Procedural terrain generation

Willy Jacquet[1]*, Parra Yoan*[2]

[1]Author Affiliation
[2]Co-author Affiliation
author@university.edu, coauthor@company.com

Abstract

Symbolic computation is seen as a trade-off between expressiveness and performance. Metaprogramming can minimize that overhead and enable further optimization.

1. Introduction

In this paper we use procedural terrain generation as our primary application and thus tweaked the model for vector manipulation.

2. State of the art

3. Symbolic computation

Manipulation of *functions* instead of *values*. *Functions* are *first class citizens*.

The following model is designed to take advantage of *template metaprogramming* facilities.

> This document is thus extended with insight about making a *C++* implementation out of it.

3.1. Generic functions

A **reduction** predicate is introduced, noted $a \rightarrow b$ (a can be *reduced* to b). This predicate is *transitive*.

> A *function* can be represented as a *type* that holds no *non-static data member*. In particular, if two instances of functions share the same type then they refer to the same function.

A **tuple** is defined as a *sequence of objects* $(x_1, ..., x_n)$.

The **application** of a function f with $x_1, ..., x_n$ is noted $f(x_1, ..., x_n)$. Generally, the *application of a tuple* $(f_1, ..., f_n)$ is defined as:

$$(f_1, ..., f_n)(x_1, ..., x_n) \rightarrow (f_1(x_1, ..., x_n), ..., f_n(x_1, ..., x_n)) \tag{1}$$

If $f(x)$ cannot be further *reduced* ($\neg\exists a, f(x) \rightarrow a$) then for $y_1, ...y_n$:

$$f(x)(y_1, ...y_n) \rightarrow f(x(y_1, ...y_n)) \tag{2}$$

A **constant** c is defined as a function that returns itself:

$$c(x_1, ..., x_n) \rightarrow c \tag{3}$$

> A *compile-time constant* can be represented as:
>
> ```
> template<typename Type, Type Value>
> struct constant {};
> ```
>
> Special constants (0, 1, π, e, ...) can be introduced as specific symbols since they appear in many (number) sets.

```
struct zero {}; struct one {}; ...
```

Given $i \in \mathbb{N}^*$, the **i-th projection** is a function that returns its i-th parameter:

$$proj_i(x_1, ..., x_n) \rightarrow \begin{cases} x_i & \text{if } i \leq n \quad (4) \\ proj_{i-n} & \text{otherwise} \quad (5) \end{cases}$$

When the i-th projection is applied to the i-th parameter of each function, the notation is abbreviated:

$$f(p_0, p_1, ...) = f \tag{6}$$

The **arity** (function) can be defined as follows:

$$\begin{aligned} arity(c) &= 0 \\ arity(proj_i) &= i \\ arity(f(g_1, ..., g_n)) &= max\{arity(g_i)\}_{i \leq n} \end{aligned} \tag{7}$$

3.2. Arithmetic functions

Elementary arithmetic functions are introduced with their usual definition:

$$\begin{aligned} (f+g)(x) &\rightarrow f(x) + g(x) \\ (f-g)(x) &\rightarrow f(x) - g(x) \\ (f \times g)(x) &\rightarrow f(x) \times g(x) \\ (f/g)(x) &\rightarrow f(x)/g(x) \end{aligned} \tag{8}$$

Partial application is made possible by (5). For example:

$$\begin{aligned} (proj_0 + proj_1)(x) &\rightarrow proj_0(x) + proj_1(x) \\ &\rightarrow x + proj_0 \end{aligned} \tag{9}$$

Lazy evaluation is introduced by establishing:

$$\begin{aligned} 0 \times g &\rightarrow 0 \\ f \times 0 &\rightarrow 0 \\ 0/g &\rightarrow 0 \end{aligned} \tag{10}$$

> This can be implemented by providing *function overloads* having the first or second operand with the zero type.

The current model can be extended with any usual function. Any function that is not mentioned in this section is defined with its usual meaning.

3.3. Differential calculus

The **partial derivative** *with respect to* x is introduced for generic functions:

$$\frac{\partial}{\partial x}(c) \to 0 \text{ with } c \text{ a constant} \tag{11}$$

$$\frac{\partial}{\partial proj_j}(proj_i) \to \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \tag{12}$$

$$\frac{\partial}{\partial x}(f(y)) \to \frac{\partial}{\partial x}(y) \times \frac{\partial}{\partial x}(f)(y) \tag{13}$$

For arithmetic functions:

$$\frac{\partial}{\partial x}(f+g) \to \frac{\partial}{\partial x}(f) + \frac{\partial}{\partial x}(g) \tag{14}$$

$$\frac{\partial}{\partial x}(f-g) \to \frac{\partial}{\partial x}(f) - \frac{\partial}{\partial x}(g) \tag{15}$$

$$\frac{\partial}{\partial x}(f \times g) \to \frac{\partial}{\partial x}(f) \times g + f \times \frac{\partial}{\partial x}(g) \tag{16}$$

$$\frac{\partial}{\partial x}(f/g) \to \frac{\frac{\partial}{\partial x}(f) \times g - f \times \frac{\partial}{\partial x}(g)}{g \times g} \tag{17}$$

3.4. Computational redundancy

Suppose that, provided a function $f \times g$, you wish to compute its value *and* partial derivative for an argument x.

3.5. Parallelization

3.6. Compilation

3.7. Performance

4. Common functions

4.1. Interpolation

Provided an interpolant $t \in [0, 1]$. Polynomial interpolation with null derivatives is defined as follows:

- linea:r t
- cubic: $-2t^3 + 3t^2$
- quintic: $6t^5 - 15t^4 + 10t^3$

4.2. Noise functions

Coherent noise can be used to model many natural phenomena and is characterized as follow [1]:

- *Referential transparency* i.e. the same input produces the same output
- A small change in the input value will produce a small change in the output value.
- A large change in the input value will produce a random change in the output value

Value noise is achieved by interpolating random values on a regular grid along each axis until it is reduced to a single number.

Listing 1: *Pseudo code implementation of 2D value noise*

```
float noise(v:float[2]):
  i=floor(v)
  t=fract(v)
  return interp(
    interp(hash(i+[0,0]),hash(i+[0,1]),t[0]),
    interp(hash(i+[1,0]),hash(i+[1,1]),t[0]),
    t[1])
```

Figure 1: *2D value noise.*

With $interp$ being an interpolation function.

4.3. Worley noise

5. Terrain representation

A **terrain** can be defined as a C^2 function $\mathbb{R}^2 \to \mathbb{R}$. Therefore it is interpreted

5.1. Coloration

5.2. Rendering

5.3. Implicit representation

6. Results

7. Appendix A: Derived functions

Many usual functions can be defined in terms of those previously introduced. Doing so makes inferring most properties (derivative, domain, ...) automatically possible.

7.1. Generic functions

$$swap = (proj_1, proj_0)$$

7.2. Arithmetic functions

$$\begin{aligned} opposite &= 0 - proj_0 \\ inverse &= 1/proj_0 \end{aligned}$$

$$\begin{aligned} translation &= p1(p0 + p2) \\ scaling &= p1(p0 \times p2) \end{aligned}$$

7.3. Calculus functions

$$\begin{aligned} \partial(cos) &= -sin \\ \partial(sin) &= cos \end{aligned}$$

8. References

[1] [Online]. Available: http://libnoise.sourceforge.net/coherentnoise/index.html