Game Engine Programming

GMT Master Program
Utrecht University

Dr. Nicolas Pronost

Course code: INFOMGEP
Credits: 7.5 ECTS

Lecture #12

Game performance tuning

Introduction

- Game performance is critical
  - gameplay will decrease with lags, low resolution rendering, memory overload *etc.*
  - performance has to be monitored
    - CPU charge, bus bandwidth use, memory footprint
- Essentially, timing and memory consumption have to be measured
  - to point out the most consuming sections

Timing

- Each OS has an internal real-time clock system
- In Win32 programming environment

```
#include <windows.h>
DWORD WINAPI GetTickCount(void);
```

  - number of milliseconds that have elapsed since the system was started, up to 49.7 days

```
#include <time.h>
clock_t clock (void);
```

  - number of clock ticks elapsed since the program was launched
    - CLOCKS_PER_SEC specifies the relation between a clock tick and a second (clock ticks per second)

Timing

- To compute time elapsed in a function

```
unsigned long t1 = GetTickCount();
update_HID();
unsigned long t2 = GetTickCount();
update_PlayerState();
unsigned long t3 = GetTickCount();
run_AI();
unsigned long t4 = GetTickCount();
render();
unsigned long t5 = GetTickCount();

unsigned long timeHID    = t2 - t1;
unsigned long timePlayer = t3 - t2;
unsigned long timeAI     = t4 - t3;
unsigned long timeRender = t5 - t4;
```

Timing

- You can have a visual feedback of these performance on screen
- Or show a percentage representation to see how the functions hold together
- To detect what sections of the game are slowing down the engine
- Optionally, switch in real-time with faster versions of them if performance decreases
  - lower level of detail in rendering or game logic, less HID updates *etc.*

Timing

- Default resolution of *GetTickCount* is 10 to 16 ms
- Some routines may perform in less time
  - 30 FPS = 33.3 ms for the whole game loop
- Multiple calls through iterations
  - do not forget to remove them later on
  - small over-estimation from the loop statement
  - small under-estimation from compiler optimization of loop statement

```
unsigned long time1 = GetTickCount();
for (int i = 0; i < iterations; i++) {
   // routine to estimate here
}
unsigned long time2 = GetTickCount();
double elapsed = (double)(time2-time1)/iterations;
```

Timing

- If you need higher resolution, and if a high-resolution performance counter exists on the hardware system
  - processor dependent (nowadays $\sim 10^{-7} s$)

```
#include <windows.h>

BOOL WINAPI QueryPerformanceFrequency(
  __out  LARGE_INTEGER *lpFrequency
);
// Retrieves the frequency of the high-resolution performance counter

BOOL WINAPI QueryPerformanceCounter(
  __out  LARGE_INTEGER *lpPerformanceCount
);
// Retrieves the current value of the high-resolution performance counter
```

Memory footprints

- To profile memory use
  - to detect memory leaks
  - to spot memory sensitive code
- By testing the total available memory before and after the routine, and doing the subtraction

```
long m1 = available_memory();
// here goes the routine to test memory for
long m2 = available_memory();
long consumed_memory = m2 - m1;
```

Memory footprints

- Implementation more complex than timing
- In multi-tasking, memory can be allocated from other processes
  - measuring total physical memory will also measure allocations from these processes
- One solution is to write your own memory manager
  - by overwriting the new and delete operators
  - and counting the memory allocation / de-allocation
  - but not only for the user type, also for the primitive types

Memory footprints

- operator new

```
unsigned long usedMemory = 0; // memory allocated so far
unsigned long maxUsedMemory = 0; // max allocated memory so far
int numAllocations = 0; // number of active allocations
unsigned long maxAllocations = 0; // max active allocations so far

// User-defined operator new
void * operator new(size_t blocksize) {
    numAllocations++;
    usedMemory += blocksize;
    if (usedMemory > maxUsedMemory) maxUsedMemory = usedMemory;
    if (numAllocations > maxAllocations) maxAllocations = numAllocations;
    return malloc(blocksize); // perform the memory allocation
}

// ...
```

Memory footprints

- operator delete

```
// ...

unsigned long minUsedMemory = 0; // min allocated memory so far

// User-defined operator delete
void operator delete(void * pointer) {
    numAllocations--;
    usedMemory -= _msize(pointer); // allocated size of a pointer created
                                   // by a call to malloc
    if (usedMemory < minUsedMemory) minUsedMemory = usedMemory;
    free(pointer); // free the memory
}
```

