DM 2231
GAMES DEVELOPMENT TECHNIQUES

2015/16 SEMESTER 1

Week 3 – User Input

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 Game Applications
  - Using good architectures to enhance development
  - How to use real-time loops in games
  - Develop good game logic

TABLE OF CONTENT

- User Input
  - The Keyboard
  - Mouse
  - Hardware Abstraction
  - Frame-Independent movements,
  - Firing control
  - Weapons Control

KEYBOARD INPUT

- Main input device for PC-based games,
  - Also for mobile phones, some consoles, and palm devices.
- Keyboard mappings (eg, W-A-S-D keys)take time to learn
  - Not easy for small children to learn.
- Keyboard input can be read using a variety of methods to retrieve,
  - full strings,
  - others work on a key-by-key basis, and so on.

KEYBOARD INPUT

- For gaming purposes, two types of routines are relevant
  - Synchronous routines
    - Wait until a key is pressed and then report it to the application.
  - Asynchronous routines
    - Return immediately after being called, and give the application information about which keys were pressed, if any.

KEYBOARD INPUT

- Synchronous read modes are used to type information
  - Enter the character name in a role-playing game (RPG).
  - They poll the controller until new key input messages arrive.
  - Not well suited for real gameplay.
    - The game code must continually check to see whether keys were pressed, and whatever the response, keep drawing, executing the AI, and so on.

KEYBOARD INPUT

- Asynchronous provide fast tests to check the keyboard state efficiently. Two main techniques…
  - First one tests the state of individual keys each time
    - The programmer passes the key code as a parameter and gets the state as a result.
  - Second one, retrieves the whole keyboard state in a single call
    - The programmer can access the data structure and check for the state of each key without further hardware checks.
    - This is generally more efficient due to lesser computing overhead.
    - Used by DirectInput

KEYBOARD INPUT:
SYNCHRONOUS READ MODE

```
#include <stdio.h>
#include <string.h>

char *enter_a_string(int
maxcharacters)  {
    char *ptr;
    int len;
    printf("Enter string : ");
    char array[50];

    do {
        scanf("%s",array);
        len = strlen(array);
        if(len > maxcharacters) {
          printf("Maximum
%d!",maxcharacters);
        }
    } while(len > maxcharacters);

    ptr = array;
    return ptr;
}
```

```
int main() {
    char array[50];
    strcpy(array,get_persona
l_elements(15));
}
```

C++

KEYBOARD INPUT: ASYNCHRONOUS READ MODE #1

```
LRESULT CALLBACK WndProc(HWND hWnd, // Handle For This Window
        UINT uMsg, // Message For This Window
        WPARAM wParam, // Additional Message Information
        LPARAM lParam) // Additional Message Information
{
    switch (uMsg) // Check For Windows Messages
    {
        case WM_KEYDOWN: // Is A Key Being Held Down?
        {
            keys[wParam] = TRUE;// If So, Mark It As TRUE
            return 0;                       // Jump Back
        }

        case WM_KEYUP:              // Has A Key Been Released?
        {
            keys[wParam] = FALSE;// If So, Mark It As FALSE
            return 0;// Jump Back
        }

    }

    // Pass All Unhandled Messages To DefWindowProc
    return DefWindowProc(hWnd,uMsg,wParam,lParam);
}
```

```
BOOL CreateGLWindow(char* title, int width, int height, int bits, bool fullscreenflag)
{
    ....
    WNDCLASS    wc;  // Windows Class Structure
    wc.lpfnWndProc= (WNDPROC) WndProc; // WndProc Handles Messages

    if (!RegisterClass(&wc)) // Attempt To Register The Window Class
    {
        MessageBox(NULL,
                "Failed To Register The Window Class.",
                "ERROR",MB_OK|MB_ICONEXCLAMATION);
        return FALSE; // Return FALSE
    }
    ....
    return TRUE;   // Success
}
```

Win32 version

KEYBOARD INPUT: ASYNCHRONOUS READ MODE #1

```
void myApplication::inputKey(int key,
int x, int y)
{
  switch (key)
  {
  case GLUT_KEY_LEFT :
    moveMeFlat(Vector3D(-1,0,0));
    break;
  case GLUT_KEY_RIGHT :
    moveMeFlat(Vector3D(1,0,0));
    break;
  case GLUT_KEY_UP :
    moveMeFlat(Vector3D(0,0,1));
    break;
  case GLUT_KEY_DOWN :
    moveMeFlat(Vector3D(0,0,-1));
    break;
  …
  }
}
```

```
int main(int argc, char **argv )
{
  …
  glutSpecialFunc(inputKey);
  …
  return 0;
}
```

