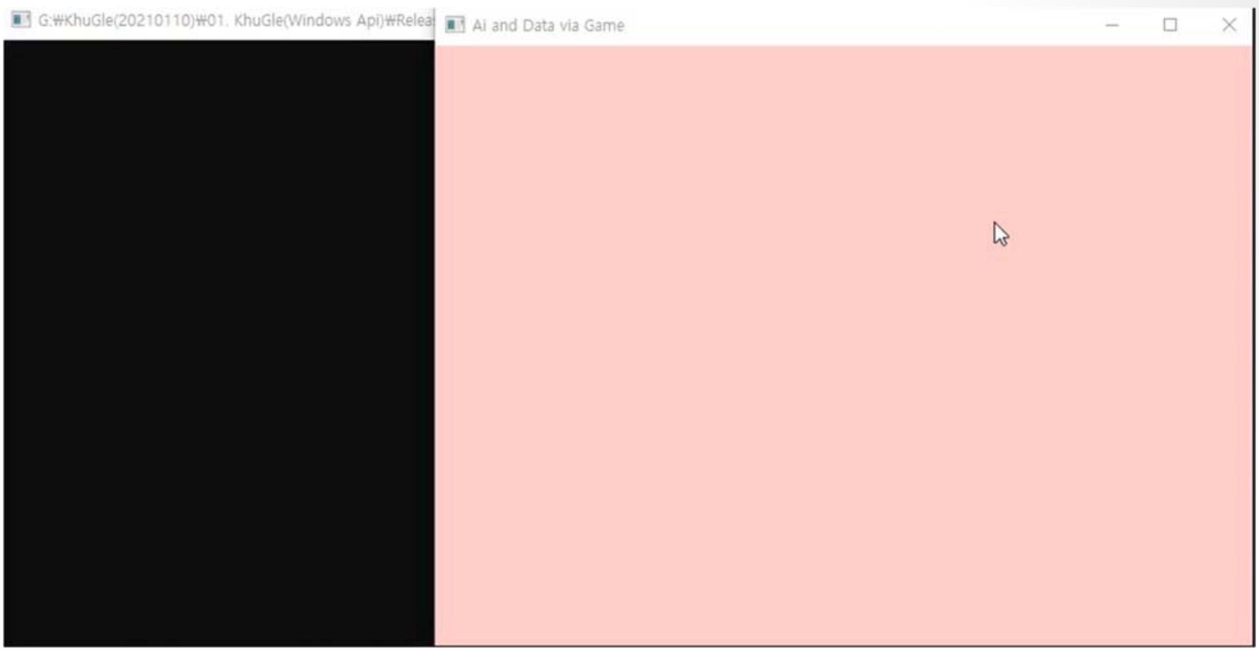
AI and Data Analysis via Game Programming

Kyung Hee University
Daeho Lee

Schedule

- Windows API
- Game Layout
- Collision and Physics
- 3D Rendering
- Sound Processing
- Image Processing
- Correlation and Clustering
- Regression
- Performance Evaluation
- Perceptron
- MLP(DNN)
- CNN

1. Windows API



Project

- Win32 Console Application
- Setting
  - General
    - Project Default
      - Character Set: Use Multi-Byte Character Set
  - C/C++
    - General
      - SDL checks: No
  - Linker
    - System
      - SubSystem: Console

WinMain

Windows API (1)

- API (application programming interface) that is used to create Windows applications
- Windows SDK (software development kit)

- int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow);

```
#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
     PWSTR szCmdLine, int CmdShow) {

     MessageBox(NULL, szCmdLine, "Title", MB_OK);

     return 0;
}
//MBCS (Multi-Byte Character Set) & Unicode Character Set
// "text" & L "text"
```

WindowProc callback function
Message handling function

Windows API (2)

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
  …
  switch (message) {
    case WM_CREATE: break;
    …
  }
  …
}
int APIENTRY WinMain(HINSTANCE hInstance,
  HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
  WNDCLASSEX windowClass;
  …
  windowClass.lpfnWndProc = WndProc;
  while(1) {
    if(PeekMessage(&msg, 0, 0, 0, PM_REMOVE)) {
        …
  }
}
```


Application class
– create window, message handling

KhuGleWin.h (1)

```
#pragma once
#include <windows.h>

class CKhuGleWin;

void KhuGleWinInit(CKhuGleWin *pApplication); // call WinMain (Global Function)

class CKhuGleWin {
public:
  HWND m_hWnd;
  int m_nW, m_nH;

  static CKhuGleWin *m_pWinApplication;

  int m_nDesOffsetX, m_nDesOffsetY;
  int m_nViewW, m_nViewH;
```

Application class
– create window, message handling

KhuGleWin.h (2)

```
  _int64 m_TimeCountFreq, m_TimeCountStart, m_TimeCountEnd;
  double m_Fps, m_ElapsedTime;

  bool m_bKeyPressed[256];
  bool m_bMousePressed[3];
  int m_MousePosX, m_MousePosY;

  WINDOWPLACEMENT m_wpPrev;

  static LRESULT CALLBACK WndProc(HWND hwnd, UINT message,  WPARAM wParam,
    LPARAM lParam);
  LRESULT CALLBACK WndProcInstanceMember(HWND hwnd,
    UINT message, WPARAM wParam, LPARAM lParam);

  void Fullscreen(); // Full screen, toggle
```


Application class
– create window, message handling

KhuGleWin.h (3)

```
  void GetFps();
  virtual void Update();  // called by 'WinMain' loop
  void OnPaint();         // Client paint

  void ToggleFpsView();

  CKhuGleWin(int nW, int nH);
  virtual ~CKhuGleWin();
  bool m_bViewFps;
};
```

Debugging new

KhuGleWin.cpp (1)

```
#include "KhuGleWin.h"
#include <cmath>
#include <cstdio>
#include <iostream>

#pragma warning(disable:4996)      // function, class member,
                                   //variable, or
                                   // typedef that's marked deprecated
#define _CRTDBG_MAP_ALLOC // memory leak detection
#include <cstdlib>        // _CrtDumpMemoryLeaks() is called in WinMAIN
#include <crtdbg.h>

#ifdef _DEBUG
#ifndef DBG_NEW
#define DBG_NEW new ( _NORMAL_BLOCK , __FILE__ , __LINE__ )
#define new DBG_NEW
#endif
#endif  // _DEBUG
```


WinMain call
Application class

KhuGleWin.cpp (2)

```
CKhuGleWin *CKhuGleWin::m_pWinApplication = 0;
void KhuGleWinInit(CKhuGleWin *pApplication) {
  CKhuGleWin::m_pWinApplication = pApplication;
  WinMain(0, 0, 0, 0);
}

CKhuGleWin::CKhuGleWin(int nW, int nH){
  m_nW = nW;      m_nH = nH;
  m_bViewFps = false;

  for(int i = 0 ; i < 256 ; ++i)
     m_bKeyPressed[i] = false; // m_bKeyPressed['A'], m_bKeyPressed[VK_LEFT]

  for(int i = 0 ; i < 3 ; ++i) // left, wheel (middle), right
     m_bMousePressed[i] = false;
}

CKhuGleWin::~CKhuGleWin() {
}
```

KhuGleWin.cpp (3)

```
LRESULT CALLBACK CKhuGleWin::WndProc(HWND hwnd, UINT message, WPARAM wParam,
  LPARAM lParam){ // static
  return m_pWinApplication->WndProcInstanceMember(hwnd,
         message, wParam, lParam);
}

LRESULT CALLBACK CKhuGleWin::WndProcInstanceMember(HWND hwnd,
  UINT message, WPARAM wParam, LPARAM lParam)
{
  int width, height;
  HDC hdc;                 // Dc, Brush, Pen, font, …
  HBRUSH hBrushGray;
  RECT rt;

  hBrushGray = (HBRUSH)GetStockObject(GRAY_BRUSH);
```


KhuGleWin.cpp (4)

```
  double AspectOrg, AspectWin;

  switch (message)   {
        case WM_CREATE:
          break;

        case WM_PAINT:
          OnPaint();
          break;

        case WM_CLOSE:
          PostQuitMessage(0);
          break;
```

KhuGleWin.cpp (5)

```
case WM_SIZE:
        height = HIWORD(lParam);width = LOWORD(lParam);
        AspectOrg = (double)m_nW/(double)m_nH;
        AspectWin = (double)width/(double)height;
        m_nDesOffsetX = 0;        m_nDesOffsetY = 0;
        m_nViewW = width;m_nViewH = height;

        if(AspectWin > AspectOrg){
          m_nDesOffsetX =
                 (int)((AspectWin-AspectOrg)*height/2.);
          m_nViewW =
                 (int)(height*AspectOrg);
        }
        else{
          m_nDesOffsetY =
                 (int)((1./AspectWin-1./AspectOrg)*width/2.);
          m_nViewH = (int)(width/AspectOrg);
        }
        break;
```


KhuGleWin.cpp (6)

```
case WM_LBUTTONDOWN:
 m_MousePosX = (LOWORD(lParam)- m_nDesOffsetX)*m_nW/m_nViewW;
 m_MousePosY = (HIWORD(lParam)-m_nDesOffsetY)*m_nH/m_nViewH;
 m_bMousePressed[0] = true;
 break;

case WM_LBUTTONUP:
 m_MousePosX = (LOWORD(lParam)-m_nDesOffsetX)*m_nW/m_nViewW;
 m_MousePosY = (HIWORD(lParam)-m_nDesOffsetY)*m_nH/m_nViewH;
 m_bMousePressed[0] = false;
 break;
```

KhuGleWin.cpp (7)

```
case WM_MBUTTONDOWN:
 m_MousePosX = (LOWORD(lParam)-m_nDesOffsetX)*m_nW/m_nViewW;
 m_MousePosY = (HIWORD(lParam)-m_nDesOffsetY)*m_nH/m_nViewH;
 m_bMousePressed[1] = true;
 break;

case WM_MBUTTONUP:
 m_MousePosX = (LOWORD(lParam)-m_nDesOffsetX)*m_nW/m_nViewW;
 m_MousePosY = (HIWORD(lParam)-m_nDesOffsetY)*m_nH/m_nViewH;
 m_bMousePressed[1] = false;
 break;
```


KhuGleWin.cpp (8)

```
case WM_RBUTTONDOWN:
 m_MousePosX = (LOWORD(lParam)-m_nDesOffsetX)*m_nW/m_nViewW;
 m_MousePosY = (HIWORD(lParam)-m_nDesOffsetY)*m_nH/m_nViewH;
 m_bMousePressed[2] = true;
 break;

case WM_RBUTTONUP:
 m_MousePosX = (LOWORD(lParam)-m_nDesOffsetX)*m_nW/m_nViewW;
 m_MousePosY = (HIWORD(lParam)-m_nDesOffsetY)*m_nH/m_nViewH;
 m_bMousePressed[2] = false;
 break;
```

KhuGleWin.cpp (9)

```
case WM_MOUSEMOVE:
 m_MousePosX = (LOWORD(lParam)-m_nDesOffsetX)*m_nW/m_nViewW;
 m_MousePosY = (HIWORD(lParam)-m_nDesOffsetY)*m_nH/m_nViewH;
 break;
case WM_KEYDOWN:
 switch (wParam){
       case VK_F11:
         Fullscreen();
         break;
       case VK_F12:
         ToggleFpsView();
         break;
       case VK_LEFT:
         break;
 }
 if(wParam >= 0 && wParam < 256)
       m_bKeyPressed[wParam] = true;
 break;
```


KhuGleWin.cpp (10)

```
case WM_KEYUP:
 if(wParam >= 0 && wParam < 256)
       m_bKeyPressed[wParam] = false;
       break;

case WM_CHAR:
 switch (wParam) {
       case 'a':
        break;
 }
 break;
```

KhuGleWin.cpp (11)

```
  case WM_ERASEBKGND:
   hdc = (HDC) wParam;
   GetClientRect(hwnd, &rt);
   SetMapMode(hdc, MM_ANISOTROPIC);
   SetWindowExtEx(hdc, 100, 100, NULL);
   SetViewportExtEx(hdc, rt.right, rt.bottom, NULL);
   FillRect(hdc, &rt, hBrushGray);
   break;

  default:
   break;
  }
  return (DefWindowProc(hwnd, message, wParam, lParam));
}
```


KhuGleWin.cpp (12)

```
void CKhuGleWin::Fullscreen() {
  DWORD dwStyle = GetWindowLong(m_hWnd, GWL_STYLE);
  if(dwStyle & WS_OVERLAPPEDWINDOW) {
    m_wpPrev.length = sizeof(WINDOWPLACEMENT);
    MONITORINFO mi = {sizeof(MONITORINFO)};
    if(GetWindowPlacement(m_hWnd, &m_wpPrev) &&
      GetMonitorInfo(MonitorFromWindow(m_hWnd, MONITOR_DEFAULTTOPRIMARY), &mi))
    {
      SetWindowLong(m_hWnd, GWL_STYLE,
        dwStyle & ~WS_OVERLAPPEDWINDOW);
      SetWindowPos(m_hWnd, HWND_TOP,
        mi.rcMonitor.left, mi.rcMonitor.top,
        mi.rcMonitor.right - mi.rcMonitor.left,
        mi.rcMonitor.bottom - mi.rcMonitor.top,
        SWP_NOOWNERZORDER | SWP_FRAMECHANGED);
    }
  }
```

Application class
Full screen

KhuGleWin.cpp (13)

```
  else {
    SetWindowLong(m_hWnd, GWL_STYLE, dwStyle | WS_OVERLAPPEDWINDOW);
    SetWindowPlacement(m_hWnd, &m_wpPrev);
    SetWindowPos(m_hWnd, NULL, 0, 0, 0, 0,
      SWP_NOMOVE | SWP_NOSIZE | SWP_NOZORDER |
      SWP_NOOWNERZORDER | SWP_FRAMECHANGED);
  }
}
```


Application class
GetFps
Update

KhuGleWin.cpp (14)

```
void CKhuGleWin::GetFps() {
  QueryPerformanceCounter((LARGE_INTEGER*)&m_TimeCountEnd);
  m_ElapsedTime = (double)(m_TimeCountEnd -
        m_TimeCountStart)/(double)m_TimeCountFreq;
  m_TimeCountStart = m_TimeCountEnd;
  m_Fps = 1./m_ElapsedTime;
}
void CKhuGleWin::Update() { // called in WinMain
  RECT Rect;
  GetClientRect(m_hWnd, &Rect);
  InvalidateRect(m_pWinApplication->m_hWnd, &Rect, false);

  char strFps[200];
  sprintf(strFps, "FPS: %7.3lf, Elapsed time: %lf", m_Fps,  m_ElapsedTime);
  if(m_bViewFps){
    std::cout << strFps << std::endl;
  }
}
```

KhuGleWin.cpp (15)

```
void CKhuGleWin::OnPaint() {
  RECT Rect;
  GetClientRect(m_hWnd, &Rect);
  int nW = Rect.right-Rect.left;
  int nH = Rect.bottom-Rect.top;

  if(nW <= 0 || nH <= 0) return;

  PAINTSTRUCT ps;
  HDC hdc = BeginPaint(m_hWnd, &ps);
  HDC hDC, hCompDC;
  hDC = GetDC(m_hWnd);
  hCompDC = CreateCompatibleDC(hDC);

  HBITMAP hBitmap;
  hBitmap = CreateCompatibleBitmap(hDC, nW, nH);
  SelectObject(hCompDC, hBitmap);
```


KhuGleWin.cpp (16)

```
  BITMAPINFOHEADER bmiHeader;

  bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
  bmiHeader.biWidth = m_nW;
  bmiHeader.biHeight = m_nH;
  bmiHeader.biPlanes = 1;
  bmiHeader.biBitCount = 24;
  bmiHeader.biCompression = BI_RGB;
  bmiHeader.biSizeImage = (m_nW*3+3)/4*4  * m_nH;
  bmiHeader.biXPelsPerMeter = 2000;
  bmiHeader.biYPelsPerMeter = 2000;
  bmiHeader.biClrUsed = 0;
  bmiHeader.biClrImportant = 0;
```

KhuGleWin.cpp (17)

```
unsigned char *Image2D24
    = new unsigned char [bmiHeader.biSizeImage];
int x, y, Offset;

for(y = 0 ; y < m_nH ; y++)
{
  Offset = (m_nW*3+3)/4*4 * (m_nH-y-1);
        // RGB, BGR, 4byte aligned (each row), bottom-up
  for(x = 0 ; x < m_nW ; x++) {
        int Offset2 = Offset+x*3;

        Image2D24[Offset2++] = 200;        // B
        Image2D24[Offset2++] = 200;        // G
        Image2D24[Offset2] = 255;          // R
  }
}
```


KhuGleWin.cpp (18)

```
  SetStretchBltMode(hCompDC, HALFTONE);

  StretchDIBits(hDC, m_nDesOffsetX, m_nDesOffsetY,
    m_nViewW, m_nViewH,0,0,
    bmiHeader.biWidth,bmiHeader.biHeight,
    Image2D24, (LPBITMAPINFO)&bmiHeader,
    DIB_RGB_COLORS,SRCCOPY);

  delete [] Image2D24;

  DeleteObject(hBitmap);

  DeleteDC(hCompDC);
  ReleaseDC(m_hWnd, hDC);

  EndPaint(m_hWnd, &ps);
}
```

KhuGleWin.cpp (19)

```
int APIENTRY WinMain(HINSTANCE hInstance,
  HINSTANCE hPrevInstance, LPSTR lpCmdLine,
  int nCmdShow)
{
  if(!CKhuGleWin::m_pWinApplication) return -1;

  WNDCLASSEX windowClass;
  MSG msg;
  DWORD    dwExStyle;
  DWORD    dwStyle;
  RECT     windowRect;

  int width = CKhuGleWin::m_pWinApplication->m_nW;
  int height = CKhuGleWin::m_pWinApplication->m_nH;
```


KhuGleWin.cpp (20)

```
  windowRect.left = (long)0;
  windowRect.right = (long)width;
  windowRect.top = (long)0;
  windowRect.bottom = (long)height;

  windowClass.cbSize = sizeof(WNDCLASSEX);
  windowClass.style = CS_HREDRAW | CS_VREDRAW;
  windowClass.lpfnWndProc
    = CKhuGleWin::m_pWinApplication->WndProc;
  windowClass.cbClsExtra = 0;
  windowClass.cbWndExtra = 0;
  windowClass.hInstance = hInstance;
  windowClass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
  windowClass.hCursor = LoadCursor(NULL, IDC_ARROW);
  windowClass.hbrBackground = NULL;
  windowClass.lpszMenuName = NULL;
  windowClass.lpszClassName = "WinClass";
  windowClass.hIconSm = LoadIcon(NULL, IDI_WINLOGO);
```

KhuGleWin.cpp (21)

```
  if(!RegisterClassEx(&windowClass))return 0;
  dwExStyle = WS_EX_APPWINDOW | WS_EX_WINDOWEDGE;
  dwStyle = WS_OVERLAPPEDWINDOW;
  AdjustWindowRectEx(&windowRect, dwStyle, FALSE, dwExStyle);
  CKhuGleWin::m_pWinApplication->m_hWnd
    = CreateWindowEx(NULL, "WinClass",
     "Ai and Data via Game",
     dwStyle | WS_CLIPCHILDREN | WS_CLIPSIBLINGS, 0, 0,
     windowRect.right - windowRect.left,
     windowRect.bottom - windowRect.top,
     NULL, NULL,hInstance,NULL);
  if(!CKhuGleWin::m_pWinApplication->m_hWnd) return 0;

  ShowWindow(CKhuGleWin::m_pWinApplication->m_hWnd, SW_SHOW);
  UpdateWindow(CKhuGleWin::m_pWinApplication->m_hWnd);
  QueryPerformanceFrequency(
         (LARGE_INTEGER*)&CKhuGleWin::m_pWinApplication->
         m_TimeCountFreq);
  QueryPerformanceCounter(
         (LARGE_INTEGER*)&CKhuGleWin::m_pWinApplication->
         m_TimeCountStart);
```

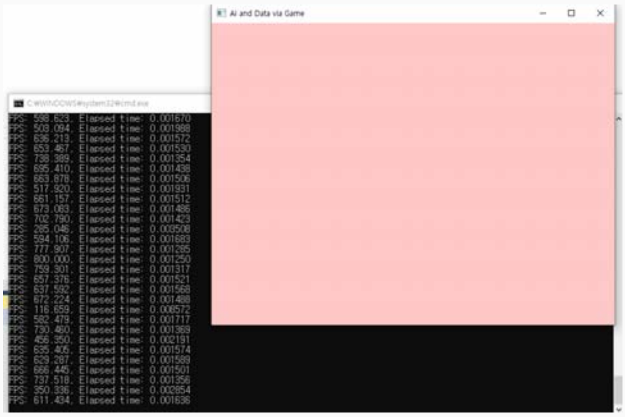

WinMain

KhuGleWin.cpp (22)

```
  while(1) {
   if(PeekMessage(&msg, 0, 0, 0, PM_REMOVE)) {
     if (msg.message == WM_QUIT) break;
     TranslateMessage(&msg);          DispatchMessage(&msg);
   }
   else {
    CKhuGleWin::m_pWinApplication->GetFps();
    CKhuGleWin::m_pWinApplication->Update();
    }
  }
  delete CKhuGleWin::m_pWinApplication;
  _CrtDumpMemoryLeaks();
  UnregisterClass("WinClass", windowClass.hInstance);
  return msg.wParam;
}
void CKhuGleWin::ToggleFpsView() {
  m_bViewFps = !m_bViewFps;
}
```

Main.cpp

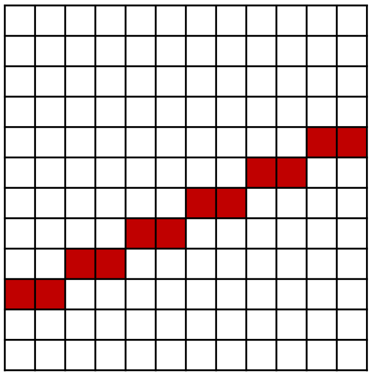
```
#include "KhuGleWin.h"
#include <iostream>

int main() {
  CKhuGleWin *pKhuGleSample = new CKhuGleWin(640, 480);
  KhuGleWinInit(pKhuGleSample);

  return 0;
}
```

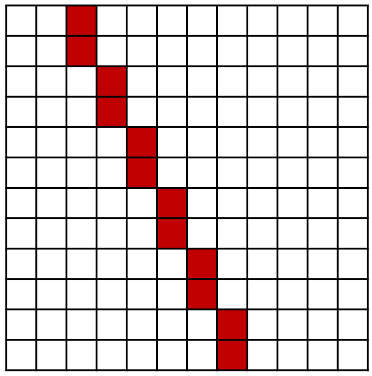


Exercise I (1)

```
class CKhuGleWin {
public:
  int m_nLButtonStatus;
  int m_LButtonStartX, m_LButtonStartY, m_LButtonEndX, m_LButtonEndY;
  CKhuGleWin(int nW, int nH) {
    m_nLButtonStatus = 0;
    …
  }
  void Update() {
    if(m_bMousePressed[0]) {
      if(m_nLButtonStatus == 0){
        m_LButtonStartX = m_MousePosX;        m_LButtonStartY = m_MousePosY;
      }
      m_LButtonEndX = m_MousePosX;      m_LButtonEndY = m_MousePosY;
      m_nLButtonStatus = 1;
    }
    else {
      if(m_nLButtonStatus == 1){
      // Save m_LButtonStartX,m_LButtonStartY,m_LButtonEndX,m_LButtonEndY
        m_nLButtonStatus = 0;
      }
    }
    …
  }
  void OnPaint() { /* Draw line */}
};
```

Exercise I (2)

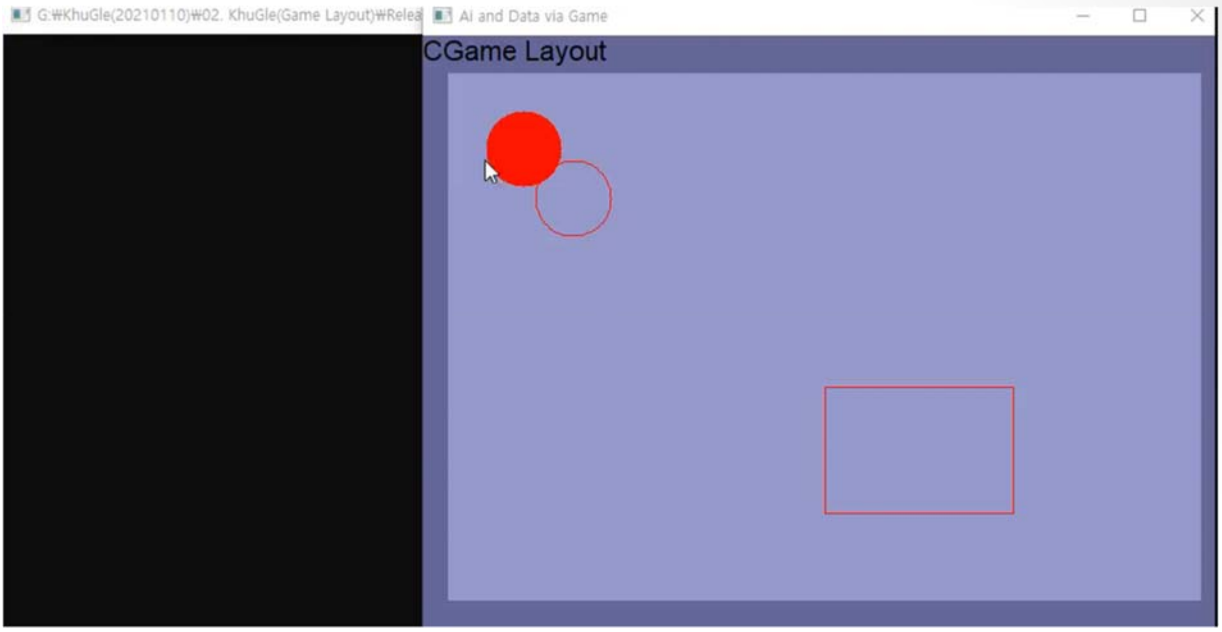




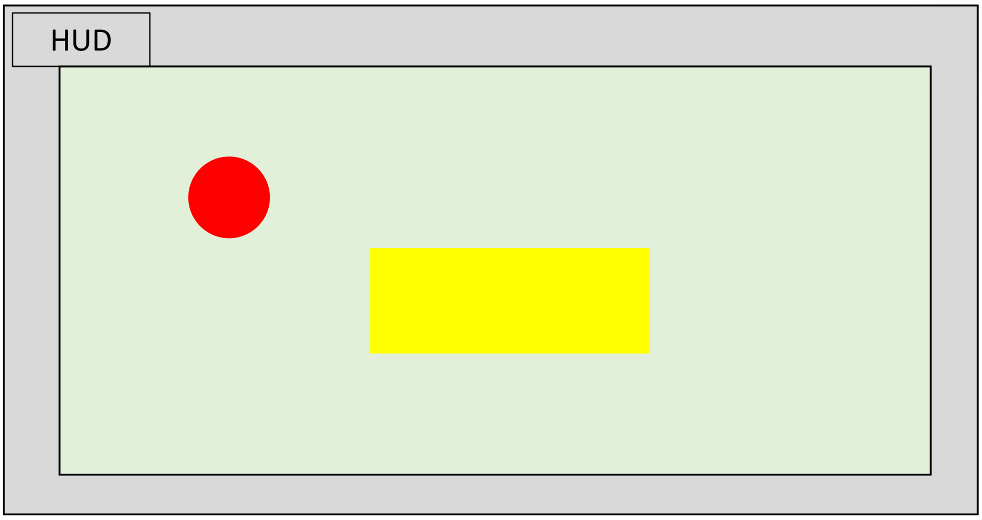
Advanced Courses

- Timer
  - WM_TIMER
  - SetTimer
    - UINT_PTR SetTimer(HWND hWnd, UINT_PTR nIDEvent, UINT uElapse, TIMERPROC lpTimerFunc);
- Thread
  - std::thread // <thread>
  - std::thread::join() // pauses until the thread finishes
  - Mutex
    - Synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads
    - std::mutex // <mutex>
    - std::mutex::lock() // locks the mutex, blocks if the mutex is not available
    - std::mutex::unlock() // unlocks the mutex

2. Game Layout



Scene/layer/sprite



Constant, color value
Point class

KhuGleBase.h (1)

```
#include <algorithm>
#include <cmath>

#define Pi3.14159

typedef unsigned long KgColor24;
#define KG_COLOR_24_RGB(R, G, B)((unsigned long)(((unsigned char)(R)|((unsigned
short)((unsigned char)(G))<<8))|(((unsigned long)(unsigned char)(B))<<16)))

#define KgGetRed(RGB)      ((RGB)& 0xff)
#define KgGetGreen(RGB)    ((((unsigned short)(RGB)) >> 8)& 0xff)
#define KgGetBlue(RGB)     (((RGB)>>16)& 0xff)

struct CKgPoint {
  int X, Y;
  CKgPoint() {}
  CKgPoint(int x, int y) : X(x), Y(y) {}
  CKgPoint operator+ (CKgPoint p1);
  CKgPoint &operator+= (CKgPoint p1);
};
```

Line class
Rectangle class

KhuGleBase.h (2)

```
struct CKgLine {
  CKgPoint Start, End;
  CKgLine() {}
  CKgLine(CKgPoint s, CKgPoint e) : Start(s), End(e) {}
  CKgLine(int sx, int sy, int ex, int ey) : Start(CKgPoint(sx, sy)),
         End(CKgPoint(ex, ey)) {}
};
struct CKgRect {
  int Left, Top, Right, Bottom;
  CKgRect::CKgRect() : Left(0), Top(0), Right(0), Bottom(0) {}
  CKgRect::CKgRect(int l, int t, int r, int b)
    : Left(l), Top(t), Right(r), Bottom(b) {}
  bool IsRect();
  int Width();
  int Height();
  CKgPoint Center();
  void Move(int x, int y);
  void Intersect(CKgRect rt);
  void Union(CKgRect rt);
  void Expanded(int e);
};
```


2D vector class

KhuGleBase.h (3)

