Game Engine Programming

GMT Master Program
Utrecht University

Dr. Nicolas Pronost

Course code: INFOMGEP
Credits: 7.5 ECTS

Lecture #5

The game loop

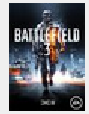
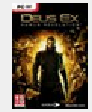
The game loop

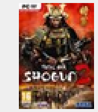
- A game is a real-time and interactive computer application
- Different kinds of time are used
  - real time (wall clock time)
  - game time (simulated time)
  - local timelines (audio, animation time…)
  - CPU cycles (functional time)
- The game loop defines how these times are combined in order to synchronize the game engine components

The game loop

- Most components use local timelines
- So usually only three tasks run concurrently
  - The HID input (player interactions)
  - The game logic (player / world state, storyline)
  - The feedback (rendering, sound, HID output)
- Limitations of real-world technology
  - 1-4 processors with limited memory and speed



	Mass Effect 3 (2012)			The Elder Scrolls V: Skyrim (2011)				Battlefield 3 (2011)			
	low 1280x720 all off -AA -AF	high 1366x768 all on onAA 4xAF	ultra 1920x1080 all on onAA 8xAF	low 1280x720 Low Preset 0xAA 0xAF	med. 1366x768 Medium Preset 4xAA 0xAF	high 1366x768 High Preset 8xAA 8xAF	ultra 1920x1080 Ultra Preset 8xAA 16xAF	low 1024x768 low -AA 2xAF	med. 1366x768 medium -AA 4xAF	high 1366x768 high -AA 16xAF	ultra 1920x1080 ultra 4x MSAA 16xAF
NVIDIA GeForce GTX 590								123	110	98	57
NVIDIA GeForce GTX 580								142	108	89	45
AMD Radeon HD 6970											
NVIDIA GeForce GTX 570											
NVIDIA GeForce GTX 480											
NVIDIA GeForce GTX 560 Ti											
AMD Radeon HD 6870				71	66	53	43	107	80	68	28
NVIDIA GeForce GTX 470			60			63	44	103	75	61	30
ATI Radeon HD 5850	60	60	60	64	61	51	41	109	78	60	24
NVIDIA GeForce GTX 460 768MB								93	68	56	18
AMD Radeon HD 6790											
ATI Radeon HD 5770											
NVIDIA GeForce GTX 550 Ti											
NVIDIA GeForce GTS 450						42	22	73	50	39	16
ATI Radeon HD 4850											
ATI Radeon HD 5670											
NVIDIA GeForce GT 240 GDDR5											
NVIDIA GeForce GT 430											
ATI Radeon HD 5570											
	Mass Effect 3			The Elder Scrolls V: Skyrim				Battlefield 3			
	low	high	ultra	low	med.	high	ultra	low	med.	high	ultra
AMD Radeon HD 6550D*											
NVIDIA GeForce GT 220											
AMD Radeon HD 6450 GDDR5	32	25	16	32	22	15		24			
ATI Radeon HD 4350											

