Game Software Design

305900
2008년 가을학기
10/9/2008
박경신

Overall Game Loop

- 전체적인 게임 프로그램 루프
  a. 게임 인트로와 인터페이스
  b. 게임 주 메뉴 인터페이스 – 예, 무기선정 같은 옵션 선택, 등
  c. 게임 레벨 시작과 게임 객체 로딩 (loading)
  d. 게임 루프 (game loop)
     i. 실제 게임 플레이에 필요한 모든 내용을 다룬다 – 가장 어려운 부분!
     ii. 만약 플레이어가 이기면, 보상을 주고 다시 goto b.
     iii. 만약 플레이어가 지면, 실패를 알려주고 이 때 만약 플레이어가 포기하면 goto a. 만약 플레이어가 다시 더 하길 원하면 goto b.

게임 루프에서 실제 게임 플레이에 필요한 모든 내용을 다루는 부분은..

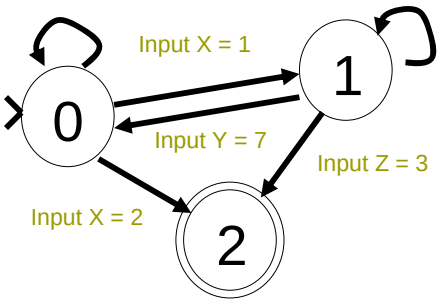
- 여러 가지 방법으로 게임플레이 구현을 할 수 있으나...
- 게임 루프에 기본적인 요소 3가지:
  - 유한상태기계 구현 (Building Finite State Machines)
  - 시뮬레이션의 항상성 유지 (Maintaining Simulation Constancy in a Game Loop)
  - 다중 스레드 게임 루프 (Multi-Threaded Game Loops)

게임 루프에 반드시 구현되어야 할 것들

- 네트워크로 들어온 데이터를 포함하여 사용자의 입력 처리
- 사용자의 입력을 바탕으로 사용자의 변인을 계산 (예, 사용자가 up arrow key를 눌렀을 때 게임 세상에서 앞으로 전진; 사용자가 벽에 부딪히게 될 상황; 등등)
- NPC (Non-player Character) AI (Artificial Intelligence) 계산
- 그래픽 처리
- 사운드 이펙트 처리

Finite State Machines는 이산수학에서 배우는 쓸모 없는 것이 아니다

- 유한상태기계 (Finite State Machines, FSMs)은 게임에서 가장 흔히 쓰이는 프로그램 구조 중 하나이다
- Quake 게임은 하나의 거대한 FSM이다
- 하나의 FSM은
  - 상태 (States)
  - 입력 (Inputs)
  - 전이 (Transitions)

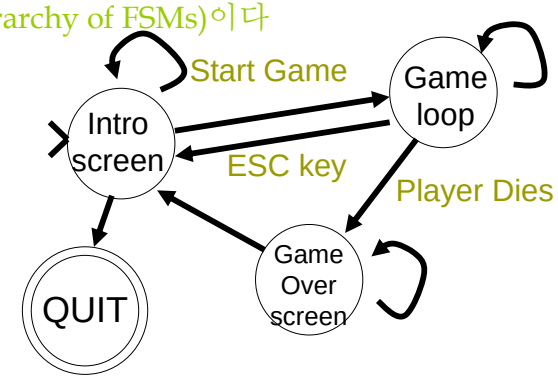


Finite State Machines 요약

- FSMs은 정보처리기계 (information processing machine) input signal set을 받아서 output signal을 생성하는 장치
- FSMs의 상태 (state)는 기계에 대해 발생한 과거 사건들의 요약 (예, 자동판매기에 투입된 동전의 액수)
- M = (S, s_0, I, O, f, g)
  - S – a finite set of states, S = {s_0, s_1, …, s_n}
  - s0 – initial state
  - I – a finite set of input, I = {i_0, i_1, …, i_n}
  - O – a finite set of output, O = {o_0, o_1, .., o_n}
  - f – S x I -> S 상태 전이 함수 (state transition function)
  - g – S -> O 출력 함수 (output function)
- 유한상태기계 M1==M2 (즉, Equivalent):
  - 동일한 입력 sequence가 주어졌을 때, 동일한 출력 sequence를 만들면 M1과 M2 는 equivalent하다고 한다 (기계의 내부 구조는 다르더라도 결과가 같으면 된다)

