JDCS
디지털콘텐츠학회논문지
Journal of Digital Contents Society
Vol. 25, No. 3, pp. 833-840, Mar. 2024

GPU 프로그래밍을 이용한 실시간 연기 상호작용 시뮬레이션

김 동 민[1] · 최 웅[2*]
[1]강남대학교 소프트웨어응용학부 학사
[2*]강남대학교 ICT융합공학부 부교수

Real-Time Smoke Interaction Simulation using GPU Programming

Dong-Min Kim[1] · Woong Choi[2*]

[1]Bachelor's degree, College of Software Applications, Kangnam University, Yongin 16979, Korea
[2*]Associate Professor, College of ICT Construction & Welfare Convergence, Kangnam University, Yongin 16979, Korea

[요 약]

몰입감은 사용자가 게임이나 VR(Virtual Reality) 콘텐츠를 즐기는 데 있어 만족감을 결정짓는 요인 중 하나이다. 현재까지 사용자의 몰입감을 증가시키기 위한 사실적이고 역동적인 시뮬레이션의 구현 방법을 제시한 사례가 많이 존재한다. 연기 또한 시뮬레이션이 가능한 요소 중 하나이고 연기와 사용자 사이의 상호작용 또한 구현이 가능하다. 본 연구에서는 GPU 프로그래밍을 이용하여 dens_step과 vel_step이라는 핵심 알고리즘으로 구성된 그리드 기반의 연기 시뮬레이션 솔버에 오브젝트와 그리드 간의 충돌이 일어난 셀을 구분하여 해당 셀을 넘겨주어 연기의 흐름을 바꾸는 방식으로 연기와 오브젝트 간의 상호작용을 구현했다. 이에 따라 오브젝트가 그리드를 통과하는 속도와 방향에 따라 연기의 흐름이 변화하는 모습을 확인할 수 있었고 오브젝트가 그리드 내부에 배치되어 있을 때 연기가 오브젝트를 통과하지 않고 오브젝트의 모양에 따라 자연스럽게 피해서 흐르는 모습을 확인할 수 있었다. 향후 다양한 최적화 방안을 모색하여 교육, 훈련, 게임 그리고 VR 콘텐츠와 같이 여러 분야에 활용이 가능할 것으로 기대된다.

[Abstract]

Immersion is one of the factors that determine satisfaction when users enjoy games or virtual reality (VR) content. To date, there have been many examples of realistic and dynamic simulation implementation methods to increase user immersion. Acting is also one of the elements that can be simulated, and interactions between acting and users can also be implemented. In this study, GPU programming was used to implement the interaction between smoke and an object by dividing the cell in which the object and the grid collided and handing the cell over to a grid-based smoke simulation solver composed of the core algorithms dens_step and vel_step to change the flow of smoke. As a result, it was possible to see how the flow of smoke changed with the speed and direction of the object passing through the grid, and when the object was placed inside the grid, it was possible to see how the smoke naturally avoided and flowed according to the shape of the object without passing through it. If various optimization measures can be successfully applied to applications in the future, it is expected to find application in various fields such as education, training, games, and VR content.

색인어 : 실시간, GPU 프로그래밍, 연기 시뮬레이션, 상호작용, 충돌 감지
Keyword : Real-Time, GPU Programming, Smoke Simulation, Interaction, Collision Detection

http://dx.doi.org/10.9728/dcs.2024.25.3.833

Received 15 February 2024; **Revised** 11 March 2024
Accepted 12 March 2024

***Corresponding Author; Woong Choi**

Tel: +82-31-280-3753
E-mail: wchoi@kangnam.ac.kr

Ⅰ. 서 론

최근 트래킹 및 디스플레이 기술의 비용 감소로 인해 가상현실이 소비자에게 제공될 수 있게 되었다. 가상현실(VR; virtual reality)은 사용자에게 높은 수준의 존재감(Presence)을 제공할 수 있는 잠재력을 가지고 있다. 여기서 존재감은 VR의 주체인 사용자가 가상 환경을 현실적으로 반응하는 정도로 간주한다[1]. 존재감은 몰입감과 상호작용과의 긴밀한 관계를 가지는데, Mütterlein의 연구에 따르면 가상 환경과 사용자 간의 상호작용이 활발할수록 존재감은 높아지게 된다. 존재감이 높아질수록 몰입감도 높아지게 되고 VR 콘텐츠에 대한 긍정적인 반응을 유도하는 핵심적인 요소로 작용한다[2]. 또한 VR 내의 환경 요소와의 상호작용이 사실적이고 그 표현이 현실감이 있을수록 몰입감의 중요한 요소로 작용한다[3].

시간이 지남에 따라 컴퓨터의 부품은 점점 소형화 되어가고 성능 또한 빠른 속도로 증가하였다. 이러한 이유로 고성능 컴퓨터를 일반적인 가정에서 접하는 일은 어렵지 않게 되었다. 그에 따라 컴퓨터 게임 이용자도 증가하였고 게임 시장의 규모도 과거에 비해 크게 발전했다. 게임 시장의 발전으로 하여금 단순히 반복된 재미를 넘어 감동과 색다른 경험을 제공하기 위한 게임들이 등장하였고 컴퓨터 게임 이용자의 몰입감도 신중하게 고려해야 할 필수적인 요소로 자리 잡았다. Cheng과 Cairns의 연구에 따르면 VR과 달리 컴퓨터 게임에서의 사실적인 시각적 표현이나 행동의 리얼리즘이 몰입감에 큰 영향을 끼치지는 않았다[4]. 하지만 모니터 디스플레이를 통한 게임에서도 몰입감은 존재감과 긴밀하게 연관되어 있으며 상상력에 의한 몰입과 같이 플레이어의 인지적 능력에 따라 몰입감의 정도는 달라질 수 있다[5]. 이와 같은 이유로 컴퓨터 게임에서의 사실적인 상호작용과 표현의 적용은 VR 콘텐츠와는 다른 목적을 가지고 있다. 예를 들어 카운터 스트라이크 2(2023, Valve)의 경우 연막탄에서 뿜어져 나온 연기가 플레이어의 행동에 맞춰 잠시 걷혔다가 다시 드리우는 표현은 전략적 요소가 될 수 있고 그에 따른 기존의 유저들의 기대도 매우 컸다[6].