```
class CKgVector2D {
public:
  double x, y;

  CKgVector2D() : x(0.), y(0.) {}
  CKgVector2D(double xx, double yy) : x(xx), y(yy) {}
  CKgVector2D(CKgPoint pt) : x(pt.X), y(pt.Y) {}

  static double abs(CKgVector2D v);
  void Normalize();
  double Dot(CKgVector2D v1);
  CKgVector2D operator+ (CKgVector2D v);
  CKgVector2D operator- (CKgVector2D v);
  CKgVector2D operator- ();
  CKgVector2D &operator+= (CKgVector2D v);
};
CKgVector2D operator*(double s, CKgVector2D v);
```

KhuGleBase.h (4)

```
unsigned char **cmatrix(int nH, int nW);
void free_cmatrix(unsigned char **Image, int nH, int nW);
double **dmatrix(int nH, int nW);
void free_dmatrix(double **Image, int nH, int nW);

void DrawLine(unsigned char **ImageGray, int nW, int nH,
  int x0, int y0, int x1, int y1, unsigned char Color);
```


KhuGleBase.cpp (1)

```
#include "KhuGleBase.h"
#include <cmath>
…
CKgPoint CKgPoint::operator+ (CKgPoint p1) {
  return CKgPoint(X+p1.X, Y+p1.Y);
}

CKgPoint &CKgPoint::operator+= (CKgPoint p1) {
  *this = *this + p1;
  return *this;
}
```

KhuGleBase.cpp (2)

```
bool CKgRect::IsRect() {
  if(Width() <= 0) return false;
  if(Height() <= 0) return false;

  return true;
}

int CKgRect::Width() {
  return Right-Left;
}

int CKgRect::Height() {
  return Bottom-Top;
}

CKgPoint CKgRect::Center() {
  return CKgPoint((Left+Right)/2, (Top+Bottom)/2);
}
```


KhuGleBase.cpp (3)

```
void CKgRect::Move(int x, int y) {
  Left += x;             Top += y;
  Right += x;            Bottom += y;
}
void CKgRect::Intersect(CKgRect rt) {
  Left = std::max(Left, rt.Left);
  Top = std::max(Top, rt.Top);
  Right = std::min(Right, rt.Right);
  Bottom = std::min(Bottom, rt.Bottom);
}
void CKgRect::Union(CKgRect rt) {
  Left = std::min(Left, rt.Left);
  Top = std::min(Top, rt.Top);
  Right = std::max(Right, rt.Right);
  Bottom = std::max(Bottom, rt.Bottom);
}
void CKgRect::Expanded(int e) {
  Left -= e;             Top -= e;
  Right += e;            Bottom += e;
}
```

Definition of 2D vector class methods

KhuGleBase.cpp (4)

```
double CKgVector2D::abs(CKgVector2D v) {
  return sqrt(v.x*v.x + v.y*v.y);
}
void CKgVector2D::Normalize() {
  double Magnitude = abs(*this);

  if(Magnitude == 0) return;
  x /= Magnitude;
  y /= Magnitude;
}
double CKgVector2D::Dot(CKgVector2D v) {
  return x*v.x + y*v.y;
}

CKgVector2D CKgVector2D::operator+ (CKgVector2D v) {
  return CKgVector2D(x+v.x, y+v.y);
}

CKgVector2D CKgVector2D::operator- (CKgVector2D v)  {
  return CKgVector2D(x-v.x, y-v.y);
}
```


Definition of 2D vector class methods

KhuGleBase.cpp (5)

```
CKgVector2D CKgVector2D::operator- ()
{
  return CKgVector2D(-x, -y);
}

CKgVector2D &CKgVector2D::operator+= (CKgVector2D v)
{
  *this = *this + v;
  return *this;
}

CKgVector2D operator*(double s, CKgVector2D v)
{
  return CKgVector2D(s*v.x, s*v.y);
}
```

Dynamic memory allocation

KhuGleBase.cpp (6)

```
unsigned char **cmatrix(int nH, int nW) {
  unsigned char **Temp = new unsigned char *[nH];
  for(int y = 0 ; y < nH ; y++)
    Temp[y] = new unsigned char[nW];
  return Temp;
}
void free_cmatrix(unsigned char **Image, int nH, int nW) {
  for(int y = 0 ; y < nH ; y++)
    delete [] Image[y];
  delete [] Image;
}
double **dmatrix(int nH, int nW) {
  double **Temp = new double *[nH];
  for(int y = 0 ; y < nH ; y++)
    Temp[y] = new double[nW];
  return Temp;
}
void free_dmatrix(double **Image, int nH, int nW) {
  for(int y = 0 ; y < nH ; y++)
    delete [] Image[y];
  delete [] Image;
}
```


Draw line

KhuGleBase.cpp (7)

```
void DrawLine(unsigned char **ImageGray, int nW, int nH, int x0, int y0,
  int x1, int y1, unsigned char Color) {
  int nDiffX = abs(x0-x1);
  int nDiffY = abs(y0-y1);

  int x, y;
  int nFrom, nTo;
  if(nDiffY == 0 && nDiffX == 0) {
    y = y0;      x = x0;
    if(!(x < 0 || x >= nW || y < 0 || y >= nH)) ImageGray[y][x] = Color;
  }
  else if(nDiffX == 0) {
    x = x0;
    nFrom = (y0 < y1 ? y0 : y1);
    if(nFrom < 0) nFrom = 0;
      nTo = (y0 < y1 ? y1 : y0);
    if(nTo >= nH) nTo = nH-1;
    for(y = nFrom ; y <= nTo ; y++) {
      if(x < 0 || x >= nW || y < 0 || y >= nH) continue;
      ImageGray[y][x] = Color;
    }
  }
```

Draw line

KhuGleBase.cpp (8)

```
  else if(nDiffY == 0) {
    y = y0;
    nFrom = (x0 < x1 ? x0 : x1);
    if(nFrom < 0) nFrom = 0;
    nTo = (x0 < x1 ? x1 : x0);
    if(nTo >= nW) nTo = nW-1;
    for(x = nFrom ; x <= nTo ; x++) {
      if(x < 0 || x >= nW || y < 0 || y >= nH) continue;
      ImageGray[y][x] = Color;
    }
  }
  else if(nDiffY > nDiffX) {
    nFrom = (y0 < y1 ? y0 : y1);
    if(nFrom < 0) nFrom = 0;
    nTo = (y0 < y1 ? y1 : y0);
    if(nTo >= nH) nTo = nH-1;
    for(y = nFrom ; y <= nTo ; y++) {
       x = (y-y0)*(x0-x1)/(y0-y1) + x0;
    if(x < 0 || x >= nW || y < 0 || y >= nH) continue;
    ImageGray[y][x] = Color;
  }
}
```


Draw line

KhuGleBase.cpp (9)

```
  else {
    nFrom = (x0 < x1 ? x0 : x1);
    if(nFrom < 0) nFrom = 0;
    nTo = (x0 < x1 ? x1 : x0);
    if(nTo >= nW) nTo = nW-1;

    for(x = nFrom ; x <= nTo ; x++) {
      y = (x-x0)*(y0-y1)/(x0-x1) + y0;
      if(x < 0 || x >= nW || y < 0 || y >= nH) continue;
      ImageGray[y][x] = Color;
    }
  }
}
```

KhuGleComponent.h

```
#include <vector>
class CKhuGleComponent {
public:
  std::vector<CKhuGleComponent*> m_Children;
  CKhuGleComponent *m_Parent;

  CKhuGleComponent();
  virtual ~CKhuGleComponent();

  void AddChild(CKhuGleComponent *pChild);

  virtual void Render() = 0;
};
```


KhuGleComponent.cpp

```
#include "KhuGleComponent.h"
…
CKhuGleComponent::CKhuGleComponent() {
  m_Parent = nullptr;
}
CKhuGleComponent::~CKhuGleComponent() {
  for(auto &Child : m_Children)
    delete Child;
}

void CKhuGleComponent::AddChild(CKhuGleComponent *pChild) {
  pChild->m_Parent = this;

  m_Children.push_back(pChild);
}
```

KhuGleScene.h

```
#include "KhuGleBase.h"
#include "KhuGleComponent.h"

class CKhuGleScene : public CKhuGleComponent {
public:
  bool m_bInit;
  int m_nW, m_nH;

  unsigned char **m_ImageR, **m_ImageG, **m_ImageB;
  KgColor24 m_bgColor;
  CKhuGleScene(int nW, int nH, KgColor24 bgColor);
  ~CKhuGleScene();
  void SetBackgroundImage(int nW, int nH, KgColor24 bgColor);
  void ResetBackgroundImage();
  void SetBgColor(KgColor24 bgColor);

  virtual void Render();
};
```


KhuGleScene.cpp (1)

```
#include "KhuGleLayer.h"
#include "KhuGleScene.h"
…
CKhuGleScene::CKhuGleScene(int nW, int nH, KgColor24 bgColor) {
  m_bInit = false;
  SetBackgroundImage(nW, nH, bgColor);
}
CKhuGleScene::~CKhuGleScene() {
  ResetBackgroundImage();
}
```

KhuGleScene.cpp (2)

```
void CKhuGleScene::SetBackgroundImage(int nW, int nH,
  KgColor24 bgColor) {
  if(m_bInit) ResetBackgroundImage();
  m_nW = nW;
  m_nH = nH;
  m_bgColor = bgColor;
  m_ImageR = cmatrix(m_nH, m_nW);
  m_ImageG = cmatrix(m_nH, m_nW);
  m_ImageB = cmatrix(m_nH, m_nW);

  for(int y = 0 ; y < m_nH ; y++)
    for(int x = 0 ; x < m_nW ; x++) {
      m_ImageR[y][x] = KgGetRed(bgColor);
      m_ImageG[y][x] = KgGetGreen(bgColor);
      m_ImageB[y][x] = KgGetBlue(bgColor);
    }
  m_bInit = true;
}
```


KhuGleScene.cpp (3)

```
void CKhuGleScene::ResetBackgroundImage() {
  if(m_bInit) {
    free_cmatrix(m_ImageR, m_nH, m_nW);
    free_cmatrix(m_ImageG, m_nH, m_nW);
    free_cmatrix(m_ImageB, m_nH, m_nW);

    m_bInit = false;
  }
}

void CKhuGleScene::SetBgColor(KgColor24 bgColor) {
  m_bgColor = bgColor;
}
```

KhuGleScene.cpp (4)

```
void CKhuGleScene::Render() {
  for(int y = 0 ; y < m_nH ; y++) {
    memset(m_ImageR[y], KgGetRed(m_bgColor), m_nW);
    memset(m_ImageG[y], KgGetGreen(m_bgColor), m_nW);
    memset(m_ImageB[y], KgGetBlue(m_bgColor), m_nW);
  }
  for(auto &Child : m_Children) {
    CKhuGleLayer *Layer = (CKhuGleLayer *)Child;
    Layer->Render();
    for(int y = 0 ; y < Layer->m_nH ; ++y) {
      if(y+Layer->m_ptPos.Y >= m_nH) break;
      int nLen = std::min(Layer->m_nW, m_nW-Layer->m_ptPos.X);
      if(nLen <= 0) continue;
      memcpy(m_ImageR[y+Layer->m_ptPos.Y] + Layer->m_ptPos.X,
        Layer->m_ImageR[y], nLen);
      memcpy(m_ImageG[y+Layer->m_ptPos.Y] + Layer->m_ptPos.X,
        Layer->m_ImageG[y], nLen);
      memcpy(m_ImageB[y+Layer->m_ptPos.Y] + Layer->m_ptPos.X,
        Layer->m_ImageB[y], nLen);
    }
  }
}
```


KhuGleLayer.h

```
#include "KhuGleBase.h"
#include "KhuGleComponent.h"
class CKhuGleLayer : public CKhuGleComponent {
public:
  bool m_bInit;
  int m_nW, m_nH;
  CKgPoint m_ptPos;

  unsigned char **m_ImageR, **m_ImageG, **m_ImageB;
  unsigned char **m_ImageBgR, **m_ImageBgG, **m_ImageBgB;
  KgColor24 m_bgColor;

  CKhuGleLayer(int nW, int nH, KgColor24 bgColor,
    CKgPoint ptPos = CKgPoint(0, 0));
  ~CKhuGleLayer();

  void SetBackgroundImage(int nW, int nH, KgColor24 bgColor);
  void ResetBackgroundImage();
  void SetBgColor(KgColor24 bgColor);

  virtual void Render();
};
```

Layer class

KhuGleLayer.cpp (1)

```
#include "KhuGleLayer.h"
…
CKhuGleLayer::CKhuGleLayer(int nW, int nH, KgColor24 bgColor, CKgPoint ptPos) {
  m_bInit = false;          m_ptPos = ptPos;
  SetBackgroundImage(nW, nH, bgColor);
}
CKhuGleLayer::~CKhuGleLayer() {
  ResetBackgroundImage();
}
void CKhuGleLayer::SetBackgroundImage(int nW, int nH, KgColor24 bgColor) {
  if(m_bInit) ResetBackgroundImage();
  m_nW = nW;       m_nH = nH;
  m_bgColor = bgColor;
  m_ImageR = cmatrix(m_nH, m_nW);     m_ImageG = cmatrix(m_nH, m_nW);
  m_ImageB = cmatrix(m_nH, m_nW);     m_ImageBgR = cmatrix(m_nH, m_nW);
  m_ImageBgG = cmatrix(m_nH, m_nW);   m_ImageBgB = cmatrix(m_nH, m_nW);

  for(int y = 0 ; y < m_nH ; y++)
    for(int x = 0 ; x < m_nW ; x++) {
      m_ImageBgR[y][x] = KgGetRed(bgColor);    m_ImageBgG[y][x] = KgGetGreen(bgColor);
      m_ImageBgB[y][x] = KgGetBlue(bgColor);
    }
  m_bInit = true;
}
```


Layer class

KhuGleLayer.cpp (2)

```
void CKhuGleLayer::ResetBackgroundImage() {
  if(m_bInit) {
    free_cmatrix(m_ImageR, m_nH, m_nW);     free_cmatrix(m_ImageG, m_nH, m_nW);
    free_cmatrix(m_ImageB, m_nH, m_nW);    free_cmatrix(m_ImageBgR, m_nH, m_nW);
    free_cmatrix(m_ImageBgG, m_nH, m_nW); free_cmatrix(m_ImageBgB, m_nH, m_nW);
    m_bInit = false;
  }
}
void CKhuGleLayer::SetBgColor(KgColor24 bgColor) {
  m_bgColor = bgColor;
}
void CKhuGleLayer::Render() {
  for(int y = 0 ; y < m_nH ; y++) {
    memcpy(m_ImageR[y], m_ImageBgR[y], m_nW);
    memcpy(m_ImageG[y], m_ImageBgG[y], m_nW);
    memcpy(m_ImageB[y], m_ImageBgB[y], m_nW);
  }
  for(auto &Child : m_Children)
    Child->Render();
}
```

KhuGleSprite.h (1)

```
#include "KhuGleBase.h"
#include "KhuGleLayer.h"
#include "KhuGleScene.h"
#include "KhuGleComponent.h"
#define GP_STYPE_LINE              0
#define GP_STYPE_RECT              1
#define GP_STYPE_ELLIPSE           2

#define GP_CTYPE_STATIC            0
#define GP_CTYPE_DYNAMIC           1
#define GP_CTYPE_KINEMATIC         2
class CKhuGleSprite : public CKhuGleComponent {
public:
  int m_nType;            // line, rect, ellipse
  int m_nCollisionType;   // static, dynamic, kinematic
  CKgLine m_lnLine;         CKgRect m_rtBoundBox;
  KgColor24 m_fgColor;
  bool m_bFill;
  int m_nWidth;
  int m_nSlice;
```


KhuGleSprite.h (2)

```
  CKgVector2D m_Center, m_Velocity, m_Acceleration;
  double m_Radius;
  double m_Mass;

  CKhuGleSprite() {}
  CKhuGleSprite(int nType, int nCollisionType, CKgLine lnLine,
    KgColor24 fgColor, bool bFill, int nSliceOrWidth = 100);
  ~CKhuGleSprite();

  static void DrawLine(unsigned char **R, unsigned char **G, unsigned char **B,
    int nW, int nH, int x0, int y0, int x1, int y1, KgColor24 Color24);

  virtual void Render();
  void MoveBy(double OffsetX, double OffsetY);
  void MoveTo(double X, double Y);
  void Move();
};
```

KhuGleSprite.cpp (1)

```
#include "KhuGleSprite.h"
…
CKhuGleSprite::CKhuGleSprite(int nType, CKgLine lnLine, KgColor24 fgColor, bool bFill,
  int nSliceOrWidth) {
  m_nType = nType;
  m_nCollisionType = nCollisionType;
  m_fgColor = fgColor;                          m_bFill = bFill;
  m_nSlice = nSliceOrWidth;                     m_nWidth = nSliceOrWidth;

  if(m_nType == GP_STYPE_LINE) m_lnLine = lnLine;
  else
    m_rtBoundBox = CKgRect(lnLine.Start.X, lnLine.Start.Y,
         lnLine.End.X, lnLine.End.Y);

  m_Center.x = (lnLine.Start.X + lnLine.End.X)/2.;
  m_Center.y = (lnLine.Start.Y + lnLine.End.Y)/2.;
  m_Velocity = CKgVector2D(0., 0.);
  m_Radius = std::max(fabs((lnLine.Start.X - lnLine.End.X)/2.),
         fabs((lnLine.Start.Y - lnLine.End.Y)/2.));

  m_Mass = m_Radius*m_Radius;
}
CKhuGleSprite::~CKhuGleSprite() {
}
```


KhuGleSprite.cpp (2)

```
void CKhuGleSprite::DrawLine(unsigned char **R,
  unsigned char **G, unsigned char **B, int nW, int nH, int x0, int y0,
  int x1, int y1, KgColor24 Color24) {
  ::DrawLine(R, nW, nH, x0, y0, x1, y1, KgGetRed(Color24));
  ::DrawLine(G, nW, nH, x0, y0, x1, y1, KgGetGreen(Color24));
  ::DrawLine(B, nW, nH, x0, y0, x1, y1, KgGetBlue(Color24));
}
```

KhuGleSprite.cpp (3)

```
void CKhuGleSprite::Render() {
  if(!m_Parent) return;

  CKhuGleLayer *Parent = (CKhuGleLayer *)m_Parent;
  if(m_nType == GP_STYPE_LINE) {
    CKgVector2D PosVec = CKgVector2D(m_lnLine.Start) - CKgVector2D(m_lnLine.End);
    CKgVector2D Normal1 = CKgVector2D(PosVec.y, -PosVec.x);
    CKgVector2D Normal2 = CKgVector2D(-PosVec.y, PosVec.x);

    Normal1.Normalize();                    Normal2.Normalize();
    Normal1 = (m_nWidth/2.)*Normal1;        Normal2 = (m_nWidth/2.)*Normal2;

    DrawLine(Parent->m_ImageR, Parent->m_ImageG, Parent->m_ImageB,
      Parent->m_nW, Parent->m_nH,
      (int)(m_lnLine.Start.X+Normal1.x), (int)(m_lnLine.Start.Y+Normal1.y),
      (int)(m_lnLine.End.X+Normal1.x), (int)(m_lnLine.End.Y+Normal1.y), m_fgColor);
    DrawLine(Parent->m_ImageR, Parent->m_ImageG, Parent->m_ImageB,
      Parent->m_nW, Parent->m_nH,
      (int)(m_lnLine.Start.X+Normal2.x), (int)(m_lnLine.Start.Y+Normal2.y),
      (int)(m_lnLine.End.X+Normal2.x), (int)(m_lnLine.End.Y+Normal2.y), m_fgColor);
  }
```


KhuGleSprite.cpp (4)

```
  else if(!m_bFill) {
    if(m_nType == GP_STYPE_RECT) {
      DrawLine(Parent->m_ImageR, Parent->m_ImageG, Parent->m_ImageB,
        Parent->m_nW, Parent->m_nH, m_rtBoundBox.Left,
        m_rtBoundBox.Top, m_rtBoundBox.Right, m_rtBoundBox.Top,
        m_fgColor);
      DrawLine(Parent->m_ImageR, Parent->m_ImageG, Parent->m_ImageB,
        Parent->m_nW, Parent->m_nH, m_rtBoundBox.Right,
        m_rtBoundBox.Top, m_rtBoundBox.Right, m_rtBoundBox.Bottom,  m_fgColor);
      DrawLine(Parent->m_ImageR, Parent->m_ImageG, Parent->m_ImageB,
        Parent->m_nW, Parent->m_nH, m_rtBoundBox.Right,
        m_rtBoundBox.Bottom, m_rtBoundBox.Left, m_rtBoundBox.Bottom,
        m_fgColor);
      DrawLine(Parent->m_ImageR, Parent->m_ImageG, Parent->m_ImageB,
        Parent->m_nW, Parent->m_nH, m_rtBoundBox.Left,
        m_rtBoundBox.Bottom, m_rtBoundBox.Left, m_rtBoundBox.Top,   m_fgColor);
    }
```

KhuGleSprite.cpp (5)

```
else {
    double RX = (m_rtBoundBox.Right - m_rtBoundBox.Left) / 2.;
    double RY = (m_rtBoundBox.Bottom - m_rtBoundBox.Top) / 2.;
    double CX = (m_rtBoundBox.Right + m_rtBoundBox.Left) / 2.;
    double CY = (m_rtBoundBox.Bottom + m_rtBoundBox.Top) / 2.;
    for(int i = 0 ; i < m_nSlice ; i++) {
      double theta1 = 2.*Pi/m_nSlice*i;
      double theta2 = 2.*Pi/m_nSlice*(i+1);

      DrawLine(Parent->m_ImageR, Parent->m_ImageG,
        Parent->m_ImageB,
        Parent->m_nW, Parent->m_nH,
        (int)(CX + cos(theta1)*RX), (int)(CY + sin(theta1)*RY),
        (int)(CX + cos(theta2)*RX), (int)(CY + sin(theta2)*RY),
        m_fgColor);
    }
  }
}
```


KhuGleSprite.cpp (6)

```
else {
  if(m_nType == GP_STYPE_RECT) {
    CKgRect interRect = CKgRect(0, 0, Parent->m_nW-1, Parent->m_nH-1);
    interRect.Intersect(m_rtBoundBox);

    if(interRect.IsRect()) {
      for(int y = interRect.Top ; y <= interRect.Bottom ; y++)
        for(int x = interRect.Left ; x <= interRect.Right ;x++)
        {
          Parent->m_ImageR[y][x] = KgGetRed(m_fgColor);
          Parent->m_ImageG[y][x] = KgGetGreen(m_fgColor);
          Parent->m_ImageB[y][x] = KgGetBlue(m_fgColor);
      }
    }
  }
```

Sprite class

KhuGleSprite.cpp (7)

```
    else {
      double RX = (m_rtBoundBox.Right - m_rtBoundBox.Left) / 2.;
      double RY = (m_rtBoundBox.Bottom - m_rtBoundBox.Top) / 2.;
      double CX = (m_rtBoundBox.Right + m_rtBoundBox.Left) / 2.;
      double CY = (m_rtBoundBox.Bottom + m_rtBoundBox.Top) / 2.;

      CKgRect interRect = CKgRect(0, 0, Parent->m_nW-1, Parent->m_nH-1);
      interRect.Intersect(m_rtBoundBox);

      if(interRect.IsRect()){
        for(int y = interRect.Top ; y <= interRect.Bottom ; y++)
          for(int x = interRect.Left ; x <= interRect.Right ; x++) {
            if((x-CX)*(x-CX)/(RX*RX) + (y-CY)*(y-CY)/(RY*RY) <= 1) {
              Parent->m_ImageR[y][x] = KgGetRed(m_fgColor);
              Parent->m_ImageG[y][x] = KgGetGreen(m_fgColor);
              Parent->m_ImageB[y][x] = KgGetBlue(m_fgColor);
            }
          }
      }
    }
  }
}
```


CKhuGleWin class
- Add: gravity, air resistance, scene object, draw text

KhuGleWin.h/cpp (1)

```
class CKhuGleWin { // CKhuGleWin.h
…
  CKgVector2D m_Gravity;
  CKgVector2D m_AirResistance;
…
  void DrawSceneTextPos(char *Text, CKgPoint ptPos);
  CKhuGleScene *m_pScene;
};

// CKhuGleWin.cpp
CKhuGleWin::CKhuGleWin(int nW, int nH) {
  m_pScene = nullptr;
  m_Gravity = CKgVector2D(0., 0.);
  m_AirResistance = CKgVector2D(0., 0.);
  …
}
CKhuGleWin::~CKhuGleWin() {
  if(m_pScene) delete m_pScene;
}
```

CKhuGleWin class
- OnPaint: draw scene

KhuGleWin.h/cpp (2)

```
void CKhuGleWin::OnPaint() {
  …
  int x, y, Offset;
  for(y = 0 ; y < m_nH ; y++) {
    Offset = (m_nW*3+3)/4*4 * (m_nH-y-1);
    for(x = 0 ; x < m_nW ; x++) {
      int Offset2 = Offset+x*3;
      if(x < 0 || x >= m_pScene->m_nW || y < 0 || y >= m_pScene->m_nH) {
        Image2D24[Offset2++] = 0;
        Image2D24[Offset2++] = 0;
        Image2D24[Offset2] = 0;
      }
      else {
        Image2D24[Offset2++] = m_pScene->m_ImageB[y][x];
        Image2D24[Offset2++] = m_pScene->m_ImageG[y][x];
        Image2D24[Offset2] = m_pScene->m_ImageR[y][x];
      }
    }
  }
  SetStretchBltMode(hCompDC, HALFTONE);
  …
}
```


CKhuGleWin class
- Draw text

KhuGleWin.h/cpp (3)

```
void CKhuGleWin::DrawSceneTextPos(char *Text, CKgPoint ptPos) {
  int nTextHeight = 25;

  HDC hDC;                HDC hCompDC;
  HBITMAP hBitmap;
  hDC = GetDC(NULL);
  hCompDC = CreateCompatibleDC(hDC);
  hBitmap = CreateCompatibleBitmap(hDC, nTextHeight*strlen(Text),
    nTextHeight+10);
  SelectObject(hCompDC, hBitmap);

  HFONT hFont;
  hFont = CreateFontA(nTextHeight, 0, 0, 0,
    FW_NORMAL, 0, 0, 0, ANSI_CHARSET, 0, 0, 0, FF_MODERN, "Arial");
  SelectObject(hCompDC, hFont);
  SetTextColor(hCompDC, RGB(0, 0, 0));
  SetBkMode(hCompDC, TRANSPARENT);

  RECT Rt;
  Rt.left = 0;    Rt.top = 0;
  DrawText(hCompDC, Text, strlen(Text), &Rt, DT_CALCRECT | DT_LEFT);
```

KhuGleWin.h/cpp (4)

```
BITMAPINFO bi;
int nW = Rt.right-Rt.left+1;
int nH = Rt.bottom-Rt.top+1;

bi.bmiHeader.biSize = sizeof(bi.bmiHeader);
bi.bmiHeader.biWidth = nW;          bi.bmiHeader.biHeight = nH;
bi.bmiHeader.biPlanes = 1;          bi.bmiHeader.biBitCount = 24;
bi.bmiHeader.biCompression = BI_RGB;
bi.bmiHeader.biSizeImage = (nW*3+3)/4*4  * nH;
bi.bmiHeader.biClrUsed = 0;         bi.bmiHeader.biClrImportant = 0;

unsigned char *Image = new unsigned char[bi.bmiHeader.biSizeImage];
for(int y = 0 ; y < nH ; y++)
  for(int x = 0 ; x < nW ; x++) {
    if(x+ptPos.X >= m_pScene->m_nW || y+ptPos.Y >= m_pScene->m_nH)
       break;
    int pos = (nW*3+3)/4*4*(nH-y-1) + x*3;
    Image[pos+2] = m_pScene->m_ImageR[y+ptPos.Y][x+ptPos.X];
    Image[pos+1] = m_pScene->m_ImageG[y+ptPos.Y][x+ptPos.X];
    Image[pos] = m_pScene->m_ImageB[y+ptPos.Y][x+ptPos.X];
  }
```


KhuGleWin.h/cpp (5)

```
  SetStretchBltMode(hCompDC, HALFTONE);
  StretchDIBits(hCompDC, 0, 0, bi.bmiHeader.biWidth,
    bi.bmiHeader.biHeight,0, 0, bi.bmiHeader.biWidth,
    bi.bmiHeader.biHeight,Image, (LPBITMAPINFO)&bi.bmiHeader,
    DIB_RGB_COLORS, SRCCOPY);
  DrawText(hCompDC, Text, strlen(Text), &Rt, DT_LEFT);

  GetDIBits(hCompDC, hBitmap, 0, nH, Image, &bi, DIB_RGB_COLORS);
  for(int y = 0 ; y < nH ; y++)
    for(int x = 0 ; x < nW ; x++) {
      if(x+ptPos.X >= m_pScene->m_nW || y+ptPos.Y >= m_pScene->m_nH)
         break;
      int pos = (nW*3+3)/4*4*(nH-y-1) + x*3;
      m_pScene->m_ImageR[y+ptPos.Y][x+ptPos.X] = Image[pos+2];
      m_pScene->m_ImageG[y+ptPos.Y][x+ptPos.X] = Image[pos+1];
      m_pScene->m_ImageB[y+ptPos.Y][x+ptPos.X] = Image[pos];
    }
  delete [] Image;
  DeleteObject(hBitmap);    DeleteObject(hFont);   DeleteDC(hCompDC);
  ReleaseDC(NULL, hDC);
}
```

Derive application class - CGameLayout

Main.cpp (1)