FSMs for Game Programming

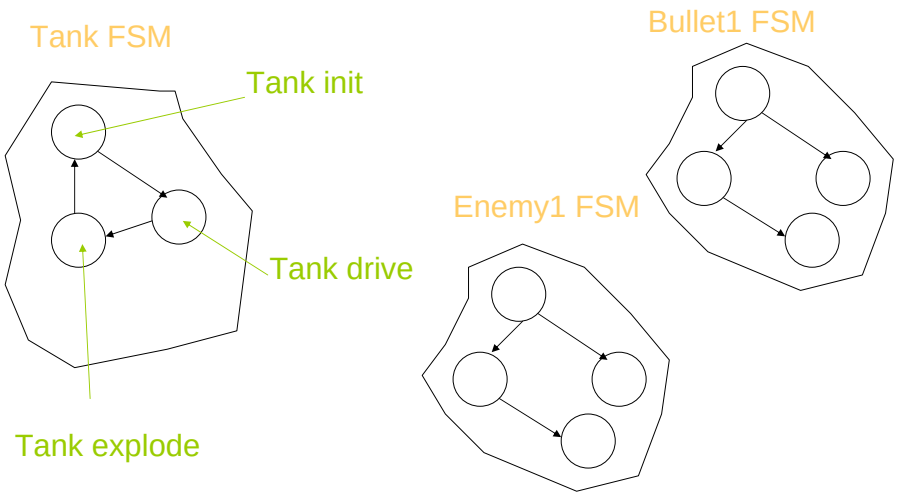
- 게임 전체는 하나의 FSM이다
- 게임의 각 단계는 하나의 FSM이다
- 게임의 각 단계에 각 객체는 하나의 FSM이다
- 그러므로, 게임은 전체적으로 FSMs의 계층구조 (a hierarchy of FSMs)이다



경고

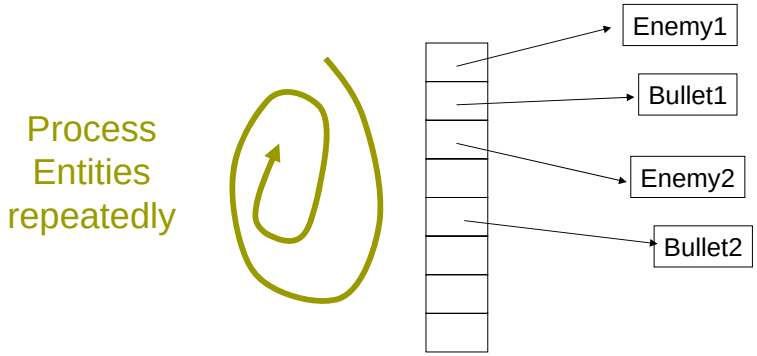
만약 게임이 FSMs의 계층구조로
처음부터 설계되지 않았다면
나중에 게임이 점점 더 복잡해지면서
새로운 기능을 요구할 때 그런 기능을
추가하는 것이 매우 어려워진다.

Each object / entity in the game loop (e.g. Tank or Bullet) contains within itself, a FSM



Consider the Game Loop

- 게임 루프는 현재 게임 세상에 보여지고 있고 처리를 필요로 하는 하나의 object / entity 배열이라고 본다



Multiple Arrays for Groups of Entities (e.g. Tanks and Bullets)

enemyArray

Enemy1
Enemy2
Enemy3
Enemy4

bulletArray

Bullet1
Bullet2
Bullet3
Bullet4

배열 (arrays)을 사용해서 게임 프로그램 실행시간 중에 할당 및 해제하지 않도록 한다. 이는 프로그램이 더 이상 할당 못하는 상황을 만들지 않지 않도록 하기 위해서이다.