게임이나 VR 콘텐츠 내에서 사용자와의 상호작용에 응용할 수 있도록 사실적인 실시간 시뮬레이션에 관한 연구는 여럿 존재해 왔다. Müller 등은 위치 기반 동역학(PBD; position based dynamics)을 통해 이해하기 쉽고 안정적인 물리 시뮬레이션을 구현했다. 대표적으로 실시간 옷감 시뮬레이션을 사실적이고 빠르며 안정적으로 구현하는 데 성공했다[7]. Pfaff 등은 유리나 부드러운 표면, 천과 같이 다양한 재질을 구체가 통과하거나 부서질 때, 사실적인 렌더링이 가능한 방법을 소개했다[8]. Chentanez 등은 여러 다른 시뮬레이션의 방법을 결합하여 효율적으로 대규모 액체 시뮬레이션을 실시간으로 구현하는 방법을 제시했다[9]. Stam은 그리드를 기반으로 안정적인 연기 시뮬레이션을 실시간으로 구현하는 방법을 제시했다[10].

현재까지 실시간 연기 시뮬레이션에 관한 연구는 여럿 존재해 왔다. 최근 연구들을 살펴보면 연기의 사실적인 렌더링과 정교한 시뮬레이션을 다양한 방식으로 제시하고 있다. Wahlqvist와 Rubini는 VR 환경에서 위험에 노출된 상황을 연출할 수 있는 방법으로 연기에 의한 가시성을 고품질로 헨더링할 수 있는 방법을 제시하고 있다[11]. Hu 등은 현실적인 가상 수술 시뮬레이션을 위해 전기 절단으로 인한 조직의 열손상이 발생할 때 3D 소용돌이 입자 큐브 알고리즘(3D-VPICA)과 보조 입자 알고리즘(APA)을 통해 연기의 흐름과 충돌의 사실적인 시뮬레이션을 구현했다[12]. Sato 등은 스트림 함수를 활용한 가이드 기반 후처리 방법을 통해 압력 투영 과정을 거치지 않아 계산의 효율성을 확보하면서 현실적인 연기의 흐름을 구현했다[13]. Wen과 Ma는 소용돌이 보존 격자 볼츠만 방법(VPLBM)을 통해 유체 내의 소용돌이 구조와 속도장을 보존하면서, 더 세밀하고 현실적인 연기 시뮬레이션을 실시간으로 구현했다[14]. 하지만 이러한 연구는 연기의 충돌이 실시간으로 일어나는 경우, 구현을 위해 고수준의 수학 능력을 요구하고 있다. 이러한 이유로 간단하게 구현이 가능하며 사용자와 동적으로 상호작용할 수 있는 환경 요소가 게임이나 VR 콘텐츠 내에 존재한다면 더 큰 몰입감과 긍정적인 반응을 불러일으킬 수 있을 것으로 판단한다. 카운터 스트라이크 2에서 공개된 주변 환경과 유기적으로 상호작용하는 연막탄 시스템에 영감을 얻어 사실적으로 상호작용할 수 있는 연기 시뮬레이션으로 그 범위를 좁힌다[6]. 본 논문은 그리드 방식의 연기 시뮬레이션을 기반으로 사용자와의 상호작용을 추가하고 안정적인 연기 시뮬레이션의 구현 방법을 제안한다.

Ⅱ. 연기 시뮬레이션과 오브젝트 상호작용 결합

2-1 하드웨어 구성 및 알고리즘

1) 하드웨어 구성

본 시뮬레이션은 렌더링을 위해 OpenGL을 이용했고 여러 가지 수학적 연산과 3차원 환경을 구성하는 데 GLM을 사용했다. 그리고 높은 성능을 위해 CUDA 12.2를 이용했다. CUDA는 NVIDIA에서 개발한 GPU의 가상 병렬 연산 요소들에 직접 접근할 수 있도록 하는 소프트웨어 계층으로 사용하려면 NVIDIA의 GPU가 필요하다[15]. 하드웨어의 시스템은 Intel i7-10700K 프로세서, 8GB의 RAM을 두 개 사용하여 총 16GB의 환경으로 구성되었으며 GPU는 NVIDIA의 RTX 3070 GPU이다. 시뮬레이션을 진행한 주변기기는 240Hz의 1920*1080 모니터 디스플레이를 사용했으며 일반적인 키보드와 마우스로 구성했다. 키보드는 카메라의 이동을 담당했으며 마우스를 이용하여 화면의 회전을 구현했다. 이렇게 구성된 시뮬레이션 환경을 통해 Visual Studio 2022에서

C와 C++를 이용하여 구현했다.

2) 알고리즘

vel_step과 dens_step이라는 루틴을 통해 그리드 기반 연기 시뮬레이션의 솔버를 구성한다. 이는 각각 연기의 밀도와 속도가 다음 시간 단계에서 어떠한 값을 가지는지 계산할 수 있도록 add_source, diffuse, advect 그리고 project라는 4개의 단계로 이루어진 함수 묶음이다. vel_step은 순서대로 add_source, SWAP, diffuse, project, SWAP, advect, project로 구성되어 있다. dens_step은 순서대로 add_source, SWAP, diffuse, SWAP, advect로 구성되어 있다. 여기서 SWAP은 두 포인터 배열을 서로 교환해 주는 매크로이다.

사용자가 정의한 오브젝트가 연기 시뮬레이션 그리드와 충돌이 일어난다면 그리드의 해당 셀을 저장한 후 오브젝트의 내부, 외부 그리고 내부와 외부 사이의 경계로 구분하여 내부와 외부는 연기 시뮬레이션 솔버의 외력 추가 단계로 전달되고 경계는 연기 시뮬레이션의 솔버의 경계 조건 설정 단계로 전달한다. 이러한 알고리즘을 통해 연기가 오브젝트를 통과하지 않으면서 오브젝트의 움직임에 따라 연기의 흐름을 변화시킬 수 있다.

