韓國 컴퓨터情報學會 論文誌
第10卷 第3號, 2005. 7.

2005-10-3-1-3

영상처리 알고리즘 구현에서 소스코드 재사용을 위한 제너릭 프로그래밍 방법에 관한 연구

이 정 헌*, 이 준 형**

A study of generic programming method for source code reuse in image processing algorithm implementation

Jeong-Heon Lee *, June-Hyung Lee **

요 약

영상처리 알고리즘 연구에 있어서 가장 어려운 부분 중 하나가 기존 알고리즘과의 성능 비교이다. 그러므로 재사용이 가능한 코드의 형태로 알고리즘을 표현하고 구현하는 것이 영상처리 연구에서는 매우 중요하다. 기존의 영상처리 시스템들은 소스코드 측면에서의 재사용을 위한 모든 조건을 지원하지 못했다. 이러한 문제를 해결하기 위하여 본 연구에서는 제너릭(generic) 프로그래밍 방법을 영상처리 알고리즘구현에 적용함으로써 소스코드 측면에서의 재사용 문제를 해결하는 방법을 제안하였다. 제안한 방법은 다른 다양한 영상 형식이 적용되더라도 같은 알고리즘 구현이 가능하도록 영상처리 알고리즘들과 알고리즘 구현을 위한 기초적인 자료 구조를 연결하는 이차원 iterator를 정의하여 사용하는 방법이다. 그리고 제안 방법의 간결함과 효율성을 확인하기 위하여 몇 개의 예제와 기존의 영상처리 알고리즘 통합 개발 환경에 이식하여 기존의 방법과 비교하였다.

Abstract

The difficulties in implementing of image processing algorithms are a major reason for the lack of research into algorithm comparison. This fact makes an image processing research with difficult. We conclude that it is important to represent algorithms in form of reusable code. Since current image processing systems do not fulfill all requirements we must pose on reusable implementations, we propose to solve the reuse problem by applying generic programming. We define two dimensional iterators, which mediate between image processing algorithms and their underlying data structures, so that the same algorithm implementation can be applied to any number of different image formats. The elegance and efficiency of this approach is illustrated by a number of useful examples and demonstrated by porting in existing image processing algorithm IDE(Integrated Development Environment).

▸ Keyword : 재사용 코드(Reusable Code), 제네릭 프로그래밍(Generic Programming), 반복자(Iterator)

• 제1저자 : 이정헌, •• 교신자자 : 이준형
• 접수일 : 2005.04.01, 심사완료일 : 2005.05.22
* 미디어 코러스 단말 개발부 부장, ** 극동정보대학 컴퓨터정보과 조교수

Ⅰ. 서 론

영상을 이용하는 디지털 기기의 일반화로 영상처리 기술이 제품 경쟁력에 중요한 부분을 차지하게 되었으며, 신속하게 영상처리 기술을 개발해내는 것이 제품의 성패를 좌우하는 시대가 도래 하였다. 하지만 공통적으로 사용할 수 있는 일반화된 알고리즘이 존재하지 않는 영상처리 분야의 특성으로 인하여 대부분의 영상처리기술들은 응용분야마다 독자적으로 개발되고 개발비용이 또한 크게 증가하는 문제점이 있다. 이러한 문제점을 해결하기 위해선 한번 개발된 알고리즘들을 적용하고자 하는 환경에 맞추어 완벽히 재사용할 수 있는 방안이 필요하며, 더 나아가 기 개발된 알고리즘들을 축적하여 주어진 조건에 적합한 해법을 자동으로 생성할 수 있다면 문제점 해결을 위한 최고의 해법이 될 것이다[1][2]. 알고리즘을 체계적 축적하여 주어진 상황에 적합한 알고리즘을 자동으로 선택하고 사용할 수 있기 위해서는 기능이 유사한 알고리즘들을 정해진 자료와 시험단계를 거쳐 얻어진 알고리즘들의 성능과 특성을 수치화하여 관리할 수 있어야 한다. 이를 위해서는 먼저 알고리즘들이 소스코드 수준에서부터 재사용이 가능한 형태로 구현되어야 하며, 표준화된 방법을 통하여 알고리즘이 구현되어야 한다[3][4][5][6][7]. 뿐만 아니라 새로운 알고리즘을 연구하는데 있어서도 기존의 개발된 알고리즘 재사용할 수 있어야 성능에 대한 비교가 가능해지므로 새로운 알고리즘에 대한 연구를 효과적으로 할 수 있다.

일반적으로 복잡한 알고리즘의 구현은 몇 주 이상이 걸리며 최소한 4 단계를 거쳐야 한다[8]. 문제를 해결하기 위한 일반적인 해법을 찾고, 알고리즘을 이해하고, 코딩하고, 마지막으로 올바로 구현되었는지 시험하는 단계이다. 하나의 문제를 해결하는데 있어서 기하학, 통계학, 지식기반 방법, 신경망과 같은 다양한 방법을 이용하여 풀 수 있으며, 문제를 풀기 위한 알고리즘을 이해한다는 것이 간단한 작업이 아님을 알 수 있다. 이러한 작업 과정에 있어서 알고리즘을 기술한 논문이나 보고서와 같은 발행물이 알고리즘의 성능에 심각한 영향을 미치는 중요한 상세 부분을 생략하고 있다면 이 같은 작업을 더욱 어렵게 만든다. 상세한 부분이라 함은 계산에 필요한 수치적인 정확도와 같은 이슈와 특정한 등식을 해결하기 위한 방법, 필터의 크기 등등을 나타낸다. 이러한 상세한 정보들은 오직 많은 시간을 투자한 실험에 의해서만 얻어질 수 있고, 알고리즘을 구현하기 위하여 더욱 깊은 이해가 필요하게 된다[2][9].

끝으로 구현된 알고리즘을 시험할 때에는 실제적인 딜레마와 마주치게 된다. 만일, 발행물에 기록되어 있는 대로 알고리즘이 수행되지 않는다면, 이것은 알고리즘이 원래부터 있는 고유한 특성이거나, 알고리즘 구현을 잘못하였거나 알고리즘이 부분적으로 최적화되어 있을 수도 있다는 것이다. 구현상의 오류가 알고리즘을 사용한지 몇 개월 후에 발견되는 경우도 일반적으로 나타나는 현상이다. 오류가 늦게 발견되었다면 모든 알고리즘에 대해 지금까지 했던 작업들을 반복하여야 한다. 그러므로, 구현이 올바르고 알고리즘이 잘못되었다는 것을 입증하는 데에는 더욱 더 오랜 시간이 걸리게 된다[1][2].

이러한 문제들을 극복하기 위해서 많은 사람들(특히 산업적인 측면에서)은 알고리즘 비교를 위한 프로젝트들을 학계에 요구하고 있다. 이러한 비교를 위한 연구를 위해 사용될 수 있는 표준 자료 집합과 결과들이 구체적으로 문서화될 수 있다면 새로운 응용 개발을 위한 적합한 알고리즘을 선택하기 위한 가이드로서 사용될 수 있는 데이터베이스가 만들어질 수 있을 것이다. 그러한 데이터베이스는 영상처리 분야의 연구를 가속화시키고 연구 결과의 질을 향상시키는데 많은 도움을 줄 것이 확실하며, 영상처리 해법을 자동 생성하기 위한 기초 자료로 이용될 수 있다. 그러나 이 데이터베이스 하나가 모든 문제들을 해결할 수는 없다. 이렇게 만들어진 데이터베이스가 존재할지라도 많은 사람들은 믿을 수 있는 비교 결과를 얻기 위해서는 연구자나 개발자 자신들의 자료로 알고리즘을 시험할 수 있어야 한다. 대부분은 비교대상이 되는 모든 후보 알고리즘들을 구현할 여유가 없기 때문에 연구자들은 구현 상태가 입증된 알고리즘을 재사용할 수 있길 원한다. 그러므로, 알고리즘의 특성을 부여하고 비교한 결과를 이용하기 위해서는 재사용할 수 있도록 알고리즘을 구현해야 하는 문제를 먼저 해결하여야 한다.