Game Loop:

```
While (not exit)
{
    // Go thru enemyArray and process enemies (some may be dormant)
    Call HandleEnemies() ;

    // Do same for bulletArray
    Call HandleBullets()

    Call HandleMyTank()
}
```


Data Structure & Member Functions for an FSM

```
Class FSM {
    currentState  Usually an enum type
    Input1
    Input2        } Single value variable or queue of messages
    Input3
    Process()     Perform all the work of the state machine
};
```

Process()

- Switch (currentState):
  - Case State1:
    - 입력 또는 입력큐의 메시지가 현 상태와 관련 있는지 점검한다
    - 만약 그렇다면, (아마도 상태 변화와 같은) 처리를 한다
    - 그렇지 않다면, Break
  - Case State2:
    - 등등..

E.g. Bullet in BZ

- DormantState: // Bullet is dormant
  - Hide particles
  - Stay in this state until it receives the activation input then set currentState = InitState
  - Break;
- InitState: // Activate the bullet
  - Init bullet position; Show it on the screen
  - currentState = MoveState
  - Break;
- MoveState: // Move bullet
  - Move bullet along trajectory
  - Check if collided with an object
  - If collided:
    - If object == tank then tank.input1="hit" // Tell tank that it is hit so that tank's FSM can deal with it.
    - currentState = BulletExplodeState
  - Break;
- BulletExplodeState: // Start explosion effect on bullet
  - Hide the bullet
  - Enable particle system explosion
  - currentState = WaitForExplosionState
  - Break;
- WaitForExplosionState: // Wait till particle explosion is over
  - explosionCounter++;
  - If explosionCounter = 100 then explosion is over; currentState = DormantState

Maintaining Simulation Constancy in a Game Loop

- 문제: 컴퓨터의 CPU속도에 상관없이 게임 장면의 탱크 또는 자동차가 같은 스피드로 움직여야 한다
- 특히 Non-threaded (그래서, 매 프레임마다 입력, 계산, 그리기를 순서대로 다 처리해야 하는) 게임 루프를 가지고 있다면 더욱 중요하다
- 과거 컴퓨터 게임은 이런 문제를 그냥 무시하고 만들었다 – 즉, 컴퓨터의 파워에 따른 처리를 하지 않았다
- 그래서 예를 들어 장면 속에 자동차가 움직이는 경우, 계산은 간단히 다음과 같이 처리했다
  - PosX = PosX + some_unit_distance
  - 이 중에 자동차가 빨리 움직이는 것들은 some_unit_distance를 크게 만들어 준다

What You Should Do Instead

- 게임 루프를 한 번씩 실행시킬 때마다 어느 정도의 시간을 필요로 하게 되는데 그 경과 시간 (elapsed time)을 dt라고 하자
- dt는 게임의 엔터티의 다음 상태가 어떻게 될 지 파악하기 위해 필요하다
- 예로, 자동차가 30-feet per second으로 움직인다고 하자
- 만약 게임 루프가 dt 시간만큼 처리를 요한다면, 게임에서 자동차의 다음 위치는
  - posX = posX + (speedX * dt)
  - posY = posY + (speedY * dt)
  - posZ = posZ + (speedZ * dt)
- Electro에서는 do_timer(dt) callback을 사용한다