	Deus Ex Human Revolution (2011)			The Witcher 2: Assassins of Kings (2011)				Crysis 2 (2011)				Total War: Shogun 2 (2011)		
	low 1024x768 DX9, all Off -AA TrilinearAF	high 1366x768 DX11, (Shadows, SSAO, DOF): Normal, Post Processing, Tessellation Simple EdgeAA 4xAF	ultra 1920x1080 DX11, Soft Shadows, SSAO High, DOF High, Post Processing, Tessellation MLAAAA 16xAF	low 1024x768 low	med. 1366x768 medium	high 1600x900 high onAA	ultra 1920x1080 very high UbersamplingAA	low 800x600 High	med. 1024x768 High	high 1366x768 Very High	ultra 1920x1080 Extreme	low 1024x768 low -AA -AF	med. 1280x720 moderate -AA trilinearAF	ultra 1920x1080 high MLAAAA 16xAF
NVIDIA GeForce GTX 590			130			47	16				101			75
NVIDIA GeForce GTX 580	258	205	92	48	48	48	20	106	105	107	68	328	124	50
AMD Radeon HD 6970														
NVIDIA GeForce GTX 570														
NVIDIA GeForce GTX 480														
NVIDIA GeForce GTX 560 Ti											43			
AMD Radeon HD 6870		88	64			44	14			121	42	349	105	41
NVIDIA GeForce GTX 470			62		48	43	13			126	44	427	97	34
ATI Radeon HD 5850	151	116	59					206	151	113	40			
NVIDIA GeForce GTX 460 768MB										87	31			
AMD Radeon HD 6790								166	113	84	30			
ATI Radeon HD 5770										73	26			
NVIDIA GeForce GTX 550 Ti										72	26			
NVIDIA GeForce GTS 450	211	83	31	47	36	23	6			59	22	282	47	17
ATI Radeon HD 4850	204			48	34	22	7			63	19	272	49	
ATI Radeon HD 5670								101	68	50	16			
NVIDIA GeForce GT 240 GDDR5														
NVIDIA GeForce GT 430														
ATI Radeon HD 5570								48	32	23	9			
	Deus Ex Human Revolution			The Witcher 2: Assassins of Kings				Crysis 2				Total War: Shogun 2		
	low	high	ultra	low	med.	high	ultra	low	med.	high	ultra	low	med.	ultra
AMD Radeon HD 6550D*								47	32	23	9			
NVIDIA GeForce GT 220														
AMD Radeon HD 6450 GDDR5	69	21	9	21	14	7		46	31	20	7	124	18	
ATI Radeon HD 4350								16	10					

source: http://www.notebookcheck.net

	Mass Effect 3			The Elder Scrolls V: Skyrim				Battlefield 3				Deus Ex Human Revolution			The Witcher 2: Assassins of Kings				Crysis 2				Total War: Shogun 2		
	low	high	ultra	low	med.	high	ultra	low	med.	high	ultra	low	high	ultra	low	med.	high	ultra	low	med.	high	ultra	low	med.	ultra
AMD Radeon HD 6755G2*																			47	32	24	9			
AMD Radeon HD 6750M								35	27	21		109	39	17					77	52	38	12			
NVIDIA GeForce GT 550M																			59	39	29	10			
AMD Radeon HD 6830M																									
ATI Mobility Radeon HD 5830																									
AMD Radeon HD 6760G2*																									
AMD Radeon HD 6740G2*				26	25	20		37	29	24		76	29						51	35	26		88	28	
AMD Radeon HD 6730M*																			56	39	28	12			
ATI Mobility Radeon HD 5770																									
AMD Radeon HD 6570M																									
AMD Radeon HD 7670M*				47	34	22	11	37	25	19	7	98	36	16					60	40	29	10			
NVIDIA Quadro 1000M																									
ATI Mobility Radeon HD 5750*																									
AMD Radeon HD 6720G2*												65	28												
NVIDIA GeForce GT 630M*				41	30	19		39	22	16		80	37						57	37	27				
NVIDIA GeForce GT 540M				41	28	19	12	36	23	18		83	37		24	15			56	36	27	9	142	24	7
ATI Mobility Radeon HD 5730																									
ATI FirePro M5800																									
AMD Radeon HD 6690G2*																									
AMD Radeon HD 6650M				27	25	20	11	35	25	19	7	89	33	16					59	39	29	10			

	Mass Effect 3			The Elder Scrolls V: Skyrim				Battlefield 3				Deus Ex Human Revolution			The Witcher 2: Assassins of Kings				Crysis 2				Total War: Shogun 2		
	low	high	ultra	low	med.	high	ultra	low	med.	high	ultra	low	high	ultra	low	med.	high	ultra	low	med.	high	ultra	low	med.	ultra
NVIDIA GeForce GT 435M																									
AMD Radeon HD 6680G2*												51	28										89	31	
AMD Radeon HD 6550M																									
AMD Radeon HD 7590M*																									
NVIDIA GeForce GTS 350M																									
AMD Radeon HD 7660G																									
AMD Radeon HD 6630M				50	32	20	10	34	22	20		80	33	14	23				51	33	25	9	116	25	9
AMD Radeon HD 7650M																									
AMD Radeon HD 7570M*																									
AMD Radeon HD 7630M																									
NVIDIA Quadro FX 1800M																									
ATI Mobility Radeon HD 5650	33	29	17	34	29	19	10	35	23	18	6	97	32	14					56	36	26	9			
AMD Radeon HD 6530M																									
NVIDIA GeForce GT 525M																			51	35	25	9			

