HSpark - Generating optimized C++ code for particle systems

Marcus Lindblom (d98macke@dtek.chalmers.se)

12th March 2002

1 Problem description

A *particle system* is defined as a set of *particles,* each having a number of *properties* such as position, velocity, size, etc. Particles can be created either simultaneously, continually during the simulation of the system or a combination of these. Each particle is given initial values upon instantiation and then the properties are updated through evaluating an expression based on time, the previous property values and perhaps some values from an "environment". The system typically expires after a while, where the actual point of time can depend on a number of reasons (a certain time has passed, all particles are dead, etc.)

Typically, particle systems are used heavily in graphically intensive applications such as computer and console games. Writing code for simulation of these systems is (as always) a compromise between code clarity and speed of execution. An application typically also has a number of similar, but yet sufficiently differencing, particle systems. Thus reuse of code between systems is also desirable.

Such applications are as of today mostly written in C++, which does, through operator overloading, allow vector expressions that closely map to the mathematical formula describing the particles properties over time. The semantics of C++ does however force the creation of temporary objects during evaluation, thus degrading performance.

2 Possible solutions

2.1 In-lining the code and deferring to the C++ compiler

By exposing the functions applicable to vector expressions, the C++ compiler should be able to perform analysis and vectorize the expressions. To which degree this can be done is not certain, as the C++ code is not a strict mathematical expression. Some compilers (i.e. VectorC) has extended the C syntax to perform vectorization on expressions (for use with SIMD capable instruction sets, such as MMX, SSE, 3DNow), but those are only available at a high cost and the issues of non-portability may not make such use feasible for all developers.

2.2 Rewriting the C++ code

By doing some of the C++ compiler's work described in the previous subsection by hand solves the performance the problem, but hand-optimization is error-prone and code reuse is almost impossible. Maintaining heavy optimized code is also very time consuming and common subexpressions in such code may not be a trivial task for the C++ compiler.

2.3 Generating C++ code from symbolic expressions

By gathering the mathematical expressions and performing optimization in software, essentially writing a code-generator, solves all these problems.

- Code can be kept at a high level, thus being easy to maintain and allows reuse.
- Optimization and expansion of vector expressions to scalar form is done by the system, thus minimizing errors introduces by hand-made optimization.

2.3.1 Writing smart C++ code

By writing a lazy-evaluated vector expressions in C++, more optimization can be done. However, writing such a library is not trivial, and due to C++'s rather large syntactic overhead when creating generic types, it is not a pleasant task.

We also defer to the compiler in doing global optimization on our final expression, which may not produce the desired result.

2.3.2 Using the Pan c-code generator

The Pan system was created to produce efficient C++ code for describing two dimensional images, changing over time and other user-modifiable properties, in the form of a Haskell-embedded language. It has a lot of nice features and has also been tested quite a lot. Using it for the purpose of particle-system code does sound promising, but there are a number of problems with this approach.

- The expression data type is ill-suited to particles, which have a number of different properties, some scalar and others vectorized (typically three-dimensional values, for position and velocity). It knows only of booleans, floats and arrays.
- When representing properties on array form forced the Haskell code drifted too far from the desired high-level of mathematics.
- Also, representing the properties of a particle as an array, the expressions produced were clearly not optimized enough.
- Extending the expression data type is difficult, due to the immense number of optimization in the constructors.
- The c-code generator is not clearly separated from the context-inclusion, thus it is difficult to extract the essential functionality.

3 Solution

The solution was to produce a code-generator more suited to particle systems. This means duplicating a lot of the work done with the Pan-system, but it has also given much insight in the inner workings of Pan and the general problem of optimizing symbolic expressions and exporting them as C code.

The expression is built in vector form through symbolic evaluation. Local optimization based on algebraic rules is applied, then the generator applies global optimization, through common subexpression elimination (CSE). Finally, the expression is converted to C++-code and put into some context, such as functions for creating or simulating a series of particles.

3.1 The expression data type

We represent our expressions internally as a dynamically typed expression, containing all possible operations and values (boolean, float, 3-vector, 4-vector) that we want to use.

```
type Id = String

data Exp =
  LitBool Bool |
  LitFloat Float |
  LitVec3 Vector3 |
  LitVec4 Vector4 |

  PropBool Id |  -- property value

  EnvBool Id | -- environment value

  IfE Exp Exp Exp |
  Variable Id Type |
  Let Id Exp Exp |

  Add Exp Exp |
  Sqrt Exp |
  ...
```

The expression is then built from operators such as (+, -, ..). As in Pan, If is written IfE and boolean operations (<,>) are postfixed with a *, due to Haskell's syntax rules. These operators do some simple verification on the types involved, but there is currently no static type checking.

