Journal of Digital Contents Society Vol. 12 No. 2 Jun. 2011(pp. 185-194)

가상 객체를 위한 스테이트챠트 기반의 점진적인 행위 LOD 모델 연구

서진석*, 윤주상**

요 약

본 논문은 게임과 가상현실 시스템을 위한 스테이트챠트(Statecharts) 기반의 점진적인(progressive) 행위 LOD 모델을 소개하고 있다. 시스템의 리소스를 최대한 활용할 수 있도록 상황에 따라 임의의 복잡도를 가진 행위 LOD를 자유롭게 생성할 수 있는 스테이트챠트 기반의 명세 프로세스를 포함하여, 모델 간 개량(refinement) 연산, 모델 간 전환(switching) 법칙, LOD 선택 정책 등을 정의하였다. 더불어, 제안된 접근방법의 가능성을 보여주기 위해, 가상 자동차가 단계적으로 설계되어가면서 점진적인 LOD 모델이 되는 예를 들고 있다.

Study on Statecharts-based Progressive Behavior LOD Model for Virtual Objects

Jinseok Seo*, Joosang Youn**

Abstract

This paper introduces a Statecharts-based progressive behavior LOD model for computer games and virtual reality systems. In order to use computing resources efficiently and generate an LOD model having arbitrary complexity, we defined a progressive behavior LOD model which including a Statecharts-based specification process, refinement operations, a switching policy, and an LOD selection policy. Additionally, in order to show the possibility of the proposed approach, we demonstrate an example of progressive LOD models by illustrating a step-by-step design of a virtual vehicle.

Keywords : Virtual Reality, Game AI, Behavior LOD

1. 서론 및 배경

최근 컴퓨터나 콘솔 게임기 등의 가격 대비 성능비가 우수해져 매우 사실적으로 표현된 3차원 콘텐츠가 대중화 되었다. 반면, 사용자들은 항상 좀 더 사실적이면서도 실시간 상호작용이 가능한 콘텐츠를 원하고 있으며, 개발자들은 제한된 리소스를 최대한 활용하여 성능을 최적화시키기 위해 노력하고 있다.

<그림 1> 3단계로 이루어진 기하 LOD 모델

성능 최적화 문제를 해결하기 위한 가장 대표적인 방법은 LOD(Levels-of-Detail) 기법을 이용하는 것이다. LOD의 기본 아이디어는 중요하지 않은 특정 개체를 덜 상세하게 렌더링(보다 적은 수의 폴리곤으로 구성된 3차원 모델을 사용) 함으로써 시스템 전체의 성능을 향상시키는

※ 제일저자(First Author) : 서진석
접수일:2011년 04월 07일, 수정일:2011년 05월 01일
완료일:2011년 06월 15일
* 동의대학교 게임공학과
jsseo@deu.ac.kr
** 동의대학교 멀티미디어공학과
▣ 이 논문은 2008년도 정부(교육과학기술부)으로 재원로 한국연구재단의 지원을 받아 수행된 기초연구사업임(KRF-2008-331-D00549).

것이다. 그림 1의 경우 3단계의 서로 다른 폴리곤 수를 가진 기하 LOD 모델의 예를 보여주고 있다.

LOD 기법은 기하 모델에서만 적용되는 것은 아니다. 인공지능, 물리 시뮬레이션, 애니메이션 등과 같은 행위 모델에도 적용이 되고 있다. 인공지능의 경우, 덜 중요한 개체를 위해서는 보다 적은 자원을 소모하는 인공지능 기법을 사용하는 대신 시뮬레이션 결과가 중요하거나 오류에 민감한 개체에게 더 많은 자원을 할당하는 방식을 사용한다.

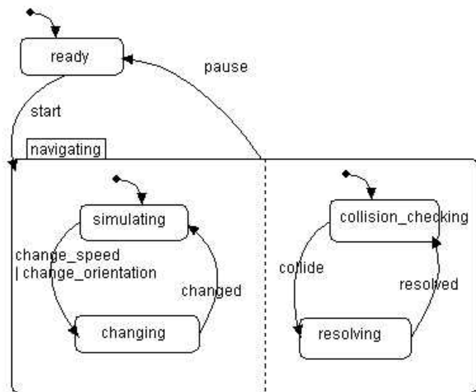
예를 들어, 사용자의 시점(카메라의 위치)으로부터 매우 먼 거리에서 움직이고 있는 자동차의 경우에는 주위의 다른 물체와 충돌하지 않도록 정확하게 도로를 운행할 필요가 없다. 비록 도로와 근접한 물체와 충돌하였다고 하더라도 사용자는 이 사실을 눈치 챌 수가 없기 때문이다.

하지만, 만약 멀리 있던 자동차로 갑자기 가까이 접근하게 되면 예측 불가능하거나, 혹은 어색한 자동차의 행위 시뮬레이션이 수행될 수 있다. 멀리 있던 자동차는 중요하지 않다고 판단되어 근처의 다른 물체와의 충돌 검사를 수행하지 않은 동시에 도로의 궤적도 정확하게 따라가고 있지 않았지만, 사용자가 어느 임계 영역 이내로 접근하게 되면 충돌 검사를 하기 시작하고 도로의 궤적도 정확하게 따라가도록 시뮬레이션 한다. 이 때, 만약 자동차가 이미 다른 물체와 충돌한 상태이거나 (겹쳐져 있는 상태) 도로로부터 비교적 멀리 떨어져 있다면 행위를 시뮬레이션 하는 알고리즘은 자동차를 매우 불안정하게 움직이게 할 가능성이 커진다.