본 연구에서는 이러한 문제점을 해결하기 위하여 기존의 재사용을 위한 연구들을 고찰하여 문제점을 분석하고 이를 해결하기 위하여 영상처리 알고리즘 구현에 적합한 제네릭 프로그래밍 방법을 제안하였다. Stepanov와 Musser에 의해 개발된 제네릭 프로그래밍 방법은 C++ 표준 라이브러리에서 많이 나타나는 설계 방법을 기반으로 하고 있다[10]. 그러나, 제안한 연구는 영상처리 알고리즘 구현에 있어서 자료 구조에 영향을 받지 않으면서 코딩이 가능한 재

사용이 강한 알고리즘 구현 방법과 기존의 C++이 가지고 있는 설계방법을 이용하였을 때 나타나는 처리 속도의 감소를 해결하기 위한 프로그래밍 방법에 대하여 제안하였다. 본 연구에서 제안한 방법이 재사용이 용이한 객체지향 방법을 사용하면서도 기존의 포인터를 이용한 직접적인 메모리 접근 방식의 구현방식과 처리속도 측면에 큰 차이가 없음을 보여주기 위하여 Linux에서 대표적인 영상처리 통합개발환경인 KHOROS와 Windows 시스템의 Hello-Vision에 이식하여 기존의 알고리즘 구현 방법과의 실행속도 차이를 비교 평가하였다.

II. 재사용을 위한 기존 연구 분석

재사용이 가능한 소스 코드를 구현한다는 것은 컴퓨터공학의 다른 분야보다 영상처리 분야에서 보다 어려울 수 있다[3][5][11]. 그 이유는 구현된 코드가 재사용이 가능하면서도 일반적으로 매우 큰 크기를 가진 영상자료를 신속하게 처리하도록 알고리즘을 구현해야 하기 때문이다. 90년대 말까지만 해도 대부분의 영상처리 알고리즘 구현은 특수 하드웨어에 내장된 처리 기능을 이용하는 방식을 중요시했다. 그러나, 하드웨어가 계속적으로 발전하고 변화함에 따라 개발된 코드 재사용은 사실상 불가능했다. CPU와 같은 기본적인 하드웨어의 성능이 증가함으로 인하여 표준 컴퓨터 코드를 가진 어플리케이션이 광범위하게 사용되기 시작하면서 재사용은 이제 필수항목이 되었고, 또한, 알고리즘의 실행속도도 아직까지는 결정적인 역할을 담당한다. 재사용을 위한 4가지 대표적인 예를 살펴봄으로써 현재까지 진행되어온 기존 연구의 특성 및 문제점을 살펴보기로 한다.

처음으로 영상처리 알고리즘 구현에 대한 표준화 시도는 1994년 ISO에 의해 IPI 표준 일부분이 된 PIKS(Programmer's Imaging Kernel System)이다. PIKS(ISO/IEC 12087-1, 12087-2)는 기본적인 이미지 자료 구조의 집합과 120개의 저수준(low-level) 영상처리 알고리즘의 정의로 이루어지지만, 이 표준은 재사용에 있어서 많은 문제를 가진 구조적 프로그래밍 방법을 사용하여 정의되었다[3].

구조적 프로그래밍 방법은 기본적으로 자료 구조를 정의하고 정의된 자료 구조의 속성을 직접적으로 접근하는 방법으로 알고리즘을 구현한다. 이것은 매우 빠른 계산 및 자료처리가 가능하지만 결과적으로 알고리즘과 자료 구조 사이의 결합이 밀접하게 이루어지나 자료 구조가 바뀔 때마다 정의된 자료 구조를 사용한 모든 알고리즘의 소스 코드가 변경되어야 함을 의미한다. 실제적으로 이 방법은 연구자가 개인적으로 정의한 자료 구조를 기반으로 표준 알고리즘들을 실행할 수 없고 연구자의 환경에 적합하도록 표준 자료 구조가 수정될 수 없음을 의미한다. 또한, 예전에 구현된 환경으로부터 표준으로의 부드러운 전환이나 두 개의 환경의 병합이 불가능해진다.

구조적 프로그래밍 방법에서 구조적 설계의 정적인 속성은 직접적이고 많은 융통성이 필요한 고수준(high-level)의 영상처리 작업(task)을 위한 자료 구조를 정의하는 것을 어렵게 한다. 이것은 PIKS가 저수준 영상처리 알고리즘에 한정적으로 적용된 이유이다. 결과적으로 PIKS 표준은 광범위하게 인정받지 못하여 최근에는 거의 사용되지 않고 있다. 그러는 사이에 구조적 프로그래밍방법의 단점을 해결할 가능성이 있는 객체지향 프로그래밍 방법이 나타났다.

여러 관련 연구 중에서 IUE는 사실상 영상처리 응용을 위한 새로운 표준화 시도로써 객체지향 프로그래밍 방법을 이용하여 정의되었다[4]. IUE는 객체지향 모델링 기법을 사용함으로써 융통성있는 자료 구조(클래스 기반)와 영상분석을 위한 모든 수준(저수준 영상처리, 기하학, 토폴로지 특징 표현, 고 수준 객체 재구성, 센서모델, 좌표시스템과 자료 교환 기능)에 대한 자료구조 및 알고리즘을 제공할 수 있었다. IUE는 계승을 이용하여 이미 만들어진 기능을 어느 정도 확장할 수 있고 새로운 환경에 적합하도록 유연하게 변경할 수 있다. 그래서, IUE는 PIKS 표준의 많은 문제점을 해결할 수 있었고 광범위한 해법을 제공할 수 있었다.

그러나, IUE는 고수준의 프로그래밍 기술을 가진 개발자가 아니면 이해가 안 될 정도로 복잡하게 이루어져 있었고 컴파일된 코드의 크기가 너무 커 실제 사용하는데 무리가 많았다. 또한, IUE를 이용한 알고리즘들은 클래스와 매우 밀접하게 결합되어 있어 영상처리 작업을 실행하는데 있어서 매우 느려지는 문제를 야기 시켰다. 이러한 IUE의 실행 속도에 대한 문제점을 해결하기 위하여 자료 구조의 할당된 객체 자료에 직접적으로 접근할 수 있도록 영상처리 알고리즘을 자료 구조 클래스의 "멤버 함수로 구현하였다. 그럼으로써 알고리즘 추가 시마다 시스템의 크기는 커지게 되었다. 또한, 보다 빠른 자료 접근을 위하여 iterator나 자료 accessor 클래스를 사용하게 됨으로써 시스템 설계를 심각할 정도로 복잡하게 하게 만들었고 객체지향 방법에 의해 얻어진 융통성을 잃어버리게 되었다.[12].

