently, the output is a tighly encapsulated C++ class. One might envision the output of a complete application in order to test and modify the esystem before it is tested in a larger context. Such an application should provide GUI-controls to manipulate different parameters in the system, just as Pan [12] and Vertigo [13] does. Another concievable target is a screen-saver application, which would simply initialize the particle system and let it run forever.

Bibliography

[1] *Reaper*, available at `http://reaper3d.sf.net`, April 2003.

[2] NVIDIA. *C for graphics*, Compilers, tutorial and other information available at `http://www.cgshaders.org`, March 2003.

[3] Simon Peyton Jones (editor). *Definition of Haskell and the Standard Libraries*, available from `http://www.haskell.org/definition/`, April 2003

[4] Pete Isensee. *Fast Math using Template Metaprogramming*, in Game Programming Gems, Charles River Media, ISBN 1-584450-049-2.

[5] Paul Hudak. *The Haskore Computer Music System*, available at `http://haskell.cs.yale.edu/haskore`, May 2003.

[6] Yale Haskell Group, *Yampa: Functional Reactive Programming with Arrows*, available at `http://haskell.cs.yale.edu/yampa`, May 2003.

[7] *The GNU Compiler Collection*, available at `http://gcc.gnu.org/`, May 2003.

[8] John vad der Burg. *Building an Advanced Particle System*, Gamasutra, available from `http://www.gamasutra.com/features/20000623/`, April 2003.

[9] Jeff Molofee. *Particle Engine Using Triangle Strips*, NeHe Productions, available from `http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=19`, April 2003.

[10] David K. McAllister. *The Design of an API for Particle Systems*, Department of Computer Science, University of North Carolina at Chapel Hill. 2000.

[11] Conal Elliott and Paul Hudak. *Functional Reactive Animation*, in the proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97).

[12] Conal Elliott, Sigbjorn Finne, Oege de Moor. *Compiling Embedded Languages*, Journal of Functional Programming, 13(2), 2003.

[13] Conal Elliott. *Vertigo: a high-level languages and compiler for graphics processors*, available from `http://www.conal.net/Vertigo`, March 2003.

[14] Koen Claessen. *Embedded Languages For Hardware Description And Verification*, PhD thesis, Department of Computing Science, Chalmers University of Technology, 2001.

[15] Levent Erkők, John Launchbury. *A recursive* `do` *for Haskell. Design and Implementation*, in the proceedings of Haskell Workshop'02, pp. 29-37, 2002.

[16] Simon Green. *Stupid OpenGL Shader tricks*, NVIDIA, available from `http://www.opengl.org/developers/code/gdc2003/`, March 2003.

[17] Matt Craighead. *NV_pixel_data_range*, NVIDIA, available from `http://oss.sgi.com/projects/ogl-sample/registry/NV/pixel_data_range.txt`, March 2003.

[18] *ARB_vertex_buffer_object*, OpenGL ARB, available from `http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_buffer_object.txt`, April 2003

Appendix A

Typing implementation excerpts

A.1 Statically typed layer

```
newtype Exp a = E ERef.ERef deriving (Eq)

data Float3 = F3 deriving (Eq, Show)
data Float4 = F4 deriving (Eq, Show)

type Cmd     = Exp ()
type Boolean = Exp Bool
type Scalar  = Exp Float
type Vector  = Exp Float3
type Color   = Exp Float4

type Point   = Vector

...

(ERef.==*),(ERef./=*) :: ERef.ERef -> ERef.ERef -> ERef.ERef

(==*), (/=*) :: Scalar -> Scalar -> Boolean
(==*) = lift2 (ERef.==*)
(/=*) = lift2 (ERef./=*)

...

class NumE a where

instance NumE Float where ...
instance NumE Float3 where ...
instance NumE Float4 where ..
instance NumE a => Num (Exp a) where ...
instance NumE a => Fractional (Exp a) where ...
```

```
instance NumE a => Floating (Exp a) where ...

instance Ord Scalar where ...
instance Enum Scalar where ...
```