Multi-Threaded Game Loops

- 만약 게임 루프가 원하는 FRAME RATE을 유지하기 위해 충분히 빠르게 돌고 있다면 Tweening (i.e., in-betweening)이 괜찮다
- 그러나 가끔 AI 시스템이 매우 복잡해 질 수도 있고 그러면 계산 시간이 오래 걸리게 된다 – 예를 들어, 침공하는 군사를 위해 보다 수준 높은 계획을 세워야 할 지능적인 인공지능 시스템 같은 경우
- 그런데 게임 루프에 느린 구성요소들로 인해 게임 전체가 느려지게 될 수 있기 때문에, 게임은 하나의 게임 루프를 가질 수 없다
- 게다가 현대 게임 시스템은 여러 개 CPU를 가지고 병렬 처리도 가능하다

트위닝 http://en.wikipedia.org/wiki/Tweening

Multi-Threaded Game Loops

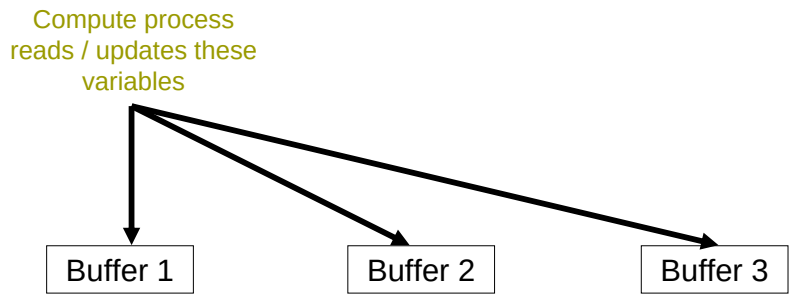
- 그러므로, 다음 루프를 위한 Multiple Threads 또는 Processes를 필요로 한다
  - Input Loop
  - Compute Loop
  - Draw Loop
  - Sound Loop
- 그리고 각 루프마다 개별적으로 또한 가능한 빠르게 처리하길 바란다
- 예를 들어, SPACEBAR를 눌러서 총을 쏘고자 하는 것에서는 사운드 루프에게 총 쏘는 소리를 내라고 지시하고 그것이 알아서 처리하도록 한다 (그래서 게임 계산으로 돌아갈 수 있도록 한다)
- 즉, 운영체제에게 일정한 간격으로 문맥 전환하도록 해서 게임 프로그램은 언제나 같은 속도로 운영되는 것처럼 보이도록 한다

Sharing Variables Efficiently

- 전역변수는 스레드들 간에 공유가 가능 하도록 해야 한다
- Forked processes의 변수는 그 프로세스의 지역 변수로 쓴다 - 그래서 그 변수를 공유해야 할 때는 공유 메모리 (shared memory) 라이브러리를 사용하도록 한다
- Threading과 Forking을 사용하되, 하나의 스레드에 변수 값을 바꾸고 다른 스레드에서 그 변수를 사용하는 그런 용도로 만들진 않도록 한다
- 공유하는 변수에는 MUTEXes 를 사용해야 한다
- 그러나 우리는 프로그램의 모든 변수마다 Mutexes를 만들기 원하지 않는다 – 왜냐하면 프로그램 전체가 느려질 수 있으니까 (due to possible blocks in mutexes)
- 해법: TRIPLE BUFFERING

Triple Buffering

- Init Step – 계산 프로세스는 공유해야 할 변수를 3개의 다른 버퍼에 복사한다



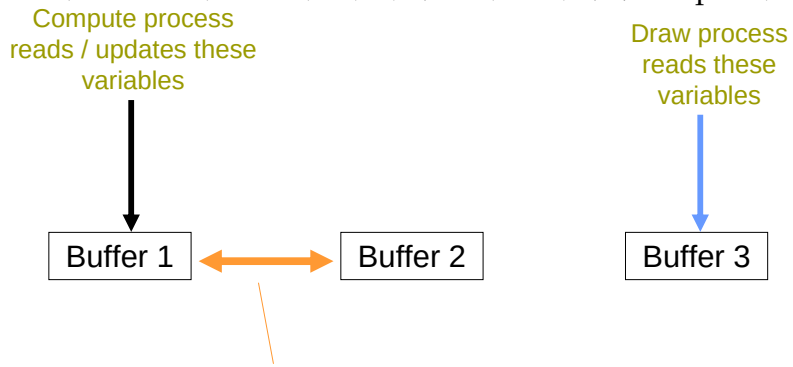
Triple Buffering

- 계산 프로세스와 그리기 프로세스는 각자 개별적으로 복사된 변수를 사용한다

NOTE: You should only triple buffer variables that you expect to share with more than 1 thread/process- obviously.