source: http://www.notebookcheck.net

The game logic loop

- Game data are usually updated in this order
  - Player related data update
    - Sense player input
    - Update player state (according to world restrictions)
  - World related data update
    - Passive elements (static items)
      - Optimized by selection of the logic area of interest
    - Logic-based elements (dynamic items)
      - Sorted according to relevance (LOD)
      - Update state
    - AI-based elements (more complex behavior)
      - Sorted according to relevance (LOD)
      - Sense internal state and goals
      - Decision and execution

The rendering loop

- Illusion of motion is obtained by a high frequency rendering loop

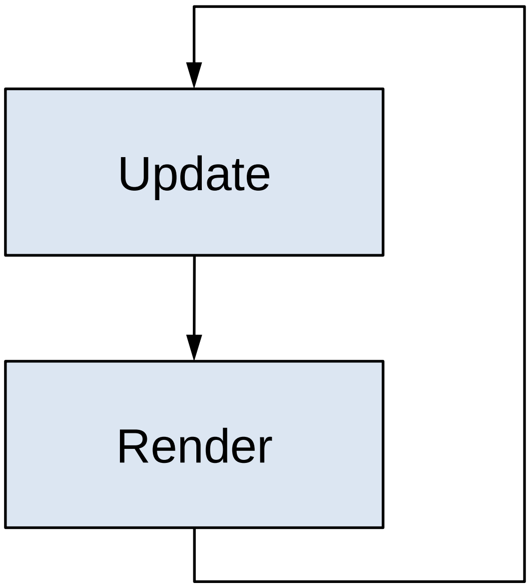
```
while (!quit) {
   // Update the camera view according to input or path
   updateCamera();

   // Update the scene graph (position/orientation of 3D objects)
   updateSceneGraph();

   // Render the scene in "Back Buffer"
   renderScene();

   // Swap Back Buffer with Front Buffer
   swapBuffers();
}
```

The real-time constraint

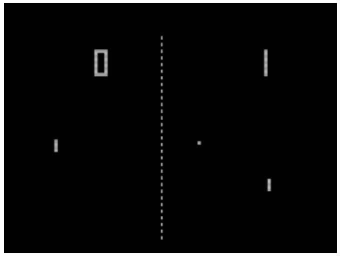
- Graphics rendering as to be performed at least at 30 FPS to get the illusion of motion
- Frequency of other subsystems may differ
  - AI (~10), input (~40), audio (~50), stereovision (~60), physics (~100), haptic feedback (~3k)
  - some need synchronization (for example physics and graphics)
- The game engine services these subsystems
  - game loop in charge of calling the components at the right time

The game loop

- 1st try: design update/render process in a single loop (coupled approach)



The game loop



- Example of what could be
  - *Pong* (1958 – *Atari Inc.*)