이러한 재사용 방법과 완전히 다른 방법을 택한 것이 바로 KHOROS다[5]. 전술한 시스템들은 소스코드를 재사용하기 위한 연구라고 보면 KHOROS는 실행코드를 재사용하기 위한 연구의 형태라고 볼 수 있다. KHOROS는 현재 사용되고 있는 영상처리 및 컴퓨터비젼 시스템 중에서 알고리즘 재사용이 가장 많이 이루어지고 있는 시스템이라 볼 수 있다. 그것은 실행파일의 형태를 가진 컴포넌트 기반의 재사용 방법을 사용하기 때문이다. KHOROS는 매우 복잡한 영상처리 작업을 완성하기 위하여 자유롭게 조합할 수 있는 작고 독립적인 영상처리 모듈의 모임으로 구성되어 있다. 모든 모듈들이 다른 모듈들과 독립적으로 구성되어 있기 때문에 매우 쉽게 새로운 모듈을 작성할 수 있고 기존에 존재하는 영상처리 알고리즘들을 KHOROS 형식으로 쉽게 변환할 수 있다. 그러나, 첫 번째 KHOROS 버전은 자료 모델링에 대하여 Polymorphic Data Model(PDM)이라는 5개의 차원(width, height, depth, time, elements)을 이용하여 모든 영상처리 자료형식을 표현하려는 구조적 프로그래밍 방법을 사용하였다. 그럼으로써 KHOROS에서 개발할 수 있는 영상처리 모듈은 PIKS와 같이 대부분 저 수준의 영상처리 모듈로 제한되어 있었다. 이러한 문제를 해결하기 위하여 버전 2.0에서는 primitives list를 중심으로 sphere, triangle, line등과 같은 geometric primitives를 가진 그래픽 자료 구조인 Geometry Data Model을 확장하였다. 하지만 이 새로운 설계도 객체지향 언어를 사용하여 구현되지 않았기 때문에 객체지향 개념의 많은 장점들을 잃어버렸으며, 처음부터 계획 하에 설계된 것이 아니라 기존에 PDM이 해결하지 못하는 부분을 채워주기 위해 개발된 것이라 확장에 한계가 가지게 되었다.

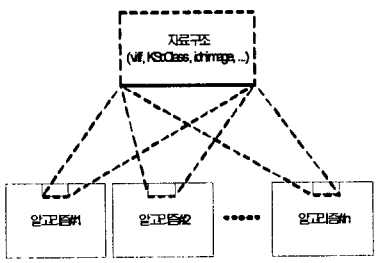
이러한 KHOROS의 단점을 보안하고 객체지향과 컴포넌트 개념으로 개발된 것이 바로 Hello-Vision 시스템이다[6][7]. 이 시스템은 기존의 영상처리 시스템과는 달리 처음부터 객체지향 방법의 모델링 기법을 사용하여 자료 구조를 정의하였으며, 이를 기반으로 하여 구현된 알고리즘이 컴포넌트 형식으로 이루어져 있어 객체지향과 컴포넌트 개념의 장점을 모두 수용한 시스템이다. 그러나, 이 시스템도 C++ 클래스로 구현된 자료 구조와 컴포넌트 형식으로 구현된 알고리즘 부분이 결합되어 있어 자료 구조의 변화가 알고리즘 코드 구현 부분에 영향을 미치는 문제점을 안고 있다.

이러한 기존 연구들의 문제점을 해결하고 재사용에 적합한 이상적인 프로그래밍 방법을 만들기 위해서는 객체지향 방법의 모델링과 컴포넌트기반의 알고리즘 구현 방법을 사용하면서 다음과 같은 고려사항들을 만족하여야 한다[10].

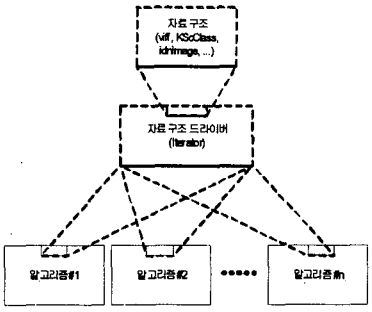
-낮은 오버헤드, 높은 성능
- 강력한 데이터 표현
- 많은 시스템으로의 이식성
- 높은 융통성 (독립적인 모듈, 낮은 결합)
- 자료 구조와 알고리즘 사이의 분리 알고리즘이 다른 자료 구조상에서 수행될 수 있어야 한다.
- 쉬운 사용법

2.1 영상처리 알고리즘 개발에 적합한 제네릭 프로그래밍 방법

전술한 고려사항들을 만족하고 재사용에 적합한 알고리즘 구현을 위하여 자료 구조에 영향을 받지 않으면서 알고리즘을 재사용을 가능하게 하는 구현 방법과 기존의 C++이 가지고 있는 설계방법을 이용하였을 때 나타나는 처리속도의 감소를 해결하기 위한 제너릭 프로그래밍 방법을 제안하였으며, 제안한 방법이 객체지향 방법을 사용하면서도 기존의 포인터를 이용한 직접적인 메모리 접근 방식의 구현 방식과 처리속도 측면에 큰 차이가 없도록 설계하였다. 본 연구에서 제안한 알고리즘 구현 방법은 (그림 1)과 같다. (그림 1)의 (a)는 기존의 일반적인 자료구조 중심의 알고리즘을 구현하였을 경우의 자료구조와 알고리즘 사이의 관계라고 볼 수 있으며, 제안된 연구는 (b)와 같은 구조로 자료구조를 포함하는 iterator 클래스를 이용하여 자료구조용 드라이버 계층을 두어 알고리즘 구현 시 자료구조를 직접 이용하지 않고 드라이버 계층 내의 클래스를 이용하여 알고리즘을 구현하는 방법으로 실제적인 자료구조가 변하더라도 알고리즘 코드에는 수정이 일어나지 않기 때문에 알고리즘의 재사용성을 강화할 수 있다는 것이 기본 아이디어이다. 단, 이러한 방법을 사용하되 기존 방법에 비해 영상처리의 처리속도를 감소시키지 않도록 자료구조 드라이버 계층을 구현하고자 하는 것이 제안된 연구의 주된 내용이다.



(a) 기존의 알고리즘 구현 구조



(b) 제안한 알고리즘 구현 구조

그림 1. 제안한 영상처리 알고리즘 구현을 위한 프로그래밍 개념도
Fig 1. Programming Diagram for Implementing Proposed Image Processing Algorithm

2.2 기본 정의

본 연구에서 영상은 범위가 x=0,…,w-1이고 y=0,…,h-1인 2 차원 사각형 배열로 정의된다. 포인트 (0,0)은 배열의 왼쪽 상단에 위치를 말하며, x가 증가하면 왼쪽으로부터 오른쪽으로 y가 증가하면 위에서부터 아래로 이동됨을 의미한다. 그리드(grid) 위치는 픽셀로써 참조되고 픽셀 형식으로써 하나의 픽셀 안에 저장된 데이터 형식을 참조한다. 하나의 픽셀형식은 프로그래밍 언어에 의해 지원되는 임의의 형식일 수 있다.

유효한 영역 외부에 있는 픽셀들은 최소한 이미지 안의 8 방향 이웃을 가진 경계를 지닌 픽셀들로써 참조된다. 액세스할 수 있는 자료 픽셀들은 '참조 가능하다(dereferencable)'라고 부른다. 일반적으로 모든 내부 픽셀들은 참조 가능하다. 영상 외부의 픽셀들은 특정한 상황에서 참조 가능 할 수 있다.

2.3 Generic 알고리즘

본 연구에서 제안한 재사용이 가능한 알고리즘과 알고리즘의 기초가 되는 자료 구조 사이가 분리될 수 있는 프로그래밍 방법은 iterator라고 부르는 자료구조와 분리된 클래스를 이용하여 네비게이션이나 자료를 액세스하는 기능을 구현함으로써 자료 구조의 논리적인 행동을 추상화하는 방식이다. 이러한 iterator의 사용법은 이미 IUE를 통하여 사용되었으나 본 연구에선 사용하는 iterator와는 아주 다르게 사용된다. 알고리즘들은 자료를 직접적으로 액세스할 수 없으나 알고리즘의 구현에 있어서 다른 자료 구조를 적용하여 구현할 수 있는 iterator를 통하여 사용자가 임의로 정의한 자료 구조를 사용할 수 있다는 것이다. 알고리즘의 형식적인 매개변수들로써 iterator가 사용됨으로써 iterator가 알고리즘의 기초가 되는 가정을 따르는 동안 알고리즘들은 다른 자료 구조상에서도 실행될 수 있다. 이것은 Stepanov와 Musser에 의해 개발된 제너릭 프로그래밍 방법의 철학이고 C++ 표준안에 STL(Standard Template Library)이 포함됨으로써 유명한 방법이 되었다[13][14]. C++에서 템플리트 메커니즘은 컴파일 시간의 다형성(Compile-time Polymorphism)의 수단으로 성능적인 측면에서 오버헤드가 매우 작고 비록 완전하지 않을 지라도 제너릭 알고리즘을 현실화하기 위해 사용된다[10][11]. 본 연구에서는 C++ 표준 라이브러리 스타일의 규칙에 가능한 가깝게 구현된 2 차원 영상 자료구조에 대한 iterator를 설계하였다.