```
#include "KhuGleWin.h"
#include <iostream>
class CGameLayout : public CKhuGleWin {
public:
  CKhuGleLayer *m_pGameLayer;

  CKhuGleSprite *m_pCircle1, *m_pCircle2, *m_pRect;
  CGameLayout(int nW, int nH);
  void Update();

  CKgPoint m_LButtonStart, m_LButtonEnd;   // mouse position
  int m_nLButtonStatus;
};
```


Derive application class – CGameLayout
Constructor

Main.cpp (2)

```
CGameLayout::CGameLayout(int nW, int nH) : CKhuGleWin(nW, nH)  {
  m_nLButtonStatus = 0;
  m_Gravity = CKgVector2D(0., 98.);
  m_AirResistance = CKgVector2D(0.1, 0.1);

  m_pScene = new CKhuGleScene(640, 480, KG_COLOR_24_RGB(100, 100, 150));

  m_pGameLayer = new CKhuGleLayer(600, 420,
        KG_COLOR_24_RGB(150, 150, 200), CKgPoint(20, 30));

  m_pScene->AddChild(m_pGameLayer);
```

Derive application class – CGameLayout
Constructor

Main.cpp (3)

```
  m_pCircle1 = new CKhuGleSprite(GP_STYPE_ELLIPSE,
    GP_CTYPE_KINEMATIC,
    CKgLine(CKgPoint(30, 30), CKgPoint(90, 90)),
    KG_COLOR_24_RGB(255, 0, 0), true, 100);
  m_pCircle2 = new CKhuGleSprite(GP_STYPE_ELLIPSE,
    GP_CTYPE_KINEMATIC,
    CKgLine(CKgPoint(70, 70), CKgPoint(130, 130)),
    KG_COLOR_24_RGB(255, 0, 0), false, 100);
  m_pRect = new CKhuGleSprite(GP_STYPE_RECT,
    GP_CTYPE_KINEMATIC,
    CKgLine(CKgPoint(300, 350), CKgPoint(450, 250)),
    KG_COLOR_24_RGB(255, 0, 0), false, 10);

  m_pGameLayer->AddChild(m_pCircle1);
  m_pGameLayer->AddChild(m_pCircle2);
  m_pGameLayer->AddChild(m_pRect);
}
```


Derive application class – CGameLayout
Update

Main.cpp (4)

```
void CGameLayout::Update() {
  if(m_bMousePressed[0]) // mouse event
  {
    if(m_nLButtonStatus == 0){
      m_LButtonStart = CKgPoint(m_MousePosX, m_MousePosY);
    }
    m_LButtonEnd = CKgPoint(m_MousePosX, m_MousePosY);
    m_nLButtonStatus = 1;
  }
  else {
    if(m_nLButtonStatus == 1) {
      std::cout << m_LButtonStart.X << "," << m_LButtonStart.Y << std::endl;
      std::cout << m_LButtonEnd.X << "," << m_LButtonEnd.Y << std::endl;

      m_nLButtonStatus = 0;
    }
  }
```

Derive application class – CGameLayout
Update

Main.cpp (5)

```
  // keyboard
  if(m_bKeyPressed[VK_LEFT]) m_pCircle1->MoveBy(-1, 0);
  if(m_bKeyPressed[VK_UP]) m_pCircle1->MoveBy(0, -1);
  if(m_bKeyPressed[VK_RIGHT]) m_pCircle1->MoveBy(1, 0);
  if(m_bKeyPressed[VK_DOWN]) m_pCircle1->MoveBy(0, 1);

  m_pScene->Render();
  DrawSceneTextPos("CGame Layout", CKgPoint(0, 0));

  CKhuGleWin::Update();
}

int main() {
  CGameLayout *pGameLayout = new CGameLayout(640, 480);
  KhuGleWinInit(pGameLayout);

  return 0;
}
```


Exercise II

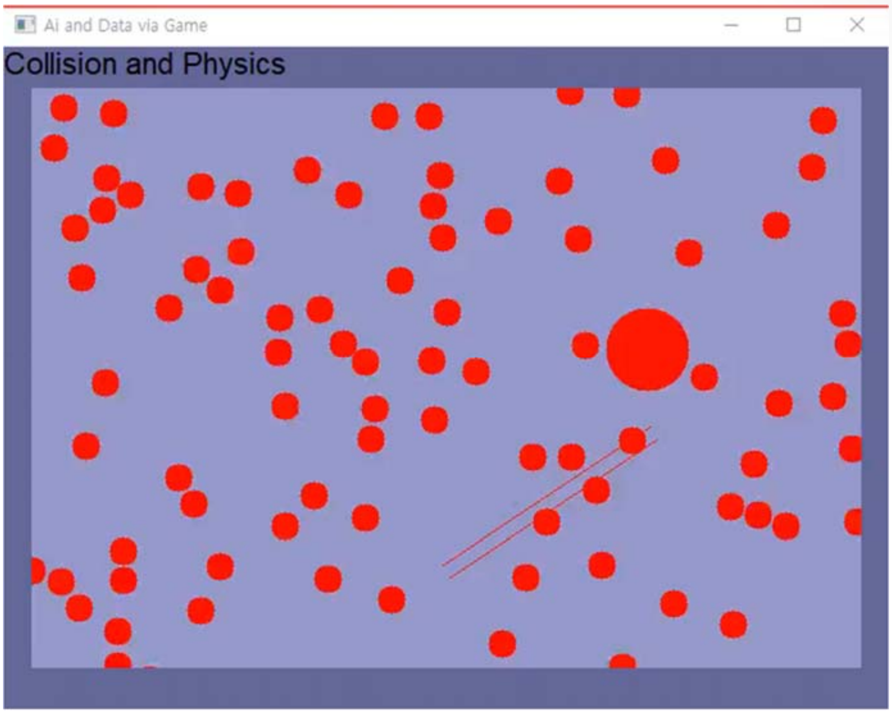
- Object movement by velocities and accelerations (gravity)

Advanced Courses

- GUI programming
  - Windows
    - Windows API
    - MFC (Microsoft foundation class)
    - UWP (universal Windows platform)

  - Cross platform
    - Qt
    - wxWidgets
    - Ultimate++

3. Collision & Physics



Main.cpp I

```
#include "KhuGleWin.h"
#include <iostream>

class CCollision : public CKhuGleWin {
public:
  CKhuGleLayer *m_pGameLayer;

  CKhuGleSprite *m_pCircle1;
  CKhuGleSprite *m_pCircle2;
  CKhuGleSprite *m_pLine;
  CKhuGleSprite *m_pNewCircle[100];

  CCollision(int nW, int nH);
  void Update();

  CKgPoint m_LButtonStart, m_LButtonEnd;
  int m_nLButtonStatus;
};
```

```
CCollision::CCollision(int nW, int nH) : CKhuGleWin(nW, nH) {
  m_nLButtonStatus = 0;

  m_Gravity = CKgVector2D(0., 98.);
  m_AirResistance = CKgVector2D(0.1, 0.1);

  m_pScene = new CKhuGleScene(640, 480, KG_COLOR_24_RGB(100, 100, 150));
  m_pGameLayer = new CKhuGleLayer(600, 420, KG_COLOR_24_RGB(150, 150, 200),
    CKgPoint(20, 30));
  m_pScene->AddChild(m_pGameLayer);

  m_pCircle1 = new CKhuGleSprite(GP_STYPE_ELLIPSE, GP_CTYPE_DYNAMIC,
    CKgLine(CKgPoint(30, 30), CKgPoint(90, 90)),
    KG_COLOR_24_RGB(255, 0, 0), true, 100);
  m_pCircle2 = new CKhuGleSprite(GP_STYPE_ELLIPSE, GP_CTYPE_DYNAMIC,
    CKgLine(CKgPoint(70, 70), CKgPoint(130, 130)),
    KG_COLOR_24_RGB(255, 0, 0), false, 100);
  m_pLine = new CKhuGleSprite(GP_STYPE_LINE, GP_CTYPE_STATIC,
    CKgLine(CKgPoint(300, 350), CKgPoint(450, 250)),
    KG_COLOR_24_RGB(255, 0, 0), false, 10);
```



```
  m_pGameLayer->AddChild(m_pCircle1);
  m_pGameLayer->AddChild(m_pCircle2);
  m_pGameLayer->AddChild(m_pLine);

  m_pCircle1->m_Velocity = CKgVector2D(900, 50);
  m_pCircle2->m_Velocity = CKgVector2D(-100, -300);
  for(int i = 0 ; i < 100 ; i++) {
    m_pNewCircle[i] = new CKhuGleSprite(GP_STYPE_ELLIPSE,
      GP_CTYPE_DYNAMIC,
      CKgLine(CKgPoint(30, 30), CKgPoint(50, 50)),
      KG_COLOR_24_RGB(255, 0, 0), true, 100);

    m_pGameLayer->AddChild(m_pNewCircle[i]);
  }
}
```

```
void CCollision::Update() {
  if(m_bMousePressed[0]) {
    if(m_nLButtonStatus == 0){
      m_LButtonStart = CKgPoint(m_MousePosX, m_MousePosY);
    }
    m_LButtonEnd = CKgPoint(m_MousePosX, m_MousePosY);
    m_nLButtonStatus = 1;
  }
  else {
    if(m_nLButtonStatus == 1) {
      std::cout << m_LButtonStart.X << "," << m_LButtonStart.Y << std::endl;
      std::cout << m_LButtonEnd.X << "," << m_LButtonEnd.Y << std::endl;

      m_nLButtonStatus = 0;
    }
  }
```



```
  if(m_bKeyPressed['S']) {
    m_pCircle1->m_Velocity = CKgVector2D(0, 0);
  }

  if(m_bKeyPressed[VK_LEFT]) m_pCircle1->m_Velocity = CKgVector2D(-500, 0);
  if(m_bKeyPressed[VK_UP]) m_pCircle1->m_Velocity = CKgVector2D(0, -500);
  if(m_bKeyPressed[VK_RIGHT]) m_pCircle1->m_Velocity = CKgVector2D(500, 0);
  if(m_bKeyPressed[VK_DOWN]) m_pCircle1->m_Velocity = CKgVector2D(0, 500);
```

```
for(auto &Layer : m_pScene->m_Children) {
  for(auto &Sprite : Layer->m_Children) {
    CKhuGleSprite *Ball = (CKhuGleSprite *)Sprite;
    Ball->m_bCollided = false;
    if(Ball->m_nType == GP_STYPE_RECT) continue;
    if(Ball->m_nType != GP_STYPE_ELLIPSE) continue;
    if(Ball->m_nCollisionType != GP_CTYPE_DYNAMIC) continue;

    Ball->m_Acceleration.x
      = m_Gravity.x - Ball->m_Velocity.x * m_AirResistance.x;
    Ball->m_Acceleration.y
      = m_Gravity.y - Ball->m_Velocity.y * m_AirResistance.y;

    Ball->m_Velocity.x += Ball->m_Acceleration.x * m_ElapsedTime;
    Ball->m_Velocity.y += Ball->m_Acceleration.y * m_ElapsedTime;

    Ball->MoveBy(Ball->m_Velocity.x*m_ElapsedTime,
      Ball->m_Velocity.y*m_ElapsedTime);
```



```
    if(Ball->m_Center.x < 0)
      Ball->MoveTo(m_nW+Ball->m_Center.x, Ball->m_Center.y);
    if(Ball->m_Center.x > m_nW)
      Ball->MoveTo(Ball->m_Center.x-m_nW, Ball->m_Center.y);
    if(Ball->m_Center.y < 0)
      Ball->MoveTo(Ball->m_Center.x, m_nH+Ball->m_Center.y);
    if(Ball->m_Center.y > m_nH)
      Ball->MoveTo(Ball->m_Center.x, Ball->m_Center.y-m_nH);

    if(CKgVector2D::abs(Ball->m_Velocity) < 0.01)
      Ball->m_Velocity = CKgVector2D(0, 0);
  }
```

```
std::vector<std::pair<CKhuGleSprite*, CKhuGleSprite*>> CollisionPairs;
for(auto &SpriteA : Layer->m_Children) {
  CKhuGleSprite *Ball = (CKhuGleSprite *)SpriteA;
  if(Ball->m_nType != GP_STYPE_ELLIPSE) continue;
  for(auto &SpriteB : Layer->m_Children) {
    CKhuGleSprite *Target = (CKhuGleSprite *)SpriteB;
    if(Ball == Target) continue;
    if(((CKhuGleSprite *)Target)->m_nType == GP_STYPE_ELLIPSE)  {
      CKgVector2D PosVec = Ball->m_Center - Target->m_Center;
      double Overlapped
         = CKgVector2D::abs(PosVec)
              - Ball->m_Radius - Target->m_Radius;
```



```
      if(Overlapped <= 0) {
        CollisionPairs.push_back({Ball, Target});
        if(CKgVector2D::abs(PosVec) == 0) {
          if(Ball->m_nCollisionType != GP_CTYPE_STATIC)
            Ball->MoveBy(rand()%3-1, rand()%3-1);
          if(Target->m_nCollisionType != GP_CTYPE_STATIC)
            Target->MoveBy(rand()%3-1, rand()%3-1);
        }
        else {
          if(Ball->m_nCollisionType != GP_CTYPE_STATIC) {
            if(Target->m_nCollisionType == GP_CTYPE_STATIC)
              Ball->MoveBy(
                -PosVec.x*Overlapped/CKgVector2D::abs(PosVec),
                -PosVec.y*Overlapped/CKgVector2D::abs(PosVec));
            else
              Ball->MoveBy(
                -PosVec.x*Overlapped/CKgVector2D::abs(PosVec)*0.5,
                -PosVec.y*Overlapped/CKgVector2D::abs(PosVec)*0.5);
          }
```

```
            if(Target->m_nCollisionType != GP_CTYPE_STATIC) {
              if(Ball->m_nCollisionType == GP_CTYPE_STATIC)
                Target->MoveBy(PosVec.x*Overlapped/CKgVector2D::abs(PosVec),
                        PosVec.y*Overlapped/CKgVector2D::abs(PosVec));
              else
                Target->MoveBy
                  (PosVec.x*Overlapped/CKgVector2D::abs(PosVec)*0.5,
                  PosVec.y*Overlapped/CKgVector2D::abs(PosVec)*0.5);
            }
          }
          Ball->m_bCollided = true;
          Target->m_bCollided = true;
        }
      }
    }
  }
```



```
  for(auto &Pair : CollisionPairs) {
    CKhuGleSprite *BallA = Pair.first;
    CKhuGleSprite *BallB = Pair.second;

    CKgVector2D PosVec = BallB->m_Center - BallA->m_Center;
    double Distance = CKgVector2D::abs(PosVec);
    if(Distance == 0) Distance = 1E-6;
    CKgVector2D Normal = (1./Distance)*PosVec;

    double kx = (BallA->m_Velocity.x - BallB->m_Velocity.x);
    double ky = (BallA->m_Velocity.y - BallB->m_Velocity.y);
    double p = 2.0
      * (Normal.x * kx + Normal.y * ky) / (BallA->m_Mass + BallB->m_Mass);

    BallA->m_Velocity.x = BallA->m_Velocity.x - p * BallB->m_Mass * Normal.x;
    BallA->m_Velocity.y = BallA->m_Velocity.y - p * BallB->m_Mass * Normal.y;

    BallB->m_Velocity.x = BallB->m_Velocity.x + p * BallA->m_Mass * Normal.x;
    BallB->m_Velocity.y = BallB->m_Velocity.y + p * BallA->m_Mass * Normal.y;
  }
}
```

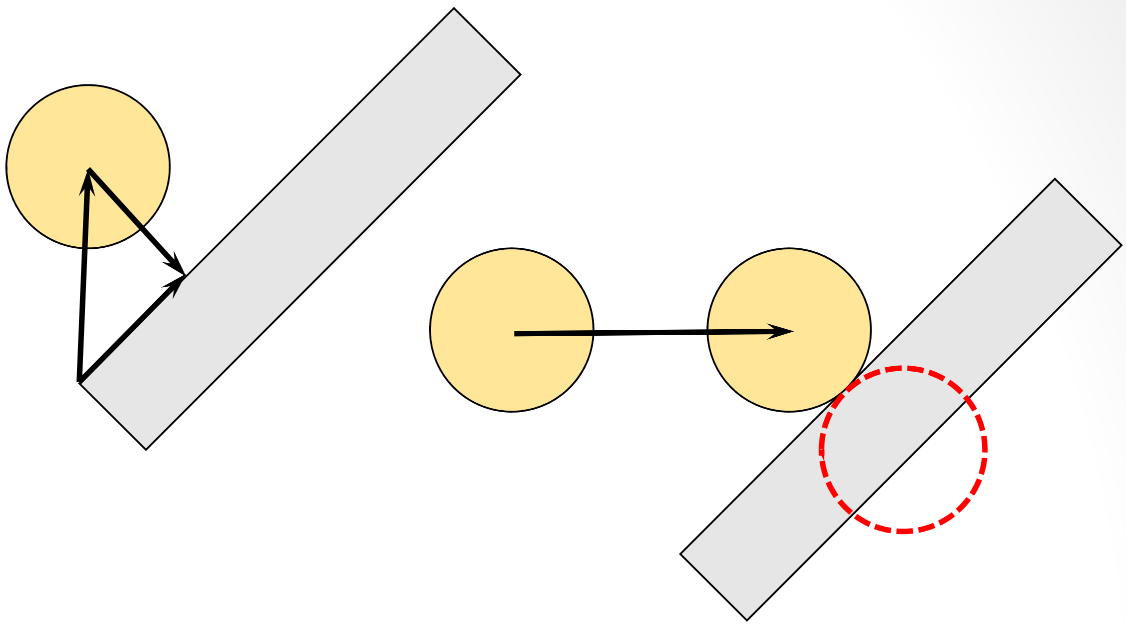
```
  m_pScene->Render();
  DrawSceneTextPos("Collision and Physics", CKgPoint(0, 0));
  CKhuGleWin::Update();
}

int main() {
  CCollision *pCollision = new CCollision(640, 480);

  KhuGleWinInit(pCollision);

  return 0;
}
```



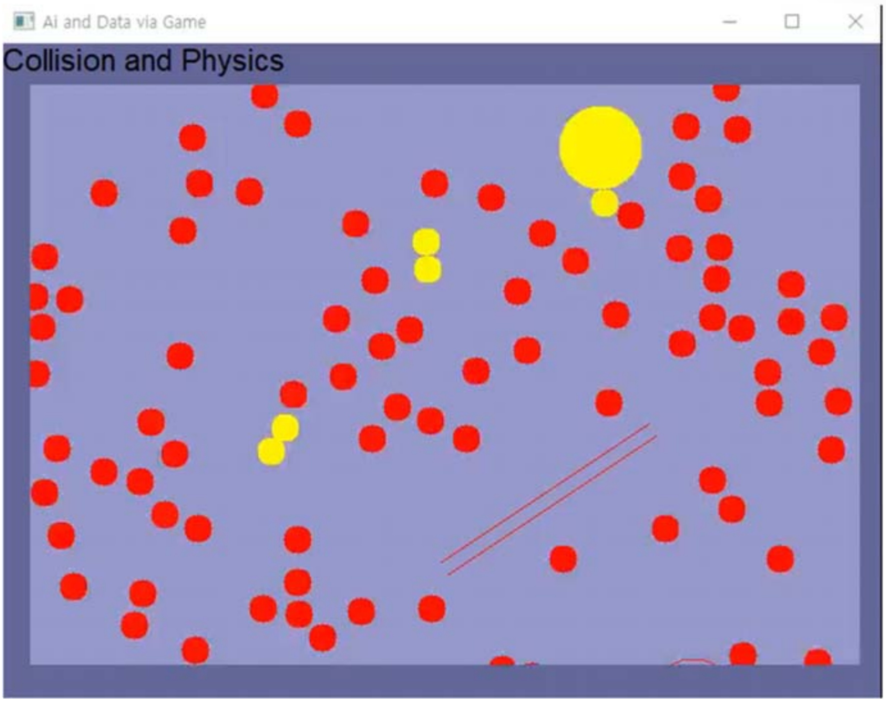
```
// Collision view: KhuGleSprite.h, KhuGleSprite.cpp
class CKhuGleSprite : public CKhuGleComponent {
  …
  int m_bCollided;
}
void CKhuGleSprite::Render() {
  if(!m_Parent) return;

  CKhuGleLayer *Parent = (CKhuGleLayer *)m_Parent;
  KgColor24 SaveColor = m_fgColor;

  if(m_bCollided) m_fgColor = KG_COLOR_24_RGB(255, 255, 0);
  …
  m_fgColor = SaveColor;
}
```

Practice I (1)

- Circle-line collision



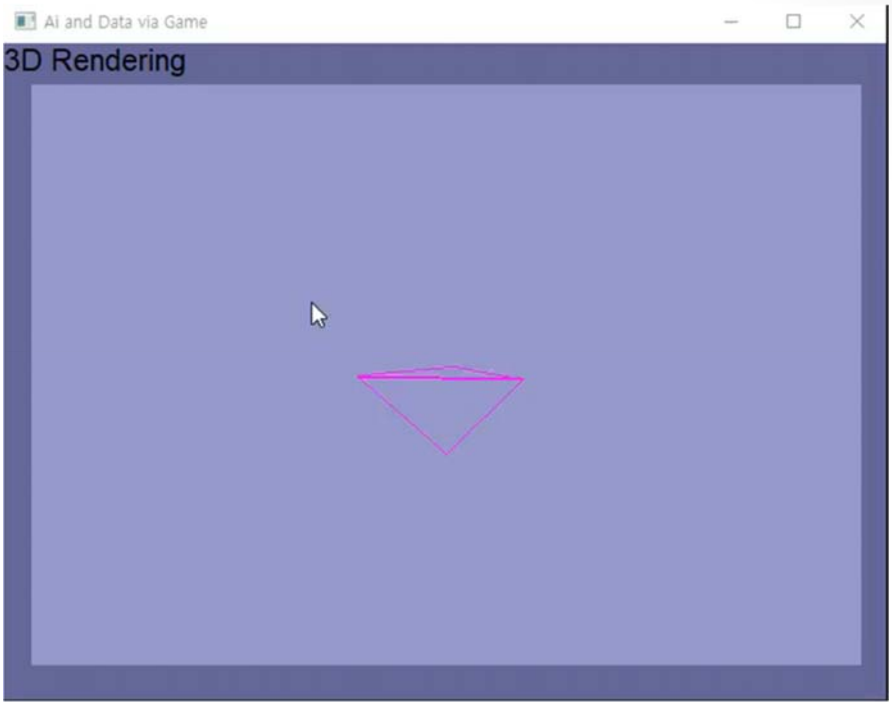
Practice I (2)



Advanced Courses

- Friction
- Elasticity

4. 3D Rendering



KhuGleBase.h (1)

```
class CKgVector3D {
public:
  double x, y, z;

  CKgVector3D() : x(0.), y(0.), z(0.) {}
  CKgVector3D(double xx, double yy, double zz) :
        x(xx), y(yy), z(zz) {}

  static double abs(CKgVector3D v);
  void Normalize();
  double Dot(CKgVector3D v1);
  CKgVector3D Cross(CKgVector3D v);
  CKgVector3D operator+ (CKgVector3D v);
  CKgVector3D operator- (CKgVector3D v);
  CKgVector3D operator- ();
  CKgVector3D &operator+= (CKgVector3D v);
};
CKgVector3D operator*(double s, CKgVector3D v);
```


KhuGleBase.cpp (1)

```
double CKgVector3D::abs(CKgVector3D v) {
  return sqrt(v.x*v.x + v.y*v.y + v.z*v.z);
}

void CKgVector3D::Normalize() {
  double Magnitude = abs(*this);
  if(Magnitude == 0) return;
  x /= Magnitude;        y /= Magnitude;          z /= Magnitude;
}

double CKgVector3D::Dot(CKgVector3D v) {
  return x*v.x + y*v.y + z*v.z;
}

CKgVector3D CKgVector3D::Cross(CKgVector3D v) {
  CKgVector3D NewV;
  NewV.x = y*v.z - z*v.y;
  NewV.y = z*v.x - x*v.z;
  NewV.z = x*v.y - y*v.x;
  return NewV;
}
```

KhuGleBase.cpp (2)

```
CKgVector3D CKgVector3D::operator+ (CKgVector3D v) {
  return CKgVector3D(x+v.x, y+v.y, z+v.z);
}

CKgVector3D CKgVector3D::operator- (CKgVector3D v) {
  return CKgVector3D(x-v.x, y-v.y, z-v.z);
}

CKgVector3D CKgVector3D::operator- () {
  return CKgVector3D(-x, -y, -z);
}

CKgVector3D &CKgVector3D::operator+= (CKgVector3D v) {
  *this = *this + v;
  return *this;
}

CKgVector3D operator*(double s, CKgVector3D v) {
  return CKgVector3D(s*v.x, s*v.y, s*v.z);
}
```


Inverse matrix (1)

$$\mathbf{A} = \mathbf{L}\mathbf{U}$$

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ \alpha_{21} & 1 & 0 & 0 \\ \alpha_{31} & \alpha_{32} & 1 & 0 \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & 1 \end{pmatrix} \begin{pmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ 0 & \beta_{22} & \beta_{23} & \beta_{24} \\ 0 & 0 & \beta_{33} & \beta_{34} \\ 0 & 0 & 0 & \beta_{44} \end{pmatrix}$$

$$\beta_{ij} = a_{ij} - \sum_{k=1}^{i-1} \alpha_{ik}\beta_{kj}$$

$$\alpha_{ij} = \frac{1}{\beta_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} \alpha_{ik}\beta_{kj} \right)$$

Inverse matrix (2)

$$\mathbf{AB} = \mathbf{I}$$
$$\mathbf{Ab}_1 = \mathbf{e}_1$$
$$\mathbf{Ab}_n = \mathbf{e}_n$$

$$\mathbf{Ab}_1 = \mathbf{e}_1$$
$$\mathbf{LUb}_1 = \mathbf{e}_1 \quad \leftarrow \mathbf{z} = \mathbf{Ub}_1$$

$\mathbf{Lz} = \mathbf{e}_1$

Forward substitution

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ \alpha_{21} & 1 & 0 & 0 \\ \alpha_{31} & \alpha_{32} & 1 & 0 \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & 1 \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$\mathbf{Ub}_1 = \mathbf{z}$

Backsubstitution

$$\begin{pmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ 0 & \beta_{22} & \beta_{23} & \beta_{24} \\ 0 & 0 & \beta_{33} & \beta_{34} \\ 0 & 0 & 0 & \beta_{44} \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} = \begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{pmatrix}$$

Inverse matrix (3)

```
bool InverseMatrix(double  **a, double **y, int nN) {
  double **CopyA = dmatrix(nN, nN);
  double *col = new double[nN];
  int *indx = new int[nN];
  double d;

  for(int r = 0; r < nN; r++)
    for(int c = 0; c < nN; c++)
        CopyA[r][c] = a[r][c];

  if(!ludcmp(CopyA, nN, indx, &d)) {
    free_dmatrix(CopyA, nN, nN);

    delete[] indx;
    delete[] col;
    return false;
  }
```

Inverse matrix (3)

```
  for(int j = 0; j < nN; j++) {
    for(int i = 0; i < nN; i++)
         col[i] = 0.0;
    col[j] = 1.0;
    lubksb(CopyA, nN, indx, col);
    for(int i = 0; i < nN; i++)
         y[i][j] = col[i];
  }

  free_dmatrix(CopyA, nN, nN);

  delete[] indx;
  delete[] col;

  return true;
}
```


LU decomposition

Inverse matrix (5)

```
bool ludcmp(double **a, int nN, int *indx, double *d) {
  int i, imax, j, k;
  double big, dum, sum, temp;
  double *vv = new double[nN];
  const double TinyValue = 1.0e-20;

  *d = 1.0;
  for(i = 0; i < nN; i++) {
    big = 0.0;
    for (j = 0; j < nN; j++)
         if ((temp = fabs(a[i][j])) > big) big = temp;
    if (big == 0.0) {
         delete[] vv;    return false; // Singular
    }
    vv[i] = 1.0 / big;
  }
```

Inverse matrix (6)