Memory footprints

- Then, regularly print *usedMemory* value (and *numAllocations*)
  - should be stable over a long period of time if each new has its corresponding delete
  - should go down to zero just before terminating the application
    - if not, most likely due to a memory leak
- You can also have a MemoryManager that keeps track of a *usedMemory* value
  - incremented after each new call
  - decremented after each delete call
  - stack variables can be ‘neglected’ as do not produce memory leaks

Memory footprints

- One extra step is to optimize yourself the memory according to its usage (access / allocation / de-allocation) in your engine
  - If access and creation patterns are known
  - Large memory pool allocation at start time
  - Implementation of homegrown new and delete operators
  - To reduce internal fragmentation that will slow down the game

Analysis tools

- Tools are available to analyze your game at run-time, called code profiler
- Programming environments such as Visual Studio have built-in profiler functionalities to display percentage of CPU time taken by functions, to track memory leaks, *etc.*

Optimization techniques

- Choose your enemy
  - Focus on small portions of code that use many resources
  - Do not focus on linear sequential code
  - Troubles are in iterations and recursions
  - Make sure each loop takes as little time as possible to execute

```
for (int i = 0; i < 10000; i++) {
    for (int j = 0; j < 10000; j++) {
      // this is usually here that you need to worry about resources
      // as you will execute this code 10^8 times
    }
}
```

Optimization techniques

- Precompute as much as possible
  - Try to tabulate mathematical functions, random numbering *etc.*
  - To perform only an array access in the loop
  - Example
    - sinus call takes 5 times longer to be evaluated than to access an array

```
float acc = 0;
for (int i = 0; i < 1000; i++)
   acc = acc + i * sin(x * i); // instead use: sinTable[x*i]
```

Optimization techniques

- Simplify your math
  - Mathematical operators are not equally fast
  - Complex function >> divide >> multiply >> addition/subtraction
  - Try to simplify equations (and/or tabulate them)
  - Try to reduce type conversion
  - Examples

```
double acc = 1000000;
for (int i = 0; i < 10000; i++) acc = acc / 2.0;
double acc = 1000000;
for (int i = 0; i < 10000; i++) acc = acc * 0.5;
//the second version takes 60% of the timing cost to execute
```

```
a*b + a*c = a*(b+c);      // gets rid of one multiply
b/a + c/a = (1/a)*(b+c); // changes one division for one multiply
          = (b+c)/a;      // gets rid of one division
```

Optimization techniques

- Store data efficiently
  - chose the right data type with the right precision
  - both code execution and memory footprint are proportional to the number of bytes used

Type	Size (B)	Range
char	1	[-128 , 127]
unsigned char	1	[0 , 255]
int	4	[-2 147 483 648 , 2 147 483 647]
unsigned int	4	[0 , 4 294 967 295]
float	4	$[-3.4*10^{38} , 3.4*10^{38}]$ (7 decimal)
double	8	$[-1.7*10^{308} , 1.7*10^{308}]$ (15 decimal)
bool	1	true / false

Optimization techniques

- Minimize malloc (new) and free (delete) calls
  - Try to avoid allocation and de-allocation in loops
  - Place them outside the critical sections
  - Example

```
for (long k=0; k<1000000; k++) {
    int i = 0;
}
// takes 3.53 ms to run (and 3.48 when i declared before for)

for (long k=0; k<1000000; k++) {
    int *i = new int;
    delete i;
}
// takes 5.58 s to run!! (and 5.56 when i declared before for)
```

Optimization techniques

- Use prefix increment/decrement operators

```
vector<double> vec;
// add 1000000 elements in the vector
vector<double>::iterator ite;
for (ite = vec.begin(); ite != vec.end(); ite++) { } // takes 2.45 s
for (ite = vec.begin(); ite != vec.end(); ++ite) { } // takes 1.07 s
```

- Pseudo-code of prefix and suffix ++

```
// prefix : ++variable
type & type::operator ++ () {
   // do the increment
   // e.g. _value += 1;
   return *this;
}
```

```
// suffix : variable++
type type::operator ++ (int) {
   type ans = *this;
   // do the increment
   // or call ++(*this);
   return ans;
}
```

Optimization techniques

- Be linear
  - CPUs come with memory caches loaded when accessing data
  - Access continuous data in memory (*e.g.* traversing an array from begin to end) produces less cache misses
    - so less loading time

Optimization techniques

- Watch out for pointers
  - Traversing a sequence of pointers can take time as the objects can be far away from each others in memory
  - Try to minimize the indirections
  - Example

```
for (int i = 0; i < numPlayer; i++ )
   Game::getInstance()->logic->world->team->players[i] = i * 2;

// can be rewritten more efficiently as

int * playersList = Game::getInstance()->logic->world->team->players;
for (int i = 0; i < numPlayer; i++ )
    playersList[i] = i * 2;
```

