Game Software Design

305900
Fall 2010
10/4/2010
Kyoung Shin Park

Overall Game Loop

- Overall Game Program Loop
  a. Game introduction and interface
  b. Game level interface – e.g., select level options like weapons, etc
  c. Game level init and loading of game objects
  d. Game loop
     i. Handle all aspects of the actual game play (The hard part!)
     ii. If player wins, goto reward sequence then goto b.
     iii. If player loses, goto failure sequence then goto a if user gives up, or b. if user wants to try again.

Handle all aspects of the actual game play (ie. The hard part!)

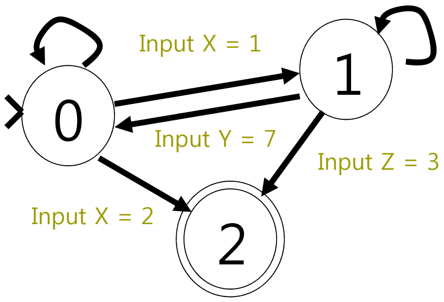
- There are *many many* ways to approach this…
- Will consider 3 important aspects here :
  - Building Finite State Machines
  - Maintaining Simulation Constancy in a Game Loop
  - Multi-Threaded Game Loops

Things that need to be done in the game loop

- Read user input (including any network data)
- Calculate user parameters based on user input (e.g. user moves forward when press “w” key; handle situations where user collides with a wall)
- Calculate NPC (Non-player Character) AI (Artificial Intelligence)
- Draw graphics
- Handle sound effects

Finite State Machines are Not Just Those Useless Things You Learned in Discrete Math

- FSMs are one of the most commonly used programming structures for games.
- Quake is 1 giant FSM.
- FSM
  - States
  - Inputs
  - Transitions

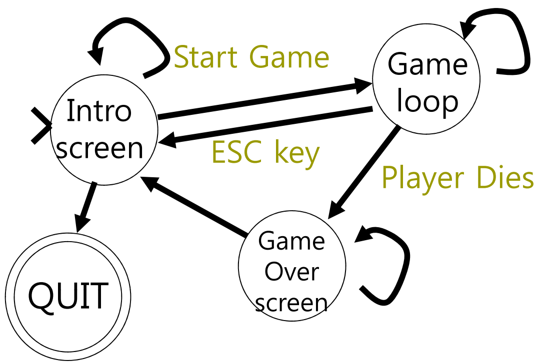


Finite State Machines

- FSM is *a state machine* that receives the *input signal set* from Information Processing Machine and then produces *output signal*.
  - FSM has a set of states that follow a certain path. A state has transitions to other states, which is caused by events or actions within a state.
- 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 (State transition function)
  - g – S -> O (Output function)
- FSM M1==M2 (i.e., Equivalent):
  - Given the same input sequence, if it produces the same output sequence, M1 is equivalent to M2

FSMs for Game Programming

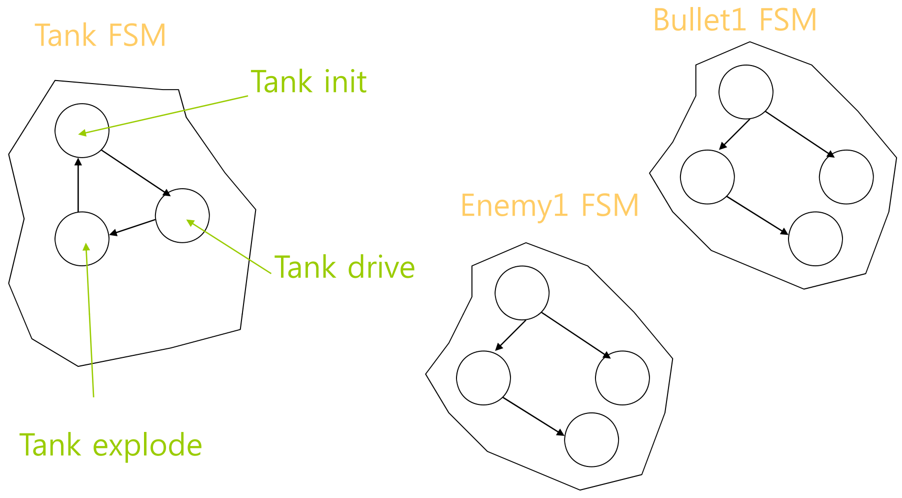
- The game, as a whole, is an FSM.
- Each phase of the game is an FSM.
- Each object in each phase of the game is an FSM.
- Hence in totality a game is a Hierarchy of FSM.