The property values refer to the individual particle being calculated. The environment contains time, time-delta and possibly other things affecting the particles collectively.

3.2 Local optimization

Local optimization works only on scalar expressions, i.e. those of float or boolean types, thus we need to expand the expression if it is of vector type.

3.2.1 Expand

Expanding a vector expression to it's scalar counterparts representing the calculation of each component is a straight-forward bottom-up operation on the expression tree. At the leaves of our expression we will find vectors, which we expand to a list of scalars.

```
expand :: Exp -> [Exp]
expand (LitVec3 (a,b,c)) = map LitFloat [a,b,c]
expand (PropVec3 id) = map (\x->PropFloat (id++"."++[x])) "xyz"
```

For our combined expressions (Add, Mul, ...), we recursively call expand on all arguments and map the constructor over the results:

```
expand (Add a b) = emap2 Add a b
expand (Mul a b) = emap2 Mul a b

emap2 f x y = zipWith f (expand x) (expand y)
```

3.2.2 Constant folding

Constant folding is trivial, if the expression is bound on literal values, we simply compute the result and replace the original expression:

```
opt :: Exp -> Exp
opt (Mul (LitFloat x) (LitFloat y)) = LitFloat (x*y)
opt (Sin (LitFloat f))              = LitFloat (sin f)
```

3.2.3 If-floating

If-floating attempts to move If-expressions as high as possible, thus putting all the computation in the leaves, hoping that there can be more optimization done there. It leads to some code replication, but does not add to the total work to be done.

```
opt (Mul (IfE x y z) e) = opt (IfE x (Mul y e) (Mul z e))
opt (Mul e (IfE x y z)) = opt (IfE x (Mul e y) (Mul e z))
```

3.2.4 Algebraic optimization

By adhering to some simple algebraic rules, we can do further optimization:

```
opt (Mul (LitFloat 1) y)   = opt y
opt (Div x y) | x == y    = LitFloat 1
              | otherwise = Div (opt x) (opt y)
```

3.3 Gathering expressions

A single particle property calculation can be time-relative, or absolute. Either we describe the derivative (with respect to time) or the absolute value of the function. Normally, there are a number of expressions that need to be summed up to produce the final value. One can f.ex. imagine gravity and air resistance affecting the velocity of a particle. We also have a third option for a calculation, namely 'Simple'. This is used when we just want to calculate a simple expression, not some particle property. The code generator uses this to determine what to do with the final value of an expression.

The "gathering" of expressions and binding them to a particle value is done by a function *genCalc*, which produces a calculation. These calculations are then fed to the CSE system for global optimization.

```
data CTime = Absolute | TimeRelative | Simple
             deriving (Eq, Show)

data Calc = C { dest :: Id,
                ctype :: Type,
                ctime :: CTime,
                exprs :: [Exp] }
            deriving (Show)

genCalc :: Id -> CTime -> [Exp] -> IO Calc
```

3.4 Global optimization

3.4.1 Hashing

In order to do CSE, we compute a hash value for each node in the expression tree. This value, combined with a pointer to the expression and its type is returned as a list of lookups, tog. We also transform the expression into a *hashed expression,* which is a duplication of our Exp-type mentioned earlier. It adds a hash-code of the expression itself on all recursive constructors, so that we avoid recomputing the hash value when doing CSE.

```
type HashCode = Int
type Ptr = StablePtr HashExp
data LookUp = LU Ptr HashCode Type
data HashExp =
  HLitFloat Float |
  HMul HashExp HashExp HashCode |

hashExp :: Exp -> IO (HashExp, [LookUp])
```

3.4.2 CSE

The CSE system takes a list of expressions and performs CSE on all these simultaneously. This allows calculation sharing between f.ex. velocity and size expressions, should they have something in common. By regarding all lookups produced by the hashing system and filtering through those who occur more than once, we can quickly determine which expressions are suitable to precalculation.

```
cse :: [Exp] -> IO [Exp]
```

3.5 C++ code generation

Using the *John Hughes's and Simon Peyton Jones's Pretty Printer Combinators* library, improved by *Conal Elliot* and some extra utilities from the Pan system, the symbolic expressions are converted to C++ code. This is fairly simple, especially when using the Pretty Printer library to make indentations, etc.