Triple Buffering

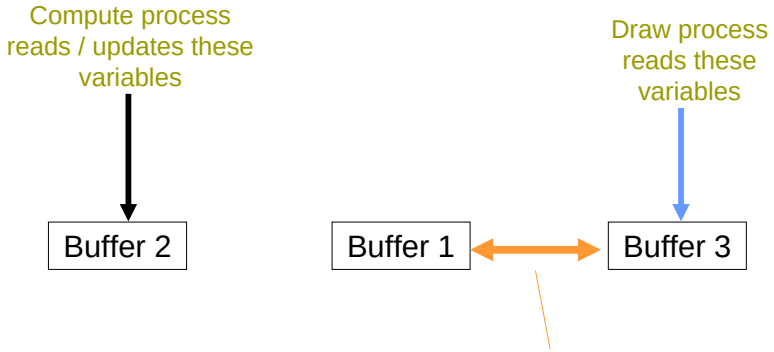
- 계산 프로세스는 그 변수의 자기 버퍼에 있는 복사된 변수만 갱신하고 난 후 이 버퍼를 다른 버퍼와 swap한다



Compute process swaps these buffers when it is done updating the variables

Triple Buffering

- 그리기 프로세스는 자기 버퍼에 복사된 변수를 가지고 그린 후 갱신된 다른 버퍼와 swap해서 다음 그릴 준비를 한다

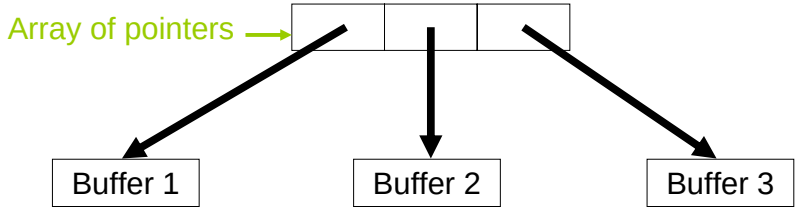


Draw process swaps buffers and draws the new buffer

Triple Buffer Implementation

- 계산 프로세스와 그리기 프로세스는 이 3개 버퍼를 가리키는 포인터의 배열을 mutex로 locking해서 swap을 안전하게 수행할 수 있도록 한다

Compute & Draw processes
lock Mutex on the array of pointers to the 3 buffers
so that they can safely do the Swap



How I Wrote A Simple Game
Day 1: Testing the Waters

- 먼저 내가 가지고 있는 시간이나 Software 지식 등 제한 요소를 고려하여 게임 설계를 한다
  - 1 bullet for user, 1 bullet for enemy, 1 enemy at a time
- 작은 프로그램 샘플을 테스트하면서 어떤 특정 기능이 있는지 확인한다
- 온라인 forum을 자주 이용한다
- 만들고자 하는 동기를 실현하도록 점진적으로 게임을 만들어 간다
- 3차원 모델링 패키지를 이용해 모델링을 시작한다
- 카메라 트래킹을 이용하여 자동차 시뮬레이션을 만들어 본다
- 내가 직접 3차원 지형과 지형 위에 장애물을 만들어 본다