Warning!

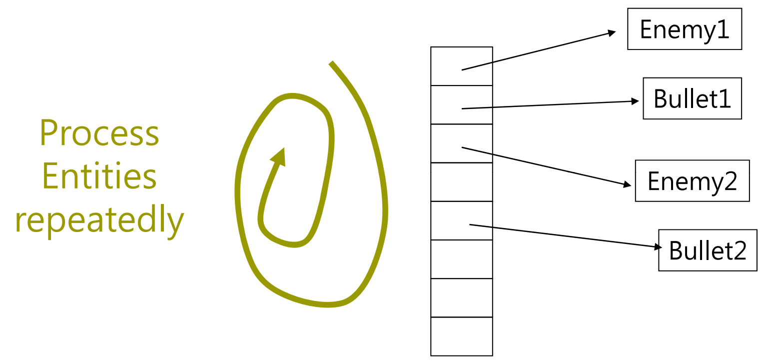
If your entire game isn't designed as a hierarchy of FSMs it will be very difficult to add new features as the game gets more complex. Your code will be spaghetti…

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



Consider the Game Loop

- Array(s) of objects/entities that are currently present in the world and need to be processed.



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

enemyArray	bulletArray
Enemy1	Bullet1
Enemy2	Bullet2
Enemy3	Bullet3
Enemy4	Bullet4

Use arrays so that your game does not do alloc and dealloc during runtime. You cannot afford to have your program fail if alloc == NULL

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:
    - Check inputs or messages on input queue to see if any are relevant to this state
    - If YES, do something (and perhaps change state)
    - Else Break
  - Case State2:
    - etc..

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

- Problem: Make sure your tank or car moves through the scene at the same speed no matter how fast your CPU is.
- Especially important if you have a non-threaded game loop where reading inputs, computing, drawing all take up time at each iteration of the game loop.
- This is a problem ignored by old computer games because computers didn't have such a wide range of performance characteristics- e.g. 1GHz to 2GhZ .
- So when they move for example a car across the screen, the calculations would simply be:
  - PosX = PosX + some_unit_distance
  - Where the bigger the some_unit_distance the "faster" the car moved

What You Should Do Instead

- Each time thru the game loop takes a certain amount of time.
- That elapsed time (say dt) is needed to determine where your entities need to be next.
- E.g. Car moving at 30 feet per second .
- If the game loop takes dt to process, the next time through the game you need to figure where the new position of the car is
  - posX = posX + (speedX * dt)
  - posY = posY + (speedY * dt)
  - posZ = posZ + (speedZ * dt)

Multi-Threaded Game Loops

- Tweening is fine if your game loop runs fast enough to keep up with the desired FRAME RATE
- But some times AI systems can get very complex and take a long time to compute.
  - E.g. an intelligent AI system that attempts to form high level plans for an invasion army .
- A game cannot afford to have 1 loop since the slower components of the loop can easily slow down the overall responsiveness of the game.
- Also modern game systems have multiple cores and can process things in parallel.

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

Multi-Threaded Game Loops

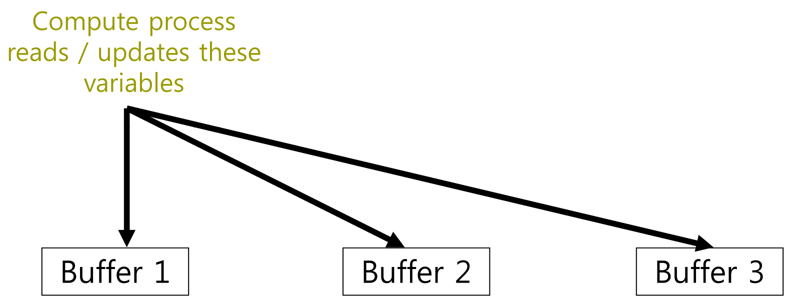
- Hence the need for **multiple Threads or Processes** for:
  - Input Loop
  - Compute Loop
  - Draw Loop
  - Sound Loop
- Want each loop to **progress independently and as fast as possible.**
- E.g. If I press the SPACEBAR to fire a bullet, I want to tell the sound loop to play the bullet sound and then handle it on its own so I can go back to computing the rest of the game.
- Ie: Allow the OS to context switch at regular intervals so that you application appears to operate at a constant rate.