위와 같은 행위 LOD 모델에서의 문제는 기하 LOD 모델에서도 오래전부터 논의되어 왔다. 기하 LOD 기법에서는 이 문제를 "Popping 효과"라고 하며 [1]의 "Progressive meshes"라는 연구로부터 시작되어 다양한 경우와 환경을 위한 연속적인 기하 LOD 모델에 대한 연구 결과물이[2][3][4] 가상현실 시스템이나 게임에 활용되고 있다. Progressive meshes의 기본 아이디어는, 원본 모델이 가장 간단한 LOD로부터 복잡한 모델까지 개량되어지는 연산을 기록해두었다가 임의의 해상도를 가지는 LOD 모델을 자유롭게 사용할 수 있다는 것이다.

기하 모델에서는 메쉬를 구성하는 삼각형의 개수라는 1차원 공간에서 문제를 해결하였지만, 행위 모델은 여러 가지 알고리즘, 기법 등으로 구성되기 때문에 연속적인 복잡도를 갖는 모델을 정의하기 어렵다. 즉, Progressive Meshes에서는 하나의 삼각형이 2개 이상의 삼각형으로 분리되는 연산과 2개 이상의 삼각형이 하나의 삼각형으로 결합되는 연산을 수학적으로 정의하기가 수월하지만, 행위 모델에서의 알고리즘이나 기법은 주로 프로그래밍 언어로 하드코딩 되거나 의사코드로 표현되기 때문에 복잡한 모델로부터 간단한 모델로의(혹은 역으로) 변환을 논리적으로 표현하기가 어렵다.

이에 본 연구에서는 행위 모델간의 변환을 논리적으로 표현할 수 있도록 스테이트챠트(Statecharts)[5] 기반의 접근방법을 사용하고자 한다. 스테이트챠트는 기존의 상태전이도(State-Transition Diagram)의 단점을 보완하고 실시간 반응이 필요한 내장형 시스템을 위한 설계 도구로써 처음 고안되었지만, 가상현실 시스템 및 게임의 특성상 3차원 공간에 존재하는 가상 객체의 행위를 설계하기에도 매우 적합한 도구이다.



<그림 2> 스테이트챠트 기반의 행위 모델

위 그림 2는 가상현실 선박시뮬레이터에서 선박의 행위를 설계한 결과이다. 스테이트챠트를 이용하게 되면 복잡한 행위 모델을 한눈에 쉽게 이해할 수 있으므로 정확하게 설계 되었는지 검증하기에 매우 수월하다. 더불어, 의미(semantics)가 명확하여 명세 결과(specification)를 직접 프로그래밍 코드로 변환하는 수작업을 거치지 않고 직접 실행 가능한 형태로도 활용할

수 있다[6].

2. 선행 연구

2.1 행위 LOD 기법

기하 모델이 아닌 행위 및 시뮬레이션 모델에 LOD를 적용하는 연구는 오래전부터 시작되었으며, 상용화 게임 및 대형 가상현실 시스템에서 활용되고 있다.

Carlson[7]은 하나의 다리로 돌아다니는 가상 객체를 설계하여 동적으로 3단계의 시뮬레이션 LOD를 적용하였을 때 전체적인 성능에 미치는 영향을 보여주었다. 이 연구는 시뮬레이션 LOD의 가능성을 처음 제시하였다는 면에서 의미가 크다.

행위 LOD 기법에 관한 연구는 주로 군집을 이루는 개체를 대상으로 많이 진행되어 왔다. 대규모 가상현실 시스템이나 게임에서는 자동으로 움직이는 에이전트 형태의 개체를 많이 넣으려는 노력을 해왔지만, 제 각각 자연스러운 행위를 시뮬레이션하기에는 언제나 자원의 제한에 부딪혀야 했다.

초기에는 주로 군집을 이루는 개체들 사이의 관계 및 특성에 근거하여 충돌 검사 여부, 애니메이션 활성화 여부 등을 결정하는 연구가 진행되어[8][9][10] 시스템의 전체적인 성능 향상에는 큰 효과가 있는 반면 Popping 효과는 해결하기 어려웠다.

비슷한 연구로, ALOHA(Adaptive Level of Detail for Human Animation)[11] 프레임워크는 군집을 이루는 개체의 행위를 묘사할 때, 중요도에 따라 애니메이션을 구성하는 각 컴포넌트를 활성화 및 비활성화 시키는 방법을 사용하였다. 컴포넌트별로 수행 여부가 결정되기 때문에 세부 애니메이션에서의 popping 효과는 있지만, 전체적인 애니메이션의 입장에서 보았을 때는 [8][9][10]에서의 연구결과에 비하여 자연스러운 LOD 모델 간 전환이 가능해졌다. [12]에서는 ALOHA를 개선하여 보다 여러 단계의 LOD 모델이 가능하도록 컴포넌트들을 좀 더 세부 조각으로 나누는 방법을 사용하였다.

위에서 언급한 군집 개체를 위한 행위 LOD에 관련된 최근의 연구들은 점진적인 메쉬와 같이 임의의 복잡도를 가지는 LOD 모델을 자유롭게 생성할 수는 없지만 비교적 여러 단계의 LOD 모델을 생성하여 시스템 성능을 최적화 하는데 많은 도움을 줄 수 있다. 하지만, 다양한 행위 LOD 모델을 명세하기 위한 방법을 제시하지 못하고 있으며, 보다 체계적인 모델간의 전환 정책이 보완되어야 한다.