```
  for(j = 0; j < nN; j++) {
    for(i = 0; i < j; i++) {
         sum = a[i][j];
         for(k = 0; k < i; k++)  sum -= a[i][k] * a[k][j];
         a[i][j] = sum;
    }

    big = 0.0;
    for(i = j; i < nN; i++) {
         sum = a[i][j];
         for (k = 0; k < j; k++)
           sum -= a[i][k] * a[k][j];
         a[i][j] = sum;
         if ((dum = vv[i] * fabs(sum)) >= big) {
           big = dum; imax = i;
         }
    }
```


Inverse matrix (7)

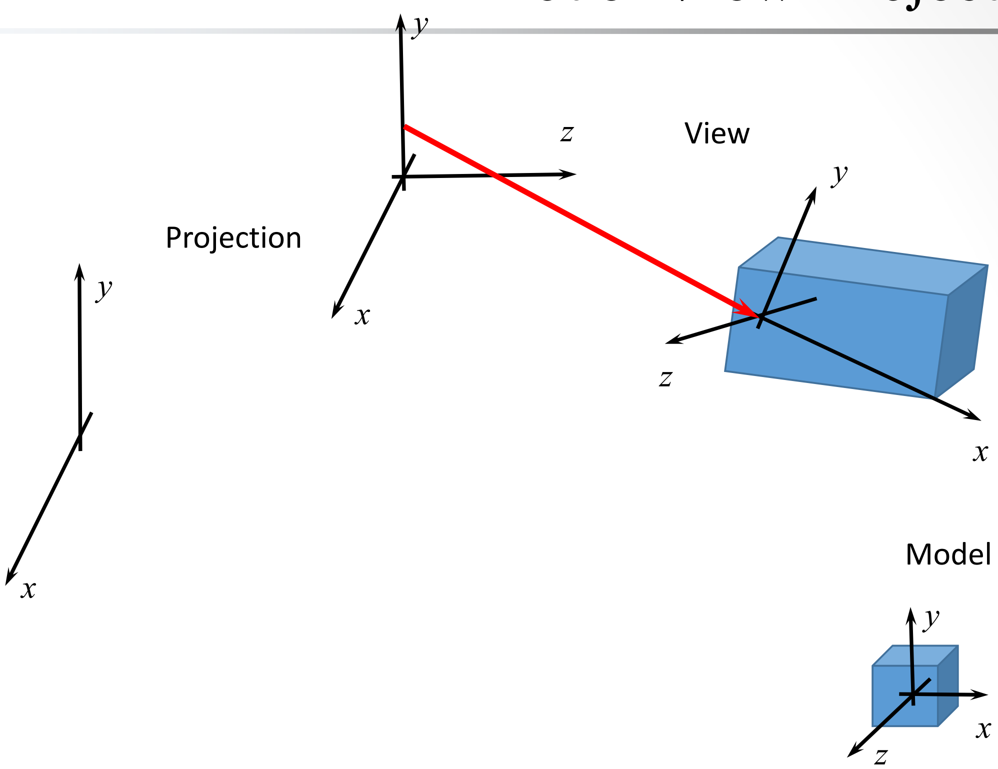
```
    if(j != imax){
         for (k = 0; k < nN; k++) {
           dum = a[imax][k];
           a[imax][k] = a[j][k];
            a[j][k] = dum;
         }
         *d = -(*d);                vv[imax] = vv[j];
    }

    indx[j] = imax;
    if(a[j][j] == 0.0) a[j][j] = TinyValue;
    if(j != nN - 1) {
         dum = 1.0 / (a[j][j]);
         for (i = j + 1; i < nN; i++)  a[i][j] *= dum;
    }
  }
  delete[] vv;
  return true;
}
```

InverseMatrix: KhuGleBase.h/cpp
Forward/back substitution

Inverse matrix (8)

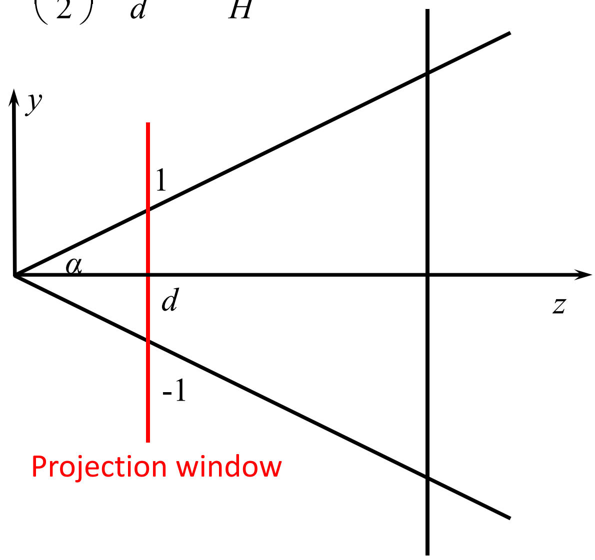
```
void lubksb(double **a, int nN, int *indx, double *b) {
  int i, ii = -1, ip, j;
  double sum;

  for(i = 0; i < nN; i++) {
    ip = indx[i];          sum = b[ip];              b[ip] = b[i];
    if(ii >= 0) {
         for(j = ii; j <= i - 1; j++) sum -= a[i][j] * b[j];
    }
    else if(sum) ii = i;
    b[i] = sum;
  }

  for(i = nN - 1; i >= 0; i--) {
    sum = b[i];
    for (j = i + 1; j < nN; j++)
         sum -= a[i][j] * b[j];
    b[i] = sum / a[i][i];
  }
}
```


Model/View/Projection



Projection (1)

$$\tan\left(\frac{\alpha}{2}\right)=\frac{1}{d},\ r=\frac{W}{H}$$



$$y_p : d = y : z$$

$$y_p = \frac{dy}{z} = \frac{y}{z\tan\left(\frac{\alpha}{2}\right)}$$

$$x_p : d = x : z$$

$$x_p = \frac{dx\left(H/W\right)}{z} = \frac{x}{rz\tan\left(\frac{\alpha}{2}\right)}$$

$$P = \begin{pmatrix} \frac{1}{r\tan\left(\frac{\alpha}{2}\right)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan\left(\frac{\alpha}{2}\right)} & 0 & 0 \\ 0 & 0 & ? & ? \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Projection (2)

- z range → [-1, 1]
- [near, far] → [-1, 1]

$$P = \begin{pmatrix} \frac{1}{r\tan\left(\frac{\alpha}{2}\right)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan\left(\frac{\alpha}{2}\right)} & 0 & 0 \\ 0 & 0 & A & B \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$z_p = \frac{Az+B}{z} = A + \frac{B}{z} \rightarrow$$

$$1 = A + \frac{B}{far},\quad -1 = A + \frac{B}{near}$$

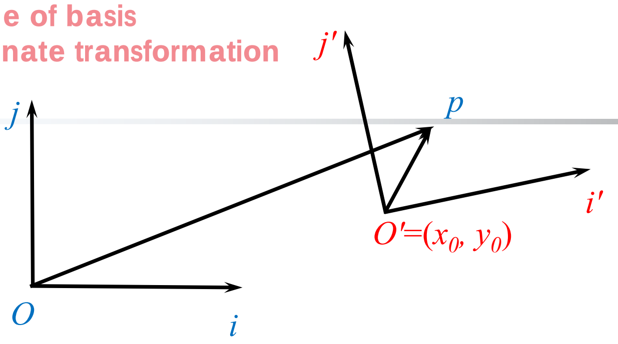
$$2 = \frac{B}{far} - \frac{B}{near} = \frac{B(near - far)}{farnear}$$

$$B = \frac{2\,far \times near}{near - far}$$

$$A = 1 - \frac{2\,far \times near}{near - far}\frac{1}{far} = \frac{-near - far}{near - far}$$

$$P = \begin{pmatrix} \frac{1}{r\tan\left(\frac{\alpha}{2}\right)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan\left(\frac{\alpha}{2}\right)} & 0 & 0 \\ 0 & 0 & \frac{-near - far}{near - far} & \frac{2\,far \times near}{near - far} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

View (1)

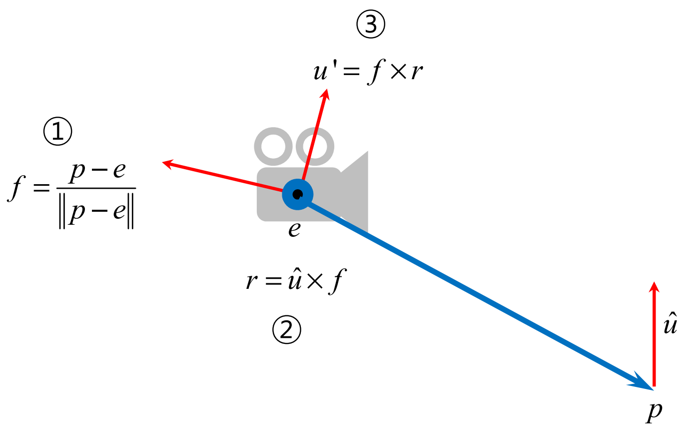


$$\overrightarrow{OP} = x\vec{i} + y\vec{j}$$
$$\overrightarrow{O'P} = x'\vec{i}' + y'\vec{j}'$$
$$\overrightarrow{OO'} = x_0\vec{i} + y_0\vec{j}$$

$$\overrightarrow{OP} = \overrightarrow{OO'} + \overrightarrow{O'P}$$
$$x\vec{i} + y\vec{j} = x_0\vec{i} + y_0\vec{j} + x'\vec{i}' + y'\vec{j}'$$
$$(x - x_0)\vec{i} + (y - y_0)\vec{j} = x'\vec{i}' + y'\vec{j}'$$
$$\begin{pmatrix}\vec{i} & \vec{j}\end{pmatrix}\begin{pmatrix}x - x_0 \\ y - y_0\end{pmatrix} = \begin{pmatrix}\vec{i}' & \vec{j}'\end{pmatrix}\begin{pmatrix}x' \\ y'\end{pmatrix}$$
$$\begin{pmatrix}x - x_0 \\ y - y_0\end{pmatrix} = \begin{pmatrix}\vec{i} & \vec{j}\end{pmatrix}^{-1}\begin{pmatrix}\vec{i}' & \vec{j}'\end{pmatrix}\begin{pmatrix}x' \\ y'\end{pmatrix} = \begin{pmatrix}\vec{i}^T \\ \vec{j}^T\end{pmatrix}\begin{pmatrix}\vec{i}' & \vec{j}'\end{pmatrix}\begin{pmatrix}x' \\ y'\end{pmatrix}$$

$$\begin{pmatrix}x \\ y \\ z\end{pmatrix} = \begin{pmatrix}\vec{i}^T\vec{i}' & \vec{i}^T\vec{j}' & x_0 \\ \vec{j}^T\vec{i}' & \vec{j}^T\vec{j}' & y_0 \\ 0 & 0 & 1\end{pmatrix}\begin{pmatrix}x' \\ y' \\ z'\end{pmatrix}$$
$$= \begin{pmatrix}i'_x & j'_x & x_0 \\ i'_y & j'_y & y_0 \\ 0 & 0 & 1\end{pmatrix}\begin{pmatrix}x' \\ y' \\ z'\end{pmatrix}$$

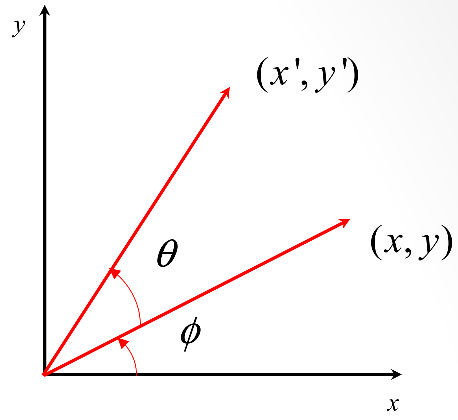
View (2)



$$\begin{pmatrix}x \\ y \\ z \\ w\end{pmatrix} = \begin{pmatrix}r_x & u'_x & f_x & C_x \\ r_y & u'_y & f_y & C_y \\ r_z & u'_z & f_z & C_z \\ 0 & 0 & 0 & 1\end{pmatrix}\begin{pmatrix}x' \\ y' \\ z' \\ w'\end{pmatrix} = V^{-1}\begin{pmatrix}x' \\ y' \\ z' \\ w'\end{pmatrix}$$

Model (1)

$$x = r\cos\phi$$
$$y = r\sin\phi$$



$$\begin{aligned} x' &= r\cos(\phi+\theta) \\ &= r\cos\phi\cos\theta - r\sin\phi\sin\theta \\ &= x\cos\theta - y\sin\theta \\ y' &= r\sin(\phi+\theta) \\ &= r\cos\phi\sin\theta + r\sin\phi\cos\theta \\ &= x\sin\theta + y\cos\theta \end{aligned}$$

Model (2)

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y\cos\theta - z\sin\theta \\ y\sin\theta + z\cos\theta \\ 1 \end{pmatrix}$$

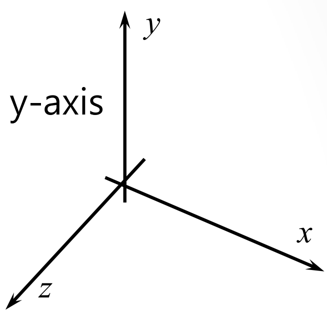
around x-axis

$R_x(\theta)$

$$\begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x\cos\theta + z\sin\theta \\ y \\ -x\sin\theta + z\cos\theta \\ 1 \end{pmatrix}$$

around y-axis

$R_y(\theta)$



$$\begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \\ z \\ 1 \end{pmatrix}$$

around z-axis

$R_z(\theta)$

Model (3)

$$R_z(\gamma)R_y(\beta)R_x(\alpha)$$

$$\begin{pmatrix} \cos\gamma & -\sin\gamma & 0 & 0 \\ \sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} \cos\beta & 0 & \sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} \cos\gamma\cos\beta & -\sin\gamma & \cos\gamma\sin\beta & 0 \\ \sin\gamma\cos\beta & \cos\gamma & \sin\gamma\sin\beta & 0 \\ -\sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} \cos\gamma\cos\beta & -\sin\gamma\cos\alpha+\cos\gamma\sin\beta\sin\alpha & \sin\gamma\sin\alpha+\cos\gamma\sin\beta\cos\alpha & 0 \\ \sin\gamma\cos\beta & \cos\gamma\cos\alpha+\sin\gamma\sin\beta\sin\alpha & -\cos\gamma\sin\alpha+\sin\gamma\sin\beta\cos\alpha & 0 \\ -\sin\beta & \cos\beta\sin\alpha & \cos\beta\cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Model (4)

- Translation

$$\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x+t_x \\ y+t_y \\ z+t_z \\ 1 \end{pmatrix}$$

- Scaling

$$\begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} s_x x \\ s_y y \\ s_z z \\ 1 \end{pmatrix}$$

```
#include "KhuGleWin.h"
#include <iostream>

struct CKgTriangle{
  CKgVector3D v0, v1, v2;

  CKgTriangle()
    : v0(CKgVector3D()), v1(CKgVector3D()), v2(CKgVector3D()) {};
  CKgTriangle(CKgVector3D vv0, CKgVector3D vv1, CKgVector3D vv2)
    : v0(vv0), v1(vv1), v2(vv2) {};
};
```



```
class CKhuGle3DSprite : public CKhuGleSprite {
public:
  std::vector<CKgTriangle> SurfaceMesh;
  double **m_ProjectionMatrix;
  CKgVector3D m_CameraPos;

  CKhuGle3DSprite(int nW, int nH, double Fov,
    double Far, double Near, KgColor24 fgColor);
  ~CKhuGle3DSprite();
```

```
  static void DrawTriangle(unsigned char **R,
    unsigned char **G, unsigned char **B, int nW, int nH,
    int x0, int y0, int x1, int y1, int x2, int y2,
    KgColor24 Color24);
  static void MatrixVector44(CKgVector3D &out,
    CKgVector3D v, double **M);
  static double **ComputeViewMatrix(CKgVector3D Camera,
    CKgVector3D Target, CKgVector3D CameraUp);

  void Render();
  void MoveBy(double OffsetX, double OffsetY, double OffsetZ);
};
```



```
CKhuGle3DSprite::CKhuGle3DSprite(int nW, int nH, double Fov, double Far,
  double Near, KgColor24 fgColor) {
  m_fgColor = fgColor;
  m_CameraPos = CKgVector3D(0., -0.2, -2);

  m_ProjectionMatrix = dmatrix(4, 4);
  for(int r = 0 ; r < 4 ; ++r)
    for(int c = 0 ; c < 4 ; ++c)
      m_ProjectionMatrix[r][c] = 0.;

  m_ProjectionMatrix[0][0] = (double)nH/(double)nW * 1./tan(Fov/2.);
  m_ProjectionMatrix[1][1] = 1./tan(Fov/2.);
  m_ProjectionMatrix[2][2] = (-Near-Far) / (Near-Far);
  m_ProjectionMatrix[2][3] = 2.*(Far * Near) / (Near-Far);
  m_ProjectionMatrix[3][2] = 1.;
  m_ProjectionMatrix[3][3] = 0.;
```

$$P = \begin{pmatrix} \frac{1}{r\tan\left(\frac{\alpha}{2}\right)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan\left(\frac{\alpha}{2}\right)} & 0 & 0 \\ 0 & 0 & \frac{-near - far}{near - far} & \frac{2\,far \times near}{near - far} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

```
 SurfaceMesh.push_back(CKgTriangle(CKgVector3D(-0.5, 0., -sqrt(3.)/6),
    CKgVector3D(0.5, 0., -sqrt(3.)/6), CKgVector3D(0., 0., sqrt(3.)/3)));
  SurfaceMesh.push_back(CKgTriangle(CKgVector3D(0., 0., sqrt(3.)/3),
    CKgVector3D(0.5, 0., -sqrt(3.)/6), CKgVector3D(0., sqrt(3.)/3, 0.)));
  SurfaceMesh.push_back(CKgTriangle(CKgVector3D(-0.5, 0., -sqrt(3.)/6),
    CKgVector3D(0, 0., sqrt(3.)/3), CKgVector3D(0., sqrt(3.)/3, 0.)));
  SurfaceMesh.push_back(CKgTriangle(CKgVector3D(0.5, 0., -sqrt(3.)/6),
    CKgVector3D(-0.5, 0., -sqrt(3.)/6), CKgVector3D(0., sqrt(3.)/3, 0.)));
};

CKhuGle3DSprite::~CKhuGle3DSprite() {
  free_dmatrix(m_ProjectionMatrix, 4, 4);
};
```



```
void CKhuGle3DSprite::DrawTriangle(unsigned char **R,
  unsigned char **G, unsigned char **B,
  int nW, int nH, int x0, int y0, int x1, int y1,
  int x2, int y2, KgColor24 Color24) {
  CKhuGleSprite::DrawLine(R, G, B, nW, nH, x0, y0, x1, y1, Color24);
  CKhuGleSprite::DrawLine(R, G, B, nW, nH, x1, y1, x2, y2, Color24);
  CKhuGleSprite::DrawLine(R, G, B, nW, nH, x2, y2, x0, y0, Color24);
}
```

```
void CKhuGle3DSprite::MatrixVector44(CKgVector3D &out,
  CKgVector3D v, double **M) {
  out.x = v.x*M[0][0] + v.y*M[0][1] + v.z*M[0][2] + M[0][3];
  out.y = v.x*M[1][0] + v.y*M[1][1] + v.z*M[1][2] + M[1][3];
  out.z = v.x*M[2][0] + v.y*M[2][1] + v.z*M[2][2] + M[2][3];

  double w = v.x*M[3][0] + v.y*M[3][1] + v.z*M[3][2]
         + M[3][3];

  if(fabs(w) > 0)
    out = (1./w)* out;
}
```



```
double **CKhuGle3DSprite::ComputeViewMatrix(CKgVector3D Camera, CKgVector3D
Target,
  CKgVector3D CameraUp) {
  CKgVector3D Forward = Target-Camera;

  Forward.Normalize();
  CameraUp.Normalize();

  CKgVector3D Right = CameraUp.Cross(Forward);
  CKgVector3D Up = Forward.Cross(Right);

  double **RT = dmatrix(4, 4);
  double **View = dmatrix(4, 4);
```

```
  RT[0][0] = Right.x;     RT[1][0] = Right.y;
  RT[2][0] = Right.z;     RT[3][0] = 0.;

  RT[0][1] = Up.x;        RT[1][1] = Up.y;
  RT[2][1] = Up.z;        RT[3][1] = 0.;

  RT[0][2] = Forward.x;   RT[1][2] = Forward.y;
  RT[2][2] = Forward.z;   RT[3][2] = 0.;

  RT[0][3] = Camera.x;    RT[1][3] = Camera.y;
  RT[2][3] = Camera.z;    RT[3][3] = 1.;

  bool bInverse = InverseMatrix(RT, View, 4);
  free_dmatrix(RT, 4, 4);

  if(bInverse) return View;
  return nullptr;
}
```

$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} r_x & u'_x & f_x & C_x \\ r_y & u'_y & f_y & C_y \\ r_z & u'_z & f_z & C_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = V^{-1} \begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix}$$


```
void CKhuGle3DSprite::Render() {
  if(!m_Parent) return;
  double NewX = m_CameraPos.x*cos(Pi/1000.) - m_CameraPos.z*sin(Pi/1000.);
  double NewZ = m_CameraPos.x*sin(Pi/1000.) + m_CameraPos.z*cos(Pi/1000.);
  m_CameraPos.x = NewX;
  m_CameraPos.z = NewZ;

  CKhuGleLayer *Parent = (CKhuGleLayer *)m_Parent;
  double **ViewMatrix = ComputeViewMatrix(m_CameraPos,
        CKgVector3D(0., 0., 0.), CKgVector3D(0., 1., 0.));
  if(ViewMatrix == nullptr) return;

  for(auto &Triangle: SurfaceMesh) {
    CKgVector3D Side01, Side02, Normal;
    Side01 = Triangle.v1 - Triangle.v0;
    Side02 = Triangle.v2 - Triangle.v0;
    Normal = Side01.Cross(Side02);
    Normal.Normalize();
```

```
    CKgTriangle ViewTriangle;
    CKgTriangle Projected;
    if(Normal.Dot(Triangle.v0 - m_CameraPos) < 0.)
    {
      MatrixVector44(ViewTriangle.v0, Triangle.v0, ViewMatrix);
      MatrixVector44(ViewTriangle.v1, Triangle.v1, ViewMatrix);
      MatrixVector44(ViewTriangle.v2, Triangle.v2, ViewMatrix);

      MatrixVector44(Projected.v0, ViewTriangle.v0, m_ProjectionMatrix);
      MatrixVector44(Projected.v1, ViewTriangle.v1, m_ProjectionMatrix);
      MatrixVector44(Projected.v2, ViewTriangle.v2, m_ProjectionMatrix);

      Projected.v0.x += 1.;              Projected.v0.y += 1.;
      Projected.v1.x += 1.;              Projected.v1.y += 1.;
      Projected.v2.x += 1.;              Projected.v2.y += 1.;
```



```
      Projected.v0.x *= Parent->m_nW/2.;
      Projected.v0.y *= Parent->m_nH/2.;
      Projected.v1.x *= Parent->m_nW/2.;
      Projected.v1.y *= Parent->m_nH/2.;
      Projected.v2.x *= Parent->m_nW/2.;
      Projected.v2.y *= Parent->m_nH/2.;
      Projected.v0.x -= 1.;              Projected.v0.y -= 1.;
      Projected.v1.x -= 1.;              Projected.v1.y -= 1.;
      Projected.v2.x -= 1.;              Projected.v2.y -= 1.;

      DrawTriangle(Parent->m_ImageR, Parent->m_ImageG, Parent->m_ImageB,
        Parent->m_nW, Parent->m_nH,
        (int)Projected.v0.x, (int)Projected.v0.y,
        (int)Projected.v1.x, (int)Projected.v1.y,
        (int)Projected.v2.x, (int)Projected.v2.y, m_fgColor);
    }
  }

  free_dmatrix(ViewMatrix, 4, 4);
}
```

```
void CKhuGle3DSprite::MoveBy(double OffsetX, double OffsetY,
  double OffsetZ) {
  for(auto &Triangle: SurfaceMesh) {
    Triangle.v0 = Triangle.v0 + CKgVector3D(OffsetX, OffsetY, OffsetZ);
    Triangle.v1 = Triangle.v1 + CKgVector3D(OffsetX, OffsetY, OffsetZ);
    Triangle.v2 = Triangle.v2 + CKgVector3D(OffsetX, OffsetY, OffsetZ);
  }
}

class CThreeDim : public CKhuGleWin {
public:
  CKhuGleLayer *m_pGameLayer;

  CKhuGle3DSprite *m_pObject3D;

  CThreeDim(int nW, int nH);
  void Update();

  CKgPoint m_LButtonStart, m_LButtonEnd;
  int m_nLButtonStatus;
};
```



```
CThreeDim::CThreeDim(int nW, int nH) : CKhuGleWin(nW, nH) {
  m_nLButtonStatus = 0;

  m_Gravity = CKgVector2D(0., 98.);
  m_AirResistance = CKgVector2D(0.1, 0.1);

  m_pScene = new CKhuGleScene(640, 480,
    KG_COLOR_24_RGB(100, 100, 150));

  m_pGameLayer = new CKhuGleLayer(600, 420,
    KG_COLOR_24_RGB(150, 150, 200), CKgPoint(20, 30));
  m_pScene->AddChild(m_pGameLayer);

  m_pObject3D = new CKhuGle3DSprite(m_pGameLayer->m_nW,
    m_pGameLayer->m_nH, Pi/2., 1000., 0.1,
    KG_COLOR_24_RGB(255, 0, 255));

  m_pGameLayer->AddChild(m_pObject3D);
}
```

```
void CThreeDim::Update() {
  if(m_bKeyPressed[VK_DOWN])
    m_pObject3D->MoveBy(0, 0.0005, 0);

  m_pScene->Render();
  DrawSceneTextPos("3D Rendering", CKgPoint(0, 0));

  CKhuGleWin::Update();
}

int main() {
  CThreeDim *pThreeDim = new CThreeDim(640, 480);

  KhuGleWinInit(pThreeDim);

  return 0;
}
```


Practice II

- Model matrix

Advanced Courses

- Depth
- Texture

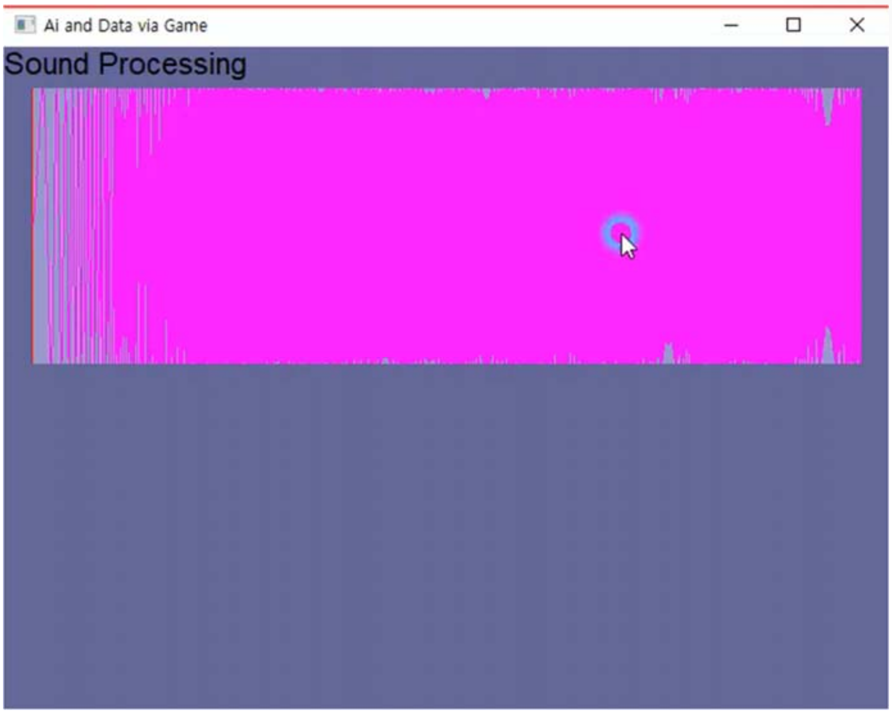
Project I

Game Design

Game Design

- Pong
- Simple platformer

5. Sound Processing



KhuGleSignal.h

```
#pragma once
typedef struct  tagWAV_HEADER_ {
  …
} WAV_HEADER_;
typedef struct tagCHUCK_ {
  …
} CHUCK_;
#pragma pack(push, 1)
typedef struct tagBITMAPFILEHEADER_ {
 …
} BITMAPFILEHEADER_;
typedef struct tagBITMAPINFOHEADER_ {
  …
} BITMAPINFOHEADER_;
typedef struct tagRGBQUAD_ {
  …
} RGBQUAD_;
#pragma pack(pop)
#define BI_RGB_        0L
```

KhuGleSignal.h

```
class CKhuGleSignal {
public:
  short int *m_Samples;
  int m_nSampleRate;
  int m_nSampleLength;

  double **m_Real, **m_Imaginary;
  int m_nWindowSize;
  int m_nFrequencySampleLength;

  int m_nW, m_nH;
  unsigned char **m_Red, **m_Green, **m_Blue;

  CKhuGleSignal();
  ~CKhuGleSignal();

  void ReadWave(char *FileName);          bool SaveWave(char *FileName);
  void ReadBmp(char *FileName);           bool SaveBmp(char *FileName);

  void MakeSpectrogram();
};
```


KhuGleSignal.cpp

```
#include "KhuGleSignal.h"
#include "KhuGleBase.h"
#include <cstdio>

CKhuGleSignal::CKhuGleSignal() {
  m_Samples = nullptr;

  m_Real = nullptr;
  m_Imaginary = nullptr;

  m_nWindowSize = 256;
  m_nFrequencySampleLength = 1024;

  m_Red = m_Green = m_Blue = nullptr;
}
```

KhuGleSignal.cpp

```
CKhuGleSignal::~CKhuGleSignal() {
  if(m_Samples) delete [] m_Samples;
  if(m_Real) free_dmatrix(m_Real, m_nFrequencySampleLength,
         m_nWindowSize);
  if(m_Imaginary) free_dmatrix(m_Imaginary,
         m_nFrequencySampleLength, m_nWindowSize);

  if(m_Red) free_cmatrix(m_Red, m_nH, m_nW);
  if(m_Green) free_cmatrix(m_Green, m_nH, m_nW);
  if(m_Blue) free_cmatrix(m_Blue, m_nH, m_nW);
}
void CKhuGleSignal::ReadWave(char *FileName){}
bool CKhuGleSignal::SaveWave(char *FileName){}

void CKhuGleSignal::ReadBmp(char *FileName){}
bool CKhuGleSignal::SaveBmp(char *FileName){}

void CKhuGleSignal::MakeSpectrogram(){}
```


SoundPlayWin.cpp

```
#include <windows.h>
#include <mmsystem.h>
#pragma comment(lib, "Winmm.lib")

