DM 2231
GAMES DEVELOPMENT TECHNIQUES

2015/16 SEMESTER 1

Week 2 – Game Application

MODULE SCHEDULE

Week	Dates	Topic	Remarks	Public Holidays
1	20-Apr-2015 to 24-Apr-2015	Module Introduction / 3D Game Programming	Issue Assignment 1	
2	27-Apr-2015 to 1-May-2015	Game Application		1 May. Labour Day
3	4-May-2015 to 8-May-2015	User Input		
4	11-May-2015 to 15-May-2015	Camera and GUI #1		
5	18-May-2015 to 22-May-2015	Camera and GUI #2		
6	25-May-2015 to 29-May-2015	Basic Game Physics		
7	1-Jun-2015 to 5-Jun-2015	Implementing Game Audio (E-learning)	Submit Assignment 1	1 Jun. Vesak Day
8	8-Jun-2015 to 12-Jun-2015	Mid-Sem Break		
9	15-Jun-2015 to 19-Jun-2015	Mid-Sem Break		
10	22-Jun-2015 to 26-Jun-2015	2D Game Programming #1	Issue Assignment 2	
11	29-Jun-2015 to 3-Jul-2015	2D Game Programming #2		
12	6-Jul-2015 to 10-Jul-2015	2D Game Programming #3		
13	13-Jul-2015 to 17-Jul-2015	Game Data		17 Jul. Hari Raya Puasa
14	20-Jul-2015 to 24-Jul-2015	Design Pattern #1		
15	27-Jul-2015 to 31-Jul-2015	Design Pattern #2		
16	3-Aug-2015 to 7-Aug-2015	Basic Artificial Intelligence (E-learning)		7 Aug. SG50 Public Holiday
17	10-Aug-2015 to 14-Aug-2015	Good Programming Practices	Submit Assignment 2	10 Aug. National Day

RECAP ON LAST WEEK'S LECTURE

- We have discussed about the main issues with 3D Game Development
  - Graphics for games are getting more complex
  - Smooth gameplay is dependent on refresh rate
  - Various Game Development techniques to reduce
    - the memory usage
    - the computation and processing of the entities

TABLE OF CONTENT

- Game Applications
  - Model-View-Controller architecture
  - Real-Time Loop
  - Game Logic

MODEL-VIEW-CONTROLLER ARCHITECTURE

- When programming a new game…
  - Do you start by first implementing some basic features?
  - As you develop more features, does your code gets more interwoven, and the classes bigger?
  - Do you spend a lot of time trying to recall what classes, methods and variables?
  - Do you spend a lot of time trying to rewriting what classes, methods and variables so that you can add new features in?
  - Do you want to easily modify your game’s display?
  - Do you work in teams?

MODEL-VIEW-CONTROLLER ARCHITECTURE

- Ponder over this…
  - Let's assume you are developing a 3D first person shooter game using C++ and OpenGL.
  - Your codes are contained in many classes and interwoven and called from your main class.
  - What if…
    - You want to change the input methods from keyboard+mouse to a joystick?
    - You want to change your codes to use DirectX instead of OpenGL?
    - You want to change the Artificial Intelligence techniques used
  - Do you need to unwoven your codes all the time?
    - Use MVC!

MODEL-VIEW-CONTROLLER ARCHITECTURE

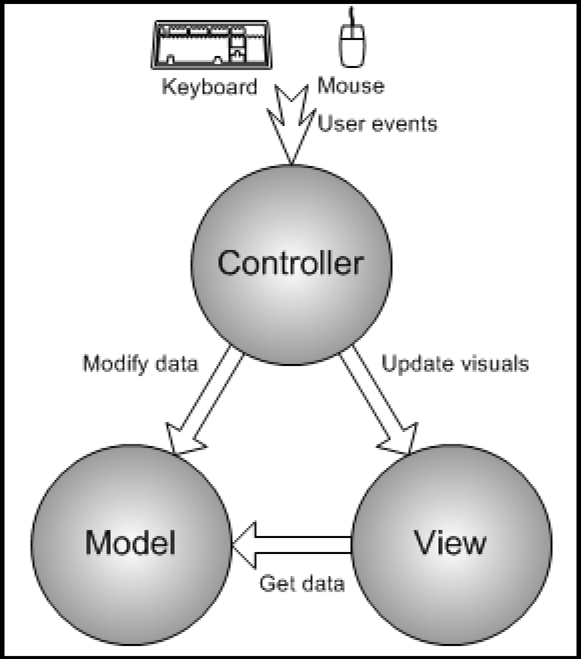
- What is MVC?
  - Model–view–controller (MVC) is a software architecture,
  - It is an architectural pattern in software engineering.
  - It isolates gameplay logic from input and rendering