제너릭 알고리즘의 구현은 꼭 필요한 액세스 함수와 네비게이션 함수를 확인하는 것으로 시작된다. 그리고 알고리즘은 전체적으로 이 기능을 제공하는데 적합한 추상 iterator의 개념으로 작성된다. 이것은 알고리즘이 기대할 수 있는 어떤 구체적인 자료 구조의 기능을 전달할 수 있는 구체적인 iterator 형식을 예제로 보여줄 것이다. 제너릭 방법은 많은 장점을 가지고 있다.

- iterator는 추상 인터페이스를 따라야 하나, 특정한 추상 베이스 클래스를 계승할 필요는 없기 때문에 이 방법은 계승 기반 방법 보다 융통성을 가진다.
- 새로운 환경에 맞게 알고리즘 코드를 이식하는 것 보다 iterator 클래스를 구현하는 것이 작업량이 적고 훨씬 쉽다는 개념이다.
- 인라인 함수와 템플리트를 사용하여 C++에서 구현한 재사용이 가능한 알고리즘은 특정한 자료 구조에 맞추어서 구현한 알고리즘처럼 빠르게 실행된다.

iterator가 객체가 아니라 할지라도 알고리즘 코드에 영향을 미치지 않고 현존하는 자료 구조를 위하여 구현할 수 있는 것이 중요하다. 그래서 거대한 알고리즘 코드 기반을 재개발할 필요 없이 임의의 시간에 제네릭 프로그래밍으로 전환할 수 있다.

예를 들면, RGB 영상을 그레이스케일 영상으로 변환한다고 하자. 이러한 작업은 보통 RGB 영상의 red, green, blue 컴포넌트를 읽을 수 있는 iterator가 필요하다 또 다른 하나는 그레이스케일 영상으로 결과를 저장하기 위한 iterator가 필요하고 세 번째로는 iterator의 끝을 표시하기 위한 방법이 필요하다. 이렇게 구현된 것이 다음의 제너릭 알고리즘이다.

```
template <class SequentialIterator, class
  RGBSequentialIterator>
void rgbToGray(SequentialIterator dest,
  SequentialIterator end,
RGBSequentialIterator source) {
for(; dst != end; ++dst, ++src)
{ *dst = 0.3 *source.red() +
  0.59*source.green() + 0.11*source.blue();
}}
```

이제 ByteImageIterator를 생성할 수 있는 하나의 영상 ByteImage와 RGBImageIterator를 생성할 수 있는 RGBImage를 가지고 있다고 가정하면 위의 코드는 다음과 같아진다.

```
void rgbToGray(ByteImage &result,
  RGBImage &src)
{
  ByteImageIterator i = result.begin();
  ByteImageIterator end = result.end();
  RGBImageIterator rgb = src.begin();
  // 제너릭 알고리즘 호출.
  rgbToGray(i, end, rgb);
}
```

물론 ByteImage는 계산된 결과가 정수로 저장되기 때문에 라운드오프(round-off) 오류가 발생한다. 그러므로, 정확한 계산 값을 저장하기 위해서는 FloatImage를 사용하게 되는데 그럴 경우의 소스 코드는 다음과 같다.

```
void rgbToGray(FloatImage &result,
  RGBImage &src)
{
  FloatImageIterator i = result.begin();
  FloatImageIterator end = result.end();
  RGBImageIterator rgb = src.begin();
  // FloatImages를 사용하는 제너릭'
    알고리즘 호출.
  rgbToGray(i, end, rgb)
}
```

그레이스케일 영상 iterator는 기본적인 함수들로 operator!=(), operator++(), operator*() 함수들을 제공하여야 한다. 이러한 operator들이 구현될 수 있는 모든 자료 구조를 알고리즘 구현시 사용할 수 있다. ByteImage가 표준 C 배열에 영상 자료를 저장하는 단순한 클래스라고 하면...

```
class ByteImage {
public:
  // ...
  ByteImageIterator begin();
  ByteImageIterator end();
private:
  int width, height;
  unsigned char * the_pixels; }
```

이 경우에 표준 C 포인터가 iterator로써 사용될 수 있다. 왜냐하면 필요한 오퍼레이터가 자동적으로 포인터에 대해 정의하기 때문이다.

```
typedef unsigned char * ByteImageIterator;
ByteImageIterator ByteImage::begin() {
  return the_pixels; }
ByteImageIterator ByteImage::end() { return
  the_pixels + width*height; }
```

이제 FloatImage안에서의 픽셀이 오직 지정된 픽셀에 대한 레퍼런스(reference : 함수 값 리턴)를 리턴 하는 함수 pixel(x, y)를 통하여 액세스할 수 있다고 가정한다. 그 때 그것의 현재 좌표를 저장하는 iterator를 구현할 수 있으나 image->pixel(x, y) 함수에게 자료 액세스를 위임한다.

```
class FloatImageIterator {
public:
  // ...
Float & operator*()
  { return image->pixel(x, y) }
private:
  FloatImage *image;
  Int x, y;
}
```

RGB iterator는 현재 픽셀의 red, green, blue 채널을 리턴 하는 함수를 제공하여야 한다. 몇 가지 예로 이것을 구현하는 방법을 충분히 보여줄 수 있다.

- RGB 이미지의 일반적인 형태는 분리된 밴드로써 각각의 컬러 채널을 저장한다. 그 때 RGB iterator는 간단히 3개의 표준 sequential iterator를 캡슐화 할 수 있다.

```
class RGBByteImageIterator {
public:
  // ...
  unsigned char red() { return *red_; }
  unsigned char green() { return *green_; }
  unsigned char blue() { return *blue_; }
  void operator++() { ++red_; ++green_;
    ++blue_; }
private:
  ByteImageIterator red_, green_, blue_; }
```

- 또 다르게 넓게 사용되는 것은 long int 값에 3 값을 저장하는 방식이다. (그리고, long int 대신에 struct RGBPixel을 사용하더라도 유사하게 설계된다.)

```
class RGBByteImageIterator2 {
public:
  // ...
  unsigned char red() { return *current_ *
    0xFF; }
  unsigned char green() { return (*current_
    >> 8) * 0xFF; }
  unsigned char blue() { return (*current_
    >> 16) * 0xFF; }
void operator++() { ++current_; }
private:
  LongIntImageIterator current_; }
```

- 컬러-맵 영상에서 iterator는 컬러 인덱스의 배열을 네비게이션하고, accessor 함수는 현재의 인덱스를 사용하여 컬러 룩업(lookup)을 수행한다.

2.4 영상 iterator를 위한 요구사항

STL에서 iterator는 정해진 시간 안에 실행할 수 있는 오퍼레이션의 개념으로 분류된다. 모든 iterator는 다음 요소(element)로 진행할 수 있어야 하고(++iter, iter++), 현재 요소를 액세스하고(*iter) 정해진 시간 안에 정의되고 실행될 다른 iterator (iter==other_iter, 출력 iterator는 제외)와 같은지 검사하여야 한다. forward iterator(그 후에 결국 입력과 출력 iterator를 고려하지 않는다.)는 정확히 이러한 기능을 제공한다. bidirectional iterators는 backward iterator(--iter, iter--)가 더해지고, random access iterator는 정해지지 않은 옵셋(iter += n, iter -= n)을 제공하고, 관련된 액세스(iter[n]), 거리 계산(이진 iter - other_iter)과 비교(iter < other_iter)를 제공한다. 게다가 각 iterator는 일정하거나(*iter가 하나의 값(value)을 리턴 하는 읽기 전용의 경우) 변한다(*iter가 레퍼런스(reference)를 리턴 하는 읽기 쓰기의 경우). 열의 끝은 항상 다른 iterator가 비교될 수 있는 past-the-end iterator로 표시된다. 이러한 특징들은 random access iterators의 모든 요구조건을 만나는 표준 C 포인터를 가진 iterator의 backward 호환성을 유지하게 한다.

iterator는 이전 절에까지는 forward iterator로 정의되었으나, random access iterator로 확장하는 것은 어렵지 않다. 그러나, iterator는 오퍼레이션을 가하는 이미지 클래스들을 불필요하게 제한하지 않음으로써 알고리즘들이 실제적으로 필요한 것 보다 많은 기능을 사용하지 않도록 하는 것이 중요하다.