군집 개체를 위한 행위 LOD가 아닌 단일 개체에 대한 행위 LOD로는 [13]에서와 같이 복잡한 모양을 갖는 식물의 움직임을 시뮬레이션 하기 위하여 LOD를 적용한 예를 들 수 있다. [13]에서는 여러 단계의 복잡도에 해당하는 LOD 모델을 미리 계산하여 만들어 둔 다음에 모델 간 전환이 일어나도 popping 효과가 없도록 하고 있다. 이는 식물이라는 개체의 특징을 잘 활용한 예로 임의의 개체에 적용하기에는 무리가 있다.

내비게이션 제어 알고리즘에 시뮬레이션 LOD를 적용한 예는 [14]에서 찾아볼 수 있다. [14]도 역시 군집의 경로 따라가기나 무리 짓기 행동의 성능 향상을 가져오기 위한 연구였지만, 위에서 언급한 동일한 개체 군집이 아닌 다양한 개체의 행위에 적용하였다는 점에서 차이가 있다.

이상으로 살펴본 행위 LOD 기법은 복잡도 문제를 해경하기 위해, 단순한 알고리즘을 사용하거나 혹은 특정 애니메이션을 비활성화 시키는 방식을 사용하고 있다. 하지만, 다양한 종류의 많은 객체로 구성된 복잡한 가상현실 시스템이나 게임은 대부분 상태기계를 기반으로 한 행위 엔진이나 인공지능 엔진을 사용한다. 이에 본 연구에서는 상태기계 기반의 행위 모델에 자연스럽게 적용이 가능하도록 스테이트챠트를 활용한 LOD 기법을 소개하고자 한다.

2.2 스테이트챠트 기반의 행위 모델

정형화된 도구를 사용하여 가상 객체의 행위를 설계하기 위한 시도는 오래전부터 시작되었다. 초기에는 주로 자료흐름도(Data-Flow Diagram)를 이용하여 그래픽 객체들 간의 함수적 기능 관계를 정의한 연구가 많이 진행되었고[15][16], 확장된 페트리넷을 이용하여 가상 객체의 행위를 설계 및 시뮬레이션 하는 연구도 있다[17].

스테이트챠트를 이용하여 가상 객체의 행위를 모델링하기 위한 시도는 [18]에서 시작되었다. 스테이트챠트는 기존의 유한상태기계(Finite

State Machine)를 위한 평면적인 상태전이도에 비해 훨씬 복잡한 시스템도 간단하게 설계할 수 있도록 해주는데, 이는 스테이트챠트가 갖는 직교성(orthogonality)과 깊이(depth)라는 특성 덕분이다. 직교성이란 한 번에 활성화 될 수 있는 상태가 2개 이상일 수 있는 특성이며, 이는 그림 2에서 "simulating" 상태와 "changing" 상태로 이루어진 상태전이도, 그리고 "collision_checking" 상태와 "resolving" 상태로 이루어진 상태전이도가 서로 독자적인 상태기계인 것처럼 작동할 수 있다는 것을 말한다. 더불어, 깊이 특성은 "navigating" 상태 내부에 다시 다른 내부 상태전이도를 가질 수 있는 것을 말한다.

[18]에서는 기하 모델과 함께 행위 모델도 가장 간단한 형태부터 복잡한 형태로 개량해가는 과정을 "개량 트리(refinement tree)"로 정의하고 있으며, 전체 시스템의 성능을 최적화하기 위해 적절한 복잡도의 기하 LOD와 행위 LOD를 조합하여 사용하는 예를 보여주고 있다. 하지만, 이 연구에서는 모든 중간 단계의 행위 LOD를 자동 생성해주는 것은 아니고 컴포넌트 형태의 각 LOD를 조합해서 사용하고 있으며, 정형화된 도구를 사용하고 있음에도 불구하고 LOD 간의 명확한 전환 정책을 제시하지는 못하였다.

[18]에서 발전된 연구로 [6]에서는 스테이트챠트로 명세된 가상객체를 직접 프로그래밍 코드로 변환하지 않고 시뮬레이션 할 수 있는 저작도구의 형태를 제시하고 있다. 비록, LOD에 관한 연구는 진행되지 않았지만 [18]의 결과와 함께 본 연구를 시작하는데 있어서 기초가 되는 선행연구라고 할 수 있다.

3. 점진적인 행위 LOD 모델

3.1 스테이트챠트를 이용한 행위 모델의 명세 프로세스

기존의 기하 LOD 기법에서는 각 단계의 LOD 모델을 생성하기 위해 가장 복잡한 모델로부터 간단한 모델로 단순화(simplification) 시켜가는 상향식(bottom-up) 방식을 사용하여 왔다. [6]에서는 이와 반대로 하향식(top-down) 접근방식을 사용하고 있는데, 본 연구에서도 이와 같은 하향식 접근방식을 기초로 하고 있다.

하향식 접근방식은 제일 간단한 행위 모델로부터 시작하여 점진적으로 개량해 나아감으로써 복잡한 최종 모델을 완성하는 것이다. 행위 모델을 개량한다는 것은 기존의 모델에 새로운 기능(feature)을 추가함으로써 그 명세 결과물인 스테이트챠트를 확장 또는 상속한다는 의미로 해석할 수 있다. 이렇게 확장된 새로운 스테이트챠트는 이 전 모델이 충족시키고자 했던 요구사항들을 무시하지 않으면서 자연스럽게 좀 더 복잡한 행위 모델을 명세하게 되는 것이다.