MODEL-VIEW-CONTROLLER ARCHITECTURE

- This “Separation of Concerns” allows each layer to be developed, tested and maintained independently.
  - Graphics programmers work solely on rendering,
  - Gameplay programmers or designers work on gameplay
  - Whoever’s left can work on input.
- How to split?
  - Model – Gameplay (game entities, eg. Player, Sword)
  - View – Rendering
  - Controller – Input and non-gameplay flow (menu’s etc)

MODEL-VIEW-CONTROLLER ARCHITECTURE

- The controller takes input from the player and changes the model.
- The controller then passes the model (and any other relevant information) to the view to be rendered.



MODEL-VIEW-CONTROLLER ARCHITECTURE: ADVANTAGES

- Cleaner Code.
  - Large teams work independently on each layer without conflict.
  - Communication across layer boundaries defined with clear interfaces
  - As game grows, complexity is minimised.
- Better Cross Platform Support.
  - Gameplay is separate and not reliant on platform specific technology.
  - Rendering and input (both heavily platform specific) are separate
    - Can easily be modified or upgraded.

MODEL-VIEW-CONTROLLER ARCHITECTURE: ADVANTAGES

- Decoupled Rendering.
  - MVC decouples the game world (and input) from the rendering.
    - Rather than calling “Render” of each game entity, the rendering system gets the data from the model when rendering.
  - Simplifies addition of multi-threaded support to the game or renderer.
  - Multiple Views of the same model!

MODEL-VIEW-CONTROLLER ARCHITECTURE: DISADVANTAGES

- Complex to develop MVC applications.
- Not right suitable for small applications
  - Adverse effect in the application’s performance and design.
- Isolated development process by UI, game logic and controller programmers
  - may leads to delay in their respective modules development.

MODEL-VIEW-CONTROLLER ARCHITECTURE: SAMPLE

main.cpp

```cpp
#include <stdio.h>

#include "DM2231_Model.h"
#include "DM2231_View.h"
#include "DM2231_Controller.h"


int main( int argc, char* args )
{
    DM2231_Model* theModel = new DM2231_Model();
    DM2231_View* theView = new DM2231_View( theModel );
    DM2231_Controller* theController = new DM2231_Controller( theModel, theView );

    theController->RunMainLoop();

    delete theController;
    theController = NULL;
    delete theView;
    theView = NULL;
    delete theModel;
    theModel = NULL;

    return 0;
}
```

MODEL-VIEW-CONTROLLER ARCHITECTURE: SAMPLE

```
#include "DM2231_Controller.h"

DM2231_Controller::DM2231_Controller(DM2231_Model* theModel, DM2231_View* theView)
: theModel(NULL)
, theView(NULL)
, m_bContinueLoop(false)
{
    this->theModel = theModel;
    this->theView = theView;
}

DM2231_Controller::~DM2231_Controller(void)
{
}

// Get the status of the stop game boolean flag
bool DM2231_Controller::RunMainLoop(void)
{
    while (m_bContinueLoop)
    {
        // Get inputs from I/O devices
        ProcessInput();

        // Update the model
        theModel->Update();

        // Display the view
        theView->Draw();
    }

    return false;
}

// Process input from I/O devices
void DM2231_Controller::ProcessInput(void)
{
}
```

DM2231_Controller

MODEL-VIEW-CONTROLLER ARCHITECTURE: SAMPLE

DM2231_Model

```
#include "DM2231_Model.h"

DM2231_Model::DM2231_Model(void)
{
}

DM2231_Model::~DM2231_Model(void)
{
}

// Update the model
void DM2231_Model::Update(void)
{
}
```

MODEL-VIEW-CONTROLLER ARCHITECTURE: SAMPLE

```
#include "DM2231_View.h"

DM2231_View::DM2231_View(DM2231_Model* theModel)
{
    this->theModel = theModel;
}

DM2231_View::~DM2231_View(void)
{
}

// Draw the view
void DM2231_View::Draw(void)
{
}
```