본 논문에서는 영상처리의 특정한 필요성에 따라 분류를 세우고 확장하려고 한다. 이를 위해서 간단함과 기능성의 측면에서 최적의 평균점(trade-off)를 찾는 것이 필요하다.

STL 모델을 확장하여 본 연구의 목적을 달성하기 위해서는 다음의 내용을 포함하여야 한다.

- 네비게이션은 2차원 안에서 가능해야 한다.
- 영상 경계는 하나의 past-the-end iterator에 의해 표시된다.
- iterator는 어떤 일정한 2차원 이웃을 쉽게 액세스할 수 있어야 한다.
- iterator는 그들의 현재 좌표에 대하여 알고 있어야 한다.
- operator*()를 통한 픽셀자료의 접근 방식이 항상 가장 좋은 방법은 아니다.

2.5 1D 영상 iterator

iterator를 지금까지는 영상 시퀀스(image sequence)와 2D 구조를 숨김으로써 영상을 해석하기 위해서 소개하였다. 그래서, 그들은 1D 영상 iterator의 인스턴스라고 볼 수 있다. 1D 영상 iterator는 x와 y 방향에서 독립적으로 iterator 위치를 변화시키기 위한 방법이 없다는 특성을 가지고 있다. ++iter와 같은 네비게이션 함수는 두 개의 좌표에 항상 영향을 미치거나 전혀 하나의 좌표도 건드리지 않는다. 영상 자료상에 다른 1D 뷰(view)를 제공하는 몇 개의 유용한 iterator는 다음에 설명하는 STL의 체계에 의해 정의될 수 있다. 예를 들어,

SequentialIterators 는 1D 픽셀 열로써 영상을 해석하는 forward iterator이다.

LineIterators는 정해지지 않은 직선을 따라서 반복하는 random access iterator이다. 예를들어 이 선을 따라서 프로파일(profile)을 추출하는데 사용된다.

ChainIterators는 정해지지 않은 커브를 따라서 반복하는 bidirectional iterator이다. 예를 들어 내부적으로 픽셀이 방문한 열은 체인코드에 의해 지정된 것일 수 있다. 그들은 커브나 영역 경계를 다루는데 유용하게 사용된다.

ROISequentialIterators는 정해지지 않은 모양을 가진 관심 있는 영역의 픽셀들을 연속적으로 방문하는 bidirectional iterator이다. 이러한 iterator들은 영역들의 통계를 계산하는데 사용된다.

후자의 3개의 iterator는 현재의 좌표를 가리키는 것이 가능해야 한다.

2.6 2차원 영상 iterator

2.6.1 네비게이션(navigation)

영상 자료 구조를 완전히 이용하기 위해선 모든 방향으로 독립적으로 움직일 수 있는 iterator가 필요하다. 이것은 좌표의 네비게이션 명령을 조정하기 위해 임의의 위치를 지정할 수 있는 매카니즘이 필요하다. 이를 해결하기 위해 오퍼레이터 오버로딩을 사용하는 것은 직접적으로 불가능하다. 대신에 iter.incX()나 iter.subY (dy)와 같은 네비게이션 함수를 정의하여 사용할 수 있다. 그러나, C++에서는 같은 자료를 다른 뷰로 정의한 nested class를 사용한 보다 나은 방법이 존재한다. 다음의 iterator 설계를 보자.

```
class ImageIterator {
public:
  // ...
  class AsX {
    // X 방향으로 내비게이트 하는데 필요한 데이터
    public:
    // X 좌표에 적용하기 위한 네비게이션 함수
    void operator++();
    // ...
  };
  class AsY {
    // data necessary to navigate in Y direction
    public:
    // Y 좌표에 적용하기 위한 네비게이션 함수
    void operator++();
    // ...
  };
  AsX x;          // 데이터에 대한 x-뷰
  AxY y;          // 데이터에 대한 y-뷰
}
```

이제 따라서 움직이는 방향을 지정하기 위한 중첩된(nested) 객체 x와 y를 사용할 수 있다.

```
ImageIterator i;
++i.x;  // x 방향으로 움직인다.
++i.y;  // y 방향으로 움직인다.
```

이 방법은 1D iterator로써 오퍼레이션을 같은 정해진 시간의 개념에서 2D iterator로 정의할 수 있다. 영상으로써 불필요하게 보이는 분류를 STL iterator와 유사하게 분류하는 것은 random access 자료 구조이다.

〈표 1〉은 ImageIterator에서 필요한 네비게이션 기능에 대한 리스트이다. (표에서 같은 영상에 대해 참조하는 i, j를 가진 i, j, k는 동일한 형식의 ImageIterator들이다. dx, dy는 int이다. s는 다음과 같이 정의된 형식 Size2D이다.)

```
struct Size2D {
    int width, height;
}
```

표 1. ImageIterator 클래스에서 필요한 네비게이션 기능 리스트
Table 1. Navigation Function List for ImageIterator Class

동작	결과	의미	비고
++i.x; i.x++	void (*i.x++는 불허)	x 좌표를 증가	내부 순환에서 사용
--i; i.x--	void (*i.x--는 불허)	x 좌표를 감소	내부 순환에서 사용
i.x += dx	ImageIterator::AsX &	dx 값을 x 좌표 값에 추가	내부 순환에서 사용
i.x -= dx	ImageIterator::AsX &	x로부터 dx값을 뺌	내부 순환에서 사용
i.x - j.x	int	x좌표의 차를 구함	결과가 다른 이미지를 조회할 수 있는지 정의가 되어있지 않음
i.x = j.x	ImageIterator::AsX &	i.x += j.x - i.x	결과가 다른 이미지를 조회할 수 있는지 정의가 되어있지 않음
i.x == j.x	bool	j.x - i.x == 0	결과가 다른 이미지를 조회할 수 있는지 정의가 되어있지 않음
i.x < j.x	bool	j.x - i.x > 0	결과가 다른 이미지를 조회할 수 있는지 정의가 되어있지 않음
++i.y; i.y++	void (*i.y++는 불허)	y 좌표를 증가	
--i.y; i.y--	void (*i.y--는 불허)	y 좌표를 감소	
i.y += dy	ImageIterator::AsY &	y 좌표에 dx를 더함	
i.y -= dy	ImageIterator::AsY &	y 좌표에서 dx를 뺌	
i.y - j.y	int	y 좌표의 차를 구한함	결과가 다른 이미지를 조회할 수 있는지 정의가 되어있지 않음
i.y = j.y	ImageIterator::AsY &	i.y += j.y - i.y	결과가 다른 이미지를 조회할 수 있는지 정의가 되어있지 않음
i.y == j.y	bool	j.y - i.y == 0	결과가 다른 이미지를 조회할 수 있는지 정의가 되어있지 않음
i.y < j.y	bool	j.y - i.y > 0	결과가 다른 이미지를 조회할 수 있는지 정의가 되어있지 않음
ImageIterator i(k)		복사 생성자	
i = k	ImageIterator &	복사	
i += s	ImageIterator &	x와 y에 옵셋을 더함	
i -= s	ImageIterator &	x와 y에서 옵셋을 뺌	
i + s	ImageIterator	x와 y에 옵셋을 더함	
i - s	ImageIterator	x와 y에서 옵셋을 뺌.	
i - j	Size2D	x와 y의 차를 구함	결과가 다른 이미지를 조회할 수 있는지 정의가 되어있지 않음
i == j	bool	i.x == j.x && i.y == j.y	내부 순환에서 이것을 사용하고, 결과가 다른 이미지를 조회할 수 있는지 정의가 되어있지 않음
typename ImageIterator::AsX멤버 i.x 의 형식			
typename ImageIterator::AsY멤버 i.y 의 형식			