A.2 Random expressions

```
class Fractional a => Rand a where
  rand :: (a -> Exp b) -> Exp b
  nrand :: (a -> Exp b) -> Exp b
  nrand f = rand (\v -> f (v-0.5))

instance Rand Scalar where
  rand f = E (ERef.rand (\v -> unE $ f $ E v))

instance (Rand a,Rand b,Fractional (a,b)) => Rand (a,b) where
  rand f = rand $ \x -> rand $ \y -> f (x,y)

instance (Rand a,Rand b,Rand c) => Rand (a,b,c) where
  rand f = rand $ \x -> rand $ \y -> rand $ \z -> f (x,y,z)

instance (Rand a,Rand b,Rand c,Rand d) => Rand (a,b,c,d) where
  rand f = rand $ \x -> rand $ \y -> rand $ \z -> rand $ \w ->
           f (x,y,z,w)

instance Rand Vector where
  rand f = rand (\e@(x,y,z) -> f (vec3 e))

instance Rand Color where
  rand f = rand (\e@(x,y,z,w) -> f (vec4 e))
```

A.3 Numerical operations

```
class (NumE a, NumE b, NumE c) => NOp a b c | a b -> c  where
  (.*),(.+),(.-), (./) :: (NOp a b c) =>
       (Exp a) -> (Exp b) -> (Exp c)
  (.*) = op (*)
  (.-) = op (-)
  (.+) = op (+)
  (./) = op (/)
  op :: (Exp c -> Exp c -> Exp c) -> Exp a -> Exp b -> Exp c
  op f a b = uncurry f $ conv a b
  conv :: Exp a -> Exp b -> (Exp c,Exp c)

instance NOp Float Float Float where
  conv a b = (a,b)
```

```
instance NOp Float Float3 Float3 where
  conv a b = (vec3f a,b)

instance NOp Float3 Float Float3 where
  conv a b = (a, vec3f b)

instance NOp Float3 Float3 Float3 where
  conv a b = (a,b)

instance NOp Float Float4 Float4 where
  conv a b = (vec4f a,b)

instance NOp Float4 Float Float4 where
  conv a b = (a, vec4f b)

instance NOp Float4 Float4 Float4 where
  conv a b = (a,b)

vec3f :: Scalar -> Vector
vec4f :: Scalar -> Color
```

Appendix B

Abstract syntax type

```
-- variable scoping
data Scope =
  Temp |
  Local |
  Global
  deriving (Eq, Ord, Show)

-- expression types
data Type =
  BoolT |
  FloatT |
  Vector3T |
  Vector4T |
  VoidT |
  CmdT
  deriving (Eq, Ord, Show)

-- variable identifiers
type Id = String

-- function expressions
data (Indir e) => Func e =
  RandVar  (e -> e)        | -- (r -> e)
  StateVar (e -> (e,e)) e  | -- (s -> (val,s')) s_0
  TimeVar  (e -> e)          -- (t -> object)

-- operations
data Op =
  -- commands
  Object |         -- [timevar,init,cmds] => Create an object
  Trigged String | -- Trigged id [cmd]    => Trig command externally

  Point |          -- [pos,col,size]    => Display point
  Emit |           -- [rate,num,cmd]    => Executes cmd at intervals
```

```
  Kill |           -- [bool]            => Kill object when true

  -- behaviours
  StateE |         -- [var,out,update,init] =>
  LetRec |         -- [var,expr] => let var = expr in var (recursive let)
  Snapshot |
  Parent |         -- reference the parent's variable

  -- assignment exprs
  New |         -- [var,expr]   => As assign, but used for declaration
  Assign |      -- [var,expr]   => var := expr
  Seq |         --              => Evaluate expressions sequentially

  -- expressions
  Let |         -- [var,expr,expr2] => let var = expr in expr2
  If |          -- [a,b,c] => If a then b else c
  Vec |         -- [..] => Build vector expression from several scalars
  Idx |         -- [i,v] => v_i (v = vec expr, i index)

  -- mathematical ops
  Add | Mul | Rcp |
  Sqrt | Abs | Exp | Log |
  Integrate | Derive |
  -- trig
  Sin | Cos |
  -- compare
  Eq | Ne | Lt | Gt | Le | Ge |
  Min | Max |
  -- boolean
  And | Or | Xor | Not

  deriving (Eq, Ord, Show)

-- indirection class (defines wrapper type)
class (Eq e, Show e) => Indir e where
  unwrap :: e -> E e
  wrap   :: E e -> e

-- expressions
data (Indir e) => E e =
  A Op [e] |
  F (Func e) |
  Var Scope Type Id |
  LitB Bool |
  LitF Double |
  RandFloat |
  Void
```

Appendix C

Working with an external framework

This appendix gives details about a possible future extension to our language, namely how it should interact with an external framework, such as a scene graph or a game engine.