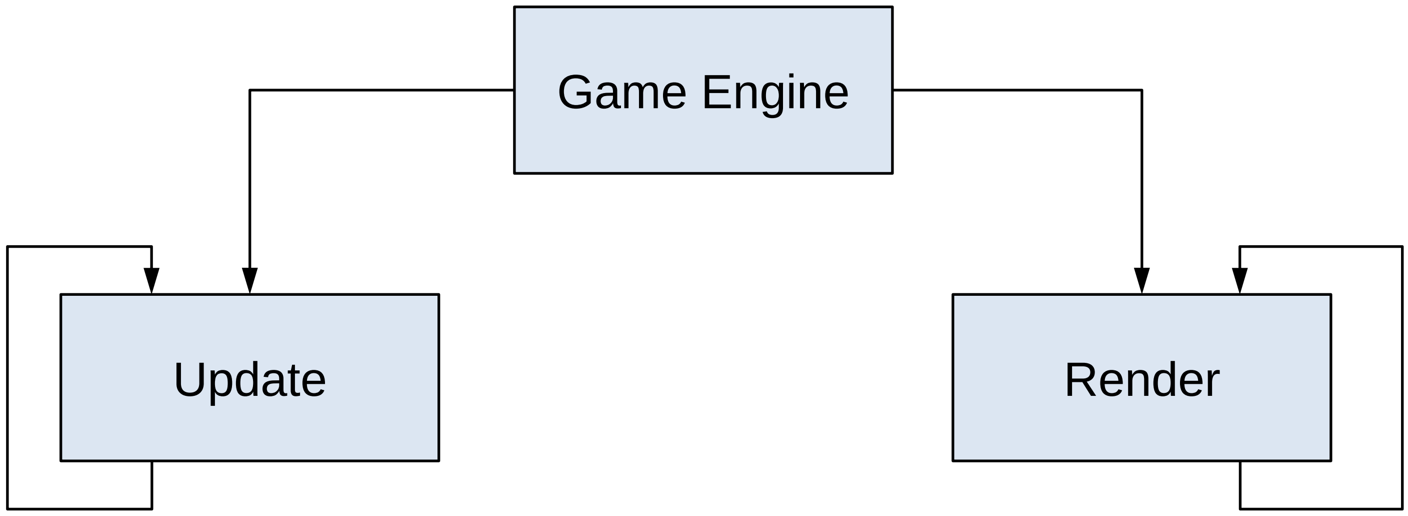
```
int main () {
   initGame(); // Set up initial configuration
   while (true) { // Game loop
        readHumanInterfaceDevices();
        if (quitButtonPressed()) break; // Exit game loop
        movePaddles();
        moveBall();
        if (scored()) {updateScore(); resetBall();}

        renderScore(); // render new game state
        renderPaddles();
        renderBall();
   }
   return 0;
}
```

The game loop

- Advantages of the coupled approach
  - Both routines are given equal importance
  - Logic and presentation are fully coupled
- Disadvantages
  - Variation in complexity in one of the two routines influences the other one
  - No control over how often a routine is updated

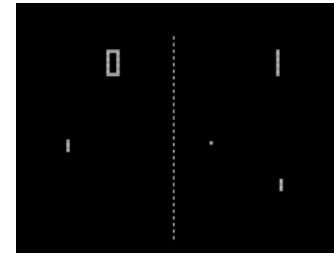
The game loop

- 2nd try: design game loop using two threads with decoupled frequencies



The game loop

- Example of what could be
  - *Pong* (1958 – *Atari Inc.*)



GameEngine.cpp

```
initGame();          // Set up initial configuration
startUpdater(60);    // Start the update loop (60 Hz)
startRenderer(30);   // Start the rendering loop (30 Hz)
```

Updater.cpp

```
while (true) { // loop
   Timer(60);
   readHumanInterfaceDevices();
   if (quitButtonPressed()) exit(0);
   movePaddles();
   moveBall();
   if (scored()) {
        updateScore();
        resetBall();
   }
}
```

Renderer.cpp

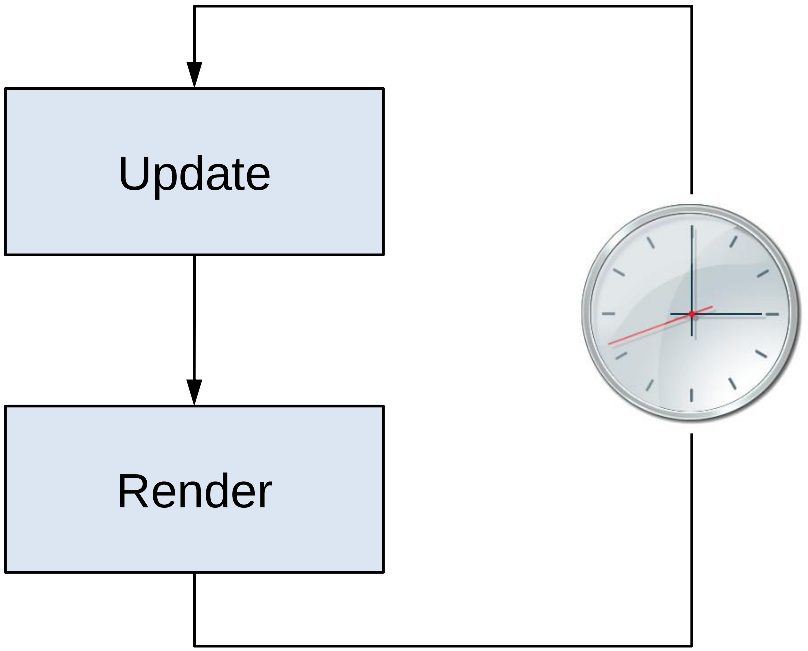
```
while (true) { // loop
   Timer(30);
   renderScore();
   renderPaddles();
   renderBall();
}
```