몇몇 연구자들이 스테이트챠트의 상속 또는 확장에 관해 연구해왔는데[19][20], 주로 모델 간 부합성과 대체 가능성 등을 기준으로 하고 있다. 부합성은 확장된 행위 명세에서도 확장되기 전의 행위 명세에서와 동일한 일련의 사건들을 인식하여 동일한 상태로 전이되는가를 말하는 것이고, 대체가능성은 확장된 모델이 기존 모델을 대체할 수 있느냐를 말하는 것이다. 이는 LOD 모델간의 교체 시 현재 상태를 결정하는데 있어서 중요한 근거가 될 뿐만 아니라 모델의 개량 연산을 정의하는 기준이 된다. 실제 본 연구에서도 모델 간 부합성과 대체가능성을 유지할 수 있도록 개량 연산(3.2절)을 정의하였으며, 행위 모델 간 전환 법칙(3.3절)과 LOD 선택 정책(3.4절)을 결정하는 기준으로 삼았다.

$$Behavior_1 = Feature_1 + Behavior_0$$
$$\vdots$$
$$Behavior_n = Feature_n + Behavior_{n-1}$$

<수식 1> 행위 모델의 점진적인 개량

점진적으로 기능을 추가하는 것을 수식으로 표현하면 수식 1과 같다. 최초의 행위 모델(비어 있는 스테이트챠트)은 새로운 기능을 하나 씩 추가할 때 마다 좀 더 복잡한 새로운 행위 모델로 개량된다. 수식 1에서 보면 최초의 행위 모델인 "$Behavior_0$"로부터 새로운 기능들인 "$Feature_1$", "$Feature_2$", ... , "$Feature_n$" 등이 추가되면서 최종 모델인 "$Behavior_n$"이 완성되는 것을 알 수 있다.

$$Feature_n = Op_1 + Op_2 + \cdots + Op_k$$

<수식 2> 새로운 기능의 정의

새로운 기능은 다시 여러 개의 개량 연산으로

정의할 수 있다. 수식 2에서 보면 새로 추가되는 기능인 "Feature_n"은 Op_1, Op_2, ... Op_k 등의 연산의 합으로 정의할 수 있다. 각 연산은 상태 분해 연산, 상태 추가 연산, 전이 추가 연산 등이 될 수 있으며, 자세한 설명은 다음 절(3.2절)에서 하고 있다.

위와 같이 새로운 기능이 추가되면서 만들어지는 중간 단계의 행위 모델은 잠정적인 LOD 모델로서의 역할을 할 수 있게 된다. 하지만 모든 중간 단계의 모델이 LOD로서 활용되는 것은 아니고 성능 평가를 통해 활용 가치가 있는지 판단하게 된다. 성능 평가에서는 각 객체가 시뮬레이션 되고 렌더링 되는 시간을 구분하여 측정하게 되며, 이 중 시뮬레이션 시간의 평균 시간을 계산하여 각 LOD 모델의 성능 지표로 삼는다. 그림 3은 선행 연구인 [6]의 예를 보여주고 있는데, 3차원 가상 공간을 구성하는 장면 그래프의 특정 노드(하나의 객체 혹은 서브 트리)의 성능을 평가하는 장면이다.

Form | Functions | Behavior | Performance

statistics : True

	current	avg.	min	max
tri.(object)	590	N/A	N/A	N/A
update(object	66.43	35.77	0.15	99.38
draw(object)	12.83	29.33	0.0	100.0
collision(objec	99.96	63.15	0.0	100.0
tri.(tree)	1810	N/A	N/A	N/A
update(tree)	75.35	43.09	0.15	99.57
draw(tree)	100.0	100.0	17.21	100.0
collision(tree)	100.0	79.17	0.0	100.0

kernel:pvKernel
classes
input devices
user:pvUser
world:pvWorld
othership:pvObject
ducks:pvObject
duck1:pvObject
duck2:pvObject

<그림 3> 가상객체의 성능을 평가하는 화면

3.2 행위 모델의 개량(refinement) 연산

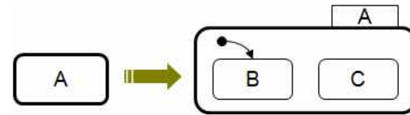
본 절에서는 행위 모델의 점진적인 개량에 필요한 연산을 정리하고 있다. 연산에는 크게 상태 분해(decomposition) 연산, 상태 추가(addition) 연산, 전이 추가 연산, 전이 조건 변화, 상태 명세 변화 등으로 나눌 수 있다.

3.2.1 상태 분해 연산

상태 분해 연산은 한 상태가 내부에 다른 상태 2개 이상을 갖게 되면서 계층 구조가 되는 것을 의미한다. 분해 연산은 다시 다음과 같이 'OR 상태로의 분해 연산'과 'AND 상태로의 분해 연산'으로 구분할 수 있다.

- OR 상태로의 분해 연산

A 상태 내부에 자식 상태인 B와 C 2개의 상태를 갖는 형태로 분해되는 연산이다. 기본적으로 A 상태가 활성화 되면 내부 디폴트 상태인 B도 활성화된다. B와 C 상태간의 전이는 자동으로 추가되지 않았는데, 이는 전이의 조건이 명시되지 않았을 경우엔 다음번 시뮬레이션 프레임에 강제로 전이가 발생하기 때문이다.

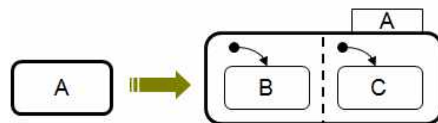


<그림 4> OR 상태로의 분해 연산

이 연산은 A라는 상태를 위한 자료구조의 자식 노드로 B와 C를 모두 포함할 수 있는 상태머신(본 연구에서는 STM이라는 클래스를 정의함)을 추가하고, 다시 새로 추가된 상태머신에 B와 C라는 2개의 상태 노드를 추가함으로써 구현되었다.