Sharing Variables Efficiently

- Global variables in threads are shared across threads.
- Variables in forked processes are local to the process. Hence in forked processes, variable sharing is done using shared memory API (at least in Unix).
- Threading and Forking are good BUT you don't want one thread to change a variable while another thread is using the variable.
- You need to set up MUTEXes.
- BUT you do not want mutexes for EVERY variable since this can slow down your application (due to possible blocks in mutexes).
- **Solution: TRIPLE BUFFERING**

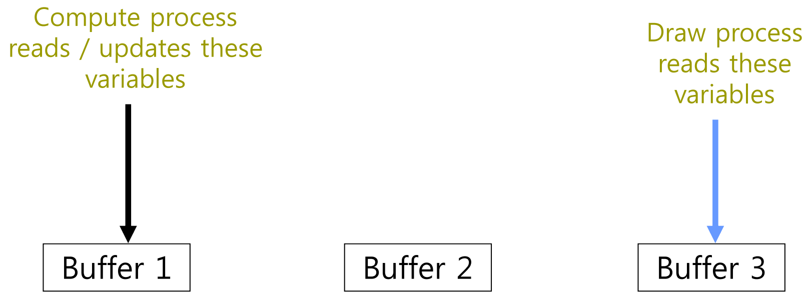
Triple Buffering

- Init Step – Variables are copied 3 times.



Triple Buffering

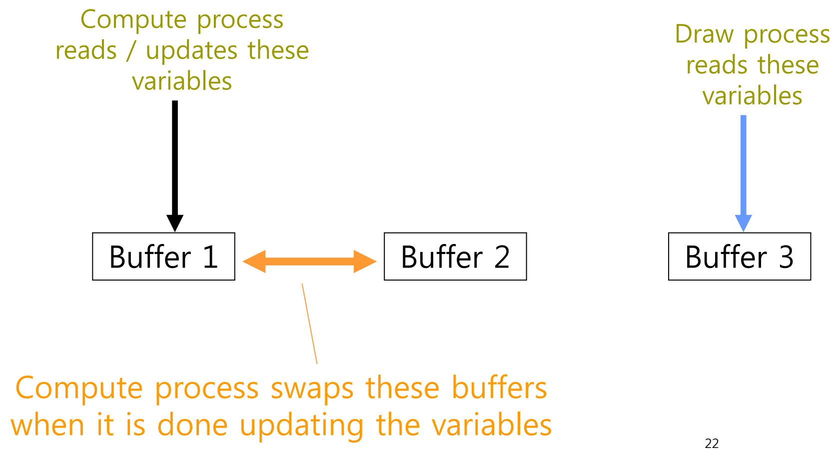
- Compute and Draw Processes use independent copies of the data.



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

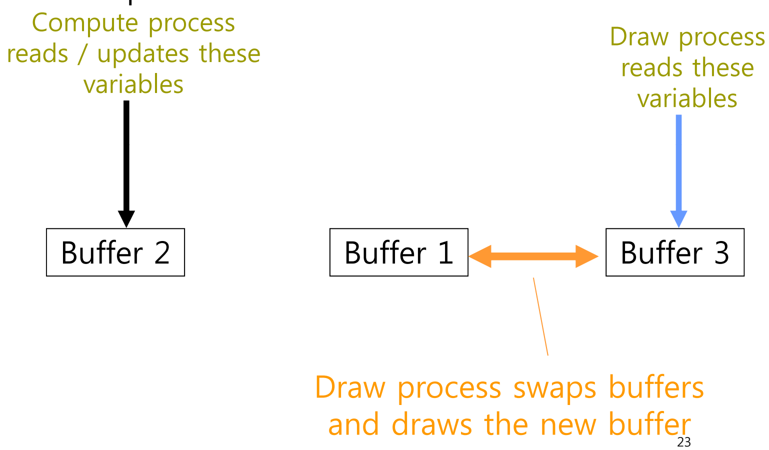
Triple Buffering

- Compute process updates its own copy of the variables.