The game loop

- Advantages of the multi-threaded approach
  – Both update and render loops run at their own frame rate
- Disadvantages
  – Not all machines are that good at handling threads (single-CPU, precise timing problems)
  – Synchronization issues (two threads accessing the same data)

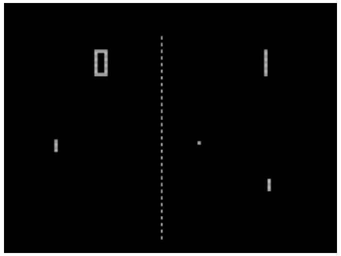
The game loop

- 3rd try: design a update/render single threaded decoupled loop



The game loop

- Example of what could be
  - *Pong* (1958 – *Atari Inc.*)



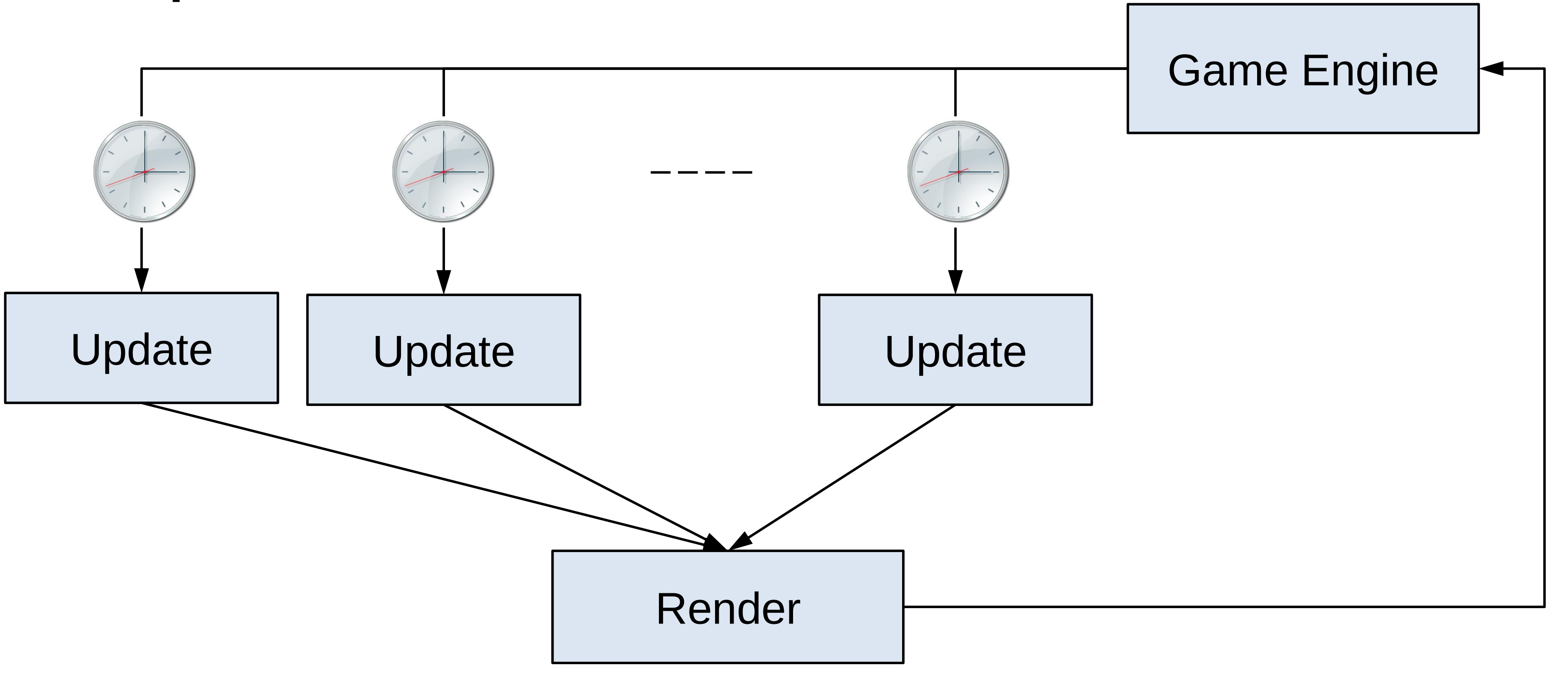
```
int main () {
    initGame();
    float lastCall = getTime(); // computer internal clock time
    while (true) { // Game loop
        if (getTime()-lastCall > 1/FREQ) {// timer
                readHumanInterfaceDevices();
                if (quitButtonPressed()) break;
                movePaddles();
                moveBall();
                if (scored()) {updateScore();resetBall();}
                lastCall = getTime();
        }
        // rendering frequency is "as fast as possible"
        renderScore();
        renderPaddles();
        renderBall();
    }
    return 0;
}
```