DM2231_View

REAL-TIME LOOPS

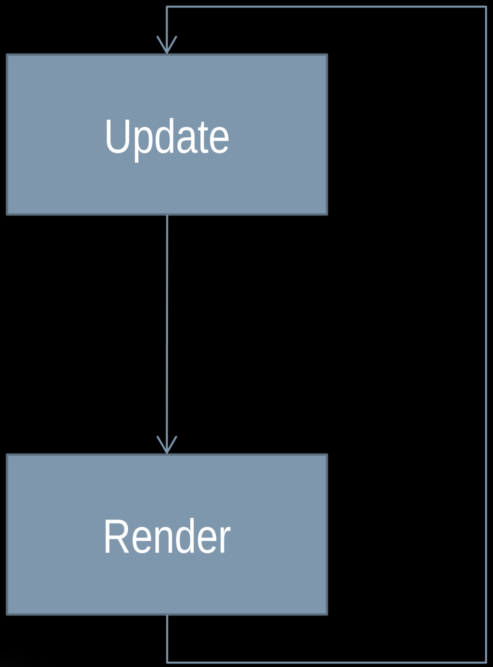
- Games are usually running in real-time
- The model, the view and the controller all need to be updated in real-time
  - Failure to update on time means lag in the game
  - Real-time loops are employed
    - Need to ensure the update rate is 30/60/100Hz
    - This rate is called Frame Rate
      - The average number of iterations through the loop per second
      - It should be consistent for playability

REAL-TIME LOOPS

- Games are usually running in real-time
- The model, the view and the controller all need to be updated in real-time
  - Failure to update on time means lag in the game
  - Real-time loops are employed
    - Need to ensure the update rate is 30/60/100Hz
    - This rate is called Frame Rate
      - The average number of iterations through the loop per second
      - It should be consistent for playability

REAL-TIME LOOPS

- Coupled Approach
  - Each update followed by a render call
  - Both have equal importance.
  - Logic and presentation are fully coupled with this approach.
- Question to ask…
  - What happens if the frames-rate varies due to changes in the level of complexity?
- Example:
  - If deploy on fast machines and slow machine…
  - Since number of logic cycles varies, will AI run slower on those slower machines?
  - What happens then?



REAL-TIME LOOPS: EXAMPLE

```
#include "DM2231_Controller.h"

DM2231_Controller::DM2231_Controller(DM2231_Model* theModel, DM2231_View* theView)
: theModel(NULL)
, theView(NULL)
, m_bContinueLoop(false)
{
    this->theModel = theModel;
    this->theView = theView;
}

DM2231_Controller::~DM2231_Controller(void)
{
}

// Get the status of the stop game boolean flag
bool DM2231_Controller::RunMainLoop(void)
{
    while (m_bContinueLoop)
    {
        // Get inputs from I/O devices
        ProcessInput();

        // Update the model
        theModel->Update();

        // Display the view
        theView->Draw();
    }

    return false;
}

// Process input from I/O devices
void DM2231_Controller::ProcessInput(void)
{
}
```

DM2231_Controller

Update

Render

REAL-TIME LOOPS: EXAMPLE

```
void Application::Run()
{
    //Main Loop
    Scene *scene = new SceneText();
    scene->Init();

    m_timer.startTimer();    // Start timer to calculate how long it takes to render this frame
    while (!glfwWindowShouldClose(m_window) && !IsKeyPressed(VK_ESCAPE))
    {
        GetMouseUpdate();
        scene->Update(m_timer.getElapsedTime());
        scene->Render();
        //Swap buffers
        glfwSwapBuffers(m_window);
        //Get and organize events, like keyboard and mouse input, window resizing, etc...
        glfwPollEvents();
        m_timer.waitUntil(frameTime);       // Frame rate limiter. Limits each frame to a s

    } //Check if the ESC key had been pressed or if the window had been closed
    scene->Exit();
    delete scene;
}
```

Update

Render

REAL-TIME LOOPS: COUPLED APPROACH

- Solution
  - The render part
    - Run as often as the hardware platform allows;
      - A faster computer should provide smoother animation, better frame rates, and so on.
  - The update part
    - Run at the speed it was designed for.
      - Characters must still walk at the speed the game was designed for or the gameplay will be destroyed

- Another problem
  - Clearly, having the render and update sections in sync makes coding complex, because one of them (update) has an inherent fixed frequency and the other does not.