2-2 그리드 기반 연기 시뮬레이션

초기 그리드 상태를 w_0으로 정의했을 때 add_source, diffuse, advect, project라는 4개의 단계를 거친 이후의 그리드 상태를 각각 w_1, w_2, w_3, w_4로 정의한다. 여기서 그리드 상태는 vel_step과 dens_step이 서로 다른 의미를 가지지만 각 단계를 설명할 때의 편의성을 고려하여 모두 w라는 공통된 기호를 이용하여 그리드 상태로 정의한다. 또한 수식은 2차원으로 표현했지만 모두 3차원으로 변환이 가능하다.

add_source는 연기의 생성 위치, 방향, 세기를 정한다. 이에 대한 수식은 다음과 같다.

$$w_1(x) = w_0(x) + \Delta t f(x,t) \tag{1}$$

수식 (1)은 현재 시간 단계의 그리드 상태에 외력을 추가해주어 다음 시간 단계의 그리드 상태를 업데이트해준다. 이 하나의 함수에 대해 dens_step에서는 연기에 해당하는 소스를 더하고 vel_step에 대해서는 외력을 더하는 방식으로 재사용이 가능하다. 여기서 Δt는 시간 단계를 의미하며 시뮬레이션이 다음 단계로 이동하는데 소요되는 시간의 양이다. 그리고 $f(x,t)$는 시간 t에서 위치 x에 작용하는 힘을 나타낸다.

diffuse는 연기가 그리드 내에서 어느 정도로 확산될지 정한다. 이에 대한 수식은 다음과 같다.

$$(I - \nu \Delta t \nabla^2) w_2(x) = w_1(x) \tag{2}$$

수식 (2)는 암시적인 방법으로 점도가 크더라도 안정적인 확산 과정을 진행할 수 있다. 여기서 I는 단위 행렬이고 ν는 기본적으로 동점성계수이지만 dens_step에서는 확산계수로 사용한다. ∇^2는 라플라시안 연산자로 이차 미분을 나타낸다. 그러나 이 방법은 대부분의 요소가 0이므로 계산의 효율을 떨어뜨릴 수 있다. 이것을 해결할 수 있는 가장 간단한 방법은 가우스 자이델(Gauss Seidel) 반복법을 사용하는 것이다. 그러나 이 방법은 병렬성이 보장되지 않기 때문에 CUDA에서 커널 함수로 사용하기에 무리가 있다. 이에 대해서는 후술할 성능 향상 섹션에서 다루겠다.

advect는 연기의 속성들이 속도장에 따라 이류하도록 하는 함수이다. 이에 대한 수식은 다음과 같다.

$$w_3(x) = w_2(p(x, -\Delta t)) \tag{3}$$

수식 (3)에서 $w_2(p(x,-\Delta t))$는 이전 시간 단계의 연기의 위치인 $p(x,-\Delta t)$에서의 그리드 상태를 나타낸다. 이류 단계를 효과적으로 풀 수 있는 방법은 밀도를 입자의 집합으로 모델링 하는 것이다. 그리드의 각 셀의 중심이 입자라고 가정하고 이전 시간 단계로부터 현재 셀의 중심점에 도달하는 입자를 역추적하고 이 입자들이 가지고 이동하는 속성은 주변 셀로부터 선형 보간한다.

project는 발산을 제거해 질량을 보존하고 안정적인 시뮬레이션이 되도록 한다. 이에 대한 수식은 다음과 같다.

$$\nabla^2 q = \nabla \cdot w_3, w_4 = w_3 - \nabla q \tag{4}$$

수식 (4)는 푸아송 방정식을 이용해 속도장의 발산을 계산하고 그것은 가우스 자이델 방법으로 해결한다. 그리고 그 결과를 기존의 속도장에 빼주어 발산을 제거해 안정적인 시뮬레이션이 되도록 한다. 여기서 q는 스칼라 퍼텐셜을 의미한다.

이 단계들을 조합하여 dens_step과 vel_step을 구성하는 것만으로도 연기 시뮬레이션은 가능하다. 하지만 그리드 내부에서 연기가 흐르는 모습을 확인하기 위해서는 set_bnd라는 경계조건 설정이 필요하다. set_bnd는 연기가 흐르다가 그리드의 경계에 부딪히면 적절하게 연기의 흐름을 바꿔주는 역할을 한다. 예를 들어 2차원 그리드에서 y 방향의 속도는 x 방향으로 이어진 그리드 경계에 대해 0이어야 하고 x 방향의 속도는 y 방향으로 이어진 그리드 경계에 대해 0이어야 한다.

또한 x 방향으로 이어진 그리드의 경계와 y 방향으로 이어진 그리드의 경계가 만나는 그리드의 모서리는 해당 셀에 인접해 있는 셀의 속도의 평균이다. 이는 각 단계가 이루어질 때마다 진행되어야 하며 이러한 설정을 통해 연기는 그리드의 외부로 유출되지 않도록 시뮬레이션 할 수 있다[10],[16].

2-3 오브젝트 상호작용 및 솔버와의 결합

1) 오브젝트 정의

연기와 사용자 간의 상호작용이 존재하려면 연기에 영향을 주는 오브젝트가 정의되어야 한다. 본 논문에서는 충돌 처리의 편의성을 고려하여 오브젝트는 모두 구체 오브젝트로 정의한다. 이 오브젝트는 클래스로 구현되어 상호작용에 필요한 정보들을 저장하고 그 정보들을 연기 시뮬레이션 솔버에 전달한다. 오브젝트 클래스에 저장되는 정보는 오브젝트의 위치, 속도, 방향, 충돌이 일어난 셀의 정보이다. 위치 정보는 현재 오브젝트가 어느 위치에 존재하고 있는지 벡터로 저장된다. 속도 정보는 오브젝트의 이동 속도로 현재 위치와 이전 위치 사이의 길이가 저장된다. 방향 정보는 오브젝트가 이동하고 있는 방향으로 이전 위치에서 현재 위치로 향하는 벡터를 정규화 한 벡터가 저장된다. 마지막으로 오브젝트와 충돌이 일어난 그리드의 셀 정보이다. 셀 정보는 그리드 사이즈에 맞는 정수형 배열로 충돌된 셀이 오브젝트 내부에 속하는지 외부에 속하는지 구분하기 위해 사용되며 충돌이 일어난 인덱스를 구분하는 방법은 오브젝트의 위치와 그리드의 각 셀의 중심점 간의 거리를 통해 알 수 있다. 구체 오브젝트 기준으로 오브젝트의 중심 위치와 각 셀의 중심점 간의 거리를 계산한 후 구해진 거리가 구체 오브젝트의 반지름보다 작다면 오브젝트와 셀 사이의 충돌이 발생한 것이다. 그러므로 해당 인덱스에 충돌 정보에 관한 고유한 값을 할당한다.