The game loop

- Advantages of the single-threaded decoupled approach
  - Better control than thread and simpler programming (no sharing and synchronization)
- Disadvantages
  - Assumes the tick takes 0 time to complete
  - No handling of Alt-Tab scenario
  - No nesting of increasing frequencies

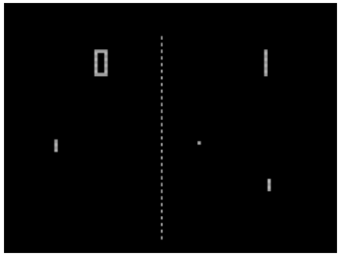
The game loop

- 4th try: design a frequency dependent update/render single threaded decoupled loop



The game loop

- Example of what could be
  - *Pong* (1958 – *Atari Inc.*)



```
int main () {
    HID.setFrequency(20);
    Paddles.setFrequency(10);
    Ball.setFrequency(10);
    while (true) { // Game loop
          HID.update();
          if (quitButtonPressed()) break;
          Paddles.update();
          Ball.update();
          if (scored()) {updateScore();resetBall();}
          lastCall = getTime();

          // rendering frequency is "as fast as possible"
          Score.render();
          Paddles.render();
          Ball.render();
    }
    return 0;
}
```

Universiteit Utrecht

The game loop

- Advantages of the frequency dependent single-threaded decoupled approach
  - Allow an individual frequency for each entity in the game
  - Same mechanism can be applied to rendering
  - Generic automatic registration mechanism
- Disadvantages
  - Need to specify the frequency ‘manually’ for each entity
  - The game engine needs an entry point for each entity to update (might be large)

The game loop

- What if the time between two updates is significantly larger than the required frequency?
  - Do nothing special: the game is ‘slowed down’
  - Update the game logic according to the actual time spend since the last call: introduce ‘visual gaps’
- Solutions
  - Speed-up update: decrease update frequency (if applicable), use game logic LoD, *etc.*
  - Speed-up rendering: use graphics LoD, lower the resolution, perform less special effects *etc.*
  - Can be done automatically with real-time profiling tools

Threads and synchronization

- Challenging task to ensure consistency
- Not all libraries and engines are thread-safe
  - A piece of code is thread-safe if it only manipulates shared data structures in a manner that guarantees safe execution by multiple threads at the same time
- What subsystem has the control at the threads?
  - input manager, core engine, game logic, thread creator component?