REAL-TIME LOOPS: COUPLED APPROACH

- Another solution:
  - Keep update and render in sync but vary the granularity of the update routine according to the elapsed time between successive calls.
    - Compute the elapsed time, t (in real-time units),
    - Use t to scale the pacing of events
      - This ensures they happen at the right speed regardless of the hardware speed.

REAL-TIME LOOPS:
COUPLED APPROACH

```
long timelastcall=timeGetTime();
while (!end)
{
    if ((timeGetTime()-timelastcall)>1000/frequency)
    {
        Update();
        timelastcall=timeGetTime();
    }
    Render();
}
```

- This will be discussed in the following slides on “Single-thread Fully Decoupled Approach”

REAL-TIME LOOPS: COUPLED APPROACH

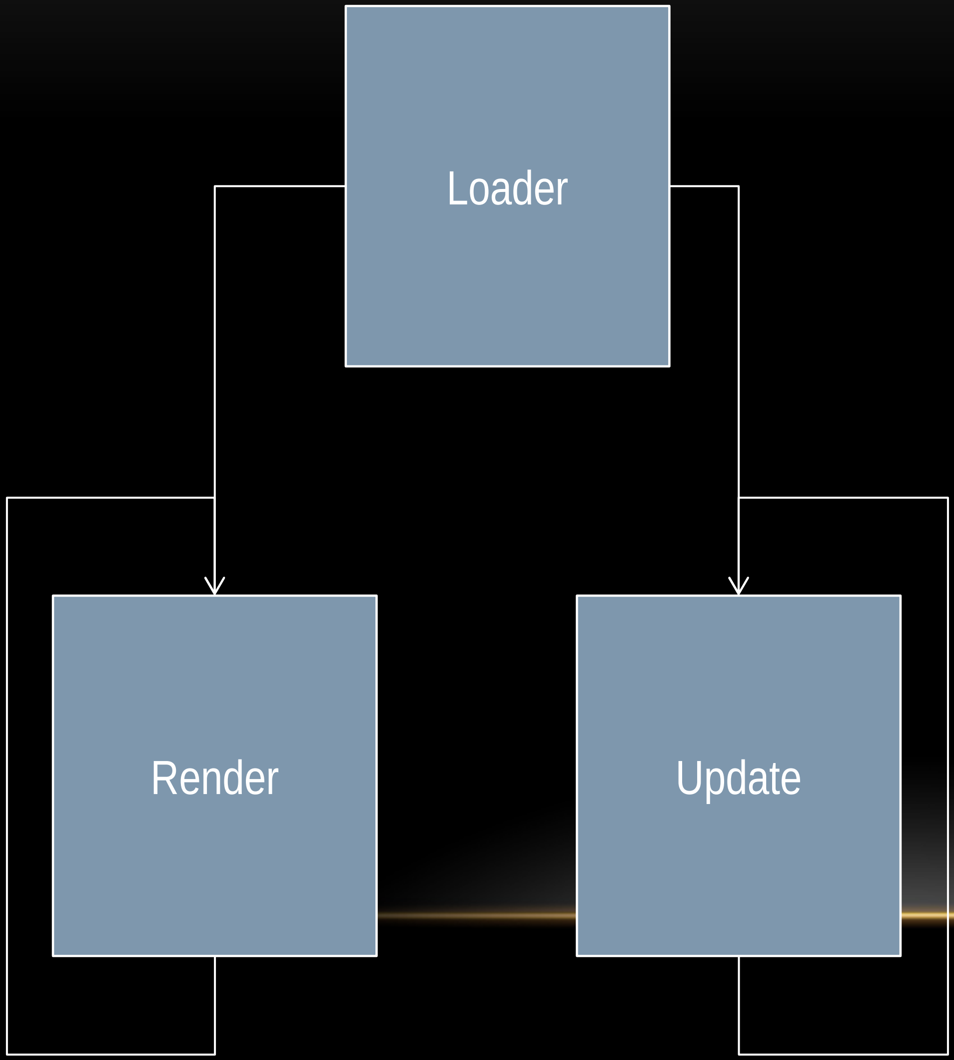
- But does it make sense to update the model more frequently?

- Question:
  - Does it make a difference between a character AI think 10 times versus 50 times per second?
- Answer:
  - No! The changes are minimal. And these precious clock cycles is wasted, and they could be used on rendering the display!