OpenGL+GLUT version

KEYBOARD INPUT: ASYNCHRONOUS READ MODE #2

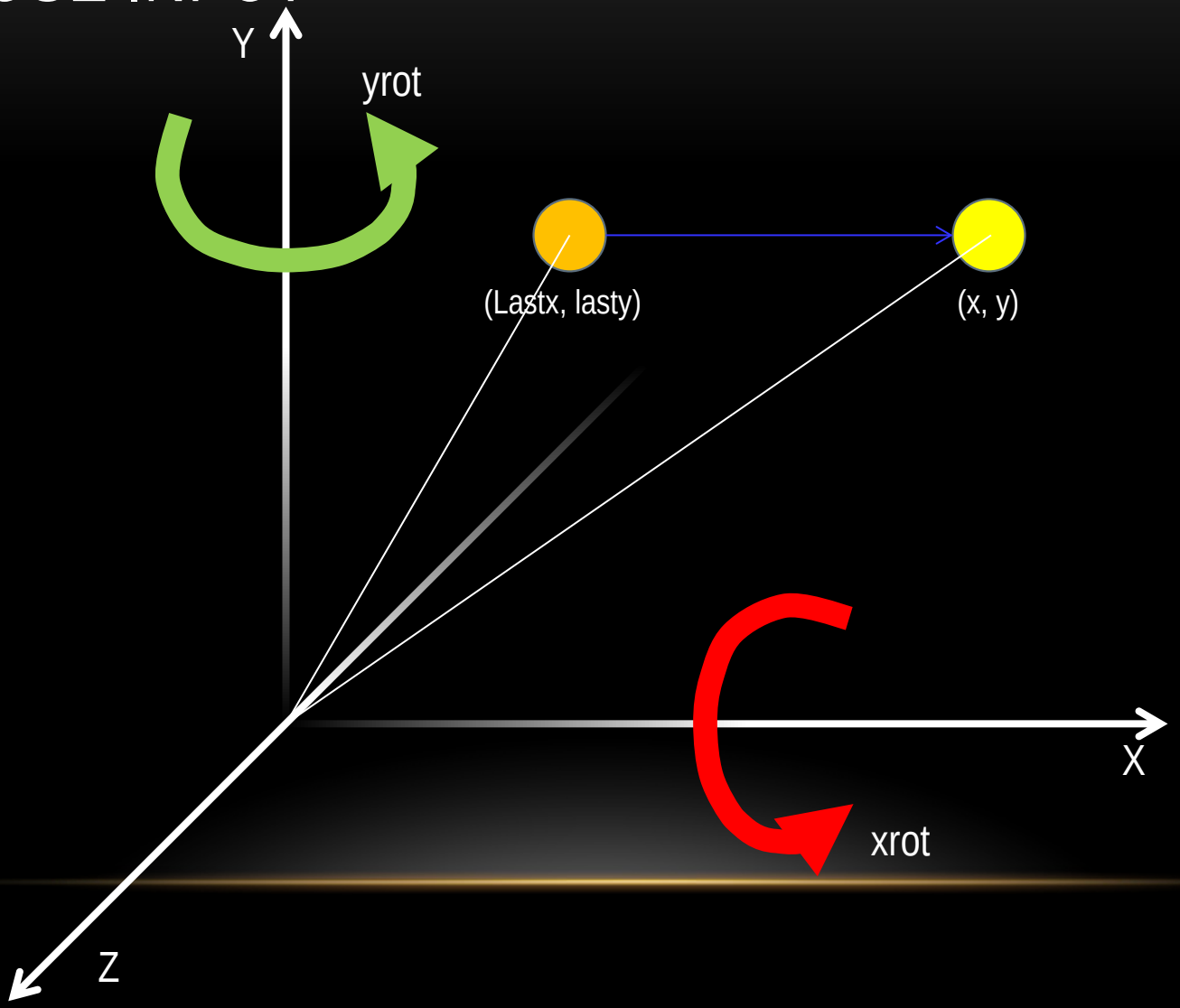
- DirectInput extracts the status of the keyboard state buffer at one go

```
BYTE diks[256]; // DirectInput keyboard state buffer
ZeroMemory( diks, sizeof(diks) );
hr = g_pKeyboard->GetDeviceState( sizeof(diks), diks );
if( FAILED(hr) )
{
        hr = g_pKeyboard->Acquire();
        while( hr == DIERR_INPUTLOST || hr== DIERR_OTHERAPPHASPRIO)
        hr = g_pKeyboard->Acquire();
}
```

MOUSE INPUT

- In 3D first person shooter games, keyboard is used to change position, while the mouse reorients the viewpoint.
  - Buttons are used to perform actions like firing weapons
- In 2D games, the mouse is used to select objects, move objects, select destination etc.