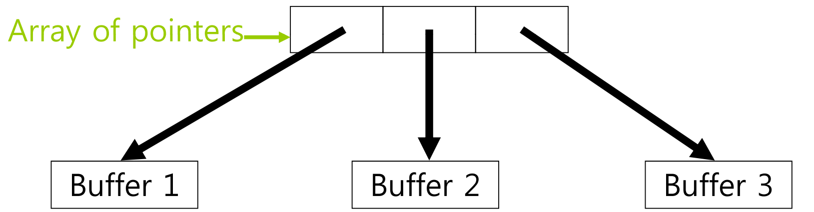
Triple Buffering

- Draw process is done drawing and ready to take in the next update.



Triple Buffer Implementation

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

- Considered design constraints of the game based on how little time I had & how little DBPro or Blitz I knew :
  - 1 bullet for user, 1 bullet for enemy, 1 enemy at a time
- A lot of testing smaller code samples to figure out how specific capabilities in DBPro worked.
- Referenced online forums a lot for help.
- Build progressively more playable game to build confidence & motivation.
- Create tank model in 3D modeling tool.
- Create driving simulator with camera tracking; try shadows.
- Create terrain obstacles- tried my own landscape models.

Day 2 : Putting Together All the Basic Game Elements

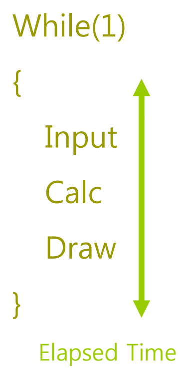
- Add shooting of bullet – simple sphere
- Attempt collision detection of sphere with landscape – could not seem to get collision to function correctly so simplified landscape to cubes
- Add explosion effect of bullets (particles) on impacting cubes and when bullets reach a max distance
- Create enemy model in 3D modeling.
- Add enemy & simple AI to move it around and shoot.
- Add simple sounds for firing & bullet impact on cubes.
- Handle when I hit enemy
  - Create enemy explosion animation in 3D modeling
- Handle when enemy hits me
  - Create me exploding in 3D modeling
- Add more sounds – ie: me exploding

Day 3 : Tuning & Adding Finishing Touches

- Tuning – in your case remember to spend a good 2 weeks tuning
  - Tweak AI – ie when to fire
  - Better bullet effect
  - Tweak lights
  - Tweak explosions effect
  - Add enemy sound volume attenuation with distance
- Finishing Touches
  - Add scoring scheme & score board
  - Add intro & outtro/replay screen
  - Add background music
  - Add better randomness
- Wishlist (if I had more time…)
  - More simultaneous enemies
  - More bullets
  - Level progression

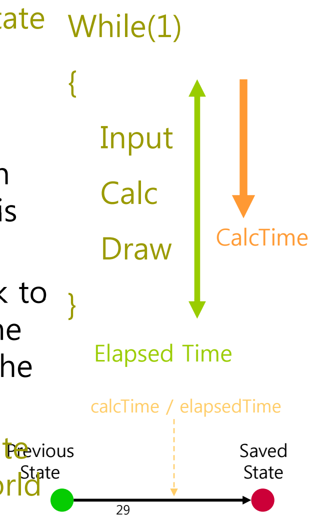
Tweening

- Main idea:
  - Game loop consists of:
    - Input/Calculation Part
    - Drawing Part
  - Figure out how much time was spent in 1 loop of the entire game loop (call this elapsedTime) (e.g. elapsedTime = 0.5 seconds)
  - Decide what is the update rate you want for your calculations (e.g. 30 updates per second) [Note: this is not the same as FRAME-RATE which typically denotes how fast the graphics refreshes]
  - Therefore given the elapsedTime figure out how many update calculations you need to perform in that elapsedTime (for 0.5 second elapsedTime you should be able to do 15 calculations)



Tweening

- Do all 15 calculations and save the state of the entire world (ie position and orientation, etc of all objects in the world).
- Find out how much time was taken in actually doing 15 calculations (call this calcTime).
- Figure out the fraction of time it took to do the calculations vs the elapsedTime (ie calcTime / elapsedTime) – this is the TWEEN value
- You use this tween value to interpolate between the previous state of the world and the saved state of the world.



Reference

- http://www.evl.uic.edu/spiff/class/cs426/