연산 적용의 예는 그림 10에서 확인할 수 있다. 처음에는 "collision_checking" 상태가 내부에 별도의 상태들을 갖지 않았지만, "OR 상태로의 분해 연산"에 의해 내부에 "waiting"과 "checking" 상태를 갖게 된 것을 볼 수 있다.

- AND 상태로의 분해 연산



<그림 5> AND 상태로의 분해 연산

A 상태 내부에 자식 상태인 B와 C 2개의 상태를 갖는 형태로 분해되는 점은 위의 연산과 동일하지만, B와 C 상태가 서로 병렬 구조가 되는 점이 다르다. 이 경우 A 상태가 활성화 되면 B와 C가 병렬 구조이므로 각각이 디폴트 상태가 되어 모두 활성화 된다.

이 연산은 A라는 상태를 위한 자료 구조의 자식 노드로 2개의 새로운 상태머신을 추가하고, 각각의 추가된 상태머신에 B와 C라는 상태 노드가 다시 자식 노드로 하나씩 추가됨으로써 구현되었다.

“AND 상태로의 분해 연산”은 주로 어떤 상태 내부에서 병렬로 두 가지 이상의 행위를 수행하고자 할 때 적용한다. 그림 9에서 보면 원래 ″moving“ 상태는 자동차를 움직였다가 멈추었다가를 반복하면서 애니메이션을 수행하지만, 그림 10에서와 같이 ″rotating″과 ″forwarding″ 상태로 분해되어 회전 움직임과 직진 움직임을 동시에 수행하게 된다.

3.2.2 상태 추가 연산

상태 추가 연산은 자식 상태가 아닌 형제 상태(sibling state)를 더하는 연산이다. 분해 연산과 마찬가지로 다음과 같이 2가지 연산의 예를 들 수 있다.

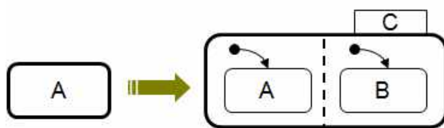
- OR 상태의 추가 연산

가장 간단하면서도 빈번하게 사용될 연산중에 하나로 새로운 상태 B가 형제 상태로 추가되는 것이다. 그림에서 볼 수 있듯이 이 연산은 A 상태에만 한정된다고 볼 수 없고 A가 이미 형제관계에 있는 다른 상태에도 같은 연산이 적용된다는 것을 알 수 있다. 자료구조로 보면 이 연산은 A를 가지고 있던 상위 상태머신의 노드에 B라는 상태 노드가 추가된다.

이 연산의 예는 그림 10과 11에서 ″collision_checking″ 상태의 내부에 ″backwarding″ 상태가 추가되는 것으로부터 찾아볼 수 있다. 그림 10에서는 충돌이 발생하면 멈추었다가 기다린 후 충돌이 해소되었을 때 다시 움직이게 되어 있지만, 그림 11에서는 후진 운동을 추가하여 보다 적극적으로 충돌을 회피하도록 설계되었다.

<그림 6> OR 상태의 추가 연산

- AND 상태의 추가 연산



<그림 7> AND 상태의 추가 연산

A 상태의 형제 상태인 B 상태가 병렬 구조로 추가되는 연산이다. 이 경우, 아무리 간단한 병렬 구조라도 A, B와 같은 상태를 자식 상태로 포함하게 될 C라는 부모 상태가 추가되어야 한다. 더불어, 원래 존재 했던 (A, B, C 상태를 제외한) 다른 상태로부터 A상태로의 전이나, A상태로부터 다른 상태로의 전이는 대상이 C로 변경되어야 한다.

자료구조상으로는 원래 A 상태를 자식 노드로 가지고 있던 상태머신에 A 대신 C 상태로 대체하게 되며, 동시에 전이의 속성도 모두 A로부터 C로 변경된다.

3.2.3 전이 추가 연산

2개의 상태 사이에 기존에는 없던 새로운 전이가 추가되는 연산이다. 전이는 출발지, 목적지, 조건 등과 같이 3가지 속성을 지니고 있으며 본 연구에서는 같은 형제 상태끼리만 전이가 정의될 수 있도록 한정하였다.

전이는 상태머신 클래스가 리스트 형태로 관리하며 새로운 전이의 추가는 이 리스트에 원소를 추가함으로써 구현되었다.

3.2.4 전이 조건 변화 연산

전이 조건 변화 연산은 전이의 속성인 출발지, 목적지, 조건 등이 변경되는 연산이다. 전이의 리스트를 지니고 있는 상태머신 자료구조에서 해당 전이를 찾아 속성을 변경한다. 이와 같이 전이는 속성이 변경될 수 있으므로 LOD 모델별로 서로 다른 속성을 지닐 수 있도록 테이블로 전이의 변경 이력을 계속 보유하도록 하였다.

3.2.5 상태 명세 변화 연산

상태 명세는 OnEnter, OnDuring, OnExit과 같이 3개의 속성을 지닌다. 각 속성은 스크립트 언어로 객체가 수행해야 할 세부 명령을 정의한다. OnEnter는 해당 상태가 활성화되었을 때 한번 수행되고, OnDuring은 해당 상태가 활성화 상태이면 계속 수행되면, OnExit은 다른 상태로 전이가 발생할 때에만 한번 수행된다. 위의 전이 조건 변화 연산과 마찬가지로 상태 명세도 LOD 모델 별로 서로 다른 속성을 지닐 수 있으므로 테이블로 명세 변화 이력을 계속 보유하도록 하였다.