MOUSE INPUT



MOUSE INPUT

```
void mouseMovement(int x, int y) {
    int diffx=x-lastx;      // difference between the current x and the last x position
    int diffy=y-lasty;      //difference between the current y and the last y position
    lastx=x;                //set lastx to the current x position
    lasty=y; //set lasty to the current y position
    xrot += (float) diffy; //add the difference in the y positions to xrot
    yrot += (float) diffx;  //add the difference in the x positions to yrot
}

int main (int argc, char **argv) {
    ....
    glutPassiveMotionFunc(mouseMovement); //check for mouse movement
    ....
}
```

MOUSE INPUT

- Linear mouse motion in X- and Y-axes are converted to rotational motion
  - Use trigonometry

```
yrotrad = (yrot / 180 * 3.141592654f);
xrotrad = (xrot / 180 * 3.141592654f);
xpos -= float(sin(yrotrad));
zpos += float(cos(yrotrad)) ;
ypos += float(sin(xrotrad));
```

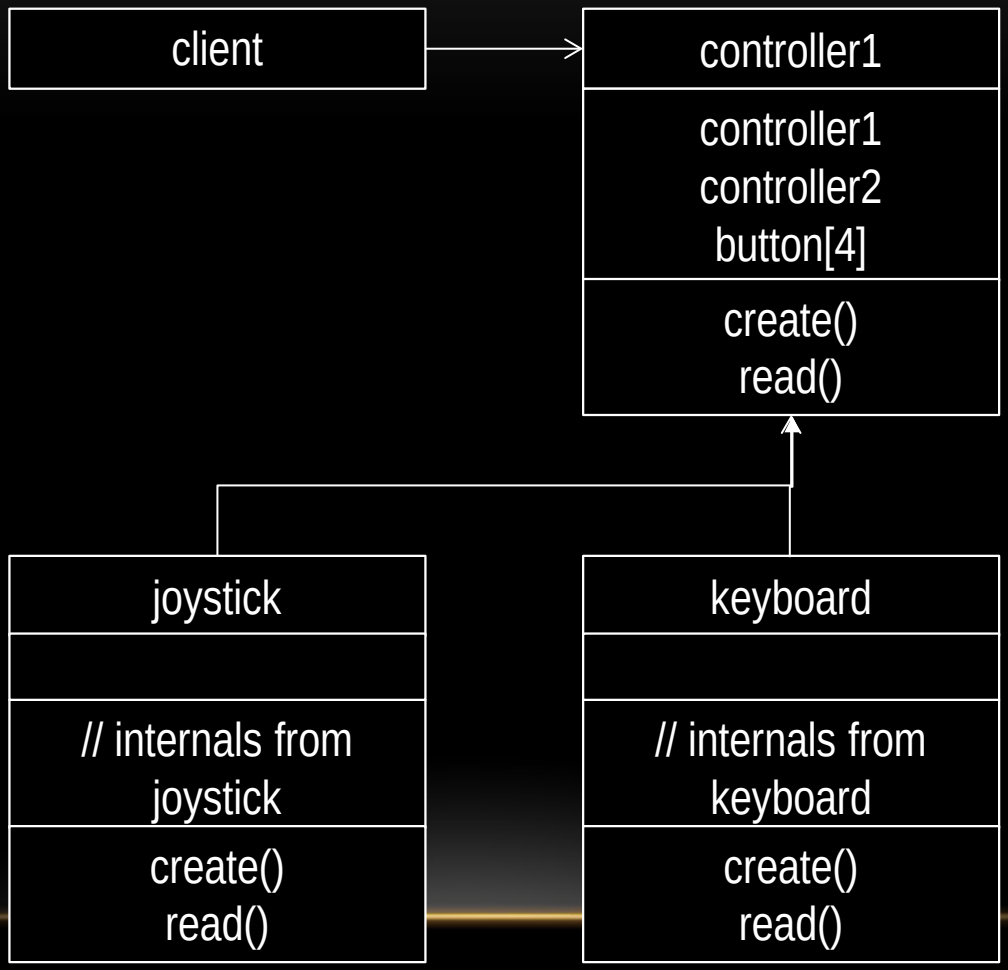
HARDWARE ABSTRACTION

- Modern games can be played a variety of input controllers,
  - Standard devices
    - keyboard, mouse, joysticks
  - Exotic devices
    - aircraft-style joysticks, snowboards, dance pads of all sorts, and even a fishing rod!
- Gamers may choose 1 or a combination of input devices to use.
- This complicates the programming of the codes.
  - How should the developer code his game to cater for this?
  - Hardware abstraction?