C.1 Running several systems in one class

Our tutorial, chapter 2, held most of the things you need to do particle systems, but in order for them to be efficient, you want bulk processing/drawing and heavy sharing. It is therefore possible to avoid creating the system at class instantiation time and rather call a function in the class each time you want something to happen. Say that we want to create explosions at random points in space, but since the particles all look the same and handle the same way, we should keep them within one class. Let's see how we can do that.

```
trigged :: String -> Command -> Command

particle p =
 obj $ \t ->
  nrand $ \v ->
   let pos = snapshot p + integrate t vel
       vel = snapshot v + integrate t acc
       acc = vec3(0,-9.82,0) - 0.5 .* vel
   in point pos red 1 <+> kill (t >* 10)

emitter p =
 obj $ \t ->
  emit 1 100 (particle p) <+> kill true

mysystem =
 let init_pos = extPoint "center_pos"
 in  trigged "explosion" (emitter init_pos)
```

Since we are taking a snapshot of the external variable, we can give it as a one-time argument when creating the system. An example of how things would be if we would like to read continously from an external variable that is specific for each trigged instance is provided below. This will give the following interface class:

```
class explosions
{
public:
  explosions();

  void explosion(const vec3 &center_pos);
  ...
};
```

Note that there is nothing stopping us from having several different particles / emitters in one class. In fact, taken to the extereme, all particles in the world can be handled by one class. This would certainly give the compiler the best opportunity to optimize things as it knows about all particles.

C.2 Communication

The one-way communication that we had in the previous example is not always desirable, depending on the framework. If some advanced occlusion culling is used, the framework can probably make intelligent decisions as as to when the system is visible It could also be that we want to track the external variable continously, which we can do by storing the reference for the duration of the system. Now it is possible that the variable that we are tracking may become invalid. The framework must have then have some way of letting us know that the reference is no longer valid.

This means that we need a way for the framework to:

1. Query for a bounding volume
2. Updating the system (thus updating the bounding volume)
3. Drawing the system
4. "Freezing" external variables so that they will no longer be accessed.
5. Killing the system completely.
6. Be informed when the system is expired so that visibility testing is no longer necessary

4 and 5 adress the same problem, but the latter also provides a way to do a clean shutdown. We solve these problems by introducing two interfaces. One is a callback that is used to notify the framework when the system has expired, the other is a tracker that can be used to query for bounding volumes and intiate different actions on the system. The callback and tracker classes look lite this:

```
class callback {
public:
  virtual void operator()() = 0;
};

class tracker {
public:
  virtual Sphere& bounding_volume() = 0;
  virtual void update(float time) = 0;
  virtual void draw() = 0;
  virtual void kill() = 0;
  virtual void snapshot() = 0;
};

class AllMyParticles {
public:
  tracker * explosion(const Point &center_pos,
                      callback* expired = 0);
...
}
```

This way, the scene graph system will have enough information to decide wether the system should be drawn or not, and the class will know when to update and draw the particles.

There are a few rules to this protocol. The class guarantees that after sending notifying through the callback, or recieving a call to kill or snapshot, it will not access any references given for this system. In return, the framework must guarantee not to call any function in the tracker after any such communication has occured.

In order to guarantee that there will be a valid reference (it is easy, but seriously wrong, to pass a local variable when instantiating the system) one could add it to the tracker instead:

```
class explosion_tracker : public tracker
{
public:
  Point center_pos;
  ~explosion_tracker() { snapshot(); } // or kill()
};
```

This would make things entirely safe, but if the system dies calling snapshot would be an error, as there is no one in the other end to recieve that call. This could be solved by letting the system store orphaned trackers until they are snapshotted or killed, whether or no there exist any particles or emitters to be tracked.