표 2. ImageIterator 클래스에서 필요한 픽셀 액세스 가능 리스트
Table 2. Pixel Access Function List for ImageIterator Class

동작	결과	의미	비고
일반적인 액세스 기능			
typename mageIterator ::PixelTypeiterator 픽셀 형식			
*i	PixelType &	픽셀 데이터를 액세스	오직 mutable iterator만
*i	PixelType	픽셀 데이터를 읽음	오직 constant iterator만
i(dx, dy)	PixelType &	옵셋(dx, dy)에서 데이터를 액세스	오직 mutable iterator만
i(dx, dy)	PixelType &	옵셋(dx, dy)에서 데이터를 읽음.	오직 constant iterator만
벡터 픽셀들을 위한 특별 엑세스 기능			
typename mageIterator ::ValueType 벡터 안에서의 value의 형식			
i(b)	ValueType &	현재 벡터의 밴드 b를 액세스	오직 mutable iterator만
i(b)	ValueType	현재 벡터의 밴드 b를 읽음.	오직 constant iterator만
i(dx, dy, b)	ValueType &	옵셋(dx, dy)에서 벡터의 밴드 b를 액세스	오직 mutable iterator만
i(dx, dy, b)	ValueType	옵셋(dx, dy)에서 벡터의 밴드 b를 읽음.	오직 constant iterator만
RGB 픽셀을 위한 특별 액세스 기능			
i.red()	ValueType &	현재 픽셀의 red 채널을 액세스	오직 mutable iterator만
i.red()	ValueType	현재 픽셀의 red 채널을 읽음	오직 constant iterator만
i.red(dx, dy)	ValueType &	옵셋(dx, dy)에서 픽셀의 red 채널을 액세스	오직 mutable iterator만
i.red(dx, dy)	ValueType	옵셋(dx, dy)에서 픽셀의 red 채널을 읽음	오직 constant iterator만
greed과 blue도 동일			

III. 구현

iterator의 구현은 영상 자료 형식에 의존한다. 그러나, 일반적인 경우를 위한 몇가지 예제는 의미가 있다. 여기서는 세부적인 내용을 잘 모르거나 내부적인 표현에 접근할 수 없는 존재하는 영상 형식을 둘러쌀 수 있는 iterator 구현을 하였다. 예제의 영상 형식을 ProtectedImage라고 부르고, 픽셀 액세스 함수 pixel(int x, int y)를 정의한다고 가정하자.

```
class ProtectedImage {
public:
  typedef PixelType;
  PixelType & pixel(int, x, int y);
// ...
}
```

그 때 iterator는 픽셀을 직접적으로 접근하기 위하여 이 함수를 사용할 수 있다. 그래서 성능은 부분 최적화될 수 있다. 그러나, 구현은 매우 간단하고 빠른 프로토타이핑을 하는 동안 좋은 선택을 한 것이다. 여기서 x와 y 방향에서 요구되는 오퍼레이터가 자동적으로 정의되기 위하여 int 형식을 가지는 AsX와 AsY 데이터 멤버를 구현한다. 다음에는 3가지 함수를 보여준다.

```
Struct ProtectedImageIterator {
 typedef ProtecedImage::PixelType
   PixelType;
 typedef int AsX;
 typedef int AsY;
ProtectedImageIterator(ProtectedImage *i)
 : image_(i), // 이미지를 기억한다.
 x(0), y(0)
 { }
```

```
// 그들의 좌표를 비교함으로써 iterator를 비
  교한다.
bool operator==( ProtectedImageIterator
  const & i) const {
  return x == i.x && y == i.y;
}
// ProtectedImage 멤버 함수를 통하여 간접
  적으로 액세스 한다.
PixelType & operator*() {
    return image->pixel(x, y);
}
  //…그에 맞게 적절히 다른 함수들을 구현한다.
  int x, y; // iterator의 현재 좌표
private:
  ProtectedImage *image_; // 포장된 이미지
}
```

또 다른 전형적인 경우는 픽셀 값들을 저장하기 위한 비압축(raw) 저장 공간(storage)에 내부적으로 할당되어 저장된 영상이다. 이 iterator는 직접적으로 포인터를 통하여 픽셀들을 접근하므로 매우 효율적이다. 그것은 자신이 소유한 영상 자료 형식에 대하여 제안 연구에서 사용한 표준 구현방식이다. 라이브러리의 사용자들은 비압축 저장 공간과 iterator 초기화에 대하여 알지 못하는 것이 필요하기 때문에 영상들은 iterator의 생성자를 캡슐화한 멤버 함수를 가진다. 다음 절에서 정의된 minPixelValue() 함수를 호출하는 것이 이 같은 방식으로 볼 수 있다.

```
OurImageType image(width, height); // 그리
고 픽셀 값을 채운다. …
// iterator를 구성한다.
OurImageType::Iterator upperleft =
  image.upperLeft();
OurImageType::Iterator lowerright =
    image.lowerRight();
// 전체 이미지에서 최소값을 찾는다.
OurImageType::PixelType min =
  minPixelValue(upperleft, lowerright);
// 보다 작은 ROI에서 최소값을 찾는다. (경계선의
픽셀들은 제외한다.)
Min = minPixelValue(upperleft + Size2D(1,1),
  lowerright  Size2D(1, 1));
```

ImageIterator 구현이 연속된 배열에 픽셀들이 저장된 것으로 가정하면 멤버함수를 가지고 있지 않은 평범한 예전의 C 자료 구조체를 가진 시스템에도 적용할 수 있다. 예를 들어, KHOROS VIFF 영상과 제너릭 알고리즘을 연결하기 원한다면, 간단히 iterator를 초기화하여 사용할 수 있다.

struct xvimag * viff_image; // 그리고 자료를 채운다.

```
if(viff_image->data_storage_type ==
  VFF_TYPE_FLOAT)
{
// float 이미지를 가지고 좌측 상단에 float
   iterator를 초기화한다.
ImageIterator<float> viff_upper_left(
(float*)(viff_image->imagedata), // 첫 번째
  픽셀에 대한 포인터
viff_image->row_size);              // 이미지의 넓이
// 우측하단은 Size2D(width, height) 크기만큼
떨어져 있다.
ImageIterator<float>
viff_lower_right(viff_upper_left +
Size2D(viff_image->row_size,
  viff_image->col_size);
float min  minPixelValue(viff_upper_left,
  viff_lower_right);
}
```

위와 같은 방법으로 Hello-Vision을 위한 iterator를 만든다면 다음과 같다.

```
KScScalarImage2DFloat * hv_image; // 그리
  고 데이터를 채운다.
if(hv_image->GetId() == KS_DATA_2D_
  FLOAT)
{
// float 이미지를 가지고 좌측 상단에 float
   iterator를 초기화한다.
ImageIterator<float> hv_upper_left(
(float*)(hv_image->GetBuffer()), // 첫 번째
픽셀에 대한 포인터
hv_image->GetXSize());              // 이미지의 넓이
// 우측하단은 Size2D(width, height) 크기만
 떨어져 있다.
```

```
ImageIterator<float>
hv_lower_right(hv_upper_left + Size2D
(hv_image->GetXSize(), hv_image->
  GetYSize());
float min = minPixelValue(hv_upper_left,
  hv_lower_right);
}
```

선택된 구체적인 iterator 구현과 상관없이 iterator가 〈표 1〉에 나열된 요구조건을 만족하는한 iterator를 이용하여 구현된 모든 제너릭 알고리즘을 사용할 수 있다.