HWAVEOUT hPlay;
bool bSoundPlaying = false;
WAVEFORMATEX WaveFormat;
WAVEHDR WaveHdr;

void CALLBACK waveOutProc(HWAVEOUT hWO, UINT uMsg, DWORD dwInstance,
  DWORD dwParam1, DWORD dwParam2) {
  …
}

void PlayWave(short int *Sound, int nSampleRate, int nLen) {
  …
}

void StopWave() {
  …
}

void GetPlaybackPosotion(unsigned long *pPosition) {
  …
}
```

KhuGleWin.h

```
#include <windows.h>
#include "KhuGleBase.h"
#include "KhuGleSprite.h"
#include "KhuGleLayer.h"
#include "KhuGleScene.h"
#include "KhuGleComponent.h"

void PlayWave(short int *Sound, int nSampleRate, int nLen);
void StopWave();
void GetPlaybackPosotion(unsigned long *Rate);

…
```


Main.cpp

```
class CKhuGleSoundLayer : public CKhuGleLayer {
public:
  CKhuGleSignal m_Sound;
  int m_nViewType;

  CKhuGleSoundLayer(int nW, int nH, KgColor24 bgColor,
    CKgPoint ptPos = CKgPoint(0, 0));
  void DrawBackgroundImage();
};

CKhuGleSoundLayer::CKhuGleSoundLayer(int nW, int nH, KgColor24 bgColor,
  CKgPoint ptPos)
  : CKhuGleLayer(nW, nH, bgColor, ptPos) {

  m_nViewType = 0;
}
```

```
void CKhuGleSoundLayer:: DrawBackgroundImage() {
  for(int y = 0 ; y < m_nH ; y++)
    for(int x = 0 ; x < m_nW ; x++) {
      m_ImageBgR[y][x] = KgGetRed(m_bgColor);
      m_ImageBgG[y][x] = KgGetGreen(m_bgColor);
      m_ImageBgB[y][x] = KgGetBlue(m_bgColor);
    }
  if(m_nViewType == 0 && m_Sound.m_Samples) {
    int xx0, yy0, xx1, yy1;
    for(int i = 0 ; i < m_Sound.m_nSampleLength ; ++i) {
      xx1 = i*m_nW/m_Sound.m_nSampleLength;
      yy1 = m_nH-(m_Sound.m_Samples[i]+32768)*m_nH/65536-1;
      if(i > 0)
        CKhuGleSprite::DrawLine(m_ImageBgR, m_ImageBgG,
            m_ImageBgB, m_nW, m_nH,
          xx0, yy0, xx1, yy1, KG_COLOR_24_RGB(255, 0, 255));
      xx0 = xx1;
      yy0 = yy1;
    }
  }
```



```
  if(m_nViewType == 1 && m_Sound.m_Real && m_Sound.m_Imaginary) {
    double Max = 0;
    for(int y = 0 ; y < m_nH ; y++)
      for(int x = 0 ; x < m_nW ; x++) {
        int yy = (m_nH-y-1)/2*m_Sound.m_nWindowSize/m_nH;
        int xx = x*m_Sound.m_nFrequencySampleLength/m_nW;

        double Magnitude = sqrt(m_Sound.m_Real[xx][yy]*m_Sound.m_Real[xx][yy] +
          m_Sound.m_Imaginary[xx][yy]*m_Sound.m_Imaginary[xx][yy]);
        if(Magnitude > Max) Max = Magnitude;
      }
```

```
    for(int y = 0 ; y < m_nH ; y++)
      for(int x = 0 ; x < m_nW ; x++) {
        int yy = (m_nH-y-1)/2*m_Sound.m_nWindowSize/m_nH;
        int xx = x*m_Sound.m_nFrequencySampleLength/m_nW;

        m_ImageBgR[y][x] =
          (int)(sqrt(m_Sound.m_Real[xx][yy]*m_Sound.m_Real[xx][yy] +
           m_Sound.m_Imaginary[xx][yy]*m_Sound.m_Imaginary[xx][yy])*255/Max);
        m_ImageBgG[y][x] = m_ImageBgR[y][x];
        m_ImageBgB[y][x] = m_ImageBgR[y][x];
      }
  }
```



```
  if(m_nViewType == 2 && m_Sound.m_Real && m_Sound.m_Imaginary) {
    double Max = 0, Min = 0;
    for(int y = 0 ; y < m_nH ; y++)
      for(int x = 0 ; x < m_nW ; x++) {
        int yy = (m_nH-y-1)/2*m_Sound.m_nWindowSize/m_nH;
        int xx = x*m_Sound.m_nFrequencySampleLength/m_nW;

        double Magnitude = sqrt(m_Sound.m_Real[xx][yy]*m_Sound.m_Real[xx][yy] +
          m_Sound.m_Imaginary[xx][yy]*m_Sound.m_Imaginary[xx][yy]);
        Magnitude = 10*log10(Magnitude*Magnitude+1.);
        if(x == 0 && y == 0) {
          Min = Magnitude;
          Max = Magnitude;
        }

        if(Magnitude > Max) Max = Magnitude;
        if(Magnitude < Min) Min = Magnitude;
      }
```

```
    for(int y = 0 ; y < m_nH ; y++)
      for(int x = 0 ; x < m_nW ; x++) {
        int yy = (m_nH-y-1)/2*m_Sound.m_nWindowSize/m_nH;
        int xx = x*m_Sound.m_nFrequencySampleLength/m_nW;

        double Magnitude = sqrt(m_Sound.m_Real[xx][yy]*m_Sound.m_Real[xx][yy] +
          m_Sound.m_Imaginary[xx][yy]*m_Sound.m_Imaginary[xx][yy]);
        Magnitude = 10*log10(Magnitude*Magnitude+1.);

        m_ImageBgR[y][x] = (int)((Magnitude-Min)*255/(Max-Min));
        m_ImageBgG[y][x] = m_ImageBgR[y][x];
        m_ImageBgB[y][x] = m_ImageBgR[y][x];
      }
  }
}
```



```
class CSoundProcessing : public CKhuGleWin {
public:
  CKhuGleSoundLayer *m_pSoundLayer;
  CKhuGleSprite *m_pSoundLine;

  CSoundProcessing(int nW, int nH, char *SoundPath);
  void Update();
};
```

```
CSoundProcessing::CSoundProcessing(int nW, int nH, char *SoundPath)
  : CKhuGleWin(nW, nH) {
  m_pScene = new CKhuGleScene(640, 480, KG_COLOR_24_RGB(100, 100, 150));

  m_pSoundLayer = new CKhuGleSoundLayer(600, 200,
    KG_COLOR_24_RGB(150, 150, 200), CKgPoint(20, 30));
  m_pSoundLayer->m_Sound.ReadWave(SoundPath);
  m_pSoundLayer->DrawBackgroundImage();
  m_pScene->AddChild(m_pSoundLayer);

  m_pSoundLayer->m_Sound.MakeSpectrogram();

  m_pSoundLine = new CKhuGleSprite(GP_STYPE_LINE, GP_CTYPE_KINEMATIC,
    CKgLine(CKgPoint(0, 0), CKgPoint(0, m_pSoundLayer->m_nH)),
    KG_COLOR_24_RGB(255, 0, 0), false, 0);
  m_pSoundLayer->AddChild(m_pSoundLine);
}
```

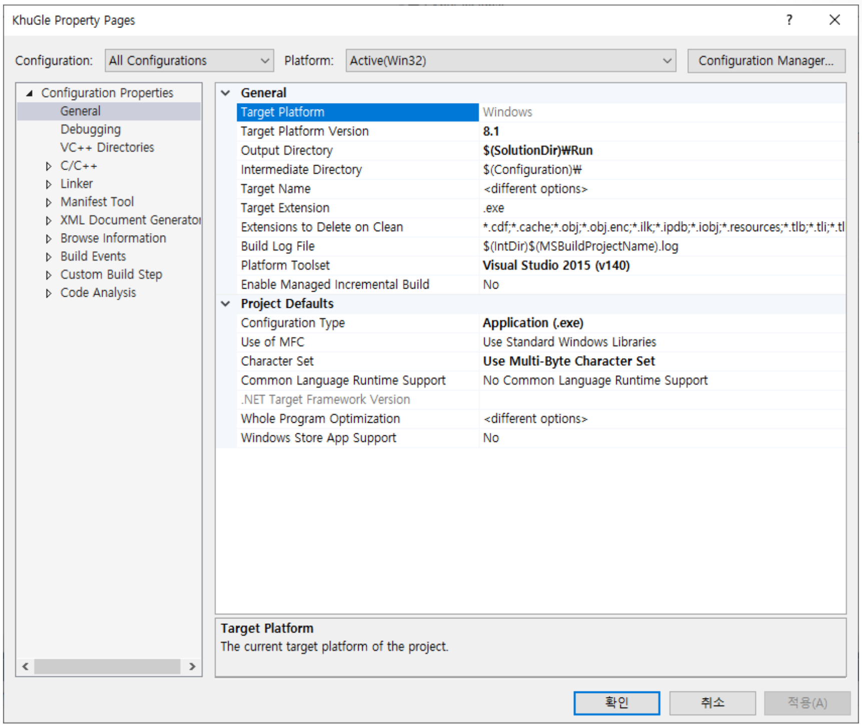


```
void CSoundProcessing::Update() {
  if(m_bKeyPressed['S']) StopWave();

  if(m_bKeyPressed['T'] || m_bKeyPressed['F'] || m_bKeyPressed['L']) {
    if(m_bKeyPressed['T']) m_pSoundLayer->m_nViewType = 0;
    if(m_bKeyPressed['F']) m_pSoundLayer->m_nViewType = 1;
    if(m_bKeyPressed['L']) m_pSoundLayer->m_nViewType = 2;
    m_pSoundLayer->DrawBackgroundImage();
  }
  if(m_bKeyPressed['M']) {
    int nLength = 3;
    for(int i = 0 ; i < m_pSoundLayer->m_Sound.m_nSampleLength-nLength ; ++i) {
      for(int ii = 1 ; ii < nLength ; ++ii)
        m_pSoundLayer->m_Sound.m_Samples[i]
          += m_pSoundLayer->m_Sound.m_Samples[i+ii];
      m_pSoundLayer->m_Sound.m_Samples[i] /= nLength;
    }
    m_pSoundLayer->m_Sound.MakeSpectrogram();
    m_pSoundLayer->DrawBackgroundImage();
    m_bKeyPressed['M'] = false;
  }
```

```
 if(m_bKeyPressed['P'])
    PlayWave(m_pSoundLayer->m_Sound.m_Samples,
      m_pSoundLayer->m_Sound.m_nSampleRate,
         m_pSoundLayer->m_Sound.m_nSampleLength);

  unsigned long nPosition;
  GetPlaybackPosotion(&nPosition);
  if(nPosition > 0)
    m_pSoundLine->MoveTo(nPosition*600/m_pSoundLayer->m_Sound.m_nSampleLength,
         m_pSoundLayer->m_nH/2);

  m_pScene->Render();
  DrawSceneTextPos("Sound Processing", CKgPoint(0, 0));
  CKhuGleWin::Update();
}
```



```
int main() {
  char ExePath[MAX_PATH], SoundPath[MAX_PATH];
  GetModuleFileName(NULL, ExePath, MAX_PATH);

  int i;
  int LastBackSlash = -1;
  int nLen = strlen(ExePath);
  for(i = nLen-1 ; i >= 0 ; i--) {
    if(ExePath[i] == '\\') {
      LastBackSlash = i;
      break;
    }
  }
  if(LastBackSlash >= 0)
  ExePath[LastBackSlash] = '\0';
  sprintf(SoundPath, "%s\\%s", ExePath, "ex.wav");

  CSoundProcessing *pSoundProcessing = new CSoundProcessing(640, 480, SoundPath);
  KhuGleWinInit(pSoundProcessing);

  return 0;
}
```



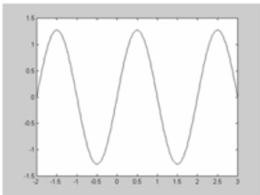
http://www.cstr.ed.ac.uk/projects/eustace/download.html

Fourier Series

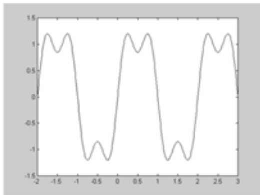
$$f(t) = \sum_{k=-\infty}^{\infty} a_k e^{jk\varpi_0 t}$$

$$f_a(t) = \sum_{k=-N}^{N} a_k e^{jk\varpi_0 t}$$

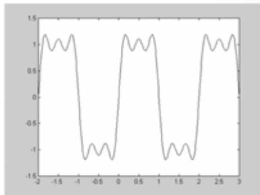
$$a_k = \frac{1}{T_0}\int_{T_0} f(t) e^{-jk\omega_0 t} dt$$



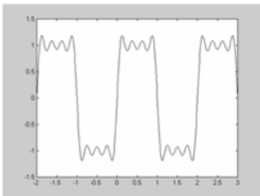
N=1



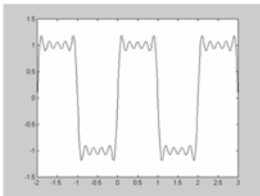
N=3



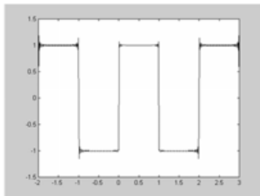
N=5



N=7



N=9



N=99

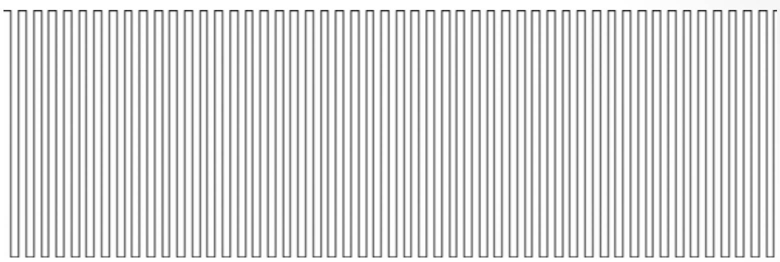
Fourier Transform

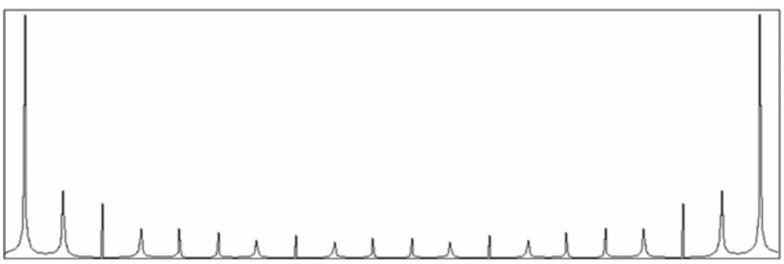
$$F(\omega) = \int_{-\infty}^{\infty} f(t) e^{-j\omega t} dt$$

$$f(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} F(\omega) e^{j\omega t} d\omega$$

$$F(u) = \sum_{n=0}^{N-1} f(n) e^{-j2\pi \frac{un}{N}}$$

$$f(n) = \frac{1}{N}\sum_{u=0}^{N-1} F(u) e^{j2\pi \frac{un}{N}}$$





$f_s/2$

Fourier Transform Properties

$$x(t) \leftrightarrow X(\omega),\ y(t) \leftrightarrow Y(\omega)$$

$$x(t-t_0) \leftrightarrow e^{-j\omega t_0} X(\omega)$$

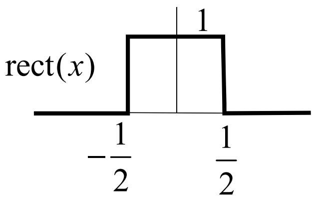
$$e^{j\omega t_0} x(t) \leftrightarrow X(\omega - \omega_0)$$

$$x(t)y(t) \leftrightarrow X(\omega) * Y(\omega)$$

$$x(t) * y(t) \leftrightarrow \frac{1}{2\pi} X(\omega)Y(\omega)$$

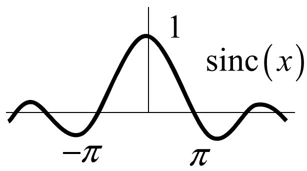
$$\delta(t) \leftrightarrow 1$$

$$1 \leftrightarrow 2\pi\delta(\omega)$$



$$\mathrm{rect}(t/\tau) \leftrightarrow \tau \mathrm{sinc}\left(\frac{\omega\tau}{2}\right)$$

$$\frac{W}{\pi} \mathrm{sinc}(Wt) \leftrightarrow \mathrm{rect}\left(\frac{\omega}{2W}\right)$$



FFT I

$$F(u) = \sum_{n=0}^{N-1} f(n) e^{-j2\pi \frac{un}{N}} = \sum_{n=0}^{N-1} f(n) W_N^{un}$$

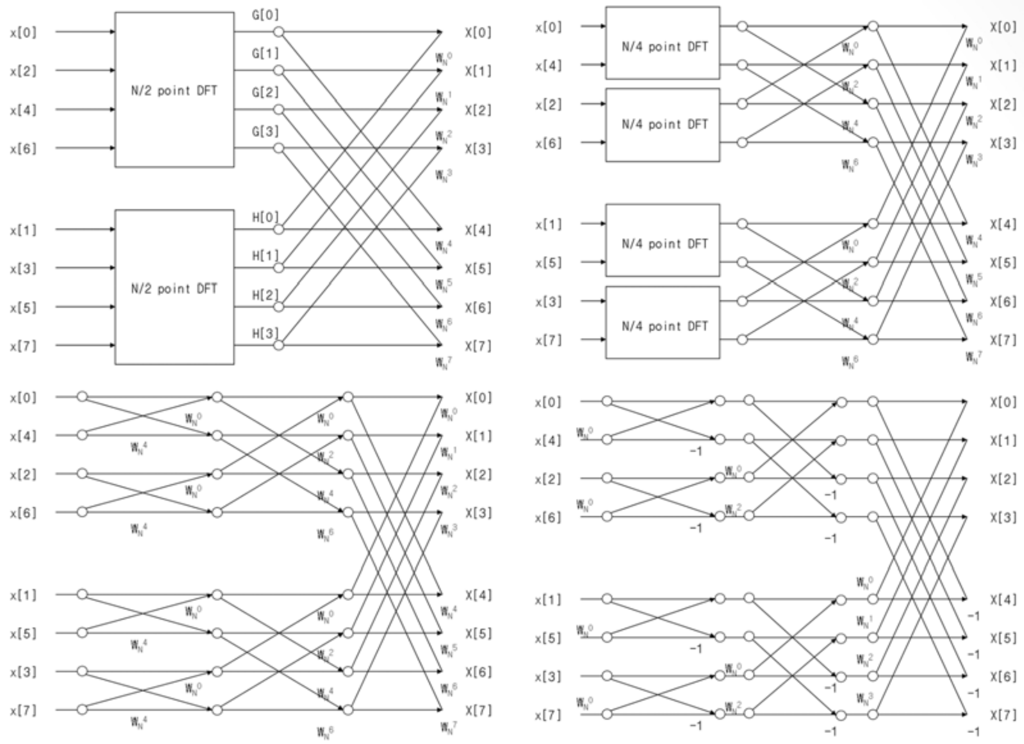
$$= \sum_{n=even} f(n) W_N^{un} + \sum_{n=odd} f(n) W_N^{un}$$

$$= \sum_{r=0}^{N/2-1} f(2r) W_N^{2ru} + \sum_{r=0}^{N/2-1} f(2r+1) W_N^{(2r+1)u}$$

$$= \sum_{r=0}^{N/2-1} f(2r) \left(W_N^2\right)^{ru} + W_N^u \sum_{r=0}^{N/2-1} f(2r+1) \left(W_N^2\right)^{ru}$$

$$= \sum_{r=0}^{N/2-1} f(2r) W_{N/2}^{ur} + W_N^u \sum_{r=0}^{N/2-1} f(2r+1) W_{N/2}^{ur}$$

FFT II



FFT III

```
void FFT2Radix(double *Xr, double *Xi, double *Yr, double *Yi,
  int nN, bool bInverse){
  int i, j, k;
  double T, Wr, Wi;

  if(nN <= 1) return;
  for(i = 0 ; i < nN ; i++) {
    Yr[i] = Xr[i];
    Yi[i] = Xi[i];
  }

  j = 0;
  for (i = 1 ; i < (nN-1) ; i++) {
    k = nN/2;
    while(k <= j) {         j = j - k;        k = k/2;}
    j = j + k;
    if (i < j) {
          T = Yr[j];         Yr[j] = Yr[i];     Yr[i] = T;
          T = Yi[j];         Yi[j] = Yi[i];     Yi[i] = T;
    }
  }

  double Tr, Ti;
  int iter, j2, pos;
  k = nN >> 1;
  iter = 1;
```

FFT IV

```
  while(k > 0) {
    j = 0;
    j2 = 0;
    for(i = 0 ; i < nN >> 1 ; i++) {
      Wr = cos(2.*Pi*(j2*k)/nN);
      if(bInverse == 0)
        Wi = -sin(2.*Pi*(j2*k)/nN);
      else
        Wi = sin(2.*Pi*(j2*k)/nN);

      pos = j+(1 << (iter-1));
      Tr =Yr[pos] * Wr - Yi[pos] * Wi;
      Ti = Yr[pos] * Wi +Yi[pos] * Wr;

      Yr[pos] = Yr[j] - Tr;
      Yi[pos] = Yi[j] - Ti;

      Yr[j] += Tr;
      Yi[j] += Ti;
      j += 1 << iter;
      if(j >= nN) j = ++j2;
    }
    k >>= 1;
    iter++;
  }
```


FFT V

```
  if(bInverse){
    for(i = 0 ; i < nN ; i++) {
      Yr[i] /= nN;
      Yi[i] /= nN;
    }
  }
}
```

Spectrogram

```
void CKhuGleSignal::MakeSpectrogram() {
  if(!m_Real) m_Real = dmatrix(m_nFrequencySampleLength, m_nWindowSize);
  if(!m_Imaginary) m_Imaginary
    = dmatrix(m_nFrequencySampleLength, m_nWindowSize);

  double *OrgReal = new double[m_nWindowSize];
  double *OrgImaginary = new double[m_nWindowSize];

  for(int t = 0 ; t < m_nFrequencySampleLength ; t++) {
    int OrgT = t*m_nSampleLength/m_nFrequencySampleLength;
    for(int dt = 0 ; dt < m_nWindowSize ; dt++) {
      int tt = OrgT+dt-m_nWindowSize/2;
      if(tt >= 0 && tt < m_nSampleLength)
        OrgReal[dt] = m_Samples[tt];
      else
        OrgReal[dt] = 0;
      OrgImaginary[dt] = 0;
    }

    FFT2Radix(OrgReal, OrgImaginary, m_Real[t], m_Imaginary[t],
      m_nWindowSize, false);
  }
  delete [] OrgReal;                 delete [] OrgImaginary;
}
```


FIR/IIR

Practice III

- Cepstrum
- Sound generation

Advanced Courses (1)

- FIR filter design
  - Window method
- IIR filter design
  - Bilinear
- Pseudo color

https://www.wavemetrics.com/sites/www.wavemetrics.com/files/images-imported/256-color-versions_3.jpg

- Window types
  - Hamming window

$$w_h(n) = \begin{cases} 0.54 - 0.46\cos\left(\dfrac{2\pi n}{N-1}\right) & 0 \le n \le N-1 \\ 0 & \text{otherwise} \end{cases}$$

Advanced Courses (2)

- Sound features
  - Short time energy

$$E_m = \sum_{n=0}^{N-1} \left| \left[x(n) w(m-n) \right]^2 \right|$$

  - ZCR (zero cross rate)
    - Speech/music classification

$$Z_m = \frac{1}{2N} \sum_{n=0}^{N-1} \left| \operatorname{sign}\left(x(n)\right) - \operatorname{sign}\left(x(n-1)\right) \right| w(m-n)$$

  - MFCC (mel-frequency cepstral coefficients)
    - MFC: representation of the short-term power spectrum and the frequency bands are equally spaced on the mel scale

6. *Image Processing*

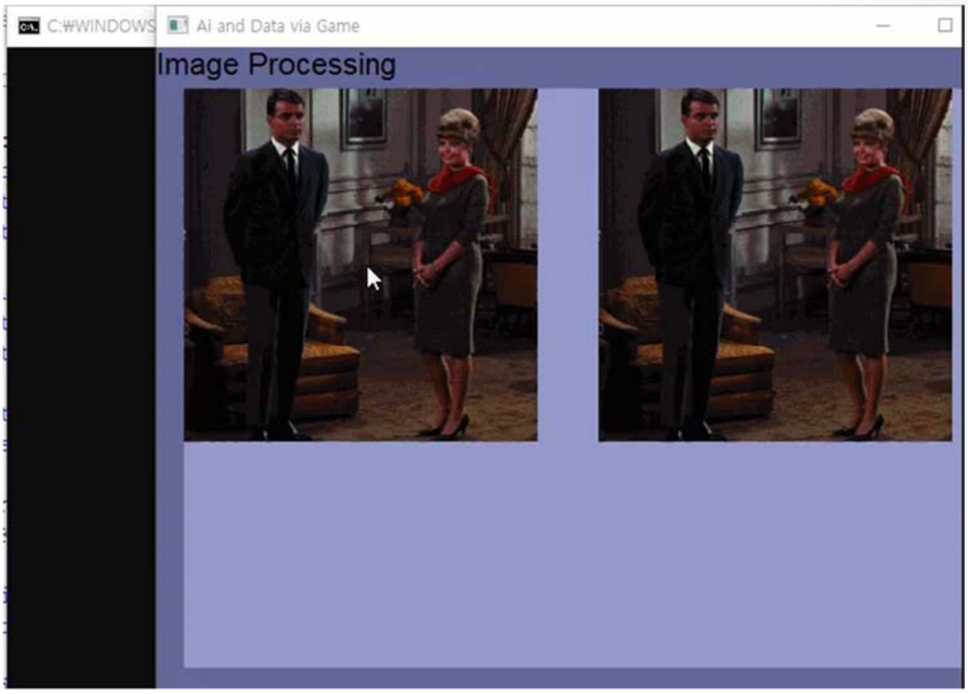


Image Load/Save

```
class CKhuGleSignal {      // KhuGleSignal.h
public:
  …
  int m_nW, m_nH;
  unsigned char **m_Red, **m_Green, **m_Blue;
  …
  void ReadBmp(char *FileName);
  bool SaveBmp(char *FileName);
};
```

Mean Filter

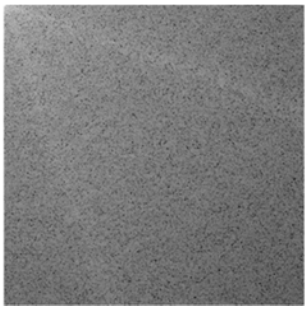
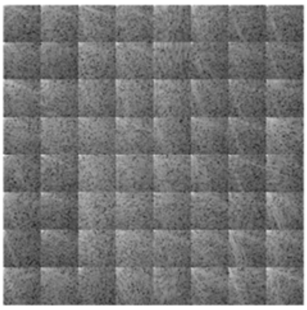
Edge

2D DCT

$$F(u,v) = C(u)C(v)\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,y)\cos\left(\frac{(2x+1)u\pi}{2N}\right)\cos\left(\frac{(2y+1)v\pi}{2N}\right)$$

$$f(x,y) = \sum_{u=0}^{N-1}\sum_{v=0}^{N-1} C(u)C(v)F(u,v)\cos\left(\frac{(2x+1)u\pi}{2N}\right)\cos\left(\frac{(2y+1)v\pi}{2N}\right)$$

$$C(\alpha) = \begin{cases} \sqrt{\frac{1}{N}} & \alpha = 0 \\ \sqrt{\frac{2}{N}} & \alpha = 1, 2, \cdots, N-1 \end{cases}$$


KhuGleBase.cpp (1)

```
void DCT2D(double **Input, double **Output, int nW, int nH, int nBlockSize) {
  int x, y;
  int u, v;
  int BlockX, BlockY;

  for(v = 0 ; v < nH ; v++)
    for(u = 0 ; u < nW ; u++)
      Output[v][u] = 0;

  for(BlockY = 0 ; BlockY < nH-nBlockSize+1 ; BlockY += nBlockSize)
    for(BlockX = 0 ; BlockX < nW-nBlockSize+1 ; BlockX += nBlockSize) {
      for(v = 0 ; v < nBlockSize ; v++)
        for(u = 0 ; u < nBlockSize ; u++) {
          Output[BlockY+v][BlockX+u] = 0;
          for(y = 0 ; y < nBlockSize ; y++)
            for(x = 0 ; x < nBlockSize ; x++) {
              Output[BlockY+v][BlockX+u] +=
                Input[BlockY+y][BlockX+x]
                * cos((2*x+1)*u*Pi/(2.*nBlockSize))
                * cos((2*y+1)*v*Pi/(2.*nBlockSize));
            }
```

KhuGleBase.cpp (2)

```
        if(u == 0)
          Output[BlockY+v][BlockX+u] *= sqrt(1./nBlockSize);
        else
          Output[BlockY+v][BlockX+u] *= sqrt(2./nBlockSize);

        if(v == 0)
          Output[BlockY+v][BlockX+u] *= sqrt(1./nBlockSize);
        else
          Output[BlockY+v][BlockX+u] *= sqrt(2./nBlockSize);
      }
    }
}
```


KhuGleBase.cpp (3)