2) 충돌 셀 분류

오브젝트와 충돌이 일어난 셀은 크게 3가지로 분류된다. 첫 번째는 내부 셀, 두 번째는 외부 셀, 마지막 세 번째는 경계 셀이다. 내부 셀은 오브젝트의 내부에 존재하는 셀로 연기가 존재하지 않아 밀도와 속도가 모두 0인 셀이다. 외부 셀은 오브젝트와 그리드가 맞닿아 있는 셀을 의미한다. 외부 셀은 그림 1처럼 오브젝트의 진행 방향에 존재하는 셀을 구분하는데 사용된다. 진행 방향에 존재하는 셀을 구분하는 방법은 다음과 같다. 먼저 구체 오브젝트의 중심 위치에서 외부 셀에 해당하는 각 셀의 중심점으로 향하는 벡터를 정규화 해준다. 이후 그 벡터를 오브젝트의 방향 정보와 내적(Dot Product) 해주어 코사인 유사도를 계산한다. 만약 코사인 유사도가 0.0 보다 크다면 외부 셀 중 오브젝트의 진행 방향에 존재하는 셀임을 알 수 있다. 이렇게 구분한 진행 방향 셀은 오브젝트의 움직임에 따라 진행 방향과 속도를 이용하여 외력을 추가해 주어 연기의 흐름을 변화시키는데 사용된다. 마지막으로 경계 셀은 내부 셀과 외부 셀 사이에 존재하는 셀로 연기가 그리드를 통과하지 못하는 것처럼 오브젝트 또한 통과하지 못하도록 set_bnd에서 오브젝트의 경계 조건을 정의하는 데 사용된다. set_bnd에 대한 설명을 고려하여 경계 셀이 오브젝트의 어느 부분에 존재하고 있는지 구분해 줄 필요가 있다. 예를 들어 경계 셀의 값이 3이라면 오브젝트의 왼쪽과 위를 향하고 있는 방향에 존재하는 셀이고 경계 셀의 값이 5라면 오브젝트의 오른쪽을 향하고 있는 방향에 존재하는 셀인 것이다. 이런 식으로 각 방향마다 고유한 값을 가지게 하여 set_bnd에서 연기가 오브젝트에 부딪힌 방향에 따라 적절한 경계 조건 처리를 가능하게 한다. 이는 그림 1처럼 외부로 향하는 모서리와 외부로 향하는 모서리를 구분해 주어야 정확한 경계 표현이 가능하다. 그림 1의 경우 내부 셀을 우선 구분하고 내부 셀 중 인접한 셀 중 어느 한 부분이라고 내부 셀로 둘러싸여 있지 않다면 외부 셀로 구분해 준다. 그리고 외부 셀과 내부 셀 사이에 존재하는 셀들을 경계 셀로 구분하는데 각 방향에 따라 서로 다른 값을 부여해 준다.

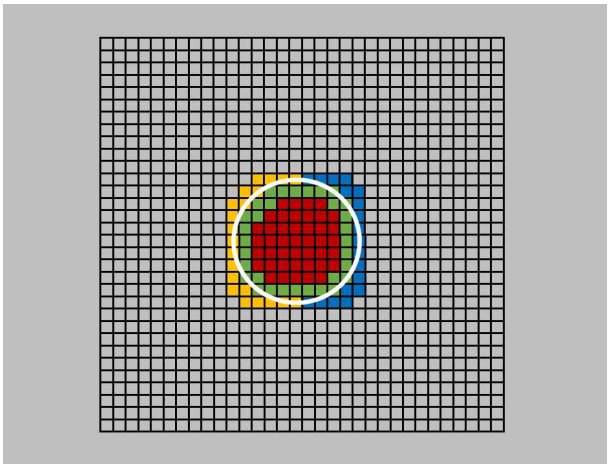

그림 1. 오브젝트와 그리드 사이의 충돌 셀 구분. (빨간색: 내부 셀, 노란색: 외부 셀, 연두색: 경계 셀, 파란색: 진행 방향 셀)

Fig. 1. Collision cell distinction between object and grid. (red: inner cell, yellow: outer cell, light green: boundary cell, blue: moving direction cell)

3) 솔버와의 결합

해당 시뮬레이션에서 오브젝트와 연기의 충돌 후 연기의 흐름을 변화시키는 과정은 기존의 그리드 기반 연기 시뮬레이션의 경계 조건 설정 과정에서 착안한 방법으로 구현 방법이 경계 조건 설정과 매우 유사하며 간단하다. 위에서 구분된 셀들은 모두 연기 시뮬레이션의 그리드의 각 셀과 동일한 인덱스를 가지고 있다. 그렇기 때문에 해당 인덱스의 값에 따라 적절한 동작을 수행해야 한다.

$$C^{curr} = 0, \mathrm{vel}^{prev} = W \cdot d^{obj} \cdot v^{obj} \quad (5)$$

수식 (5)에서 C^{curr}는 현재 셀 상태 중 내부 셀에 해당하는 각 셀의 상태를 의미한다. 내부 셀의 경우 오브젝트의 내부에 해당하므로, 해당 셀의 밀도와 속도를 모두 0으로 설정해 준다. 또한 vel^{prev}는 이전 시간 단계의 각 셀에 대한 속도를 의미한다. W, d^{obj}, v^{obj}는 각각 가중치, 오브젝트의 진행 방향, 오브젝트의 진행 속도를 의미한다. 가중치의 크기에 따라 연기와 오브젝트의 충돌이 일어났을 때, 연기의 흐름이 크게 변화하

거나 적게 변화할 수 있다. 이는 상황에 맞추어 조절할 수 있으며 해당 시뮬레이션에서는 150으로 설정했다. 외부 셀의 경우 오브젝트의 진행 방향에 존재하는 셀을 구분하여 오브젝트의 방향과 속도에 맞게 외력을 전달해 준다. 하지만 하나의 충돌 셀 변수로 외력을 전달할 셀을 구분할 경우 그리드에 여러 개의 오브젝트가 존재한다면 서로 다른 진행 방향을 가지고 있음에도 불구하고 모두 같은 방향으로 외력을 전달할 수 있다. 그렇기 때문에 각 오브젝트마다 서로 다른 충돌 셀 변수를 가지고 있거나 오브젝트를 구분할 수 있는 고유한 ID를 부여해 주어야 한다. 경계 셀은 set_bnd에서 각 방향에 따라 서로 다른 경계 조건 처리가 필요하다. 예를 들어 위를 향하고 있는 셀의 경우 아래의 방향으로 흐르는 연기를 만났다면 그 흐름을 반대 방향, 즉 위로 바꿔주어야 한다. 반대로 아래를 향하고 있는 셀의 경우 위의 방향으로 흐르는 연기를 만났다면 그 흐름을 반대 방향, 즉 아래로 바꿔주어야 한다.