REAL-TIME LOOPS: TWIN-THREADED APPROACH

- Two threads
  - one thread runs the rendering portion
  - the other runs the update portion
- By controlling the frequency at which each routine is called,
  - the rendering portion gets as many calls as possible
  - while keeping a constant, hardware-independent resolution in the update portion.
- Executing the AI between 10 and 25 times per second is more than enough for most games.

REAL-TIME LOOPS: TWIN-THREADED APPROACH



- Advantages
  - Render can run at 60fps
  - AI can run at 15fps
  - Real AI code runs at fixed time step
    - Decision making routines
  - Simpler AI routines run on a per-frame basis.
    - Animation interpolators
    - Trajectory update

REAL-TIME LOOPS:
TWIN-THREADED APPROACH

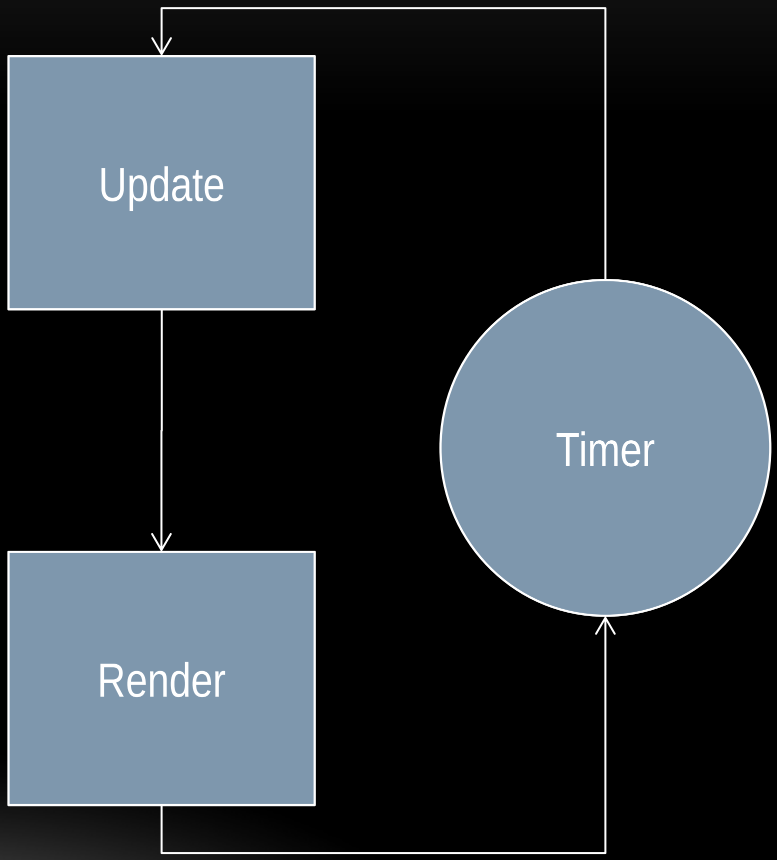
- Disadvantages
  - May not implement well on single-CPU machines
    - They use pre-emptive multi-tasking to do threading
      - Some threads may hold the CPU time longer than others
    - Not all threads may not return at the same time
    - Threads accessing shared memory? Locks?
    - Render thread have to wait for update thread to complete.
      - Degrades gameplay experience

REAL-TIME LOOPS:
SINGLE-THREAD FULLY DECOUPLED APPROACH

- Run update and render calls sequentially
  - Just like Coupled Approach
- Skip update calls
  - To maintain a fixed called rate
- Decouple the render from the update routine.
  - Render is called as often as possible,
  - update is synchronized with time.

REAL-TIME LOOPS: SINGLE-THREAD FULLY DECOUPLED APPROACH

- How to do this?
  - Compute the elapsed time between loops (using the time stamp)
    - and compare it with the inverse of the desired frequency.
    - Test if we need to run Update routine to maintain the desired call frequency.



REAL-TIME LOOPS: SINGLE-THREAD FULLY DECOUPLED APPROACH

- Example:
  - if you want to run the AI 20 times per second, you must call the update routine every 50 milliseconds.
  - Store the time at which you perform each call to update, and only execute it if 50 milliseconds have elapsed since then.
- Very popular mechanism because many times it offers better control than threads
- Simpler programming than thread programming.
- No concern over shared memory, synchronization, and so on.