```
void IDCT2D(double **Input, double **Output, int nW, int nH, int nBlockSize) {
  int x, y;
  int u, v;
  int BlockX, BlockY;

  for(y = 0 ; y < nH ; y++)
    for(x = 0 ; x < nW ; x++)
      Output[y][x] = 0;

  for(BlockY = 0 ; BlockY < nH-nBlockSize+1 ; BlockY += nBlockSize)
    for(BlockX = 0 ; BlockX < nW-nBlockSize+1 ; BlockX += nBlockSize) {
      for(y = 0 ; y < nBlockSize ; y++)
        for(x = 0 ; x < nBlockSize ; x++) {
          Output[BlockY+y][BlockX+x] = 0;
```

KhuGleBase.cpp (4)

```
          for(v = 0 ; v < nBlockSize ; v++)
            for(u = 0 ; u < nBlockSize ; u++) {
              double Cu, Cv;
              if(u == 0) Cu = sqrt(1./nBlockSize);
              else Cu = sqrt(2./nBlockSize);

              if(v == 0) Cv = sqrt(1./nBlockSize);
              else Cv = sqrt(2./nBlockSize);

              Output[BlockY+y][BlockX+x] +=
                Cu*Cv*Input[BlockY+v][BlockX+u]
                * cos((2*x+1)*u*Pi/(2.*nBlockSize))
                * cos((2*y+1)*v*Pi/(2.*nBlockSize));
            }
        }
    }
}
```


MSE/PSNR

- MSE: mean squared error

$$\frac{1}{N}\sum_{i=1}^{N}\left(x_i - \hat{x}_i\right)^2$$

- PSNR: peak signal-to-noise ratio
  - Color: MSE of average mean

$$10\log_{10}\left(\frac{\text{Max}^2}{\text{MSE}}\right)$$

KhuGleBase.cpp

```
double GetMse(unsigned char **I, unsigned char **O, int nW, int nH) {
  double Mse = 0;
  for(int y = 0 ; y < nH ; ++y)
    for(int x = 0 ; x < nW ; ++x)
      Mse = (I[y][x] - O[y][x])*(I[y][x] - O[y][x]);

  Mse /= nW*nH;
  return Mse;
}
double GetPsnr(unsigned char **IR, unsigned char **IG, unsigned char **IB,
  unsigned char **OR, unsigned char **OG, unsigned char **OB, int nW, int nH) {
  double MseR, MseG, MseB, Mse;

  MseR = GetMse(IR, OR, nW, nH);
  MseG = GetMse(IG, OG, nW, nH);
  MseB = GetMse(IB, OB, nW, nH);
  Mse = (MseR + MseG + MseB)/3.;

  if(Mse == 0) return 100.;
  return 10*log10(255*255 / Mse);
}
```



```
…
class CKhuGleImageLayer : public CKhuGleLayer {
public:
  CKhuGleSignal m_Image, m_ImageOut;
  CKhuGleImageLayer(int nW, int nH, KgColor24 bgColor,
    CKgPoint ptPos = CKgPoint(0, 0))
    : CKhuGleLayer(nW, nH, bgColor, ptPos) {}
  void DrawBackgroundImage();
};

void CKhuGleImageLayer::DrawBackgroundImage() {
  for(int y = 0 ; y < m_nH ; y++)
    for(int x = 0 ; x < m_nW ; x++) {
      m_ImageBgR[y][x] = KgGetRed(m_bgColor);
      m_ImageBgG[y][x] = KgGetGreen(m_bgColor);
      m_ImageBgB[y][x] = KgGetBlue(m_bgColor);
    }
```

```
  if(m_Image.m_Red && m_Image.m_Green && m_Image.m_Blue) {
    for(int y = 0 ; y < m_Image.m_nH && y < m_nH ; ++y)
      for(int x = 0 ; x < m_Image.m_nW && x < m_nW ; ++x) {
        m_ImageBgR[y][x] = m_Image.m_Red[y][x];
        m_ImageBgG[y][x] = m_Image.m_Green[y][x];
        m_ImageBgB[y][x] = m_Image.m_Blue[y][x];
      }
  }
  if(m_ImageOut.m_Red && m_ImageOut.m_Green && m_ImageOut.m_Blue) {
    int OffsetX = 300, OffsetY = 0;
    for(int y = 0 ; y < m_ImageOut.m_nH && y + OffsetY < m_nH ; ++y)
      for(int x = 0 ; x < m_ImageOut.m_nW && x + OffsetX < m_nW ; ++x) {
        m_ImageBgR[y + OffsetY][x + OffsetX] = m_ImageOut.m_Red[y][x];
        m_ImageBgG[y + OffsetY][x + OffsetX] = m_ImageOut.m_Green[y][x];
        m_ImageBgB[y + OffsetY][x + OffsetX] = m_ImageOut.m_Blue[y][x];
      }
  }
}
```



```
class CImageProcessing : public CKhuGleWin {
public:
  CKhuGleImageLayer *m_pImageLayer;
  CImageProcessing(int nW, int nH, char *ImagePath);
  void Update();
};
CImageProcessing::CImageProcessing(int nW, int nH, char *ImagePath)
  : CKhuGleWin(nW, nH) {
  m_pScene = new CKhuGleScene(640, 480, KG_COLOR_24_RGB(100, 100, 150));
  m_pImageLayer = new CKhuGleImageLayer(600, 420,
    KG_COLOR_24_RGB(150, 150, 200), CKgPoint(20, 30));
  m_pImageLayer->m_Image.ReadBmp(ImagePath);
  m_pImageLayer->m_ImageOut.ReadBmp(ImagePath);
  m_pImageLayer->DrawBackgroundImage();
  m_pScene->AddChild(m_pImageLayer);
}
void CImageProcessing::Update() {
  if(m_bKeyPressed['D'] || m_bKeyPressed['I'] || m_bKeyPressed['C']
    || m_bKeyPressed['E'] || m_bKeyPressed['M']) {
    bool bInverse = m_bKeyPressed['I'];
    bool bCompression = m_bKeyPressed['C'];
    bool bEdge = m_bKeyPressed['E'];
    bool bMean = m_bKeyPressed['M'];
```

```
double **InputR = dmatrix(m_pImageLayer->m_Image.m_nH,
  m_pImageLayer->m_Image.m_nW);
double **InputG = dmatrix(m_pImageLayer->m_Image.m_nH,
  m_pImageLayer->m_Image.m_nW);
double **InputB = dmatrix(m_pImageLayer->m_Image.m_nH,
  m_pImageLayer->m_Image.m_nW);

double **OutR = dmatrix(m_pImageLayer->m_Image.m_nH,
  m_pImageLayer->m_Image.m_nW);
double **OutG = dmatrix(m_pImageLayer->m_Image.m_nH,
  m_pImageLayer->m_Image.m_nW);
double **OutB = dmatrix(m_pImageLayer->m_Image.m_nH,
  m_pImageLayer->m_Image.m_nW);

for(int y = 0 ; y < m_pImageLayer->m_Image.m_nH ; ++y)
  for(int x = 0 ; x < m_pImageLayer->m_Image.m_nW ; ++x) {
    InputR[y][x] = m_pImageLayer->m_Image.m_Red[y][x];
    InputG[y][x] = m_pImageLayer->m_Image.m_Green[y][x];
    InputB[y][x] = m_pImageLayer->m_Image.m_Blue[y][x];
                  }
```



```
if(bEdge)  {
  for(int y = 0 ; y < m_pImageLayer->m_ImageOut.m_nH ; ++y)
    for(int x = 0 ; x < m_pImageLayer->m_ImageOut.m_nW ; ++x) {
      OutR[y][x] = OutG[y][x] = OutB[y][x] = 0.;
      if(x > 0 && x < m_pImageLayer->m_ImageOut.m_nW-1 &&
        y > 0 && y < m_pImageLayer->m_ImageOut.m_nH-1) {
        double Rx = InputR[y-1][x-1] + 2*InputR[y][x-1] + InputR[y+1][x-1]
          - InputR[y-1][x+1] - 2*InputR[y][x+1] - InputR[y+1][x+1];
        double Ry = InputR[y-1][x-1] + 2*InputR[y-1][x] + InputR[y-1][x+1]
          - InputR[y+1][x-1] - 2*InputR[y+1][x] - InputR[y+1][x+1];
        double Gx = InputG[y-1][x-1] + 2*InputG[y][x-1] + InputG[y+1][x-1]
          - InputG[y-1][x+1] - 2*InputG[y][x+1] - InputG[y+1][x+1];
        double Gy = InputG[y-1][x-1] + 2*InputG[y-1][x] + InputG[y-1][x+1]
          - InputG[y+1][x-1] - 2*InputG[y+1][x] - InputG[y+1][x+1];
        double Bx = InputB[y-1][x-1] + 2*InputB[y][x-1] + InputB[y+1][x-1]
          - InputB[y-1][x+1] - 2*InputB[y][x+1] - InputB[y+1][x+1];
        double By = InputB[y-1][x-1] + 2*InputB[y-1][x] + InputB[y-1][x+1]
          - InputB[y+1][x-1] - 2*InputB[y+1][x] - InputB[y+1][x+1];
        OutR[y][x] = sqrt(Rx*Rx + Ry*Ry);  OutG[y][x] = sqrt(Gx*Gx + Gy*Gy);
        OutB[y][x] = sqrt(Bx*Bx + By*By);
      }
    }
    std::cout << "Edge" << std::endl;
}
```

```
else if(bMean) {
  for(int y = 0 ; y < m_pImageLayer->m_ImageOut.m_nH ; ++y)
    for(int x = 0 ; x < m_pImageLayer->m_ImageOut.m_nW ; ++x) {
      OutR[y][x] = OutG[y][x] = OutB[y][x] = 0.;
      if(x > 0 && x < m_pImageLayer->m_ImageOut.m_nW-1 &&
        y > 0 && y < m_pImageLayer->m_ImageOut.m_nH-1) {
        for(int dy = -1 ; dy < 2 ; ++dy)
          for(int dx = -1 ; dx < 2 ; ++dx) {
            OutR[y][x] += InputR[y+dy][x+dx];   OutG[y][x] += InputG[y+dy][x+dx];
            OutB[y][x] += InputB[y+dy][x+dx];
          }
      }
    }
  std::cout << "Mean filter" << std::endl;
}
else {
  DCT2D(InputR,OutR,m_pImageLayer->m_Image.m_nW,m_pImageLayer->m_Image.m_nH,8);
  DCT2D(InputG,OutG,m_pImageLayer->m_Image.m_nW,m_pImageLayer->m_Image.m_nH,8);
  DCT2D(InputB,OutB,m_pImageLayer->m_Image.m_nW,m_pImageLayer->m_Image.m_nH,8);
  std::cout << "DCT" << std::endl;
}
```



```
if(!bInverse && ! bCompression) {
  double MaxR, MaxG, MaxB, MinR, MinG, MinB;
  for(int y = 0 ; y < m_pImageLayer->m_ImageOut.m_nH ; ++y)
    for(int x = 0 ; x < m_pImageLayer->m_ImageOut.m_nW ; ++x) {
      if(x == 0 && y == 0) {
        MaxR = MinR = OutR[y][x];
        MaxG = MinG = OutG[y][x];
        MaxB = MinB = OutB[y][x];
      }
      else {
        if(OutR[y][x] > MaxR) MaxR = OutR[y][x];
        if(OutG[y][x] > MaxG) MaxG = OutG[y][x];
        if(OutB[y][x] > MaxB) MaxB = OutB[y][x];

        if(OutR[y][x] < MinR) MinR = OutR[y][x];
        if(OutG[y][x] < MinG) MinG = OutG[y][x];
        if(OutB[y][x] < MinB) MinB = OutB[y][x];
      }
    }
```

```
    for(int y = 0 ; y < m_pImageLayer->m_ImageOut.m_nH ; ++y)
      for(int x = 0 ; x < m_pImageLayer->m_ImageOut.m_nW ; ++x) {
        if(MaxR == MinR) m_pImageLayer->m_ImageOut.m_Red[y][x] = 0;
        else m_pImageLayer->m_ImageOut.m_Red[y][x]
               = (int)((OutR[y][x]-MinR)*255/(MaxR-MinR));
        if(MaxG == MinG) m_pImageLayer->m_ImageOut.m_Green[y][x] = 0;
        else m_pImageLayer->m_ImageOut.m_Green[y][x]
               = (int)((OutG[y][x]-MinG)*255/(MaxG-MinG));
        if(MaxB == MinB) m_pImageLayer->m_ImageOut.m_Blue[y][x] = 0;
        else m_pImageLayer->m_ImageOut.m_Blue[y][x]
               = (int)((OutB[y][x]-MinB)*255/(MaxB-MinB));
      }
  }
```



```
  else {
    if(bCompression) {
      for(int y = 0 ; y < m_pImageLayer->m_ImageOut.m_nH ; ++y)
        for(int x = 0 ; x < m_pImageLayer->m_ImageOut.m_nW ; ++x) {
          if(x%8 > 3 || y %8 > 3) {
            OutR[y][x] = 0; OutG[y][x] = 0; OutB[y][x] = 0;
          }
        }
      std::cout << "Compression" << std::endl;
    }
    else
      std::cout << "Non compression" << std::endl;

    IDCT2D(OutR, InputR, m_pImageLayer->m_Image.m_nW,
      m_pImageLayer->m_Image.m_nH, 8);
    IDCT2D(OutG, InputG, m_pImageLayer->m_Image.m_nW,
      m_pImageLayer->m_Image.m_nH, 8);
    IDCT2D(OutB, InputB, m_pImageLayer->m_Image.m_nW,
      m_pImageLayer->m_Image.m_nH, 8);
```

```
double MaxR, MaxG, MaxB, MinR, MinG, MinB;
for(int y = 0 ; y < m_pImageLayer->m_ImageOut.m_nH ; ++y)
  for(int x = 0 ; x < m_pImageLayer->m_ImageOut.m_nW ; ++x) {
    if(x == 0 && y == 0) {
      MaxR = MinR = InputR[y][x];
      MaxG = MinG = InputG[y][x];
      MaxB = MinB = InputB[y][x];
    }
    else {
      if(InputR[y][x] > MaxR) MaxR = InputR[y][x];
      if(InputG[y][x] > MaxG) MaxG = InputG[y][x];
      if(InputB[y][x] > MaxB) MaxB = InputB[y][x];
      if(InputR[y][x] < MinR) MinR = InputR[y][x];
      if(InputG[y][x] < MinG) MinG = InputG[y][x];
      if(InputB[y][x] < MinB) MinB = InputB[y][x];
    }
  }
```



```
for(int y = 0 ; y < m_pImageLayer->m_ImageOut.m_nH ; ++y)
  for(int x = 0 ; x < m_pImageLayer->m_ImageOut.m_nW ; ++x) {
    if(MaxR == MinR) m_pImageLayer->m_ImageOut.m_Red[y][x] = 0;
    else m_pImageLayer->m_ImageOut.m_Red[y][x]
      = (int)((InputR[y][x]-MinR)*255/(MaxR-MinR));
    if(MaxG == MinG) m_pImageLayer->m_ImageOut.m_Green[y][x] = 0;
    else m_pImageLayer->m_ImageOut.m_Green[y][x]
      = (int)((InputG[y][x]-MinG)*255/(MaxG-MinG));
    if(MaxB == MinB) m_pImageLayer->m_ImageOut.m_Blue[y][x] = 0;
    else m_pImageLayer->m_ImageOut.m_Blue[y][x]
      = (int)((InputB[y][x]-MinB)*255/(MaxB-MinB));
  }
}
```

```
    if(bMean || bCompression || bInverse) {
      double Psnr = GetPsnr(m_pImageLayer->m_Image.m_Red,
        m_pImageLayer->m_Image.m_Green, m_pImageLayer->m_Image.m_Blue,
        m_pImageLayer->m_ImageOut.m_Red, m_pImageLayer->m_ImageOut.m_Green,
        m_pImageLayer->m_ImageOut.m_Blue,
        m_pImageLayer->m_Image.m_nW, m_pImageLayer->m_Image.m_nH);
      std::cout << Psnr << std::endl;
    }

    free_dmatrix(InputR, m_pImageLayer->m_Image.m_nH,
      m_pImageLayer->m_Image.m_nW);
    free_dmatrix(InputG, m_pImageLayer->m_Image.m_nH,
      m_pImageLayer->m_Image.m_nW);
    free_dmatrix(InputB, m_pImageLayer->m_Image.m_nH,
      m_pImageLayer->m_Image.m_nW);
    free_dmatrix(OutR, m_pImageLayer->m_Image.m_nH,
      m_pImageLayer->m_Image.m_nW);
    free_dmatrix(OutG, m_pImageLayer->m_Image.m_nH,
      m_pImageLayer->m_Image.m_nW);
    free_dmatrix(OutB, m_pImageLayer->m_Image.m_nH,
      m_pImageLayer->m_Image.m_nW);
```



```
    m_pImageLayer->DrawBackgroundImage();
    m_bKeyPressed['D'] = m_bKeyPressed['I'] = m_bKeyPressed['C']
      = m_bKeyPressed['E'] = m_bKeyPressed['M'] = false;
  }

  m_pScene->Render();

  DrawSceneTextPos("Image Processing", CKgPoint(0, 0));

  CKhuGleWin::Update();
}
```

```
int main() {
  char ExePath[MAX_PATH], ImagePath[MAX_PATH];
  GetModuleFileName(NULL, ExePath, MAX_PATH);
  int i;
  int LastBackSlash = -1;
  int nLen = strlen(ExePath);
  for(i = nLen-1 ; i >= 0 ; i--) {
    if(ExePath[i] == '\\') {
      LastBackSlash = i;
      break;
    }
  }
  if(LastBackSlash >= 0) ExePath[LastBackSlash] = '\0';
  sprintf(ImagePath, "%s\\%s", ExePath, "couple.bmp");

  CImageProcessing *pImageProcessing
    = new CImageProcessing(640, 480, ImagePath);
  KhuGleWinInit(pImageProcessing);
  return 0;
}
```


Practice IV (1)

- Quantization

$$\hat{c}(x,y) = ROUND\left(\frac{c(x,y)}{q(x,y)}\right)$$

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	66
14	13	16	24	40	57	69	57
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	36	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

Practice IV (2)

- Interpolation
- Anti-aliasing

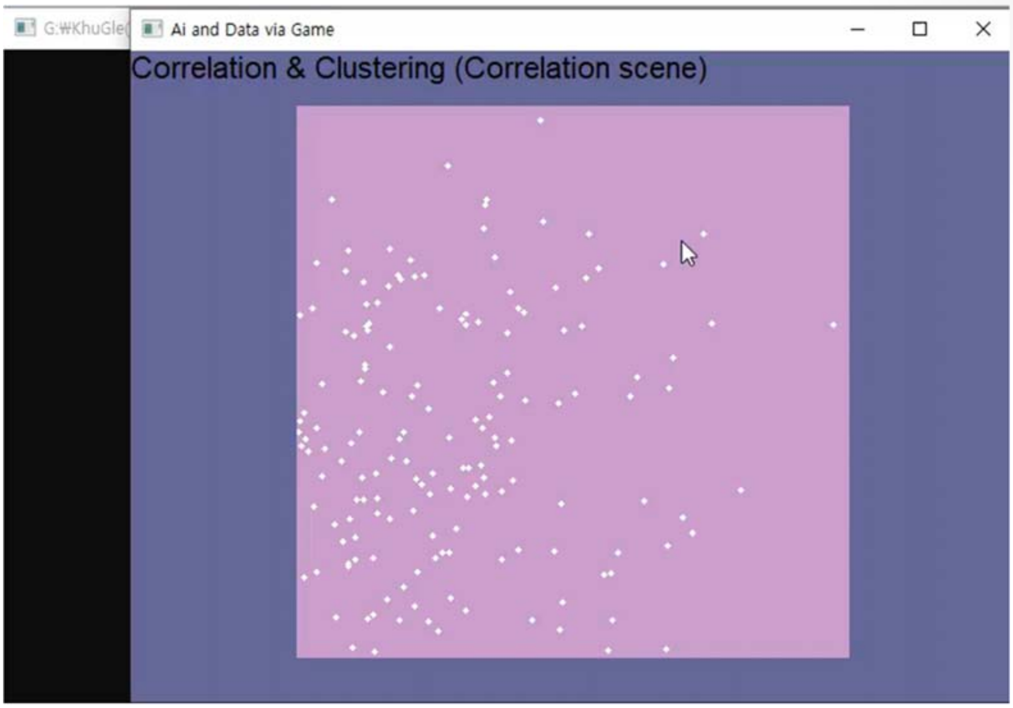
Advanced Courses

- Png
- Animation

- Variable length code(VLC, VLE)

- RMSE: root mean squared error
- MAE: mean absolute error
- MAPE: mean absolute percentage error
- SSIM: structural similarity index measure

7. Correlation & Clustering



Pearson Correlation Coefficient

- Pearson correlation coefficient (PCC)
  - Pearson product-moment correlation coefficient

$$\rho_{x,y} = \frac{\operatorname{cov}(X,Y)}{\sigma_x \sigma_y} = \frac{\mathrm{E}\left[(X-\mu_x)(Y-\mu_y)\right]}{\sigma_x \sigma_y}$$

$$= \frac{\mathrm{E}[XY]-\mathrm{E}[X]\mathrm{E}[Y]}{\sqrt{\mathrm{E}[X^2]-(\mathrm{E}[X])^2}\sqrt{\mathrm{E}[Y^2]-(\mathrm{E}[Y])^2}}$$

Pseudo Random Number

```
int rand();       // pseudo random number, [0, RAND)MAX]
void srand();     // random seed initialization
srand(time(0));


// <random>
unsigned int seed = (unsigned int)std::chrono::
         system_clock::now().time_since_epoch().count();
std::default_random_engine generator(seed);

std::uniform_real_distribution<double> uniform_dist(0, 1);
std::normal_distribution<double> normal_dist(0, 1);

double number1 = uniform_dist(generator);
double number2 = normal_dist(generator);
```

KhuGleBase.cpp (1)

```
double GetPearsonCoefficient(std::vector<std::pair<double, double>> Data) {
  double Mean1 = 0, Mean2 = 0, Mean12 = 0;
  double SquaredMean1 = 0, SquaredMean2 = 0;

  for(auto EachData : Data) {
    Mean1 += EachData.first;
    Mean2 += EachData.second;
    Mean12 += EachData.first*EachData.second;

    SquaredMean1 += EachData.first*EachData.first;
    SquaredMean2 += EachData.second*EachData.second;
  }
```


KhuGleBase.cpp (2)

```
  Mean1 /= Data.size();
  Mean2 /= Data.size();
  Mean12 /= Data.size();

  SquaredMean1 /= Data.size();
  SquaredMean2 /= Data.size();

  double sigma1 = sqrt(SquaredMean1-Mean1*Mean1);
  double sigma2 = sqrt(SquaredMean2-Mean2*Mean2);

  if(sigma1 == 0 || sigma2 == 0) return 0;

  return (Mean12 - Mean1*Mean2)/(sigma1*sigma2);
}
```

Main.cpp (1)

```
…
class CCorrelationLayer : public CKhuGleLayer {
public:
  std::vector<CKhuGleSprite *> m_Point;

  CCorrelationLayer(int nW, int nH, KgColor24 bgColor,
    CKgPoint ptPos = CKgPoint(0, 0)): CKhuGleLayer(nW, nH, bgColor, ptPos) {
    GenerateData(200);
  }
  void GenerateData(int nCnt);
};
void CCorrelationLayer::GenerateData(int nCnt) {
  unsigned int seed = (unsigned int)std::chrono::
                       system_clock::now().time_since_epoch().count();
  std::default_random_engine generator(seed);
  std::uniform_real_distribution<double> uniform_dist(0, 1);

  for(auto &Child : m_Children)
    delete Child;
  m_Children.clear();
  m_Point.clear();
```


Main.cpp (2)

```
  double mean1 = uniform_dist(generator);
  double mean2 = uniform_dist(generator);

  double sigma1 = uniform_dist(generator)/2.;
  double sigma2 = uniform_dist(generator)/2.;

  double rotate = uniform_dist(generator)*Pi;

  std::normal_distribution<double> normal_dist1(mean1, sigma1);
  std::normal_distribution<double> normal_dist2(mean2, sigma2);
```

Main.cpp (3)

```
  double x, y;
  for(int i = 0 ; i < nCnt ; i++) {
    double xx = normal_dist1(generator);
    double yy = normal_dist2(generator);

    x = (xx-mean1)*cos(rotate) - (yy-mean2)*sin(rotate) + mean1;
    y = (xx-mean1)*sin(rotate) + (yy-mean2)*cos(rotate) + mean2;

    x = (x*m_nW - m_nW/2)*0.6 + m_nW/2;
    y = (y*m_nH - m_nH/2)*0.6 + m_nH/2;

    CKhuGleSprite *Point = new CKhuGleSprite(GP_STYPE_ELLIPSE, GP_CTYPE_DYNAMIC,
      CKgLine(CKgPoint((int)x-2, (int)y-2), CKgPoint((int)x+2, (int)y+2)),
      KG_COLOR_24_RGB(255, 255, 255), true, 30);

    m_Point.push_back(Point);
    AddChild(Point);
  }
}
```


Main.cpp (4)

```
class CClusterLayer : public CKhuGleLayer {
  …
};

class CCorrelationClustering : public CKhuGleWin {
public:
  CKhuGleScene *m_pCorrelationScene;
  CKhuGleScene *m_pClusteringScene;

  CCorrelationLayer *m_pCorrelationLayer;
  CClusterLayer *m_pClusteringLayer;

  bool m_bCorrelationScene;

  CCorrelationClustering(int nW, int nH);
  virtual ~CCorrelationClustering() {
    m_pScene = nullptr;
    delete m_pCorrelationScene;
    delete m_pClusteringScene;
  }
  void Update();
};
```

Main.cpp (5)

```
CCorrelationClustering::CCorrelationClustering(int nW, int nH)
  : CKhuGleWin(nW, nH) {
  m_pCorrelationScene = new CKhuGleScene(640, 480,
    KG_COLOR_24_RGB(100, 100, 150));
  m_pClusteringScene = new CKhuGleScene(640, 480,
    KG_COLOR_24_RGB(100, 100, 150));

  m_pCorrelationLayer = new CCorrelationLayer(400, 400,
    KG_COLOR_24_RGB(200, 150, 200), CKgPoint(120, 40));
  m_pCorrelationScene->AddChild(m_pCorrelationLayer);

  m_pClusteringLayer = new CClusterLayer(400, 400,
    KG_COLOR_24_RGB(150, 150, 200), CKgPoint(120, 40));
  m_pClusteringScene->AddChild(m_pClusteringLayer);

  m_pScene = m_pCorrelationScene;
  m_bCorrelationScene = true;
}
```


Main.cpp (6)

```
void CCorrelationClustering::Update() {
  if(m_bKeyPressed['M']) {
    m_bCorrelationScene = !m_bCorrelationScene;
    if(m_bCorrelationScene)
      m_pScene = m_pCorrelationScene;
    else
      m_pScene = m_pClusteringScene;
    m_bKeyPressed['M'] = false;
  }
  if(m_bKeyPressed['S']) {
    if(m_bCorrelationScene) {
      std::vector<std::pair<double, double>> Data;
      for(auto Point : m_pCorrelationLayer->m_Point)
        Data.push_back({Point->m_Center.x, Point->m_Center.y});

      double pcc = GetPearsonCoefficient(Data);
      std::cout << pcc << std::endl;
    }
    else {
    …
    }
    m_bKeyPressed['S'] = false;
  }
```

Main.cpp (7)

```
  if(m_bKeyPressed['N']) {
    if(m_bCorrelationScene)
      m_pCorrelationLayer->GenerateData(200);
    else {
    …
    }
    m_bKeyPressed['N'] = false;
  }
  m_pScene->Render();
  if(m_bCorrelationScene)
    DrawSceneTextPos("Correlation && Clustering (Correlation scene)", CKgPoint(0, 0));
  else
    DrawSceneTextPos("Correlation && Clustering (Clustering scene)", CKgPoint(0, 0));
  CKhuGleWin::Update();
}
int main() {
  CCorrelationClustering *pCorrelationClustering
    = new CCorrelationClustering(640, 480);
  KhuGleWinInit(pCorrelationClustering);
  return 0;
}
```


k-Means Clustering (1)

- k-Means Clustering
  - Centroid-based method
  - Iterative refinement method

  - $\mathbf{c}_0=\{c_1, c_2, \ldots c_k\} \leftarrow$ random
    while iteration or **c** is not change ($\mathbf{c}_i = \mathbf{c}_{i-1}$) **do**
    assign each sample to the cluster which has the closest **c**
    compute new centroids ($\mathbf{c}_i$) for each cluster (sample mean)
    end while

k-Means Clustering (2)

- Static k value
- Dependent results on initial centroids
- Spherical and equally sized clustering

Main.cpp (1)

```
…
class CKhuGleSprite2 : public CKhuGleSprite {
public:
  int m_nClusterIndex;
  CKhuGleSprite2(int nType, int nCollisionType, CKgLine lnLine,
    KgColor24 fgColor, bool bFill, int nSliceOrWidth = 100,
    int nClusterIndex = 0)
  : CKhuGleSprite(nType, nCollisionType, lnLine, fgColor, bFill, nSliceOrWidth)
  {
    m_nClusterIndex = nClusterIndex;
  }
};
```

Main.cpp (2)

```
class CClusterLayer : public CKhuGleLayer {
public:
  std::vector<CKhuGleSprite2 *> m_Center;
  std::vector<CKhuGleSprite2 *> m_Point;
  int m_nClusterNum, m_nStep;

  CClusterLayer(int nW, int nH, KgColor24 bgColor,
    CKgPoint ptPos = CKgPoint(0, 0)) : CKhuGleLayer(nW, nH, bgColor, ptPos) {
    m_nClusterNum = 3;

    GenerateData(m_nClusterNum, 50);
    m_nStep = 0;
  }

  void GenerateData(int nCluster, int nCnt);
};
```


Main.cpp (3)