2-4 성능 향상

1) GPU 프로그래밍

CUDA를 이용하면 병렬성이 존재하는 반복문의 경우 반복문을 제거하고 GPU의 성능을 극대화하여 기존의 반복문의 처리 속도를 향상시킬 수 있다[17]. 하지만 연기 시뮬레이션 솔버의 가우스 자이델 방법은 병렬성이 존재하지 않는다. 그렇기 때문에 병렬 시스템에 특화된 Red-Black 가우스 자이델 방법을 사용한다. Red-Black 가우스 자이델은 짝수 인덱스에 대해서는 Red 셀, 홀수 인덱스에 대해서는 Black 셀로 정의하여 Red 셀에 대한 연산을 먼저 진행하고 Black 셀에 대한 연산을 진행하는 방법이다.

2) 충돌 셀 변수 관리 및 성능 실험

각 오브젝트마다 충돌 셀을 저장하는 변수, 즉 충돌이 일어난 인덱스에 값을 저장하는 그리드를 가지고 있다면 충돌이 일어난 부분을 찾기 위한 그리드의 순회 횟수가 많아진다. 그렇기 때문에 하나의 그리드를 정의하고 각 오브젝트마다 ID를 부여하여 해당 인덱스가 어떤 ID를 가진 오브젝트에 대한 충돌 처리를 관리하는지 명확하게 표시해 줘야 한다. 또한 충돌이 일어난 인덱스만 저장해 두었다가 연기 시뮬레이션 솔버에 외력을 전해줄 때 해당 인덱스만 처리를 해주는 방식과 충돌을 관리하는 변수 그리드 전체를 순회하는 방식에 대해 고려해야 할 사항이 있다. 이 두 가지 방식은 각각 장단점을 가지고 있다. 첫 번째 방식은 전체 그리드를 순회할 필요가 없어 분기 미스가 적다. 하지만 커널 함수의 특성상 동적 할당이 이루어지지 않은 변수의 경우 사용할 수가 없어 매 반복마다 충돌이 일어난 셀의 개수만큼 동적 할당을 해주어야 한다. 반면에 두 번째 방식은 매 반복마다 동적 할당은 해줄 필요가 없다. 하지만 그리드 전체를 순회하므로 충돌이 일어나지 않은 셀에 대해 분기 미스가 자주 발생한다. 이 두 가지 방식에 대해서는 어느 방식이 성능이 더 좋을지 예측하기가 어렵다. 그렇기 때문에 두 가지 방식에 대한 실험을 진행해 보았다.

성능 향상이 실제로 이루어졌는지 확인한 실험은 총 두 가지이다. 첫 번째는 CUDA를 이용하지 않고 연기 시뮬레이션을 구현했을 때와 CUDA를 이용하여 연기 시뮬레이션을 구현했을 때이다. 여기서 그리드 사이즈가 $N \times N \times N$일 때 $N = 30$부터 N의 크기를 5씩 늘려가며 초당 프레임 수(FPS; frame per seconds)를 확인하는 실험이다. 두 번째는 CUDA를 이용하여 충돌 셀을 관리하는 변수를 선언하는 방식의 차이에 따른 성능의 차이를 확인하는 실험이다. 각 오브젝트마다 충돌 셀을 관리하는 변수를 가지며 동적 할당을 통해 필요한 인덱스만 솔버에 전달하는 방식과 하나의 충돌 셀을 관리하는 변수를 통해 오브젝트를 ID로 구분하며 동적할당 없이 전체 그리드를 순회하는 방식을 구현했다. 그리고 그리드의 사이즈가 $N \times N \times N$일 때 $N = 50$부터 N의 크기를 5씩 늘려갔으며 그리드와 한 번에 충돌하는 오브젝트의 개수도 1개부터 3개까지 늘려가는 방법으로 FPS를 측정해 보았다. 위의 두 가지 실험은 모두 chrono 라이브러리를 이용하여 시뮬레이션의 한 프레임 당 소요된 시간을 누적한 후 그것을 총 프레임 수로 나누어 평균 FPS를 방식으로 진행했다. 시뮬레이션은 10초를 넘어가면 자동으로 종료되게 했다. 또한 프로그램의 알고리즘은 변화를 확인할 부분 이외에는 통일했으며 시뮬레이션의 초기 변수값도 통일하였다.

Ⅲ. 연구결과 및 고찰

3-1 결과

vel_step과 dens_step을 이용하여 연기 시뮬레이션을 구성할 경우 그림 2와 같이 그리드 내에서 연기가 이류하는 모습을 확인할 수 있다. 연기 상호작용 시뮬레이션은 총 세 가지 종류로 구성했다. 첫 번째는 FPS 게임과 같이 사용자가 마우스 좌클릭을 할 때마다 사용자의 위치로부터 일직선으로 오브젝트를 던진다. 두 번째는 사용자가 바라보는 화면의 중앙에 오브젝트가 고정되어 마우스를 이용하여 카메라를 회전할 경우 오브젝트가 따라오며 원하는 방향과 속도로 오브젝트의 위치를 조종할 수 있다. 세 번째는 그리드의 경계선에 오브젝트를 배치했다. 위의 세 가지 시뮬레이션을 통해 이동하는 오브젝트가 연기와 충돌하거나 배치된 오브젝트에 연기가 충돌될 때 연기의 흐름이 어떻게 변화하는지 관찰할 수 있다. 첫 번째와 두 번째 시뮬레이션의 경우 오브젝트의 이동 방향과 속도에 따라 연기가 갈라지며 흐르는 모습을 관찰할 수 있다. 세 번째 오브젝트의 경우 연기가 오브젝트를 통과하지 않고 오브젝트의 모양에 맞춰 자연스럽게 피해서 흐르는 모습을 관찰할 수 있다. 이러한 모습은 그림 3을 통해 확인할 수 있다.