HARDWARE ABSTRACTION

- Coding your game with a "virtual" controller in mind,
- Any controller that conforms to that "virtual" controller’s profile can be used by the game engine seamlessly.
- How to do this?
  - Write a generic controller handler (usually, a pure abstract class) from which specific controllers are derived via inheritance.
  - Then, at runtime, only the kind of controller selected by the player is created
  - Advantage: Provides a seamless and elegant way of integrating different controllers.

HARDWARE ABSTRACTION



FRAME-INDEPENDENT MOVEMENTS

- Faster computer = higher frame rate
- Slower computer = lower frame rate

- For games with frame dependent movements, will the game be played faster?
- How to make the speed of gameplay consistent for fast and slow computers?
  - Use frame-independent movements

FRAME-INDEPENDENT MOVEMENTS

- Frame-independent movements detects the time difference between frames
  - Uses this difference to compute the movement of characters
    - Faster computer = characters move lesser between frames
    - Slower computer = characters move more between frames

FRAME-INDEPENDENT MOVEMENTS

```
// In order to do framerate independent movement, we have to know
// how long it was since the last frame
u32 then = device->getTimer()->getTime();

// This is the movemen speed in units per second.
const f32 MOVEMENT_SPEED = 5.0f;

while(device->run())
{
        // Work out a frame delta time.
        const u32 now = device->getTimer()->getTime();
        const f32 frameDeltaTime = (f32)(now - then) / 1000.f; // Time in seconds
        then = now;
```

FRAME-INDEPENDENT MOVEMENTS

```
        core::vector3df nodePosition = node->getPosition();

        if(receiver.IsKeyDown(irr::KEY_KEY_W))
                nodePosition.Y += MOVEMENT_SPEED * frameDeltaTime;
        …
        node->setPosition(nodePosition);

        // Do other stuff
        …
        // draw the 3D scene
        drawAll();
}
```

FIRING CONTROL

- In real world, various guns have different firing rates.
  - Need to consider reloading time, automatic, semi-automatic or manual gun, single barrel or multiple barrels.
- If we are making a 3D FPS game, then how can we vary the rate of the guns firing the bullets?
  - Use Firing Control to manage the firing rates!
    - Prevent the gamer from firing another round until a certain time had lapsed after firing one round.
    - Prevent the gamer from firing another round until a certain time had lapsed after initiating a change of magazine.

FIRING CONTROL

- Key considerations for firing control
  - Firing rate
  - Ammunition per magazine
  - Magazine changing time
- Example: SAR 21
  - 450–650 rounds/min
    - 7.5 – 10.833 rounds/s
    - 0.1333s – 0.092s delay between each round fired
  - 30-round box magazine

FIRING CONTROL

```
bool CWeaponObject::UseWeapon( float fTimeStamp )
{
    if (theWeaponType > EMPTY)
    {
        if (( iWeaponAmmunition > 0 ) && (fTimeStamp - fTimeSinceLastRound > 133.3f))
        {
            iWeaponAmmunition--;
            bWeaponFired = true;
            fTimeSinceLastRound = fTimeStamp;
            return true;
        }
    }

    bWeaponFired = false;
    return false;
}
```