```
void CClusterLayer::GenerateData(int nCluster, int nCnt) {
  unsigned int seed = (unsigned int)std::chrono
    ::system_clock::now().time_since_epoch().count();
  std::default_random_engine generator(seed);

  std::uniform_real_distribution<double> uniform_dist(0, 1);

  for(auto &Child : m_Children)
    delete Child;

  m_Children.clear();
  m_Center.clear();
  m_Point.clear();

  for(int i = 0 ; i < m_nClusterNum ; ++i) {
    CKhuGleSprite2 *Center = new CKhuGleSprite2(GP_STYPE_ELLIPSE,
      GP_CTYPE_DYNAMIC, CKgLine(CKgPoint(m_nW/2-10, m_nH/2-10),
      CKgPoint(m_nW/2+10, m_nH/2+10)),
      KG_COLOR_24_RGB(i%2*255, i/2%2*255, i/4%2*255), false, 100);

    m_Center.push_back(Center);
    AddChild(Center);
  }
```

Main.cpp (4)

```
  for(int k = 0 ; k < nCluster ; ++k) {
    double mean1 = uniform_dist(generator);
    double mean2 = uniform_dist(generator);

    double sigma1 = uniform_dist(generator)/10.;
    double sigma2 = uniform_dist(generator)/10.;

    double rotate = uniform_dist(generator)*Pi;

    std::normal_distribution<double> normal_dist1(mean1, sigma1);
    std::normal_distribution<double> normal_dist2(mean2, sigma2);
```


Main.cpp (5)

```
    double x, y;
    for(int i = 0 ; i < nCnt ; i++) {
      double xx = normal_dist1(generator);
      double yy = normal_dist2(generator);

      x = (xx-mean1)*cos(rotate) - (yy-mean2)*sin(rotate) + mean1;
      y = (xx-mean1)*sin(rotate) + (yy-mean2)*cos(rotate) + mean2;

      x = (x*m_nW - m_nW/2)*0.6 + m_nW/2;
      y = (y*m_nH - m_nH/2)*0.6 + m_nH/2;

      CKhuGleSprite2 *Point = new CKhuGleSprite2(GP_STYPE_ELLIPSE,
        GP_CTYPE_DYNAMIC,
        CKgLine(CKgPoint((int)x-2, (int)y-2), CKgPoint((int)x+2, (int)y+2)),
        KG_COLOR_24_RGB(255, 255, 255), true, 30);

      m_Point.push_back(Point);
      AddChild(Point);
    }
  }
}
```

Main.cpp (6)

```
…
void CCorrelationClustering::Update() {
  …
  if(m_bKeyPressed['S']) {
    if(m_bCorrelationScene) {
      …
    }
  else {
    if(m_pClusteringLayer->m_nStep == 0) {
      for(auto &Center : m_pClusteringLayer->m_Center)
        Center->MoveTo((double)rand()/RAND_MAX*m_pClusteringLayer->m_nW,
          (double)rand()/RAND_MAX*m_pClusteringLayer->m_nH);
    }
    else {
      std::vector<int> ClusterCnt;
      std::vector<std::pair<double, double>> NewCenter;

      for(auto &Center : m_pClusteringLayer->m_Center) {
        NewCenter.push_back({0., 0.});
        ClusterCnt.push_back(0);
      }
```


Main.cpp (7)

```
      for(auto &Point : m_pClusteringLayer->m_Point) {
        int Index = Point->m_nClusterIndex;

        NewCenter[Index].first += Point->m_Center.x;
        NewCenter[Index].second += Point->m_Center.y;

        ClusterCnt[Index]++;
      }
      for(int k = 0 ; k < m_pClusteringLayer->m_nClusterNum ; ++k) {
        if(ClusterCnt[k] > 0)
          m_pClusteringLayer->m_Center[k]->MoveTo
            (NewCenter[k].first/ClusterCnt[k],
             NewCenter[k].second/ClusterCnt[k]);
      }
    }
```

Main.cpp (8)

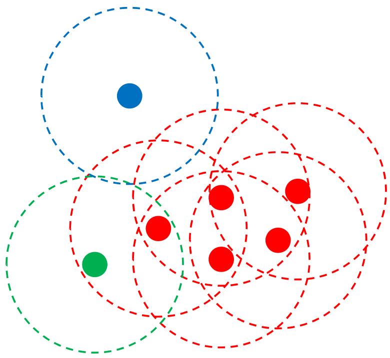
```
    for(auto &Point : m_pClusteringLayer->m_Point) {
      double MinDist, Dist;
      for(int k = 0 ; k < m_pClusteringLayer->m_nClusterNum ; ++k) {
        Dist = sqrt((Point->m_Center.x
            - m_pClusteringLayer->m_Center[k]->m_Center.x)
           *(Point->m_Center.x - m_pClusteringLayer->m_Center[k]->m_Center.x) +
           (Point->m_Center.y - m_pClusteringLayer->m_Center[k]->m_Center.y)
           *(Point->m_Center.y - m_pClusteringLayer->m_Center[k]->m_Center.y));

        if(k == 0) {
          Point->m_nClusterIndex = k;
          MinDist = Dist;
        }
        else if(Dist < MinDist) {
          Point->m_nClusterIndex = k;
          MinDist = Dist;
        }
      }
      Point->m_fgColor = KG_COLOR_24_RGB(Point->m_nClusterIndex%2*255,
        Point->m_nClusterIndex/2%2*255, Point->m_nClusterIndex/4%2*255);
    }
    ++(m_pClusteringLayer->m_nStep);
    }
    m_bKeyPressed['S'] = false;
  }
```


Main.cpp (9)

```
  if(m_bKeyPressed['N']) {
    if(m_bCorrelationScene)
      m_pCorrelationLayer->GenerateData(200);
    else {
      m_pClusteringLayer->GenerateData(m_pClusteringLayer->m_nClusterNum, 50);
      m_pClusteringLayer->m_nStep = 0;
    }
    m_bKeyPressed['N'] = false;
  }

  m_pScene->Render();
  …
}
int main() {
  …
}
```

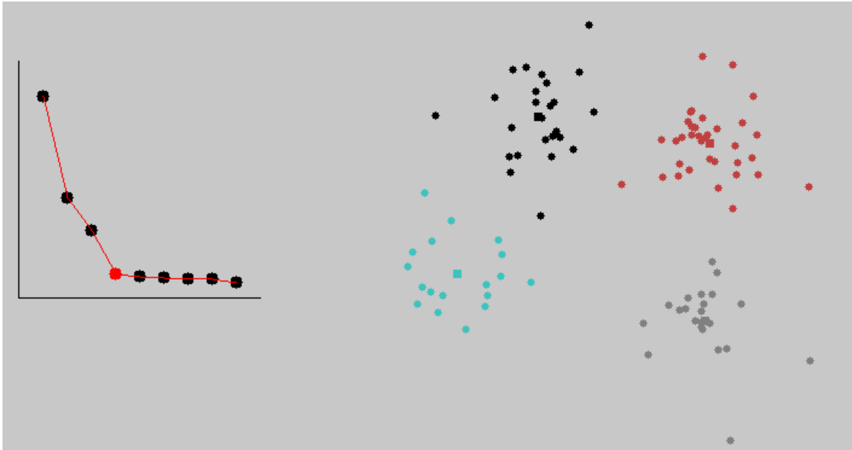
Practice VI (1)

- DBSCAN
  - Density-based spatial clustering of application with noise
  - Density-based clustering
  - Core points: at least τ points with distance ε
  - Border points: reachable from a core points
  - Outliers: not core points and not reachable form any core points



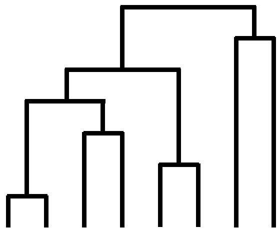
Practice VI (2)

- Elbow method
  - Determining the number of clusters
  - # of clusters vs. sum of squared distance
  - SSD (sum of squared distance, sse: sum of squared error)
    - Sum of squared distance from the cluster cenroid



Advanced Courses (1)

- Connectivity-based clustering
  - Merge for split



- Clustering evaluation
  - Known class labels
    - Precision
    - RI (rand index)
  - Unknown class labels
    - Sum of squared distance
    - Silhouette value

Advanced Courses (2)

- Cross correlation
  - Similarity, matching score
  - Dot product

$$f(t) \otimes g(t) \triangleq \int_{-\infty}^{\infty} f^*(\tau) g(\tau + t) d\tau$$

$$f(t) * g(t) \triangleq \int_{-\infty}^{\infty} f(\tau) g(t - \tau) d\tau$$

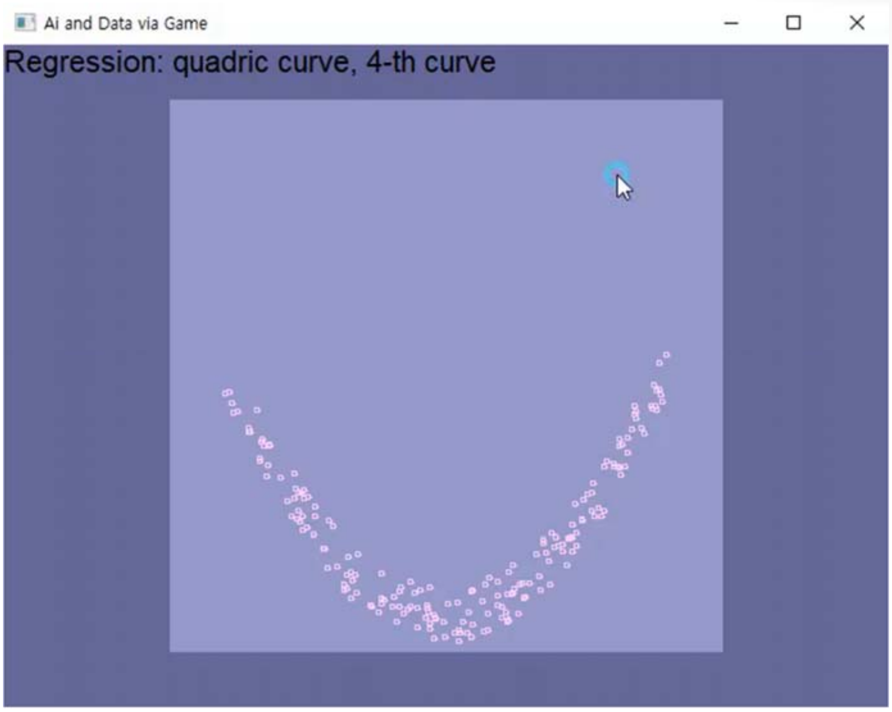
$$\rho_{x,y} = \frac{\mathrm{cov}(X,Y)}{\sigma_x \sigma_y} = \frac{\mathrm{E}\left[(X - \mu_x)(Y - \mu_y) \right]}{\sigma_x \sigma_y} = \frac{1}{N} \frac{(X - \mu_x)(Y - \mu_y)}{\sigma_x \sigma_y}$$

- Cosine similarity

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \|\mathbf{b}\| \cos\theta$$

$$\cos\theta = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \|\mathbf{b}\|}$$

8. Regression



Regression (1)

- Regression
  - Modeling the relationship between a dependent variable and one or more independent variables

    $y = ax + b + \varepsilon$

    $y = ax_1 + bx_2 + c + \varepsilon$
    - ε: residual (error)

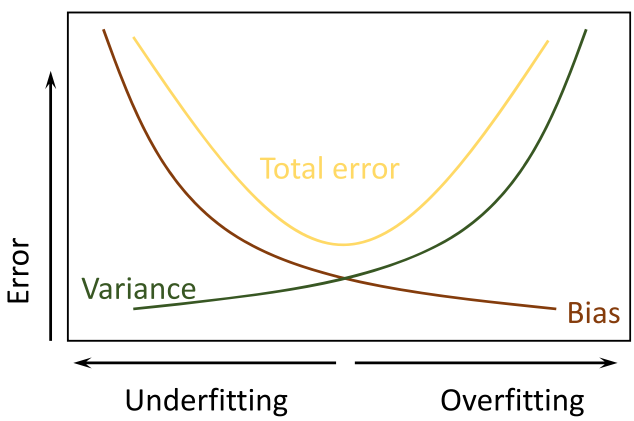
  - Linear regression, multiple linear regression, nonlinear regression

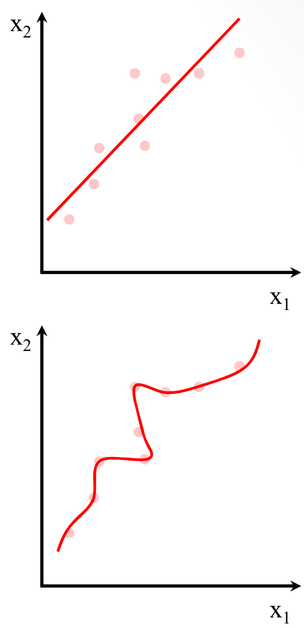
    $y = ax^3 + bx^2 + cx + d$

    $y = ax_1 + bx_2 + cx_3 + d$

Regression (2)

- Bias-variance trade off





Regression (3)

- Least squares regression

$$y_1 = ax_1 + b$$
$$y_2 = ax_2 + b$$
$$y_3 = ax_3 + b$$
$$\vdots$$

SSE (Sum of squared errors)

$$\mathbf{Xw} = \hat{\mathbf{y}}$$

$$\text{SSE} = Q = \sum\left(y_i - \hat{y}_i\right)^2 = \sum\left(y_i - ax_i + b\right)^2 = \sum\left(y_i - w_1 x_i + w_0\right)^2$$

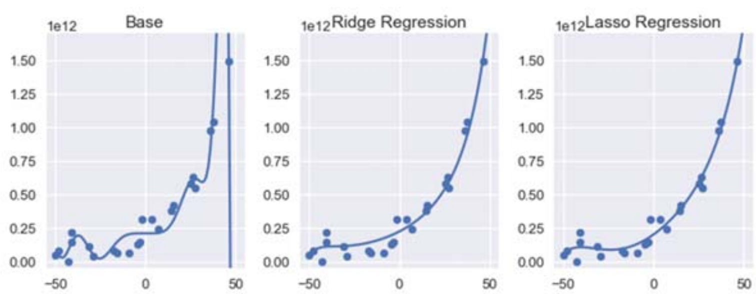
$$\frac{\partial Q}{\partial \mathbf{w}} = \frac{\partial}{\partial \mathbf{w}}\left(\mathbf{y} - \mathbf{Xw}\right)^2 = \frac{\partial}{\partial \mathbf{w}}\left(\mathbf{y}^T\mathbf{y} - 2\mathbf{w}^T\mathbf{X}^T\mathbf{y} + \mathbf{w}^T\mathbf{X}^T\mathbf{Xw}\right) = -2\mathbf{X}^T\mathbf{y} + 2\mathbf{X}^T\mathbf{Xw}$$

$$\frac{\partial Q}{\partial \mathbf{w}} \to 0$$
$$-2\mathbf{X}^T\mathbf{y} + 2\mathbf{X}^T\mathbf{Xw}$$
$$\mathbf{X}^T\mathbf{y} = \mathbf{X}^T\mathbf{Xw},\ \mathbf{w} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{y}$$

Regression (4)

- Ridge regression
  - Error + L2 regularization

$$y = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + \lambda \sum_{i=0}^{n} w_i^2$$



$$\frac{\partial Q}{\partial \mathbf{w}} = \frac{\partial}{\partial \mathbf{w}}\left(\left(\mathbf{y} - \mathbf{Xw}\right)^2 + \lambda\left\|\mathbf{w}\right\|_2^2\right) \to 0$$

$$\mathbf{w} = \left(\mathbf{X}^T\mathbf{X} + \mathbf{\Gamma}^T\mathbf{\Gamma}\right)^{-1}\mathbf{X}^T\mathbf{y},\ \mathbf{\Gamma} = \alpha\mathbf{I}$$

https://www.textbook.ds100.org/ch/16/reg_lasso.html

- Lasso regression
  - Least absolute shrinkage and selection operator
  - Error + L1 regularization

$$+\lambda \sum_{i=0}^{n} |w_i|$$

$$\lambda \frac{\partial}{\partial \mathbf{w}}\left(\left\|\mathbf{w}\right\|_1\right) = \lambda \sum \frac{\partial}{\partial w_i}\left(|w_i|\right)$$

$$\lambda \frac{\partial}{\partial w_i}\left(|w_i|\right) = \begin{cases} -\lambda & w_i < 0 \\ [-\lambda, \lambda] & w_i = 0 \\ \lambda & w_i > 0 \end{cases}$$

Least Squares (1)

```
bool LeastSquared(double **X, double *w, double *y, int nRow, int nCol,
  bool bRidge, double alpha) {
  double **Xt = dmatrix(nCol, nRow);
  double **XtX = dmatrix(nCol, nCol);
  double **InverseXtX = dmatrix(nCol, nCol);
  double **PseudoInverseX = dmatrix(nCol, nRow);

  for(int r = 0 ; r < nCol ; ++r)
    for(int c = 0 ; c < nRow ; ++c)
      Xt[r][c] = X[c][r];

  for(int r = 0 ; r < nCol ; ++r)
    for(int c = 0 ; c < nCol ; ++c) {
      XtX[r][c] = 0;
      for(int k = 0 ; k < nRow ; ++k)
        XtX[r][c] += Xt[r][k] * X[k][c];

      if(bRidge)
        if(r == c) XtX[r][c] += alpha*alpha;
    }
```


Least Squares (2)

```
  if(InverseMatrix(XtX, InverseXtX, nCol)) {
    for(int r = 0 ; r < nCol ; ++r)
      for(int c = 0 ; c <  nRow ; ++c) {
        PseudoInverseX[r][c] = 0;
        for(int k = 0 ; k < nCol ; ++k)
          PseudoInverseX[r][c] += InverseXtX[r][k] *Xt[k][c];
      }
    for(int r = 0 ; r < nCol ; ++r) {
      w[r] = 0;
      for(int k = 0 ; k < nRow ; ++k)
        w[r] += PseudoInverseX[r][k] * y[k];
    }
  }
  else {
    free_dmatrix(Xt, nCol, nRow);
    free_dmatrix(XtX, nCol, nCol);
    free_dmatrix(InverseXtX, nCol, nCol);
    free_dmatrix(PseudoInverseX, nCol, nRow);

    return false;
  }
```

Least Squares (3)

```
  free_dmatrix(Xt, nCol, nRow);
  free_dmatrix(XtX, nCol, nCol);
  free_dmatrix(InverseXtX, nCol, nCol);
  free_dmatrix(PseudoInverseX, nCol, nRow);
  return true;
}
```


Main.cpp (1)

```
…

class CLsmLayer : public CKhuGleLayer {
public:
  std::vector<CKhuGleSprite *> m_Point;
  bool m_bQuadricCurve;
  int m_nPointCnt;
  double **m_X, *m_y, *m_w;

  CLsmLayer(int nW, int nH, KgColor24 bgColor, CKgPoint ptPos = CKgPoint(0, 0),
    int nPointCnt = 100) : CKhuGleLayer(nW, nH, bgColor, ptPos) {
    m_X = nullptr;
    m_y = nullptr;
    m_w = nullptr;

    m_bQuadricCurve = true;

    GenerateData(nPointCnt, false);
  }
  virtual ~CLsmLayer() {
    if(m_X) free_dmatrix(m_X, m_nPointCnt, 3);
    if(m_y) delete [] m_y;
    if(m_w) delete [] m_w;
  }
  void GenerateData(int nCnt, bool bExtremeNoise);
};
```

Main.cpp (2)

```
void CLsmLayer::GenerateData(int nCnt, bool bExtremeNoise) {
  if(m_X) free_dmatrix(m_X, m_nPointCnt, 3);
  if(m_y) delete [] m_y;
  if(m_w) delete [] m_w;

  m_nPointCnt = nCnt;
  m_X = dmatrix(m_nPointCnt, 3);
  m_y = new double[m_nPointCnt];
  m_w = new double[3];

  unsigned int seed
    = (unsigned  int)std::chrono::system_clock
      ::now().time_since_epoch().count();
  std::default_random_engine generator(seed);
```


Main.cpp (3)

```
  std::uniform_real_distribution<double> uniform_dist1(0.005, 0.01);
  std::uniform_real_distribution<double> uniform_dist2(m_nW*0.4, m_nW*0.6);
  std::uniform_real_distribution<double> uniform_dist3(m_nH*0.9, m_nH*0.95);
  std::uniform_real_distribution<double> uniform_dist4(m_nW*0.1, m_nW*0.9);
  std::uniform_real_distribution<double> uniform_dist5(0, m_nW*0.1);
  double a = -uniform_dist1(generator);
  double x0 = uniform_dist2(generator);
  double y0 = uniform_dist3(generator);
  double ExtremeNoisePos = uniform_dist4(generator);
```

Main.cpp (4)

```
  for(auto &Child : m_Children)
    delete Child;
  m_Children.clear();
  m_Point.clear();

  double x, y, noise;
  double m = (rand()%2?1:-1)*a*100;
  for(int i = 0 ; i < m_nPointCnt ; ++i) {
    noise = uniform_dist5(generator)-m_nW*0.05;
    x = uniform_dist4(generator);

    if(m_bQuadricCurve)    y = a*(x-x0)*(x-x0) + y0 + noise;
    else    y = m*(x-x0) + y0 + noise;

    if(bExtremeNoise) {
      if(x > ExtremeNoisePos-m_nW*0.05 && x < ExtremeNoisePos+m_nW*0.05) {
        if(m_bQuadricCurve)
          y = a*(x-x0)*(x-x0) + y0 + (noise-m_nW*0.05)*3;
        else
          y = m*(x-x0) + y0 + (noise-m_nW*0.05)*3;
    }
  }
```


Main.cpp (5)

```
  m_X[i][0] = x*x;
  m_X[i][1] = x;
  m_X[i][2] = 1;

  m_y[i] = y;

  CKhuGleSprite *Point = new CKhuGleSprite(GP_STYPE_ELLIPSE, GP_CTYPE_DYNAMIC,
  CKgLine(CKgPoint((int)x-2, (int)y-2), CKgPoint((int)x+2, (int)y+2)),
  KG_COLOR_24_RGB(255, 200, 255), false, 30);

  m_Point.push_back(Point);
  AddChild(Point);
  }
  SetBackgroundImage(m_nW, m_nH, m_bgColor);
}

class CRegression : public CKhuGleWin {
public:
  CLsmLayer *m_pLsmLayer;

  CRegression(int nW, int nH);
  void Update();
};
```

Main.cpp (6)

```
CRegression::CRegression(int nW, int nH) : CKhuGleWin(nW, nH) {
  m_pScene = new CKhuGleScene(640, 480, KG_COLOR_24_RGB(100, 100, 150));
  m_pLsmLayer = new CLsmLayer(400, 400, KG_COLOR_24_RGB(150, 150, 200),
    CKgPoint(120, 40), 200);
  m_pScene->AddChild(m_pLsmLayer);
}
void CRegression::Update() {
  if(m_bKeyPressed['Q']) {
    m_pLsmLayer->m_bQuadricCurve = !m_pLsmLayer->m_bQuadricCurve;
    m_pLsmLayer->GenerateData(200, false);
    m_bKeyPressed['Q'] = false;
  }
```


Main.cpp (7)

```
 if(m_bKeyPressed['S']) {
    LeastSquared(m_pLsmLayer->m_X, m_pLsmLayer->m_w, m_pLsmLayer->m_y,
      m_pLsmLayer->m_nPointCnt, 3, false, 0);

    int y0;
    for(int x = 0 ; x < m_pLsmLayer->m_nW ; ++x) {
      int y = (int)(m_pLsmLayer->m_w[0]*x*x + m_pLsmLayer->m_w[1]*x
                   + m_pLsmLayer->m_w[2]);
      if(x > 0) {
        CKhuGleSprite::DrawLine(m_pLsmLayer->m_ImageBgR,
          m_pLsmLayer->m_ImageBgG, m_pLsmLayer->m_ImageBgB,
          m_pLsmLayer->m_nW, m_pLsmLayer->m_nH, x-1, y0,
          x, y, KG_COLOR_24_RGB(255, 0, 0));
      }
      y0 = y;
    }
```

Main.cpp (8)

```
    LeastSquared(m_pLsmLayer->m_X, m_pLsmLayer->m_w, m_pLsmLayer->m_y,
      m_pLsmLayer->m_nPointCnt, 3, true, 0.9);

    for(int x = 0 ; x < m_pLsmLayer->m_nW ; ++x) {
      int y = (int)(m_pLsmLayer->m_w[0]*x*x + m_pLsmLayer->m_w[1]*x
                    + m_pLsmLayer->m_w[2]);
      if(x > 0) {
        CKhuGleSprite::DrawLine(m_pLsmLayer->m_ImageBgR,
          m_pLsmLayer->m_ImageBgG, m_pLsmLayer->m_ImageBgB,
          m_pLsmLayer->m_nW, m_pLsmLayer->m_nH, x-1, y0, x, y,
          KG_COLOR_24_RGB(255, 255, 0));
      }
    y0 = y;
    }
    m_bKeyPressed['S'] = false;
  }
```

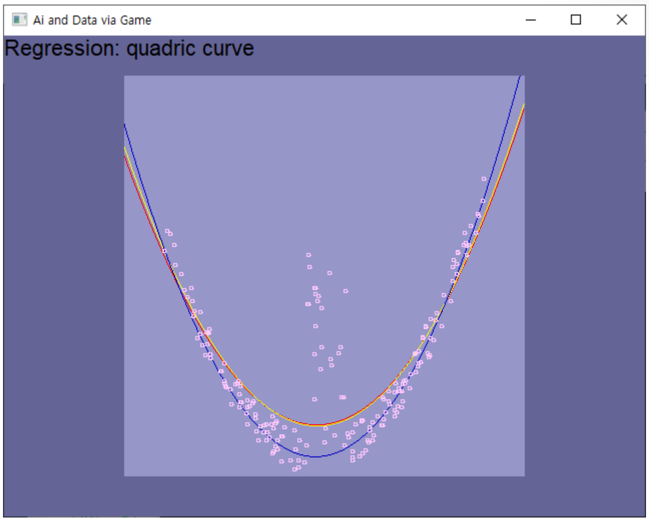

Main.cpp (9)