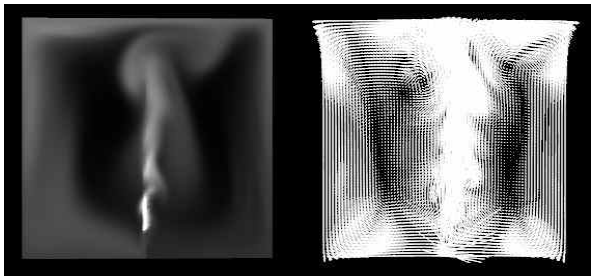

그림 2. 80 × 80 사이즈의 그리드에서 실행한 연기 시뮬레이션. (왼쪽은 밀도장, 오른쪽은 속도장)

Fig. 2. Smoke Simulation executed on an 80 × 80 grid size. (The left side shows the density filed, and the right side shows the velocity foled)

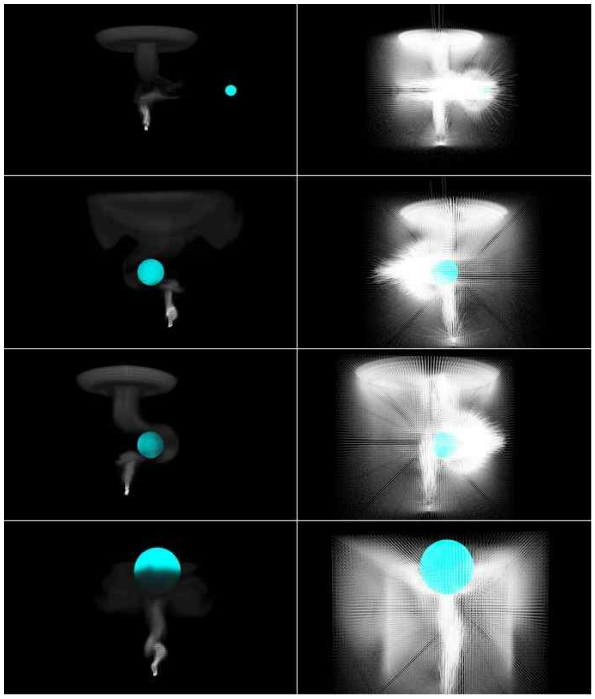

그림 3. 왼쪽은 연기 시뮬레이션, 오른쪽은 그에 대응하는 속도장. (위에서부터 아래까지 순서대로 오브젝트를 던질 때, 왼쪽으로 이동시킬 때, 오른쪽으로 이동시킬 때, 그리드 위에 배치했을 때)

Fig. 3. On the left are smoke simulations, and on the right are the corresponding velocity fields. (From top to bottom in order: when throwing an object, when moving it to the left, when moving it to right, and when placed above the grid)

성능 향상 섹션에서 언급한 대로 OpenGL만을 이용하여 CPU로 연기 시뮬레이션을 수행했을 때와 CUDA와 함께 GPU로 연기 시뮬레이션을 수행했을 때의 평균 FPS 차이를 분석했다. 또한 충돌이 일어난 셀을 처리할 때 충돌 셀 관리 변수를 선언하는 방식과 충돌 셀을 관리하는 방식의 차이에 따른 평균 FPS 차이를 분석했다. 그림 4를 통해 GPU를 이용했을 때의 평균 FPS는 CPU를 이용했을 때보다 약 921.32%의 향상률을 보인다는 결과를 확인할 수 있다. 그리고 표 1을 통해 각 오브젝트마다 충돌 셀 관리 변수를 순회하고 동적 할당을 통해 필요한 인덱스만 이용한 방식은 하나의 충돌 셀 관리 변수를 모든 오브젝트가 공유하며 그리드 전체를 순회하는 방식에 비해 평균 FPS가 약 3.91만큼 높다는 결과를 확인할 수 있다. 그러나 첫 번째 방식은 각 오브젝트마다 그리드를 순회해야 하는 방식의 특성상 그리드와 충돌되는 오브젝트의 개수가 많아질수록 평균 FPS의 하락률이 약 4.08%가 되는 것을 확인할 수 있다. 그러나 기존의 연기 충돌 시뮬레이션은 NVIDIA TITAN X GPUs 환경에서 CUDA를 이용한 GPU 병렬처리와 VPLBM을 통해 두 개의 오브젝트와 연기가 충돌이 일어나는 시뮬레이션을 $64 \times 64 \times 128$ 사이즈의 그리드에서 약 85 FPS라는 결과를 얻었다[14]. 이는 오브젝트와 충돌이 일어난 셀을 분류하고 외력을 솔버에 전달하는 방식의 그리드 사이즈가 $64 \times 64 \times 64$ 일 때, 약 53 FPS라는 결과가 나온 것에 비해 월등히 높은 수치라는 것을 알 수 있다. 하지만 VPLBM은 소용돌이와 속도장을 보존하기 위해 격자 볼츠만 방법에 기반하여 보다 복잡한 접근 방식을 사용하는데 반해 Stam의 방법을 기반으로 한 충돌 검증 방식은 설계 방식이 간단하여 구현이 편리하고 충돌에 따른 외력 전달 부분이 솔버와 분리되어 있어 사용자의 요구에 따라 충돌 시의 흐름 변화 조절이 간편하다. 또한 오브젝트가 그리드 내부를 넘어 외부에서 넘나들어도 안정적으로 충돌을 감지할 수 있다.