REAL-TIME LOOPS:
SINGLE-THREAD FULLY DECOUPLED APPROACH

```
long timelastcall=timeGetTime();
while (!end)
{
   if ((timeGetTime()-timelastcall)>1000/frequency)
   {
      Update();
      timelastcall=timeGetTime();
   }
   Render();
}
```

- Problems:
  - Assumes that Update() is completed in 0 seconds
  - Does not cater for Alt-Tab

REAL-TIME LOOPS:
SINGLE-THREAD FULLY DECOUPLED APPROACH

More complete version…

```
time0 = getTickCount();
while (!bGameDone) {
     time1 = getTickCount();
     frameTime = 0;
     int numLoops = 0;
     while ((time1 - time0) > TICK_TIME && numLoops < MAX_LOOPS) {
          GameTickRun();
          time0 += TICK_TIME;
          frameTime += TICK_TIME;
          numLoops++;
     }
     IndependentTickRun(frameTime);
     // If playing solo and game logic takes way too long, discard pending time.
     if (!bNetworkGame && (time1 - time0) > TICK_TIME)
        time0 = time1 - TICK_TIME;
     if (canRender) {
          // Account for numLoops overflow causing percent > 1.
          float percentWithinTick = Min(1.f, float(time1 - time0)/TICK_TIME);
          GameDrawWithInterpolation(percentWithinTick);
     }
}
```

Update()

Render()

REAL-TIME LOOPS

- Question: How about real-time loops which is able to adapt to the speed of the machines?
  - Game will run smoothly on fast machines
  - Yet, game will compensate for slow machines (and framerate) by moving the camera or objects are a faster speed.
- Solution: FrameRate-Independent Movements. This will be discussed more in a later lecture.

REAL-TIME LOOPS: COUPLED APPROACH

```
// Get the status of the stop game boolean flag
bool DM2231_Controller::RunMainLoop(void)
{
	while (m_bContinueLoop)
	{
		theTimeControl.Now = theDevice->getTimer()->getTime();
		theTimeControl.DeltaTime = (f32)(theTimeControl.Now - theTimeControl.Then) / 1000.f; // Time in seconds
		theTimeControl.Then = theTimeControl.Now;

		// Get inputs from I/O devices
		ProcessInput();

		// Update the model
		theModel->Update();

		// Display the view
		theView->Draw();
	}

	return false;
}

// Process input from I/O devices
void DM2231_Controller::ProcessInput(void)
{
	// Move forward
	if(theEventReceiver.keyDown(irr::KEY_KEY_W))
		camera->setPosition( camera->getPosition() + oldTargetVec * MOVEMENT_SPEED * theTimeControl.DeltaTime );

	// Move backward
	if(theEventReceiver.keyDown(irr::KEY_KEY_S))
		camera->setPosition( camera->getPosition() - oldTargetVec * MOVEMENT_SPEED * theTimeControl.DeltaTime );
}
```

GAME LOGIC

- In the real-time loop, it is recommended that the game world is updated at a rate of 10 to 25 times per second.
- 3 main blocks to update:
  - the player
  - the world
  - the nonplaying characters (NPCs).

GAME LOGIC

- Player Update / World Update / NPC Update
  - Player Input: Read values from input devices
    - User inputs?
  - Player Restrictions: Computes restrictions to player interaction.
    - Collision Detection?
  - Player Update: impose restrictions on inputs

GAME LOGIC

- Player Update / <u>World Update </u>/ NPC Update
  - Updating the game environment
  - Two main categories of elements to update
    - Passive elements
      - Walls, trees, terrain.
    - Logic-based elements
      - Embedded behaviour
        - Flying birds in Left 4 Dead
        - Clouds moving past the sky
        - Fog moving across the town

GAME LOGIC

- Player Update / World Update / <u>NPC Update</u>

- This is covered under Basic Artificial Intelligence later this semester

GAME LOGIC

The 3 main block to update

Player Update

World Update

NPC Update

The breakdown

- Player update
  - Sense Player input
  - Compute restrictions
  - Update player state
- World update
  - Passive elements
    - Pre-select active zone for engine use
  - Logic-based elements
    - Sort according to relevance
    - Execute control mechanism
    - Update state
- AI based elements
  - Sort according to relevance
  - Sense internal state and goals
  - Sense restrictions
  - Decision engine
  - Update world

SUMMARY

- We have discussed about the main issues with Game Applications
  - Using good architectures to enhance development
  - How to use real-time loops in games
  - Develop good game logic