Day 2 : Putting Together All the Basic Game Elements

- 총알(간단한 구 모양) 쏘기를 추가한다
- 간단한 지면(입방체 모양)과 구의 충돌 검사를 시도한다
- 총알이 최대 거리에 도달하여 (간단한 입방체에) 맞으면 총알 터지는 효과 (particle systems)을 추가한다
- 적군을 3차원 모델링 한다
- 움직이는 적군과 간단한 인공지능을 추가한다
- 총을 쏠 때와 총알이 입방체에서 터지는 사운드를 추가한다
- 내가 적군과 부딪쳤을 때를 처리한다
  - 적군이 터지는 장면의 애니메이션을 3차원 모델링 한다
- 적군이 나를 쳤을 때를 처리한다
  - 내가 터지는 장면의 애니메이션을 3차원 모델링 한다
- 적군이나 내가 터질 때의 사운드 등을 더 추가한다

Day 3 : Tuning & Adding Finishing Touches

- Tuning – 기본적인 부분은 적어도 2주 이상 튜닝 해야 한다
  - Tweak AI – ie when to fire
  - Better bullet effect
  - Tweak lights
  - Tweak explosions effect
  - Add enemy sound volume attenuation with distance
- Finishing Touches – 최종 다듬기
  - 점수 계산 방법 & 점수판 추가
  - 시작장면 (intro)과 끝장면 (outtro) 또는 반복장면 (replay) 추가
  - 배경음악 추가
  - 보다 나은 돌출상황들 추가
- Wishlist – 만약에 내가 시간이 더 있었다면, 뭘 더하지..
  - 보다 많은 동시 다발적인 적군들
  - 보다 많은 총알들
  - 레벨 디자인

Tweening

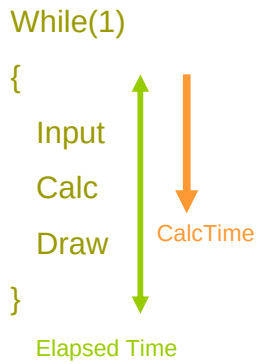
- Main idea:
  - 게임루프는 다음을 가지고 있다
    - 입력처리 (input) 부분과 계산 (calculation) 부분
    - 그리기 (draw) 부분
  - 전체 게임 루프가 한번 돌기 위해 얼만큼 시간이 드는지 알아본다 (이것을 elapsedTime 이라 부른다. elapsedTime = 0.5 seconds)
  - 필요로 하는 계산을 위한 갱신 속도 (update rate)를 결정한다 (예, 30 updates per second) [Note: 게임의 갱신속도는 그래픽스 refresh정도를 나타내는 FRAME-RATE과 같은 것은 아니다]
  - 그러므로 주어진 경과시간을 가지고, 그 시간 안에 얼마나 많은 갱신에 필요한 계산을 해야 하는지 알아본다 (예, 0.5초 경과 시간 안에 15개의 계산을 수행할 수 있어야 한다는 등)

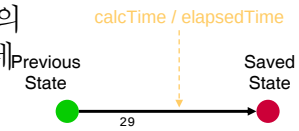
```
While(1)
{
  Input
  Calc
  Draw
}
```

Elapsed Time

Tweening

- 모든 15개의 계산을 수행하고 (모든 물체에 대한 위치, 방향 등) 전체 월드의 상태를 저장한다
- 그리고 실제로 15개의 연산을 수행하고 나서 걸린 시간을 잰다 (이것을 calcTime 이라 부른다)
- 실제 계산을 위해 걸린 시간과 경과 시간 간의 소수값 (fraction)를 알아낸다 - 즉 이 소수값 (calcTime / elapsedTime)이 TWEEN 값이 된다
- 그리고 전체 월드의 전 상태와 현 월드의 저장상태 간을 보간 (interpolate)하는데 이 tween 값을 이용한다





Tweening

- In Electro:
  - do_timer(dt) callback은 마지막으로 do_timer가 불린 시간으로부터 경과 시간 (elapsed time)을 반환한다
  - 그리고 이 경과 시간 dt를 그리기에 사용한다