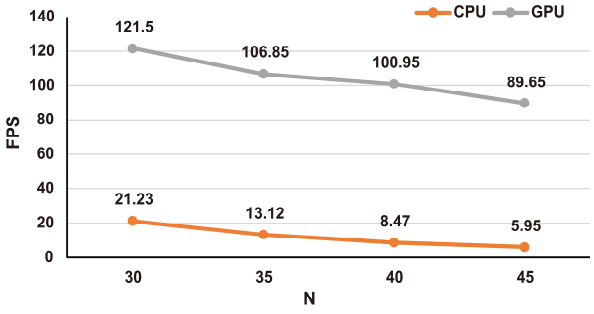

그림 4. N × N × N 사이즈의 3차원 연기 시뮬레이션에서 OpenGL만을 사용했을 때의 FPS와 CUDA를 사용하여 GPU 프로그래밍을 했을 때의 FPS

Fig. 4. FPS in a 3D smoke simulation of N × N × N size when using only OpenGL versus using CUDA for GPU programming

3-2 고찰

사실적인 연기 시뮬레이션에 간단한 오브젝트의 경계 조건 정의, 속도와 방향에 따른 외력 추가에 의한 연기의 흐름 변화는 확실히 여러 상황에 응용이 가능했다. 구체 오브젝트의 발사, 이동, 배치에 관한 시뮬레이션에 더해 다양한 오브젝트에 대한 충돌 처리를 구현한다면 그 외의 시스템은 모두 쉽게 적용이 가능하다는 것이 큰 장점으로 다가왔다. 또한 GPU 프로그래밍을 이용하여 FPS를 눈에 띄게 높여 실시간 시뮬레이션도 끊김 없이 동작하게 한다는 점을 통해 성능 향상에 있어 병렬 처리의 중요성을 알 수 있다.

표 1. N × N × N 사이즈의 연기 시뮬레이션에서 오브젝트 개수에 따른 FPS. (Malloc: 매 반복마다 동적 할당 진행, Many: 많은 수의 그리드 순회, Miss: 충돌 셀 검출 과정에서 많은 분기 미스, Few: 적은 수의 그리드 순회)

Table 1. FPS according to the number of objects in a smoke simulation of size N × N × N. (Malloc: Dynamic allocation per iteration, Many: large number of grid traversal, Miss: large number of branch misses in collision cell detection, Few: small number of grid traversal)

N	Number of Object	Malloc, Many FPS	Miss, Few FPS
50	1	82.01	75.54
50	2	80.98	75.24
50	3	77.67	74.81
55	1	76.61	69.82
55	2	73.83	69.66
55	3	70.63	67.86
60	1	67.49	60.98
60	2	64.17	60.42
60	3	61.39	59.58
65	1	54.92	50.73
65	2	52.26	50.23
65	3	49.54	49.77

동적 할당은 오버헤드를 일으키는 원인으로 실시간 시뮬레이션의 경우 매 프레임마다 이루어지는 것은 권장되는 방식이 아니다[18]. 하지만 본 시뮬레이션에서 진행한 실험의 결과를 통해 GPU 프로그래밍에서 사용되는 스레드의 비효율적인 관리가 매 프레임마다 동적 할당을 해두는 경우보다 FPS를 떨어뜨리는 수치가 더 높게 나왔다는 것을 확인할 수 있다. 즉 충돌 셀을 관리할 때 그리드 전체를 순회하며 분기 미스가 더 많이 일어난다는 것은 스레드를 비효율적으로 사용하여 성능 저하의 원인이 된다는 것이다. 그러므로 만약 동적 할당 없이 필요한 인덱스만을 이용하여 충돌 셀에 대해 처리가 가능하다면 더 높은 성능의 시뮬레이션을 얻을 수 있을 것이다. 또한 각 오브젝트마다 충돌 셀을 관리하는 변수가 존재하는 경우는 오브젝트의 개수가 성능의 영향을 준다는 것을 알 수 있었다. 하나의 변수를 모든 오브젝트가 공유하는 방식은 오브젝트 개수가 늘어나더라도 FPS가 거의 일정한 모습을 볼 수 있지만 각 오브젝트마다 변수가 존재하는 경우는 오브젝트의 개수가 늘어남에 따라 FPS가 점점 낮아지는 모습을 볼 수 있다. 그러므로 연기가 여러 오브젝트와의 충돌이 일어나는 응용 프로그램을 계획할 경우 하나의 변수를 이용하면서 필요한 인덱스만을 솔버에 넘겨주는 방식으로 두 가지 방식을 조합하면 더욱 빠른 성능의 시뮬레이션을 얻을 수 있다는 사실을 알 수 있다.

하지만 본 시뮬레이션은 어디까지나 연기와의 상호작용이 존재할 때 시각적으로 조금 더 역동적인 표현을 위한 시뮬레이션으로 환경 요인이 모두 고려된 사실적인 시뮬레이션과는 거리가 있다. 또한 시뮬레이션을 실시간으로 동작시키기 위한 최적화를 진행하였지만 게임이나 VR 콘텐츠에 적용하기에는 무리가 있을 것이다. 왜냐하면 GPU 프로그래밍을 이용한 시뮬레이션의 경우 $N \times N \times N$ 사이즈의 그리드 연기 시뮬레이션에서 $N = 45$일 때 약 89 FPS가 나왔다. 하지만 이는 상호작용 없이 3차원의 연기 시뮬레이션 하나만 동작시켰을 때이다. 응용 프로그램에서 여러 연기 시뮬레이션이 상호 작용하는 경우가 있다면, 이를 실시간으로 처리하는 것은 어려울 수 있다. 향후, 더 연구를 진행하여 이러한 점을 극복할 수 있다면 게임에서의 다양한 볼거리와 전략적 요소가 되어 사용자의 만족도를 높일 수 있을 것이며 VR을 이용한 소방 훈련이나 상호작용 체험과 같은 콘텐츠에서 몰입감을 높일 수 있을 것이다.

Ⅳ. 결 론

본 연구는 GPU 프로그래밍을 활용한 그리드 기반 연기 시뮬레이션과 충돌 검출된 셀들을 솔버로 전송해 외력을 적용하는 방식을 융합함으로써, 연기와 오브젝트의 상호작용을 가능하게 하는 새로운 접근법을 제안하였다. 향후 연구에서는 충돌 셀을 보다 효율적으로 처리하여 다양한 응용 분야에서 활용될 수 있는 추가적인 최적화 전략을 모색할 예정이다.