C.3 Advanced particle trickery

Assume that we are making a spooky adventure game, where our hero (or heroine) is wading through a swamp in the middle of the night with only a

lantern as light source. Naturally, all sorts of moths and mosquitos, whichof there exists plenty, since this is a swamp after all, fly towards the lamp. The game engine spawns swarms of flies that moves to the lantern, circle around it for some time but then depart back to their original position and die. Occasionally, our main character throws fireballs at these swarms, as he or she has been in the swamp since noon and are quite fed up with their abominable biting, buzzing and generally annoying attitude. It may also happen that the lantern goes out (it was gotten rather cheap from a suspicious looking old witch living at the edge of the swamp), in which case our poor adventurer soon ends up being consumed, lantern and all, by the ancient, unfathomable horror that lurks beneath the dark, oily surface. The flies however hangs around for a while, simply because they are too stupid to do anything else.

So, first we define our particle systems. It will consist of three types of particles:

- The mosquitos
- The fireball
- An explosion

Their definition would be something like:

```
mosquito :: Point -> Point -> Command
mosquito lantern_pos init_pos  =
 obj $ \t ->
 rand $ \x::Vector ->
  let dir = lantern_pos - pos
      time_to = 30
      pos = snapshot init_pos .+ integrate t vel
      vel = x .+ t integrate acc
      acc = clamp $ ifE (time <* time_to) dir (-dir)
      near_lantern = (length dir <* 10) &&* (time <* time_to)
      movement = ifE near_lantern
                     (lantern_pos + rotatey (1,0,0) t)
                     pos
  in point movement grey 1 <+> kill (t >* 50)

firespark :: Vector -> Point -> Command
firespark init_dir init_pos =
 obj $ \t ->
 rand $ \x ->
  let pos = snapshot init_pos + integrate t (snapshot init_dir)
      col = lerp (clamp (x+sin(time*10))) red yellow
  in point pos col 2 (t >* 5)

circle-emit :: Point -> Scalar -> Scalar -> (Point -> Scalar) ->
               Command
circle-emit init_pos r qty obj =
 obj $ \t ->
 nrand $ \x ->
```

```
  let pos = snapshot $ r*(normalize x)+swarm_pos
  in emit 1 qty (obj pos) <+> kill true

swarm =
  let pos = extPoint "start_pos"
      lantern = extPoint "lantern"
      emit = circle-emit pos 2 50 (mosquito lantern)
  in trigged "swarm" emit

fireball =
  let pos = extPoint "start_pos"
      dir = extVector "direction"
      emit = circle-emit pos 0.25 100 (firespark dir)
  in trigged "fireball" emit

system = swarm <+> fireball <+> explosion
```

Our mosquitos should move towards the lantern and circle it for the first 30 seconds, then move away and die after 50 seconds. They have a bit of randomness to their velocity, which should give them some of that erratic insectoid flight pattern. Grey in color and a size of 1.

The fireballs are much simpler, and just fly along a given direction. They do vary a bit with their color and size.

We use the same emitter for both systems, which will calculate points on the surface of a sphere with given center and radius and apply these points to the objects that it emits.

Finally, we just put these together so that we can export them into the same class. So, compiling our system would give us the following interface:

```
class SwampParticles
{
public:
  SwampParticles();

  tracker* swarm(const Point &start_pos,
                 const Point &lantern,
                 callback *expired = 0);

  tracker* fireball(const Point &start_pos,
                    const Vector &init_dir,
                    callback *expired = 0);

  tracker* explosion(const Point &start_pos,
                     callback *expired = 0);

  ...
};
```

There, now we can spawn swarms of flies, they track the lantern. If the lantern dies we can snapshot the system so that they still flie towards the lantern.

Or, we could change the lantern position to the center of that swarm's bounding sphere, making the swarm gather on the spot for a while. Fireballs can be thrown, and with some collision checking from the game engine, an explosion can be triggered when a hit occurs, destroying the fireball but creating another system in its place.

C.4 Beyond particle systems

If the external variables represent key states, we can make a small game in this language. Take as an example the old arcade hit Moonlander, where the player was set to steer a rocket and try to land softly on a landing pad, avoiding surrounding mountains and conserving as much fuel as possible.

Since we can integrate values, we can keep track of fuel and calculate thrust effect. Checking if the position is near the landing pod and the angle of the ship is straight, we can decide wether a landing is succesful or not. The mountains can be generated with a few randomized sine-waves and the landing pod can be put at the minumum position. At a few of the maxima we can have lava or smoke coming out (no, the moon does not have active volcanoes, but games aren't meant to be realistic, they are meant to have fancy graphics :)