3.3 행위 모델 간 전환 법칙

전이 추가 연산은 모델 간 전환 시에 특별한 처리를 하지 않아도 무관하고, 전이 조건변화 연산과 상태 명세 변화 연산은 각 테이블에서 해당 조건 및 명세를 불러오면 되므로 큰 문제가 되지 않는다. 단, 상태 명세 변화 연산은 다음과 같이 특별한 경우에는 전환 시에 두 상태의 어느 속성(OnEtner, OnDuring, OnExit)에 해당하는 스크립트를 수행해야 할 지 결정해야 한다.

3.3.1 전환 전 상태의 OnExit 수행 여부

기본적으로 모델 간 전환이 이루어진다고 해서 반드시 OnExit을 수행해야할 필요는 거의 없다. 하지만, 특별한 경우에는 OnExit을 수행해야 어색한 결과를 피할 수 있는 행위 모델이 존재할 수 있는데, 이 경우에는 행위 모델을 명세할 때 개발자가 결정하도록 해주었다.

예를 들어 그림 11의 "backwarding" 상태는 후진 기능이 없는 모델로 갑자기 전환 될 경우 계속 후진 운동을 유지할 위험성이 있기 때문에, 반드시 OnExit(후진 운동을 멈춤)을 수행해야 한다.

3.3.2 전환 후 상태의 OnEnter 수행 여부

디폴트로는 전환 후 상태의 OnDuring부터 수행되어야 하는 것이 보통이나, 위와 마찬가지로 반드시 OnEnter부터 수행되어야만 하는 경우가 존재할 수 있다. 이 경우에도 개발자가 명세단계에서 명시해야 한다.

이 예는 그림 10과 11의 "rotating"과 "forwarding"에서 찾을 수 있다. 그림 10의 모델에서 11로 전환이 이루어질 때, 그림 11의 "rotating"과 "forwarding"에서는 가속, 정속, 감속 순으로 애니메이션을 수행하므로 OnEnter(멈춤 상태에서 가속을 시작)부터 수행해야 한다.

3.3.3 전환 법칙

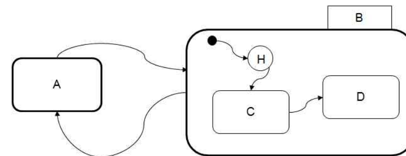
위의 두 연산과는 달리 두 개의 상태 분해 연산 및 두 개의 상태 추가 연산은(총 네 개의 연산) 전환될 때 활성화할 상태를 정의해야 하는데, 각각 복잡한 모델로 전환될 때의 전환법칙(switching-down rule)과 간단한 모델로 전환될 때의 법칙(switching-up rule)을 모두 정의하였다.

<표 1> 개량 연산 별로 본 모델간 전환 법칙

연산	switching-down	switching-up
OR 분해	1) A → A and B 2) A → A and C	1) A and B → A 2) A and C → A
AND 분해	1) A → A and B and C	1) A and B and C → A
OR 추가	1) A → A 2) A → B	1) A → A 2) B → A
AND 추가	1) A → A and B and C	1) A and B and C → A

표 1에서 볼 수 있듯이 switching-up 법칙은 복잡한 모델에서 간단한 모델로 전환되는 것이기 때문에 활성화될 상태를 결정하는 것은 매우 간단하다. 하지만, OR 상태가 수반되는 switching-down 법칙은 원래는 없던 상태가 추가되는 경우기 때문에 활성화될 상태를 선택할 때 신중해야 한다. 예를 들어, "OR 상태로의 분해 연산"의 경우엔 B를 활성화 시킬지 C를 활성화 시킬지 결정해야 한다. 본 연구에서는 다음과 같이 2가지 기준에 대하여 고찰해보았다.

- 히스토리에 근거한 결정



<그림 8> 스테이트챠트의 enter-by-history

스테이트챠트의 기능 중에 "히스토리에 의한 상태 입장(enter-by-history)"이 있다. 이는 여러 상태의 계층 구조로 이루어진 부모 상태로 전이가 발생할 때 자식 상태 중 활성화할 상태를 무조건 디폴트 상태로만 하지 않는다는 것이다. 위의 그림 8에서 보면 상태 A에서 B로 전이가 발생할 때 기존에는 디폴트 상태인 C가 활성화 되지만 히스토리 기능을 사용하게 되면 "가장 최근에 활성화 되었던 상태"로 전이가 발생하게 된다. 예를 들어, B 상태가 활성화 되었을 때 동시에 D가 활성화 상태였다면, A 상태로 전이가 된 후 다시 B 상태로 돌아올 때 디폴트 상태 C가 아닌 D가 활성화 되는 것을 말한다.

서로 다른 복잡도를 가지는 LOD 모델 간 전환

이 빈번할 때에는 위와 같이 히스토리에 근거한 활성화 상태 결정이 가장 자연스러운 선택이 될 수도 있다. 그렇지만, 히스토리 기능을 모든 전환 법칙에 적용하여서는 안 된다. 대표적인 이유로, 특정 행위는 반드시 다른 행위 뒤에 수행되어야만 하는 경우를 들 수 있다.

- 우선순위에 근거한 결정

기하 모델의 경우 개체별로 우선순위를 두어 특정 개체는 사용자의 시점에서 멀리 떨어져 있더라도 복잡한 모델을 사용하는 방법이 있으며, 혹은 여러 가지 기준에 의하여 메쉬의 단순화를 수행할지 여부를 결정하게 된다. 기준의 예로는 "거리가 멀더라도 화면의 중앙에 위치할 때는 단순화를 하지 않는다.", "모델의 가장자리 부분은 단순화를 하지 않는다." 등이 있다.