Ⅳ. 예제 알고리즘

이 장에서는 지금까지 설계한 iterator 기반의 정의들을 이용하여 지금까지 설명한 2차원 제너릭 알고리즘을 위한 다른 구현 방법을 설명하기 위한 몇가지 예를 보여준다. 아래와 같은 간단한 알고리즘을 재사용하는 것이 큰 도움이되지 않는다고 볼 수 있다. 그러나, 간단한 알고리즘들이 커다란 알고리즘들로 확장된다는 것을 생각한다면 제안된 연구결과가 규모가 큰 알고리즘을 재사용하는데 이용할 수 있다고 볼 수 있다.

일반적으로 upper_left의 점을 관심영역의 첫 번째 픽셀로 정하는 것은 영상처리에서 습관적으로 사용하는 방법이다. 그러므로 lower_right는 영역 밖의 첫 번째 픽셀이라고 볼 수 있다. (즉, 영역 안의 마지막 픽셀은 (—lower_right.x, —lower_right.y)의 위치에 있다.) 또한, 영상 안에서 알고리즘이 수정없이 사각의 부분영역에 적용될 수 있기 위해서 upper_left와 upper_right는 정해지지 않은 임의의 좌표로써 사용된다.

첫 번째 알고리즘은 ROI에서 최소의 픽셀 값을 계산한다. 그것은 인덱스 오퍼레이터(operator()(int x, int y))를 통한 루프 기반의 간단한 정수로써 계산된다.

```
template <class ImageIterator>
ImageIterator::PixelType
minPixelValue(ImageIterator upper_left,
  ImageIterator lower_right)
{
  int width = lower_right.x  upper_left.x;
  // ROI의 넓이
  int height = lower_right.y  upper_left.y;
  // ROI의 높이
  ImageIterator::PixelType min =
    *upper_left; // 최소값을 초기화한다.
  for (int y=0; y<height; ++y) // 정수 좌표를 사용한다.
  {
   for (int x=0; x<width; ++x)
   {
    // index 오퍼레이터를 통하여 픽셀을 액세스 한다.
   if (min > *upper_left) min =
     upper_left(x, y);
   }
  }
  return min;
}
```

위의 변형인 다른 방식의 사용 예는 매우 명확하고 즉시 이해할 수 있다. 그러나, 주소 계산을 픽셀을 액세스하는 것에 포함함으로 인하여 각 픽셀들이 여러 번 액세스되더라도 부분적으로 최적화되어 수행할 수도 있다. 주소 계산은 순환(loop)를 제어하기 위하여 iterator의 네비게이션 함수를 사용하는 다음 변형의 예에서 없어진다. 다음의 알고리즘은 ROI 안에서 모든 픽셀 값의 합을 계산하는 알고리즘이다.

```
template <class ImageIterator, class
  SumType>
SumType
sumOfPixelValues(ImageIterator upper_left,
ImageIterator lower_right, SumType initial_sum)
{
// lower_right가 upper_left의 왼쪽이나 위쪽
   에 있다면 아무 동작도 하지 않는다.
if((lower_right.x  upper_left.x <= 0) &&
  (lower_right.y  upper_left.y <= 0)) return;
ImageIterator y(upper_left); // ROI 시작에서
```

```
  iterator 설정한다.
ImageIterator yend(y);      // 경계를 표시
yend.y = lower_right.y;     // ROI의 첫 번째 칸의
끝을 설정
// iterator를 직접적으로 사용하여 첫 번째 칸
  아래쪽으로 반복
for(; y != yend; ++y.y)
{
  ImageIterator x(y);   // 현재 열의 시작에
    서 iterator 설정한다.
  ImageIterator xend(x);      // 경계를 표시한다.
  xend.x = lower_right.x;   // 현재 열의 끝을
    설정한다.
  // iterator를 직접적으로 사용하여 현재 열
    을 가로질러 반복한다.
  for(; x != xend; ++x.x)
{
  // dereferencing 오퍼레이터를 통하여 픽셀
    을 액세스한다.
  Initial_sum += *x;
}
return initial_sum;}
```

다음 예는 두 개의 이미지를 포함하기 때문에 조금 복잡하다. 알고리즘은 하나의 이미지를 정해지지 않은 크기의 정사각형 윈도우를 이용하여 평균화함으로써 영상을 뿌옇게 한다.(크기는 체스 보드 메트릭안에 window_radius로써 주어진다.) 각 소스 픽셀에 대하여, 주변 윈도우 안의 모든 픽셀들의 평균이 계산되고 결과가 대상 픽셀에 기록된다. 윈도우가 부분적으로 이미지의 외곽에 있다면 그것은 따라서 가장자리가 짤리게 된다. 특성(traits) 기법[15]은 픽셀 값의 합을 보관하기 위한 변수의 형식을 결정하는데 사용된다.

```
template <class ImageIterator>
void average(ImageIterator dest_upper_left,
  ImageIterator source_upper_left, Image Iterator
  source_lower_right, int
  window_radius){
// SumType을 결정하기 위하여 traits를 사용한다.
Typedef ImageIterator::PixelType PixelType;
Typedef numeric_traits <PixelType>
  ::Tpromote SumType;
// 이미지의 넓이와 높이를 계산한다.
int w = source_lower_right.x
  source_upper_left.x + 1;
int h = source_lower_right.y
  source_upper_left.y + 1;
// y iterator를 생성한다.
ImageIterator yd(dest_upper_left);
ImageIterator ys(source_upper_left);
for(int y=0; y < h; ++y, ++ys.y, ++yd.y)
{
  // x iterator를 생성한다.
ImageIterator xd(yd);
ImageIterator xs(ys);
for(int x=0; x < w; ++x, ++xs.x, ++xd.x)
{
  // 이미지를 맞추기 위한 윈도우의 크기
int x0 = min(xs.x  source_upper_left.x,
  window_radius);
int y0 = min(xs.y  source_upper_left.y,
  window_radius);
int x1 = min(source_upper_left.x - xs.x,
  window_radius + 1);
int y1 = min(source_upper_left.y - xs.y,
  window_radius + 1);
int ww = x0 + x1 + 1; // 윈도우의 넓이
int wh = y0 + y1 + 1; // 윈도우의 높이
//합을 초기화한다.
SumType sum = 0;
//윈도우를 위하여 sumOfPixelValues()
  함수를 호출한다.
  sum = sumOfPixelValues(xs  Size2D(x0,
   y0), xs + Size2D(x1, y1), sum); }}}
```

이 알고리즘은 포인터 기반 버전보다 덜 복잡하면서도 실행 속도 측면에서는 거의 비슷한 속도를 낸다.

V. 성 능

본 연구에서 제안한 방식으로 구현한 제너릭 알고리즘 성능을 평가하기 위하여 영상처리 알고리즘 통합 개발 환경으로 대표되는 두 개의 시스템에 기존의 방법부터 제안방법까지 알고리즘을 구현하여 벤치마크 테스트를 수행하였다. 최근 일반화되어 있는 400백만 픽셀의 디지털 카메라로 찍은 2272 * 1704 크기의 RGB 이미지에 대하여 9*9 크기의 윈도우 크기로 평균을 구하는 5가지의 다른 구현 방법들에 대하여 KHOROS, Hello-Vision에 이식하여 벤치마크 하였다.