Optimization techniques

- Size does matter
  - To compile arrays of structures, the compiler performs a multiplication by the size to create the array indexing
    - if the structure size is a power of 2, the multiplication is replaced by a shift operation (much faster)
    - you can round array sizes aligned to a power of 2 even if you do not use all of it
  - Example

```
int arrayPlayersID [38];
int arrayPlayersID [64]; // faster allocation
```

Optimization techniques

- Breaking switches
  - To reduce the number of comparisons in a switch (or if-else-if statement), place all the less frequent (error) cases at the end

```
switch (event.msg) {
   case FREQUENT_MSG1:
        handleFreqMsg1();
        break;
   // other frequent messages ...
   case INFREQUENT_MSG1:
        handleInFreqMsg1();
        break;
   // other infrequent messages ...
   default: //...
}
```

Optimization techniques

- Local variable in inner most scope
  - Do not declare local variables directly after function declaration
  - Use them only when necessary
  - Example

```
void function () {
    if (!error()) {
        // do something if no error
    }
    else {
        LargeMemoryObject o;
        // do something with o
    }
}
```

Optimization techniques

- Returning value
  - Do not return a value if not used
  - Returning a value has a cost (CPU and memory)
  - Example

```
bool calledFunction (float x) {
    // do something with x
    return true;
}

void function () {
    calledFunction(2.0); // is not using return value
}
```

Optimization techniques

- Prefer initialization over assignment
  - Direct initialization saves a call to the default constructor comparing to initialization followed by instantiation
  - Example

```
void function (Object value) {
    Object o; // default constructor
    o = value; // assignment operator
}
void function_optimized (Object value) {
    Object o = value; // copy constructor
}
```

Optimization techniques

- "Just in case" virtual functions
  - Do not declare virtual functions just in case they might be overridden one day
    - look-up v-table cost (CPU to check it and memory to store it)
  - If the day comes, change them to virtual
  - Example

```
class MyClass {
   // ...
   virtual void function (float);
}
// no subclasses of MyClass
// or subclasses that do not need to override function
```

Optimization techniques

- Keep the running fresh
  - *E.g.* perform reset operations between levels
  - Some consoles can even be completely rebooted
  - You can run manually some form of garbage collector (and easier if you manage yourself the memory allocation)

Physical structure

- A normal PC/console game deals with at least 4000 files (more for MMOGs)
- A proper physical structure is crucial
- Structure is determined by which files need other files in order to compile
- In C++ this ‘need’ translates into #include directives

Physical structure

- In an ideal world, every file would compile by itself
- Not possible as a program is made up of interacting objects
- We can try to minimize the amount of connections between files
- The level of connections between a file and the rest of the code is called insulation
  - fewer connections mean more insulated

Physical structure

- A class that hides well its implementation will have better encapsulation
  - Fewer classes depend on it
  - Cleaner logical structure
  - Easier debugging
  - Simpler testing
- Whenever a file is modified, all files that include that file need to be recompiled
  - so the more insulated files you have, the faster the code compiles

Header files

- Header files should contain the minimum amount of code that still allows everything to compile and run
- Move non-essential information out of the header file into the implementation file
  - Constants only used in the implementation, local structure definitions, algorithms *etc.*
- However, C++ does not provide for a very clear distinction
  - We still need to define the non-public members of a class in the header file, templates code *etc.*

Header files

- Indirect inclusion issue (nested include)

GameConst.h

```
#ifndef GAMECONST_H_
#define GAMECONST_H_

#define GAMELEVELS 42
// ...
#endif
```

Player.h

```
#ifndef PLAYER_H_
#define PLAYER_H_

#include "GameConst.h"
// ...
#endif
```

Game.cpp

```
#include "Player.h"
int main() {
    for (int i = 0; i < GAMELEVELS; ++i)
         InitializeLevel(i);
    // ...
}
```

  - if Player class does not need GameConst.h anymore, compiler error in Game.cpp
    - can be difficult to debug as unrelated files

Header files

- Forward declaration: #include in body file

GameCamera.h

```
#include "GameEntity.h"
class GamePlayer; // forward declaration

class GameCamera : public GameEntity {
   public:
        GamePlayer* getPlayer();
   private:
        GamePlayer* player_;
};
```

GameCamera.cpp

```
#include "GameCamera.h"
#include "GamePlayer.h"
// ...
```

  - prevents to try to include GamePlayer (and each related includes) for each class that includes GameCamera
  - solves the problem of dual inclusion

Precompiled header

- Often, many classes need to include the same files such as the standard libraries

```
#include "Player.h"                                    Player.cpp
#include <vector>
#include <string>
#include <iostream>
// ...
```