3.4 LOD 선택 정책

LOD 선택 정책은 시스템 전체의 성능과 직결된다. 여러 개체 중 어떤 것을 좀 더 간단한 LOD로 전환할지를 결정해야 하고, 한 개체 내에서도 어떤 LOD 모델을 사용해야 할지 결정해야 한다. 개체의 선택 기준은 다음과 같이 3가지를 마련하였다.

- 거리에 따른 선택

대체로 카메라로부터 멀리 떨어진 개체일수록 덜 중요하다고 판단할 수 있다. 시스템의 성능을 높이기 위해서 카메라로부터 가장 멀리 있는 개체부터 선택하여 보다 덜 복잡한 LOD 모델을 선택할 수 있다.

- 크기에 따른 선택

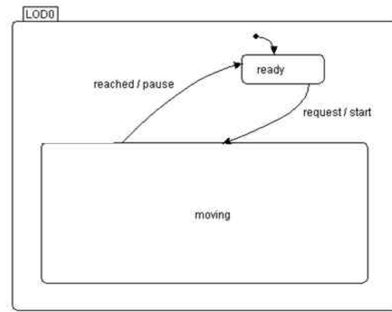
위의 거리에 따른 선택이 항상 옳지는 않다. 멀리 있더라도 화면에 렌더링 되었을 때 차지하는 면적이 넓으면 함부로 덜 복잡한 LOD 모델을 선택하여서는 안 된다. 거리에 따른 선택을 하되 크기가 크면 다른 개체를 선택하도록 하여야 한다.

- 중요도에 따른 선택

아무리 카메라로부터 먼 거리에 위치하고 크기가 작다고 하더라도 콘텐츠의 성격상 매우 중요한 역할을 하는 개체가 존재할 수 있다. 이 경우 개발자가 개체별로 서로 다른 중요도를 부여할 수 있도록 하여 매우 중요한 역할을 하는 개체는 가장 나중에 고려대상으로 삼아야 한다.

4. 가상 자동차의 LOD

자동차의 초기 행위 모델은 그림 9와 같이 설계하였다. 이 모델에서의 자동차는 주어진 목적지까지 정속도록 이동하며, 도착하면 멈추었다가 다시 회전하는 식의 간단한 행위를 지닌다. 또한, 다른 자동차와의 충돌도 무시하였다.



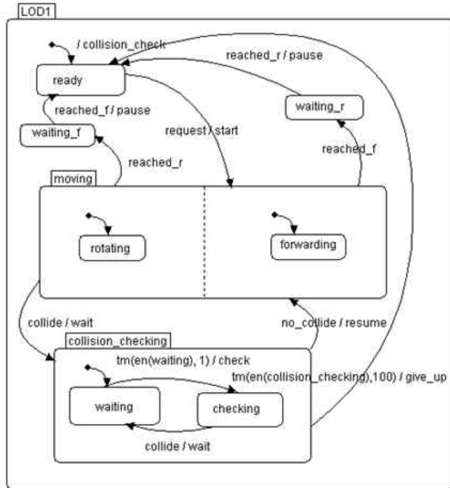
<그림 9> 자동차의 초기 행위 모델

다음 단계에서는 우선 충돌검사 기능(Feature$_1$)을 추가하였는데, 충돌이 감지되면 멈추었다가 일정 시간(1단위의 프레임타임)동안 기다린 후, 다시 충돌검사를 되풀이하고, 100단위의 프레임타임이 흐른 뒤에도 되풀이 된다면 포기하고 다음 목적지로 이동하는 식이다. 이렇게 새롭게 추가된 기능인 "Feature$_1$"은 그림 10에서 확인 해보면 "collision_checking"이라는 상태가 추가된 것을 알 수 있다. 더불어, "collision_checking" 상태는 다시 "OR 상태로의 분해 연산"에 의해 내부에 "waiting"과 "checking" 상태를 갖게 되었다.

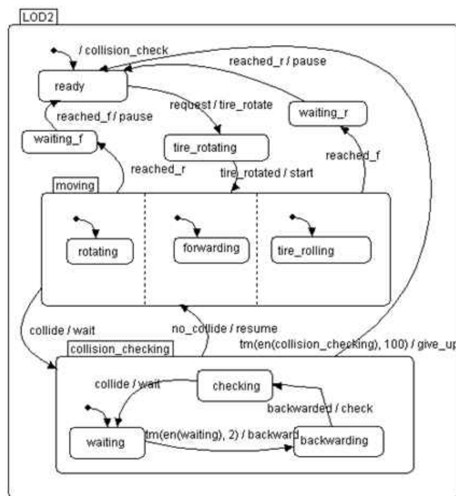
다음에 추가되는 기능인 "Feature$_2$"는 회전 움직임과 직진 움직임의 분리이다. 초기의 끊임이 있는 움직임과는 달리 부드러운 회전 운동으로 변경하였다. "Feature$_2$"는 그림 10에서와 같이 "moving" 상태가 "AND 상태로의 분해 연산"에 의해 내부에 "rotating"과 "forwarding" 상태로 분해되는 것으로 정의된다.

위와 같은 방법으로 기능을 계속 추가하여 최종 모델(그림 11)이 완성되는데, 그림 10으로부터 추가된 기능은 바퀴의 움직임(Feature$_3$), 회전 운동의 개선(Feature$_4$), 직인 운동의 개선(Feature$_5$), 충돌 회피 운동(Feature$_6$) 등이다. 그림 11에서 생략된

부분은 "Feature4"와 "Feature5"로 각각은 "rotating"상태와 "forwarding" 상태가 "OR 상태로의 분해 연산"에 의해 내부 상태전이도를 갖게 되면서 개량된다. 이 두 개의 기능이 추가되기 전에는 정속도로 갑자기 움직이기 시작했다가 멈추는 식으로 애니메이션이 수행되었지만, 개량된 후에는 가속, 정속, 감속의 순으로 부드러운 애니메이션을 수행하게 된다.