Process vs. Thread

- What is a process?
  - An OS entity that provides the context for
    - Executing program instructions
    - Managing resources (memory, I/O handles, ...)
  - A process is protected from other OS processes via memory management
  - Every process has its own address space

Process vs. Thread

- Each process must have at least one 'path of execution': main thread
- A thread is a path of execution
  - Threads share the same OS address space
    - Cheap data exchange
  - Threads can individually be stopped, started, paused, and new threads can be created
  - Threads are not 'protected': blocking or aborting a thread could influence the whole program

Threads

- Multithreading does not automatically increase performance
  - Multiple threads accessing the same data can result in a lot of synchronization overhead
  - But allows independent execution of code
- Win32 thread scheduling
  - Multi-processor machines management
  - In a cycle, each thread gets allocated a ‘time slice’
  - The threads can have different priorities

Win32 thread

```
#include <windows.h> // including Win32 threads declaration
#include <iostream>
#include <string>
using namespace std;

DWORD WINAPI MyRenderThread(LPVOID n) {
   string name = string(n);
   cout << "Executing render thread " << name << endl;
   while (true) {
        // code to render scene
   }
   return 0;
}

...
```

Win32 thread

```
...

int main() {

   DWORD iID;            // id number
   HANDLE RenderThread; // the Win32 thread
   DWORD waiter;        // flag

   // create the thread
   RenderThread = CreateThread(NULL,0,
                            MyRenderThread,"rendering",
                            0,&iID);
   // check for creation errors
   if (RenderThread == NULL) {
        DWORD dwError = GetLastError();
        cout << "Error in creating thread: "<< dwError << endl ;
        return 0;
   }

   // wait until thread has finished
   waiter = WaitForSingleObject(RenderThread,INFINITE);
   return 0;
}
```

Win32 thread

- The Win32 thread function

```
DWORD WINAPI threadName (LPVOID parameter) {
   Type typedParameter = (Type)parameter;
   // thread code
   return 0;
}
```

  - input parameter as *void ** type
    - any amount of data
    - type casting to use it in the function
    - usually custom *struct* to be send to thread
  - return type as *DWORD*

Win32 thread

- The Win32 thread creation

```
HANDLE WINAPI CreateThread(
    __in_opt LPSECURITY_ATTRIBUTES lpThreadAttributes,
    __in SIZE_T dwStackSize,
    __in LPTHREAD_START_ROUTINE lpStartAddress,
    __in_opt LPVOID lpParameter,
    __in DWORD dwCreationFlags,
    __out_opt LPDWORD lpThreadId );
```

  - *lpThreadAttributes*: pointer to structure to determine whether the handle can be inherited by child processes (NULL = cannot be inherited)
  - *dwStackSize*: initial size of stack (0 = default size)
  - *lpStartAddress*: pointer to the function to execute
  - *lpParameter*: pointer to the parameters of the function
  - *dwCreationFlags*: flags controlling the thread creation (run time)
  - *lpThreadId*: pointer to variable receiving identifier
  - returns
    - HANDLE: used for further operations like waiting, pausing, ending...
    - NULL if creation failed

Threads

- How to work on threads that are C/C++ OO-compliant?
  - This implementation is Windows-specific
  - For Linux or other OS, reimplementation is required (fork function)
- Solution: use platform-independent OO thread library, such as OpenThreads, or Boost::Thread
  - include both Win32 and pthread libraries
  - selection using pre-processor directives

OpenThreads

```
class MyThread : public OpenThreads::Thread {
   public:
        MyThread() : Thread() {
                // constructor
        }

        virtual ~MyThread() {
                // destructor
        }

        // Overriding thread running method from OpenThreads
        void run() {
                // thread execution code
        }
};
```

```
MyThread t;
t.run();
```

Thread issue example

- Two threads accessing the same data

```
if (!ptrInstance) ptrInstance = new Object();
```

1. Thread A evaluates condition (pointer NULL)
2. Thread A suspended
3. Thread B evaluate condition (pointer NULL)
4. Thread B creates new instance
5. Thread B suspended
6. Thread A creates new instance

⚠ Two instances have been created!