- These APIs have large and complex header
- They do not change during the development
- One solution: the precompiled header
  - not platform independent
  - take advantage of it if possible

Precompiled header

- A precompiled header will be loaded and processed only once
- So we put only the headers that do not change during the development process
- Visual Studio supports a precompiled header (usually called stdafx.h)

```
#include "stdafx.h" // Precompiled header                    Warrior.cpp
#include "Warrior.h"

// other includes
#include "Player.h"
#include "Enemy.h"

// ...
```

Precompiled header

- If we decide to compile with a precompiled header, greatly decreased compile time
- But as number of included headers grows, every file that uses the precompiled header automatically knows about all the files that are part of the precompiled header

The PIMPL pattern

- The PIMPL (pointer to implementation) pattern allows us to avoid including files required only for private variables
- Create a simple structure or class that contains the private implementation
- We create and destroy it along with the object itself

The PIMPL pattern

Before

Warrior.h

```
#include "Player.h"
#include "Position.h"
#include <string>
class Enemy;

class Warrior : public Player {
   public :
        Enemy * getEnemy();
   private :
        Enemy * enemy_;
        Position pos;
        std::string name_;
        // require the headers include
};
```

After

Warrior.h

```
#include "Player.h"


class Enemy;

class Warrior : public Player {
   public :
        Enemy * getEnemy();
   private :
        class PIMPL ;
        PIMPL * pimpl_;


};
```

The PIMPL pattern

Warrior.cpp

```
#include "Warrior.h"
#include "Position.h"
#include "Enemy.h"
#include <string>

class Warrior::PIMPL {
    public:
          Enemy * enemy_;
          Position pos;
          std::string name_;
};

Warrior::Warrior() {
    pimpl_ = new PIMPL();
}

Warrior::~Warrior() {
    delete pimpl_;
}

Enemy * Warrior::getEnemy() {
    return pimpl_->enemy_;
}
```

The PIMPL pattern

- Advantages
  - We removed all includes related to private implementation
  - Reduction in dependencies between headers
  - We have also hidden all implementation details to the user
- Limitations
  - An added complexity to program the class (access private members through PIMPL object)
  - Minor performance cost for dynamic allocation and redirection

Dealing with bad data

```
void Vector3D::normalize() {
    float fLength = sqrt(x*x + y*y + z*z);
    x /= fLength;
    y /= fLength;
    z /= fLength;
}
```

- In this function we might divide by 0!
- Tendency to propagate the NaN value to calling functions while we would prefer to deal with it here

Dealing with bad data

- Solution 1: use assert
  - but is removed in the released application

```
void Vector3D::normalize() {
    float fLength = sqrt(x*x + y*y + z*z);
    assert(fLength != 0);
    x /= fLength;
    y /= fLength;
    z /= fLength;
}
```

Dealing with bad data

- Solution 2: cope with it
  - but adds extra logic (performance decreases)
  - still produces 'incorrect data'

```
void Vector3D::normalize() {
    float fLength = sqrt(x*x + y*y + z*z);
    if (fLength != 0) {
          x /= fLength;
          y /= fLength;
          z /= fLength;
    }
    else { // a unit vector
          x = 1.0f;
          y = 0.0f;
          z = 0.0f;
    }
}
```

Dealing with bad data

- Solution 3: cope with it with error code or exception
  - change the signature or need to be caught

```
bool Vector3D::normalize() {
    float fLength = sqrt(x*x + y*y + z*z);
    if (fLength != 0) {
          x /= fLength;
          y /= fLength;
          z /= fLength;
          return true;
    }
    else { // a unit vector
          x = 1.0f;
          y = 0.0f;
          z = 0.0f;
          return false;
    }
}
```

Dealing with bad data

- Solution 4: a compromise
  - potential performance hit from the conditions

```
void Vector3D::normalize() {
    float fLength = sqrt(x*x + y*y + z*z);
    #ifdef ASSERTBADDATA
    assert(fLength != 0);
    #endif
    if (fLength != 0) {
          x /= fLength;
          y /= fLength;
          z /= fLength;
    }
    else {
          x = 1.0f;
          y = 0.0f;
          z = 0.0f;
    }
}
```

Dealing with bad data

- Solution 5: add an “unsafe” function
  - used only if already guaranteed that no zero-length vector will be normalized

```
void Vector3D::normalizeUnsafe() {
    float fLength = sqrt(x*x + y*y + z*z);
    x /= fLength;
    y /= fLength;
    z /= fLength;
}
```

End of lecture #12

Next lecture

Game network programming