이러한 기술은 게임 및 가상현실 콘텐츠에서 사용자에게 이전에는 경험할 수 없었던 새로운 경험을 제공하며, 사용자가 원하는 시나리오를 연출해 더 깊은 몰입감을 제공할 수 있는 가능성을 제시할 것이다. 또한 효율적인 화재 대응 훈련 및 교육 프로그램의 개발에도 기여할 수 있을 것이다.

이를 통해 더욱 현실적이고 역동적인 디지털 콘텐츠를 구현하여 사용자의 만족도를 높일 수 있을 것으로 기대된다.

감사의 글

이 성과는 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(NRF-2022R1A2C1092178).

참고문헌

[1] J. Hvass, O. Larsen, K. Vendelbo, N. Nilsson, R. Nordahl, and S. Serafin, “Visual Realism and Presence in a Virtual Reality Game,” in *Proceedings of 2017 3DTV Conference: The True Vision - Capture, Transmission and Display of 3D Video (3DTV-CON)*, Copenhagen, Denmark, pp. 1-4, June 2017. https://doi.org/10.1109/3DTV.2017.8280421

[2] J. Mütterlein, “The Three Pillars of Virtual Reality? Investigating the Roles of Immersion, Presence, and Interactivity,” in *Proceedings of the 51st Hawaii International Conference on System Sciences (HICSS 2018)*, Waikoloa Village: HI, pp. 1407-1415, January 2018. https://doi.org/10.24251/HICSS.2018.174

[3] J. H. Lee, “VR System Environment Technologies and User Input Elements,” *Journal of the Korean Society of Design Culture*, Vol. 24, No. 2, pp. 585-596, June 2018. https://doi.org/10.18208/ksdc.2018.24.2.585

[4] K. Cheng and P. A. Cairns, “Behaviour, Realism and Immersion in Games,” in *Proceedings of CHI ’05 Extended Abstracts on Human Factors in Computing Systems (CHI EA ’05)*, Portland: OR, pp. 1272-1275, April 2005. https://doi.org/10.1145/1056808.1056894

[5] P. Cairns, A. Cox, and A. I. Nordin, “Immersion in Digital Games: Review of Gaming Experience Research,” in *Handbook of Digital Games*, Hoboken, NJ: Wiley-IEEE Press, ch. 12, pp. 337-361, 2014.

[6] Counter-Strike 2. Introducing Counter Strike 2 [Internet]. Available: https://www.counter-strike.net/cs2?l=english.

[7] M. Müller, B. Heidelberger, M. Hennix, and J. Ratcliff, “Position Based Dynamics,” *Journal of Visual Communication and Image Representation*, Vol. 18, No. 2, pp. 109-118, April 2007. https://doi.org/10.1016/j.jvcir.2007.01.005

[8] T. Pfaff, R. Narain, J. M. de Joya, and J. F. O’Brien, “Adaptive Tearing and Cracking of Thin Sheets,” *ACM Transactions on Graphics*, Vol. 33, No. 4, 110, July 2014. https://doi.org/10.1145/2601097.2601132

[9] N. Chentanez, M. Müller, and T.-Y. Kim, “Coupling 3D Eulerian, Heightfield and Particle Methods for Interactive Simulation of Large Scale Liquid Phenomena,” *IEEE Transactions on Visualization and Computer Graphics*, Vol. 21, No. 10, pp 1116-1128, October 2015. https://doi.org/10.1109/TVCG.2015.2449303

[10] J. Stam, “Stable Fluids,” in *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH ’99)*, Los Angeles: CA, pp. 121-128, August 1999. https://doi.org/10.1145/311535.311548

[11] J. Wahlqvist and P. Rubini, “Real-Time Visualization of Smoke for Fire Safety Engineering Applications,” *Fire Safety Journal*, Vol. 140, 103878, October 2023. https://doi.org/10.1016/j.firesaf.2023.103878

[12] L. Hu, M. Chen, P. X. Liu, and S. Xu, “A Vortex Method of 3D Smoke Simulation for Virtual Surgery,” *Computer Methods and Programs in Biomedicine*, Vol. 198, 105813, January 2021. https://doi.org/10.1016/j.cmpb.2020.105813

[13] S. Sato, Y. Dobashi, and T. Kim, “Stream-Guided Smoke Simulations,” *ACM Transactions on Graphics*, Vol. 40, No. 4, 161, August 2021. https://doi.org/10.1145/3450626.3459846

[14] J. Wen and H. Ma, “Real-Time Smoke Simulation Based on Vorticity Preserving Lattice Boltzmann Method,” *The Visual Computer*, Vol. 35, No. 9, pp. 1279-1292, September 2019. https://doi.org/10.1007/s00371-018-1514-x

[15] Wikipedia. CUDA [Internet]. Available: https://ko.wikipedia.org/wiki/CUDA.

[16] GDCVault. Real-Time Fluid Dynamics for Games [Interner]. Available: https://gdcvault.com/play/1022708/Real-Time-Fluid-Dynamics-for

[17] B. S. Kim and I. S. Kim, “Speed up Power Flow Analysis Calculations with CUDA-based GPU Computing,” in *Proceedings of the 54th Korean Institute of Electrical Engineers Summer Conference*, Pyeongchang, pp. 543-544, July 2023.

[18] I. Puaut, “Real-Time Performance of Dynamic Memory Allocation Algorithms,” in *Proceedings of the 14th Euromicro Conference on Real-Time Systems (Euromicro RTS 2002)*, Vienna, Austria, pp. 41-49, June 2002. https://doi.org/10.1109/EMRTS.2002.1019184

김동민(Dong-Min Kim)

2024년 : 강남대학교 소프트웨어응용학부 (공학사)

2018년～2024년: 강남대학교 소프트웨어응용학부

※관심분야 : 가상현실(Virtual Reality), 물리 시뮬레이션(Physical Simulation), 게임 엔진 등

최웅(Woong Choi)

2000년 : 조선대학교 대학원 (공학석사)

2005년 : Tokyo Institute of Technology (공학박사-지능시스템과학)

2005년～2010년: Ritsumeikan University

2010년～2022년: National Institute of Technology, Gunma College

2022년～현 재: 강남대학교 ICT융합공학부 부교수

※관심분야 : 가상현실(Virtual Reality), 휴먼 컴퓨터 인터랙션(Human Computer Interaction) 등