3.6 Context

After having the expressions generated into c-code, we put this into some context. The current contexts have been produced:

- The simulation of all particles, where the code computing one particle is nested inside a loop statement.
- The creation of particles into a STL::vector<>, the number of particles computed as an expression (can thus depend on environment) and each particle given initial values.
- The creation of particles, similar to the above, but their initial values can depend on a property that is iterated over an interval.

4 Improvements

There are many improvements possible for this library. Some are needed in order to make it behave properly in all cases, others lie in the support framework which makes the system actually useful in a working environment.

4.1 Static typing

As described in *Compiling Embedded Languages,* static typing can be added as a layer on top of the dynamically typed expression. It is not difficult, but there is a lot of typing involved.

4.2 Local optimization

The system does not take advantage of the fact that additions, multiplications and a few other calculations can be reordered. This hinders some optimization. Allowing this however produces some problems; f.ex. in the expression (a*b*c + a*c + b*c), a decision has to be made whether a*b or b*c should be optimized out of a*b*c. Currently, the expression a*b*c is treated as (a*b)*c, where only a*b can be optimized away.

4.3 CSE

4.3.1 Hashing

Currently, hashing is done very naively (hashing of the string representing the expression) which results in an exponential time algorithm. A better hashing algorithm would do a bottom-up computation so that each node is only looked upon once, and the hash values of the children are used in the computation of the parent's hash value.

4.3.2 Algorithmic complexity

The CSE-system is not efficient, calling itself recursively to do the same work several times over. This has not proven a problem yet, but it will probably run into problems with deeply nested expressions that has many CSE:s.

4.3.3 CSE:s in If statements

These are currently not handled in an efficient manner, i.e. they are moved out always and are thus always calculated, even if the result isn't used.

4.4 C++ code generation

The current implementation of the code generator adds unnecessary parenthesis around some expressions. This is most likely due to bad management of operator precedences.

4.5 Context

The context system is far from complete. Some improvements are very simple, others require considerable work.

- More contexts should be added, especially emitter functions that work continuously.
- Build a system that handles death of particles, choosing proper data structure for all particles and manages it during simulation.
- Analysis on the properties set/used by the calculations, for error-checking and automatic definition of the particle object/struct.
- Add an option to produce straight C functions that works on arrays (currently using STL-iterators to define an interval).

Currently, the generation of a context is a bit 'clumsy'. Moving the computation to a 'list-like' monad would not be difficult and would also produce really neat code. What one would ideally want is something like this, to describe a particle system:

```
simfunc =
  do velCR (do gravity 0.2  -- particle mass 0.2 kg
               airres 0.05) -- air resistance at 5 %
     sizeC (do 0.5                -- base radius .5 m
               0.3 * sin(time*3)) -- radius will vary +/- .3 m
     deadWhen (lifetime > 2)

initfunc =
  do posC (do origo)
     velC (do vec3(0,10,0))
    sizeC (do 2)
```

```
particleSystem =
  do classname “ShiningSparks”
     creation (emit 200 initfunc)
     emit (timeInterval 2 initfunc)
     simulate simfunc
     display (point red)
     deadWhen (time > 20)
```

Also, some standard functions for displaying the particles would be necessary, which is commonly done as points, lines, billboards or as a triangle set. This allows the code-generator to produce a complete c++-class, upon which it also should invoke the C++-compiler to produce executable code.

5 Language discussion

5.1 Recursive data types

Recursive data types is used in the heart of the code generator, namely the dynamic expression type. This also allow heavy use of pattern matching throughout the code.

5.2 List operations & partial application

A lot of the CSE code is list operations. Haskell's powerful support for lists makes it easy to express any operation over lists, especially in combination with partial application.

5.3 Tuples & data types

There is a lot of gathering/scattering of data, i.e. putting things into tuples, removing things from tuples, etc. The same goes for some data types and their constructors. Due to the fact that it is so easy to build new types, marshaling between them is done often.

5.4 Reference types

The use of StablePtr requires the CSE computation to be in the IO-monad. The ST monad had probably been a better choice.

5.5 Dot syntax

One often wants to access individual components in a vector by indexing them with the name of the component (vel.x). In Haskell, this can only be written as (x vel), which is acceptable, but a bit awkward. However, the optimizing system allows one to use vector expressions all the way, since it can safely optimize away computations resulting in zero.