표 3. 벤치마크 테스트 결과
(2272 * 1704 크기의 컬러 영상을 9 * 9 크기의 윈도우로 평균을 구하는 알고리즘)
Table 3. Result of Benchmark Test

시스템	알고리즘 구현 방법				
	방법1	방법2	방법3	방법4	방법5
시스템1	1.8s (100%)	8.5s (472%)	3.1s (172%)	2.0s (111%)	1.9s (105%)
시스템2	2.1s (100%)	10.7s (509%)	5.7s (271%)	2.5s (120%)	2.2 s (104%)
	하드웨어 : PC Pentium 4, 2.4GHz, 512 MB 시스템 1 : Windows 2000, Hello-Vision, Microsoft VC++ 6.0 시스템 2 : Linux , KHOROS, GCC 3.4.3				

- 방법 1: 경험적으로 최적화한 포인터 기반의 C 코드로 구현한 경우 (구조적 프로그래밍 방법)
- 방법 2: 가상 접근 함수를 사용하는 추상 이미지 인터페이스를 사용하는 전통적인 객체지향 방식으로 구현한 경우 (IUE 에서 사용한 방법)

```
class AbstractImage {
public:
  Virtual float & pixel(int x, int y) = 0; }
```

- 방법 3: 4장의 ProtectedImageIterator 방식으로 평균 알고리즘을 구현한 경우 (IUE에서 사용한 방법)
- 방법 4: 기존의 자료구조의 포인터를 통하여 직접적으로 이용하는 ImageIterator를 사용하여 평균 알고리즘을 구현한 경우 (제안 방법)
- 방법 5 : 방법 2와 같은 알고리즘이나, 추상 이미지의 가상 함수 대신에 ImageIterator의 operator()(int x, int y)를 사용하여 알고리즘(cf. 함수 minPixelValue())을 구현한 경우 (제안 방법)

이러한 벤치마크 테스트의 결과가 〈표 3〉이다. 〈표 3〉을 보면 본 연구에서 제안한 ImageIterator(방법 4와 방법 5)가 포인터로 직접 액세스한 알고리즘(방법 1)의 성능과 가까운 결과가 나옴을 확인할 수 있었다. 반대로 C++의 정통적인 객체지향 방법을 사용하는 추상 이미지 인터페이스를 사용한 방법들은 매우 느리게 나타났다.

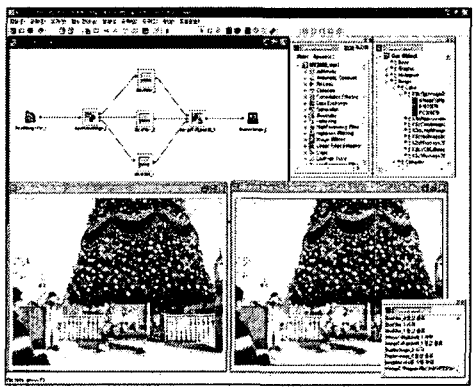


그림 1. Hello-Vision에서 벤치마크 테스트하는 화면
Fig 1. Scene of Benchmark Test using Hello-Vision

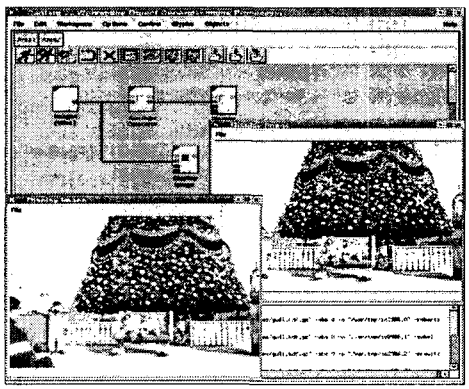


그림 2. KHOROS 에서 벤치마크 테스트하는 화면
Fig 2. Scene of Benchmark Test using KHOROS

Ⅵ. 결 론

본 연구는 영상처리 알고리즘 구현 시 자료구조의 변경에 영향을 받지 않으면서 재사용 가능한 알고리즘 개발을 위한 제너릭 프로그래밍 방법에 대한 연구이다. 본 연구에서 제안한 방법은 영상처리에서 재사용 소프트웨어 설계에 대한 전통적인 방법의 많은 문제점을 극복할 수 있다.

- iterator를 이용한 알고리즘의 구현은 사용자 가정한 임의의 자료 구조를 그대로 이용할 수 있도록 해주며, 현재 존재하는 알고리즘 소스 코드에는 영향을 미치지 않는다.
- 포인터 기반의 알고리즘의 최적화 버전에 비해 실행 속도의 차이가 매우 적은 반면에 영상 변환을 할 필요가 없다.
- 재사용 가능한 알고리즘을 구현한다는 것이 전통적인 프로그래밍 방법에 비해 결코 어려운 일이 아니며, 복잡한 계승 계층을 설계할 필요가 없다.

제안한 방법을 사용하면 영상처리 알고리즘의 재사용이 좀 더 쉬워진다. 그래서, 제안한 연구는 알고리즘 특징과 성능 비교를 위한 연구를 위한 기초적 연구라고 할 수 있다. 향후 보다 고수준의 이미지 분석 기능을 위하여 그래프 알고리즘을 구현하기 위한 다양한 종류의 iterator를 정의하여 전체적인 영상처리 분야를 지원할 수 있도록 하여야 할 것이다.

참고문헌

[1] R. Clouard, et al, "Borg: a knowledge-based system for automatic generation of image processing programs", Pattern Analysis and Machine Intelligence, IEEE Transactions on,2,Feb. 1999.

[2] R. Taniguchi, M. Amamiya and E. Kawaguchi, "Knowledge-based image processing system: IPSSENS-II", Image Processing and its Applications, 1989., Third International Conference on,pp. 462466, Jul 1989.

[3] "Image Processing and Interchange Standard", ISO/IEC Document 12087, 1994.

[4] "Image Understanding Environment Documentation", Amerinex Applied Imaging Inc. 1996 (http://www.aai.com/AAI/IUE/IUE.html).

[5] "Khoros Documentation", Khoral Research Inc. 1996. (http://www.khoral.com)

[6] OkSam Chae and JeongHeon Lee, "Integrated Image Processing Software Development Environment: Hello-Vision", CISST 2001 Proceedings, 2001.

[7] 이정헌, 안용학, 채옥삼, "효율적인 영상처리 교육을 위한 통합 환경 개발에 관한 연구", 대한 전자 공학회 논문지 제 41권 sp편 제6호, pp.127-135, 2004년 11월.

[8] M. Klinger, "Reusable test executive and test programs methodology and implementation comparison between HP VEE and LabView", AUTOTESTCON '99. IEEE Systems Readiness Technology Conference, 1999. IEEE ,Aug.-2, pp.305 312, Sept. 1999.

[9] R. Clouard, et al, "BORG: a knowledge-based system for the automation of image segmentation tasks", Image Processing and its Applications, 1995., Fifth International Conference on ,pp.524528, Jul 1995.

[10] D.Musser, A.Stepanov, "Generic Programming", 1stIntl. Joint Conf. of ISSAC-88 and AAEC-6, Springer LNCS 358, 1989.

[11] D.Musser, A.Stepanov, "Algorithm-Oriented Generic Libraries", Software Practice and Experience, Vol. 24(7), pp.623-642, 1994.

[12] VXL, http://vxl.sourceforge.net

[14] "The Programming Language C++", ISO/IEC Document CD 14882, 1996.

[13] D.Musser, A. Saini, "STL Tutorial and Reference Guide", Addison-Wesley, 1996.

[15] N. Myers, "A New and Useful Template Techniques: Traits", C++ Report, June 1995.

저 자 소 개

이 정 헌

e-mail: opendori@paran.com

1994년 경희대학교 전자계산공학과 (공학석사)

1999년~2004년 (주)엠지시스템 부설연구소 연구소장

2005년 2월 경희대학교 전자계산공학과(공학박사)

2005년 3월 미디어 코러스 단말 개발부 부장

〈관심분야〉 통합개발환경, 멀티미디어자료처리, 디지털영상처리, 소프트웨어공학 등

이 준 형

1996년~현재 : 경희대학교 컴퓨터공학과 박사과정

1999년~현재 : 극동정보대학 컴퓨터정보과 교수

〈관심분야〉 영상처리