Locking mechanisms

- Semaphores
- Mutexes and Guards
- Other types of locking mechanisms
  - Condition Variables
    - notify locked thread from another thread
  - Monitor
    - uses condition variables
    - its methods are executed with mutual exclusion

Semaphores

- A semaphore is an object that limits the number of threads gaining simultaneous access to itself
  - dutch inventor Edsger Dijkstra
  - keeps an internal count of accessing threads
  - may optionally store references to the threads
- Can be used for
  - Limiting the number of concurrent database connections
  - Controlling the number of players connected to a server
  - *etc.*

Semaphores

- Three functions available
  - Init(int) to initialize the semaphore
  - P (Proberen) also called wait, waits for resource and decrements semaphore
  - V (Verhogen) also called signal, makes a resource available and increments semaphore

```
semaphore.Init(3);
...
semaphore.P();
// do something with semaphore resource
semaphore.V();
```

Mutex

- Mutex = mutually exclusive

```
OpenThreads::Mutex mutex; // shared by threads (e.g. static)
mutex.lock();
if (!ptrInstance) ptrInstance = new Object();
mutex.unlock();
```

  – Similar to binary semaphore behavior
  – Execute code without interruption
  – Disadvantages
    - unlock required before each return statement
    - bad efficiency

Guard

- Mutex as an object (Boost::Guard)

```
class Guard {
   public:
         Guard(OpenThread::Mutex& m) : _mutex(m) {
                  _mutex.lock();
         }
         virtual ~Guard() {
                  // automatic unlock when out of scope
                  _mutex.unlock();
         }
   private:
         OpenThreads::Mutex& _mutex;
};
```

```
OpenThreads::Mutex mutex;
Guard guard (mutex);
if (!ptrInstance) ptrInstance = new Object();
```

Critical section

- Lock/unlock mutex/guard at different places in the code to establish critical section

```
void MyClass::method() {

   // do some stuff here

   mutex.lock(); // enter critical section

   // do critical (uninterruptable) stuff here

   mutex.unlock(); // exit critical section

   // continue with more stuff

}
```

Critical section

⚠ Be very careful with the scope of the mutex/guard

```
OpenThreads::Mutex mutex;
if (!ptrInstance) { // <- not guarded
   Guard guard (mutex);
   ptrInstance = new Object();
}
```

Critical section

- Execution of the statement

```
ptrInstance = new Object();
```

1. Allocate memory for Object
2. Assign memory location to ptrInstance
3. Construct the object in the memory

```
ptrInstance = // step 2
    operator new (sizeof(Object)); // step 1
    new (ptrInstance) Object(); // step 3
```

Critical section

- Consider the following scenario
  – thread A executes (1) and (2) then is suspended
    - ptrInstance is not NULL but instance not constructed
  – thread B checks the NULL condition
    - do not enter as not NULL
  – thread B continues and uses a non fully initialized object!
- Solutions
  – To keep the mutex/guard before the check
  – To keep a local copy of ptrInstance

Deadlock

- Example

```
Guard (mutex1);                              thread A
// do critical stuff here
// <- interrupted here !
Guard (mutex2);
// do very critical stuff here
```

```
Guard (mutex2);                              thread B
// do critical stuff here
// <- interrupted here !
Guard (mutex1);
// do very critical stuff here
```

  - This can lead to deadlock if each thread is waiting for the other one
  - Deadlock can be avoided by careful design!

Volatile keyword

- Example

```
class GameEntity {
   public:
         void render();
         void update();
   private:
         bool updateFinished;
};
```

```
void GameEntity::render() {                              thread A
   while (!updateFinished) sleep(100);         // loops of 100ms
   GraphicsEngine::render(this);
}
```

```
void GameEntity::update() {                              thread B
   updateFinished = false;
   // update the Game Entity
   updateFinished = true;
}
```

Volatile keyword

- Due to optimizations, this will not work
  - as sleep has no effect on the instance, updateFinished is not re-evaluated by default
  - Thread A will deadlock
- Optimizations can be turned off using the volatile keyword
- In the GameEntity class

```
volatile bool updateFinished;
```

End of lecture #5

Next lecture

Design Patterns for Games