<그림 10> 중간 단계의 행위 모델



<그림 11> 완성된 행위 모델 ("rotating"과 "forwarding"의 내부 상태전이도는 생략)

5. 결론

최근 3차원 그래픽 관련 SW 및 HW 발전으로 매우 사실적인 3차원 영상을 렌더링 할 수 있는 시스템이 대중화되었다. 대신, 복잡한 인공지능 테크닉이나 물리 시뮬레이션으로 인하여 CPU 연산에 병목현상이 발생하는 경우가 생겨, 물리 연산만을 위한 가속기도 등장하고 있다. 점진적인 행위 LOD 기법을 활용하게 되면 제한된 자원을 최대한 활용하면서도 사실적인 행위 시뮬레이션이 가능한 응용프로그램 개발이 가능해질 것으로 기대된다.

더불어, 본래 본 연구는 점진적인 행위 LOD의 가능성을 살펴보는 것이 주목적이지만, 3차원 기하 LOD 기법과의 통합을 통해 보다 완전한 LOD 기법의 완성을 꾀할 수 있다. 이 경우엔 [6]에서 언급한 바와 같이 기하 모델과 행위 모델과의 호환성(compatibility) 문제를 해결하는 것이 가장 중요하다.

참 고 문 헌

[1] H. Hoppe, "Progressive meshes," Proceedings of the 23rd annual conference on Computer Graphics and interactive techniques, pp. 99-108, 1996.

[2] H. Hoppe, "Smooth View-Dependent Level-of-Detail Control and Its Application to Terrain Rendering," Ninth IEEE Visualization, pp. 35-42, 1998.

[3] C. Prince, "Progressive Meshes for Large Models of Arbitrary Topology," Master's thesis, University of Washington, 2000.

[4] J. Kim, S. Lee, and L. Kobbelt, "View-dependent Streaming of Progressive Meshies," Shape Modeling International 2004, pp. 209-220, 2004.

[5] D. Harel, On Visual Formalism, Communication of ACM, 31(5), pp. 514-530, 1988.

[6] J. Seo, "A Structured Methodology for Virtual Reality Systems Development and Interactive Support Tools", PhD thesis, Postech, 2005.

[7] D. A. Carlson and J. K. Hodgins, "Simulation Levels of Detail for Real-time Animation," Proceedings of Graphics Interface 1997, pp. 1-8, 1997.

[8] O. C. Cordeiro et al, "Concurrency on social forces simulation model," First International Workshop on Crowd Simulation, 2005.

[9] N. Pelechano et al, "Crowd simulation incorporating agent psychological models, roles and communication," First International Workshop on Crowd Simulati

on, 2005.
[10] S. R. Musse, and D. Thalmann, "A model of human crowd behavior: Group inter-relationship and collisi on detection analysis," Computer Animation and Si mulation, 1997.
[11] C. O'Sullivan et al., "Levels of Detail for Crowds and Groups," Computer Graphics Forum, 21(4), pp. 733-741, 2002.
[12] C. Brom, O. Sery, and T. Poch, "Simulation Level of Detail for Virtual Humans," Lecture Notes in Com puter Science, 4277, 1-14, 2007.
[13] J. Beaudoin and J. Keyser, "Simulation levels of det ail for plant motion," Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation, pp. 297-304, 2004.
[14] D. C. Brogan and J. K. Hodgins, "Simulation Level of Detail for Multiagent Control," Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part1, pp. 199-206, 2002.
[15] C. Gaskell and R. Phillips, "Software architecture of the executable graphical specification tool EGS," □Software-Concepts and Tools, 16(3), pp. 124-135, 1995.
[16] Y. Adachi, T. Kumano, and K. Ogino, "Intermediate representation for stiff virtual objects," Proceedings of IEEE Virtual Reality Annual International Sympo sium, pp. 203-210, 1995.
[17] J. Willians and M. M. Harrison, "A toolset supporte d approach for designing and testing virtual environ ment interaction techniques," International Journal of Human-Computer Studies, 55(2), pp. 145-165, 20 00.
[18] J. Seo, G. J. Kim, "Levels of Detail(LOD) Engineerin g of VR Objects," ACM Virtual Reality Systems an d Technology Conference (VRST99), pp. 104-110, 1999.
[19] S. Cook and J. Daniels, Designing Object Systems: Object-Oriented Modeling with Syntropy, New Yor k: Prentice Hall, 1994.
[20] D. Harel and O. Kupferman, "On the inheritance of state-based object behavior," tech. rep., the Belfer Institute of Mathematics and Computer Science, 199 9.

서 진석

2000년 : 포스텍 대학원 (공학석사)
2005년 : 포스텍 대학원 (공학박사)
2005년～현 재: 동의대학교 게임공학과 조교수
관심분야 : 가상현실, 증강현실, 컴퓨터 게임, 저작도구

윤 주상

2003년 : 고려대학교 대학원 (공학석사)
2008년 : 고려대학교 대학원 (공학박사)
2008년～현 재: 동의대학교 멀티미디어공학과 조교수
관심분야 : 무선 멀티미디어 전송, 모바일 애드혹 네트워크, 유비쿼터스 컴퓨팅, 무선 센서 네트워크