WEAPONS CONTROL

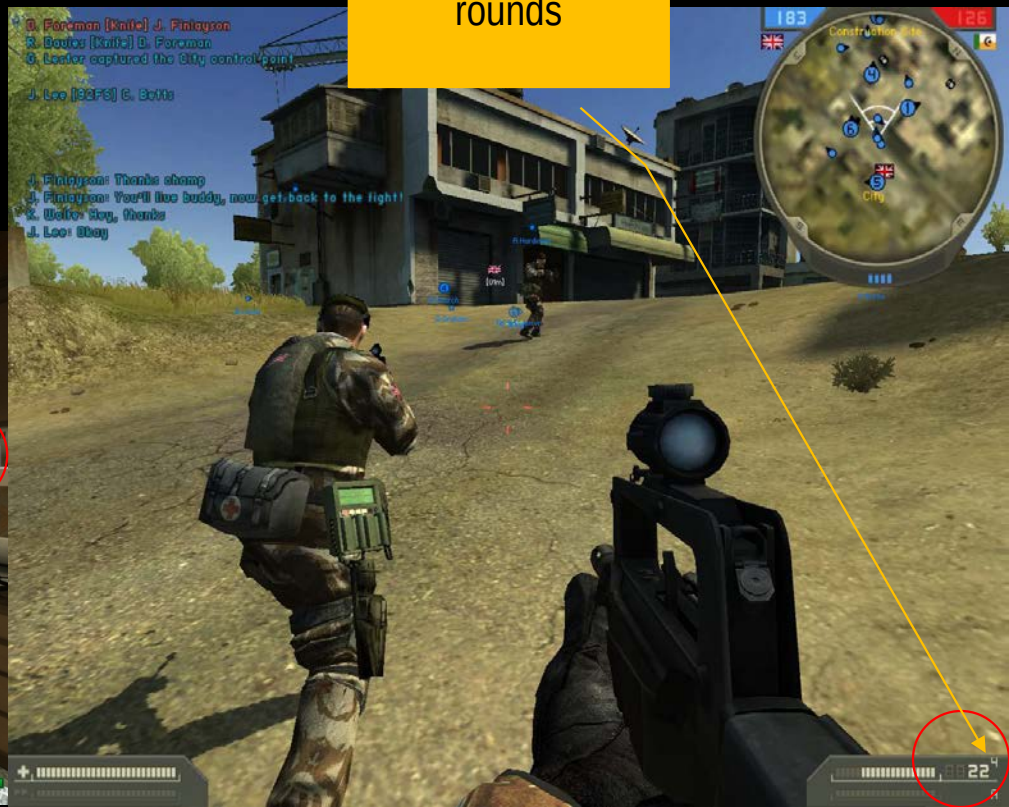
- Most 3D First Person shooter games has players who can have more than 1 ammunition clip for their guns.
  - If a player can have 30-rounds in 1 clip and he is carrying the maximum of 5 clips, then he has a total of 150 bullets.
  - If he picks up a new clip of 30-rounds, then does he have 6 clips?
  - If he picks up a new clip of 30-rounds, but he has already used 20 rounds from his current clip, then how many clips should he have?
  - If he has already used 20 rounds from his current clip, and he does a RELOAD, then how many clips should he have?
- How to code all these?

WEAPONS CONTROL

- Question:
  - Is it better to record the ammunition in the game as total clips or total rounds?

total ammunitions - 480 rounds

total clips - 480 rounds



WEAPONS CONTROL

```
bool CWeaponObject::UpdateWeaponAmmunition(const int iAmmunitionUsed)
{
        if (iWeaponAmmunition < iAmmunitionUsed)
                return false;

        iWeaponAmmunition -= iAmmunitionUsed;
        if (iWeaponAmmunition < 0)
                ReloadWeapon();

        return true;
}
```

Check if the ammunition used is more than what the player has

Reduce the current ammunition by the ones used up.

If the ammunition is less than 0, then reload weapon

WEAPONS CONTROL

```
// Functions to operate on this class
bool CWeaponObject::ReloadWeapon(void)
{
    if (iWeaponAmmunitionClip<=0)
        return false;

    int totalAmmo = iWeaponAmmunition + iWeaponTotalAmmunition;
    if (totalAmmo > iWeaponAmmunitionClipCapacity)
    {
        iWeaponAmmunition = iWeaponAmmunitionClipCapacity;
        iWeaponTotalAmmunition -= iWeaponAmmunitionClipCapacity;
        iWeaponAmmunitionClip = (int) (iWeaponTotalAmmunition / iWeaponAmmunitionClipCapacity);
    }
    else
    {
        iWeaponAmmunition = iWeaponAmmunitionClipCapacity;
        iWeaponAmmunitionClipCapacity = 0;
        iWeaponAmmunitionClip = 0;
    }

    return true;
}
```

If the player has more than 1 clip, then fill up the current clip with ammo, and update number of clips

If the player has only 1 clip, then use it.

WEAPONS CONTROL

```
bool CWeaponObject::AddWeaponAmmunitionClip(void)
{
        if (iWeaponAmmunitionClip == iWeaponAmmunitionMaxClip)
                return false;

        iWeaponAmmunitionClip++;
        return true;
}
```

Check if the player has the maximum number of clips

Add the number of clip by 1

SUMMARY

- We have discussed about the main issues with User Input
  - Techniques to get input from the keyboard and mouse
  - Using Hardware Abstraction to create codes for generic input
  - Using Frame-Independent Movements to run with the same gameplay speed on both fast and slow hardwares
  - Use Firing and Weapons Control to make the 3D FPS games more realistic