```
  if(m_bKeyPressed['N'] || m_bKeyPressed['M']) {
    if(m_bKeyPressed['M'])   m_pLsmLayer->GenerateData(200, true);
    else   m_pLsmLayer->GenerateData(200, false);

    m_bKeyPressed['N'] = false;
    m_bKeyPressed['M'] = false;
  }
  m_pScene->Render();

  if(m_pLsmLayer->m_bQuadricCurve)
    DrawSceneTextPos("Regression: quadric curve", CKgPoint(0, 0));
  else
    DrawSceneTextPos("Regression: line", CKgPoint(0, 0));
  CKhuGleWin::Update();
}
int main() {
  CRegression *pRegression = new CRegression(640, 480);
  KhuGleWinInit(pRegression);
  return 0;
}
```

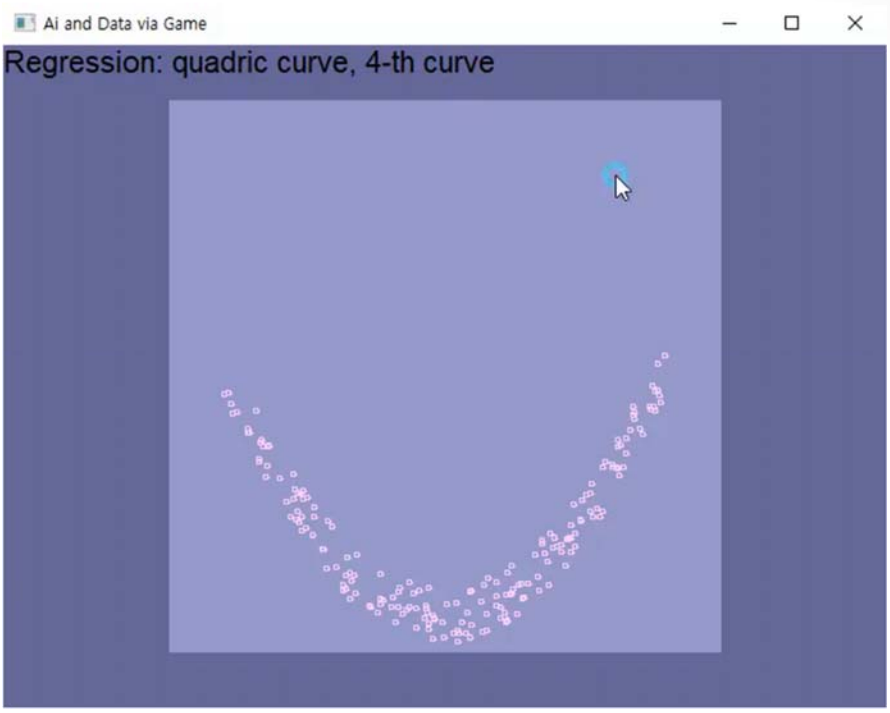
Practice VI (1)

- High order polynomial regression
- RANSAC (Random sample consensus)
  - Iterative parameter estimation method

  - $\mathbf{W} \leftarrow \varnothing$
    $C_M \leftarrow 0$
    while iteration **do**
    randomly subset selection
    estimate parameter ($\mathbf{w}_i$ or $\mathbf{W}_i$)
    inlier count (C_i)
    if $C_i > C_M$ **then**
    $C_M \leftarrow C_i$
    $\mathbf{W} \leftarrow \mathbf{W}_i$
    end if
    end while



Practice VI (2)

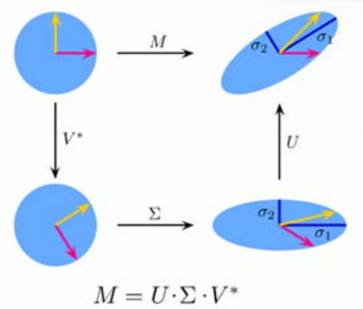


Advanced Courses (1)

- Singular value decomposition (SVD)

  $\mathbf{M} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T$

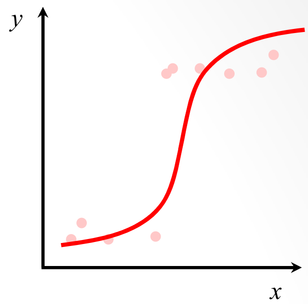
  - $\mathbf{U}$ and $\mathbf{V}$ are singular vectors, orthonormal and unitary matrices
  - $\boldsymbol{\Sigma}$ is a diagonal matrix having singular values
  - Applications
    - Pseudo inverse
    - Truncated SVD
      - Regularization
      - Dimensionality reduction



https://upload.wikimedia.org/wikipedia/commons/thumb/b/bb/Singular-Value-Decomposition.svg/220px-Singular-Value-Decomposition.svg.png

Advanced Courses (2)

- Logistic regression
  - Classification using a logistic function

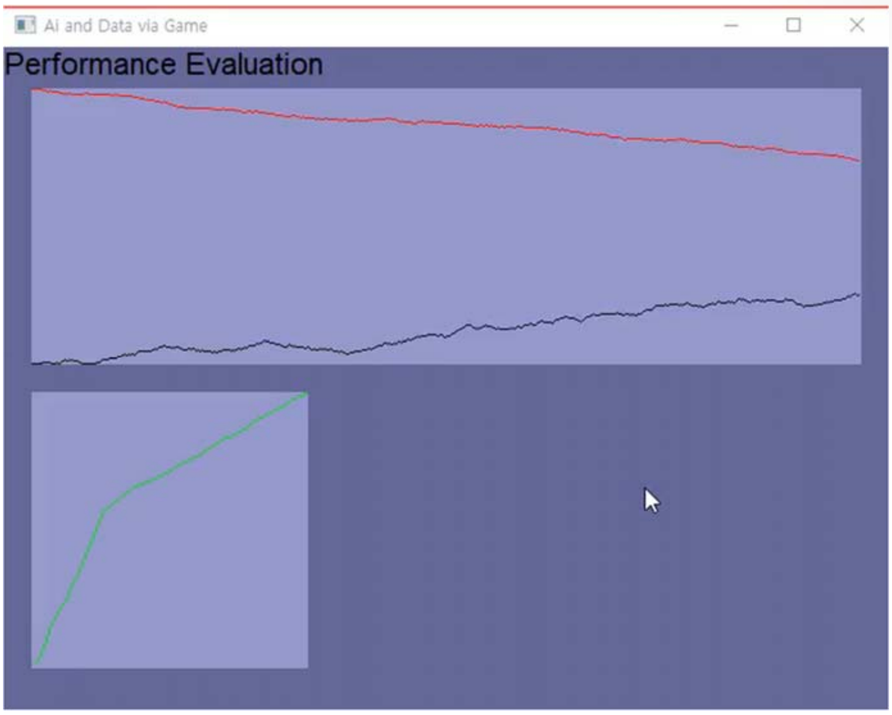


$$P(y=1) = \frac{1}{1+e^{-(b+\mathbf{wX})}}$$

$$\frac{1}{P(y=1)} - 1 = e^{-(b+\mathbf{wX})}, \quad \frac{1-P(y=1)}{P(y=1)} = \frac{1}{e^{(b+\mathbf{wX})}}$$

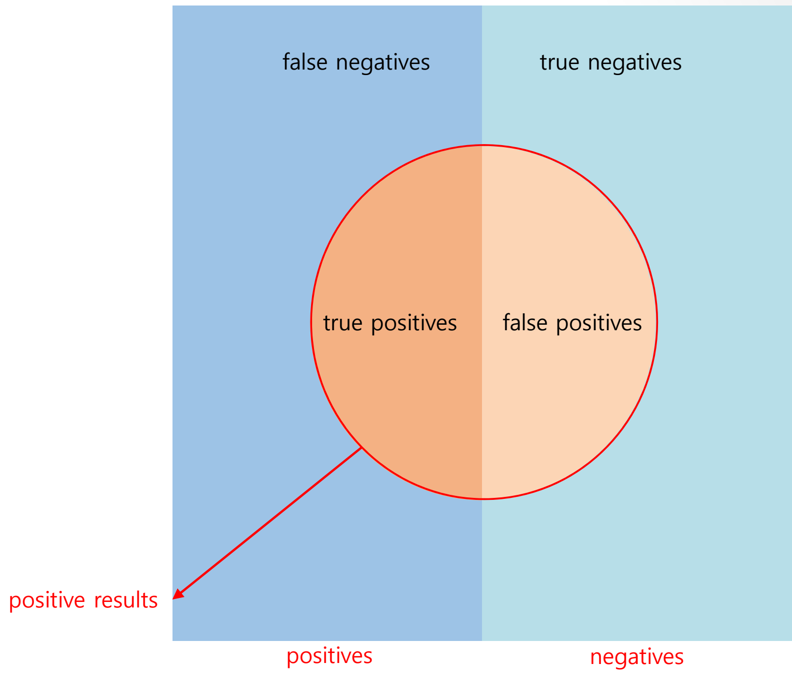
$$\log\left(\frac{P(y=1)}{1-P(y=1)}\right) = \log\left(\frac{P(y=1)}{P(y=0)}\right) = b + \mathbf{wX}$$

9. Performance Evaluation



Classification accuracy I

- True positives
- False positives
  - error
- True negatives
- False negatives
  - error



Classification accuracy II

- Precision
  - (true positives)/(true positives + false positives)
  - (true positives)/(positive results)
- Recall, sensitivity
  - (true positives)/(true positives + false negatives)
  - (true positives)/(positives)
- False positive rate (FPR)
  - (false positives)/(negatives)
- False negative rate (FNR)
  - (false negatives)/(positives)
- Accuracy
  - (true positives + true negatives)/(positives + negatives)

F1 score & confusion matrix

- F1 score
  - Harmonic mean of precision and recall

$$F_1 = \left(\frac{\text{recall}^{-1} + \text{precision}^{-1}}{2} \right)^{-1} = 2\frac{\text{precision} \cdot \text{recall}}{\text{reccsion+recall}}$$

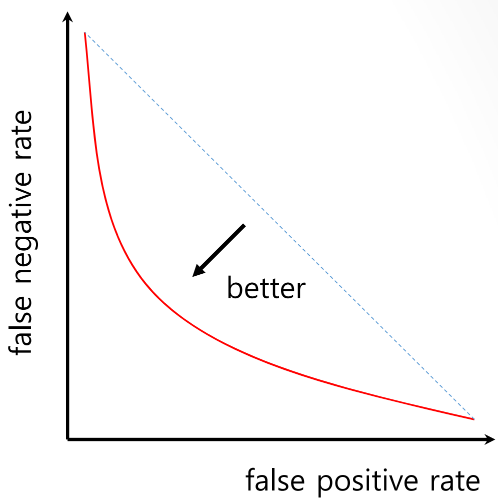
$$F_\beta = \frac{1}{\frac{1}{\beta^2+1}\frac{1}{\text{precision}} + \frac{\beta^2}{\beta^2+1}\frac{1}{\text{recall}}} = \left(1+\beta^2\right)\frac{\text{precision} \cdot \text{recall}}{\beta^2\text{preccsion+recall}}$$

- Confusion matrix
  - Classification performance

		Actual class	
		A	B
Predicted Class	A	10	4
	B	3	15

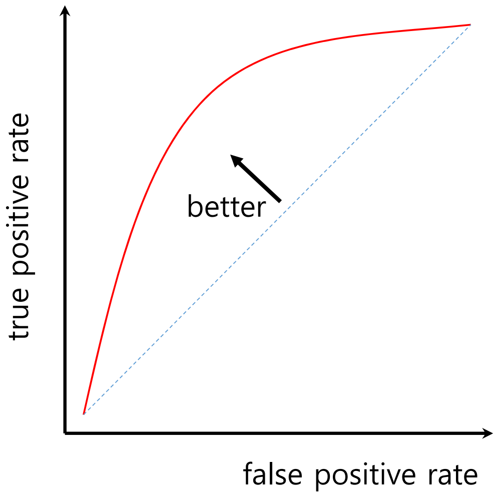
ROC

- ROC
  - Receiver operating characteristic



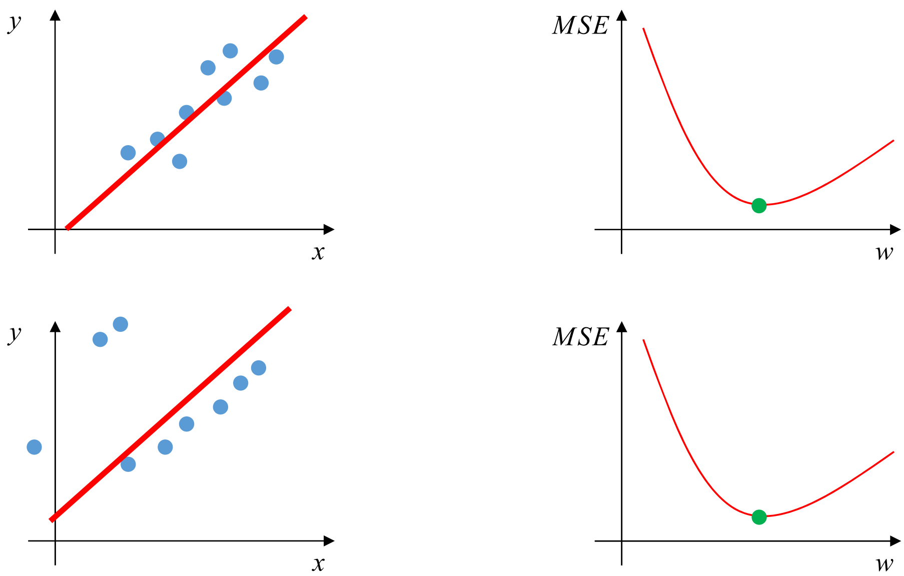
AUC

- AUC
  - Area Under the ROC curve
  - AUROC



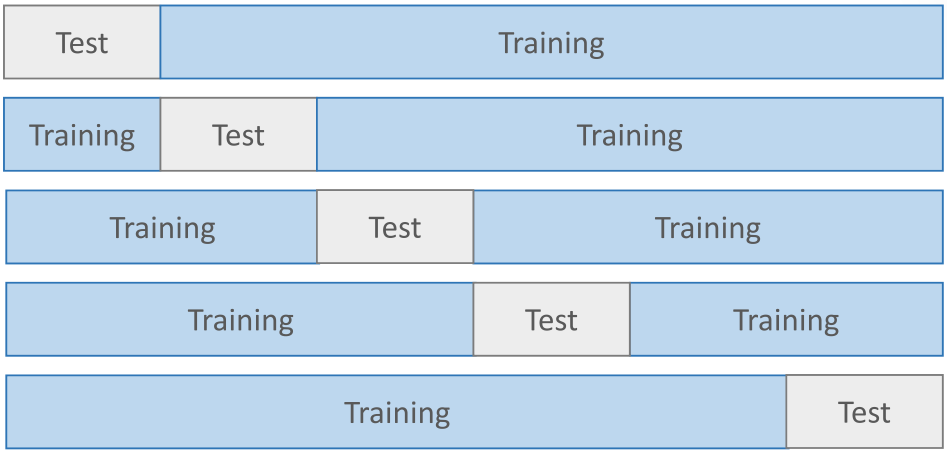
Error

- MSE (Mean square error)



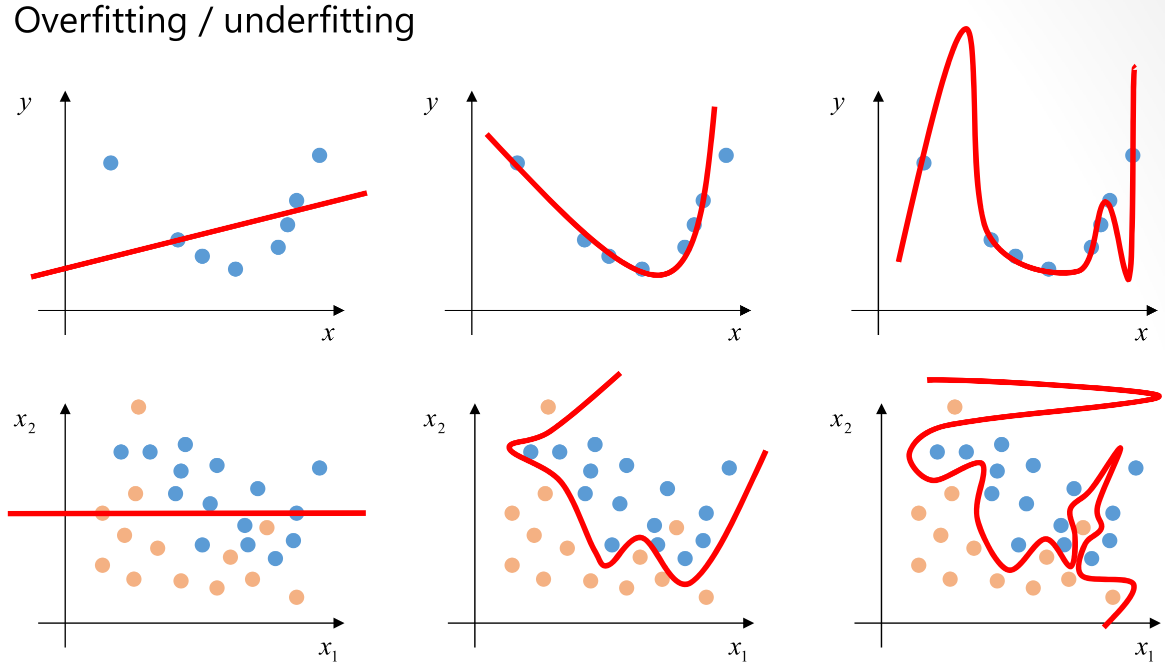
Validation

- k-fold cross validation



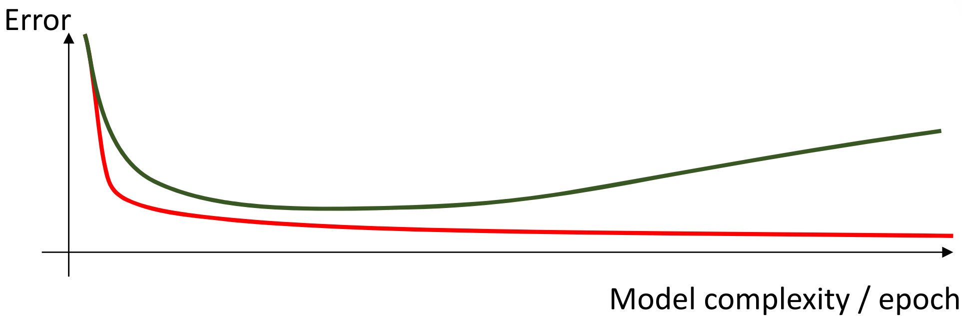
Overfitting / underfitting I

- Overfitting / underfitting



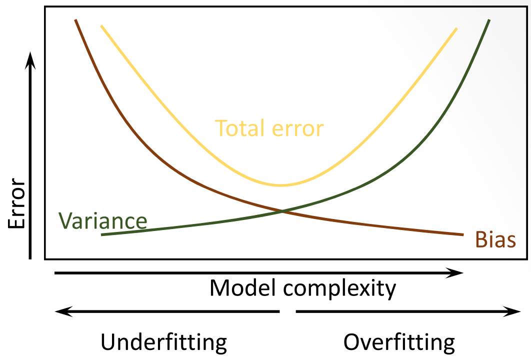
Overfitting / underfitting II

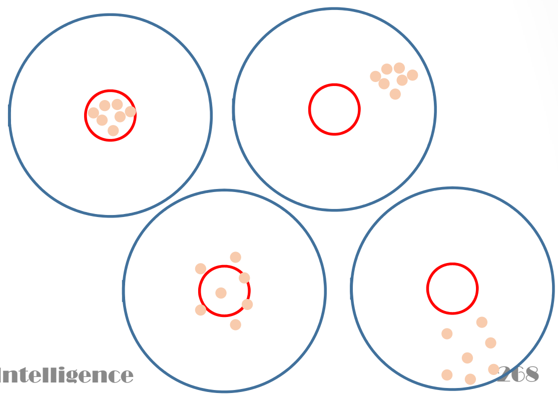
- Train/validation



Generalization / Regularization

- Generalization
- Regularization
  - Controlling overfitting
- Bias
  - Difference between predictions and true values
  - High bias: underfitting
- Variance
  - High variance: overfitting

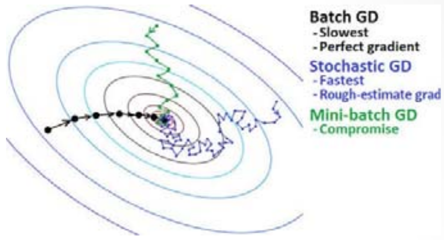




Hyperparameter I

- Hyperparameter : parameter used to control the learning process
  - Learning rate
  - Batch size
  - Regularization hyperparameter

- (Mini-)Batch
  - Batch gradient descent: All samples
  - Stochastic gradient descent: 1 sample
  - Mini batch gradient descent: batch size samples



https://miro.medium.com/max/527/1*tRhocv_8nr4CwGbc3CaPXQ.jpeg

Hyperparameter II

- Epoch
  - Updated by an entire dataset

- Iteration
  - Updated by an batch size

```
class CKhuGleGraphLayer : public CKhuGleLayer {
public:
  int m_nCurrentCnt;
  std::vector<std::vector<double>> m_Data;
  std::vector<double> m_MaxData;
  int m_nDataTotal;
  double m_TrainAccuacy, m_TrainLoss;

  CKhuGleGraphLayer(int nW, int nH, KgColor24 bgColor, int nDataTotal,
    CKgPoint ptPos = CKgPoint(0, 0));
  void SetMaxData(int nIndex, double Value);
  void DrawBackgroundImage();
};
```



```
CKhuGleGraphLayer::CKhuGleGraphLayer(int nW, int nH, KgColor24 bgColor,
  int nDataTotal, CKgPoint ptPos)
  : m_MaxData(nDataTotal), m_Data(nDataTotal),
    CKhuGleLayer(nW, nH, bgColor, ptPos) {
  m_bgColor = bgColor;
  m_nCurrentCnt = 0;
  m_nDataTotal = nDataTotal;

  m_TrainAccuacy = 0.;
  m_TrainLoss = 2.5;
}

void CKhuGleGraphLayer::SetMaxData(int nIndex, double Value) {
  m_MaxData[nIndex] = Value;
}
```

```
void CKhuGleGraphLayer::DrawBackgroundImage() {
  for(int y = 0 ; y < m_nH ; y++)
    for(int x = 0 ; x < m_nW ; x++) {
      m_ImageBgR[y][x] = KgGetRed(m_bgColor);
      m_ImageBgG[y][x] = KgGetGreen(m_bgColor);
      m_ImageBgB[y][x] = KgGetBlue(m_bgColor);
    }
```



```
  int xx0, yy0, xx1, yy1;
  for(int k = 0 ; k < m_nDataTotal ; ++k) {
    KgColor24 Color = KG_COLOR_24_RGB(k%2*255, k/2%2*255, k/4%2*255);
    for(int i = 0 ; i < m_nCurrentCnt ; ++i) {
      xx1 = i*m_nW/m_nCurrentCnt;
      yy1 = (int)(m_nH - m_Data[k][i]*m_nH/m_MaxData[k] - 1);
      if(yy1 < 0) yy1 = 0;
      if(yy1 >= m_nH) yy1 = m_nH-1;
      if(i > 0)
        CKhuGleSprite::DrawLine(m_ImageBgR, m_ImageBgG, m_ImageBgB, m_nW, m_nH,
          xx0, yy0, xx1, yy1, Color);
      xx0 = xx1;
      yy0 = yy1;
    }
  }
}
```

```
class CKhuGleRocLayer : public CKhuGleLayer {
public:
  std::vector<std::pair<int, double>> m_Data;
  std::vector<std::pair<double, double>> m_Positive;

  CKhuGleRocLayer(int nW, int nH, KgColor24 bgColor,
    CKgPoint ptPos = CKgPoint(0, 0))
  : CKhuGleLayer(nW, nH, bgColor, ptPos) {
    MakeData();
    ComputePositives();
    DrawBackgroundImage();
  }
  void MakeData();
  void ComputePositives();
  void DrawBackgroundImage();
};
```



```
void CKhuGleRocLayer::MakeData() {
  unsigned int seed = (unsigned int)std::chrono::
    system_clock::now().time_since_epoch().count();
  std::default_random_engine generator(seed);
  std::poisson_distribution<int> poisson_dist(85);

  m_Data.clear();
  for(int i = 0 ; i < 2000 ; ++i) {
    double score = poisson_dist(generator)/100.;
    if(score < 0) continue;
    if(score > 1) continue;

    if(rand()%10 > 3) {
      if(score > 0.85) m_Data.push_back({1, score});
      else if(score > 0.15) m_Data.push_back({rand()%10<5?0:1, score});
      else m_Data.push_back({0, score});
    }
    else
      m_Data.push_back({rand()%10 < 5?0:1, score});
  }
}
```

```
void CKhuGleRocLayer::ComputePositives() {
  m_Positive.clear();

  for(int nThreshold = 0 ; nThreshold <= 100 ; nThreshold += 1) {
    double TP = 0, FP = 0;
    int nPositiveCnt = 0;
    for(auto &Data : m_Data) {
      if(Data.second >= nThreshold/100.) {
        if(Data.first == 1) TP++;
        else FP++;
      }
      if(Data.first == 1)
      nPositiveCnt++;
    }

    TP /= nPositiveCnt;
    FP /= (m_Data.size()-nPositiveCnt);

    m_Positive.push_back({TP, FP});
  }
}
```



```
void CKhuGleRocLayer::DrawBackgroundImage() {
  for(int y = 0 ; y < m_nH ; y++)
    for(int x = 0 ; x < m_nW ; x++) {
      m_ImageBgR[y][x] = KgGetRed(m_bgColor);
      m_ImageBgG[y][x] = KgGetGreen(m_bgColor);
      m_ImageBgB[y][x] = KgGetBlue(m_bgColor);
    }
  int xx0, yy0, xx1, yy1;
  bool bFirst = true;
  for(auto &Positive : m_Positive) {
    xx1 = (int)(Positive.second * (m_nW-1));
    yy1 = m_nH-(int)(Positive.first * (m_nH-1))-1;

    if(!bFirst)
      CKhuGleSprite::DrawLine(m_ImageBgR, m_ImageBgG, m_ImageBgB, m_nW, m_nH,
        xx0, yy0, xx1, yy1, KG_COLOR_24_RGB(0, 255, 0));
    bFirst = false;
    xx0 = xx1;
    yy0 = yy1;
  }
}
```

```
class CPerformance : public CKhuGleWin {
public:
  CKhuGleGraphLayer *m_pTrainGraphLayer;
  CKhuGleRocLayer *m_pRocLayer;

  CPerformance(int nW, int nH);
  void Update();
};

CPerformance::CPerformance(int nW, int nH) : CKhuGleWin(nW, nH)  {
  m_pScene = new CKhuGleScene(640, 480, KG_COLOR_24_RGB(100, 100, 150));

  m_pTrainGraphLayer = new CKhuGleGraphLayer(600, 200,
    KG_COLOR_24_RGB(150, 150, 200), 2, CKgPoint(20, 30));
  m_pTrainGraphLayer->SetMaxData(0, 100.);
  m_pTrainGraphLayer->SetMaxData(1, 2.5);
  m_pScene->AddChild(m_pTrainGraphLayer);

  m_pRocLayer = new CKhuGleRocLayer(200, 200,
    KG_COLOR_24_RGB(150, 150, 200), CKgPoint(20, 250));
  m_pScene->AddChild(m_pRocLayer);
}
```



```
void CPerformance::Update() {
  if(m_bKeyPressed['N']) {
    m_pRocLayer->MakeData();
    m_pRocLayer->ComputePositives();
    m_pRocLayer->DrawBackgroundImage();

    m_bKeyPressed['N'] = false;
  }

  unsigned int seed = (unsigned int)std::chrono::
    system_clock::now().time_since_epoch().count();
  std::default_random_engine generator(seed);

  std::uniform_real_distribution<double> uniform_dist1(-0.43, 0.5);
  std::uniform_real_distribution<double> uniform_dist2(-0.7, 0.5);

  double alpha = uniform_dist1(generator);
  double beta = uniform_dist2(generator);
```

```
  m_pTrainGraphLayer->m_TrainAccuacy
    = 0.99*m_pTrainGraphLayer->m_TrainAccuacy
      + 0.01*(alpha+m_pTrainGraphLayer->m_TrainAccuacy);
  m_pTrainGraphLayer->m_TrainLoss
    = 0.99*m_pTrainGraphLayer->m_TrainLoss
      + 0.01*(beta+m_pTrainGraphLayer->m_TrainLoss);
  if(m_pTrainGraphLayer->m_TrainAccuacy > 1)
    m_pTrainGraphLayer->m_TrainAccuacy = 1;
  if(m_pTrainGraphLayer->m_TrainLoss < 0) m_pTrainGraphLayer->m_TrainLoss = 0;

  m_pTrainGraphLayer->m_Data[0].push_back(
    100*m_pTrainGraphLayer->m_TrainAccuacy);
  m_pTrainGraphLayer->m_Data[1].push_back(m_pTrainGraphLayer->m_TrainLoss);
  m_pTrainGraphLayer->m_nCurrentCnt++;
  m_pTrainGraphLayer->DrawBackgroundImage();

  m_pScene->Render();
  DrawSceneTextPos("Performance Evaluation", CKgPoint(0, 0));

  CKhuGleWin::Update();
}
```



```
int main() {
  CPerformance *pPerformance = new CPerformance(640, 480);

  KhuGleWinInit(pPerformance);

  return 0;
}
```

Practice VII (1)

- k-NN analysis
  - Diabetes prediction
  - https://www.kaggle.com/saurabh00007/diabetescsv
  - Pregnancies, Glucose, BloodPressure, SkinThickness, Insulin, BMI, DiabetesPedigreeFunction, Age, Outcome

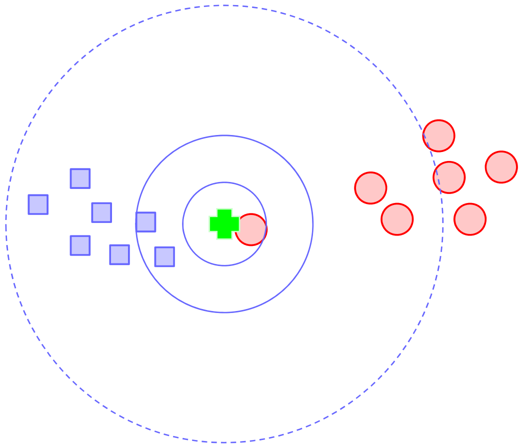
```
std::string CsvPath;
CsvPath = ExePath + std::string("\\diabetes.csv");

std::vector<std::vector<std::string>> ReadData;
ReadCsv(CsvPath, ReadData);

for(auto &read : ReadData) {
  for(auto &column : read) {
    std::cout << column << ",";
  }
  std::cout << std::endl;
}
```


Practice VII (2)

```
void ReadCsv(std::string FileName, std::vector<std::vector<std::string>> &Data) {
  std::ifstream ifs;                          // #include <fstream>
                                              // #include <string>
  ifs.open(FileName);                         // #include <algorithm>
  if(!ifs.is_open()) return;

  std::string LineString = "";
  std::string Delimeter = ",";
  while(getline(ifs, LineString)) {
    std::vector<std::string> RowData;
    unsigned int nPos = 0, nFindPos;
    do {
      nFindPos = LineString.find(Delimeter, nPos);
      if(nFindPos == std::string::npos) nFindPos = LineString.length();

      RowData.push_back(LineString.substr(nPos, nFindPos-nPos));
      nPos = nFindPos+1;
    } while(nFindPos < LineString.length());
    Data.push_back(RowData);
  }
  ifs.close();
}
```

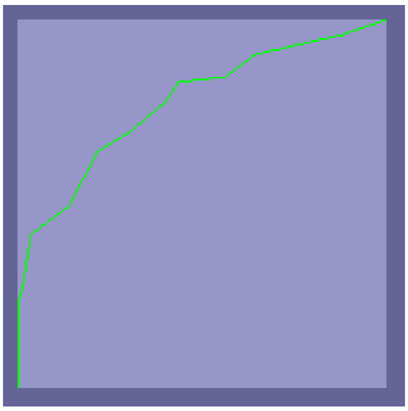
Practice VII (3)

- k-NN classification
  - Instance-based learning



Practice VII (4)

- Analysis
  - String to numeric data
    - std::stod(), std::stoi()
  - # of neighbors (k) vs. Accuracy (k-fold cross validation)
  - Confusion matrix
  - Accuracy, precision, recall, f1-score
  - ROC
    - Probability: (positive instances)/k
  - Normalization ?



Advanced Courses

- ROUGE: recall-oriented understudy for gisting evaluation
  - ROUGE-N: n-gram based co-occurrence statistics
    - n-gram: contiguous sequence of n items from a given sample

$$ROUGE\_N_{signle}(c,r) = \frac{\sum_{r_i \in r} \sum_{n-gram \in r_i} Count(n-gram, c)}{\sum_{r_i \in r} numNgrams(r_i)}$$

  - ROUGE-L, ROUGE_W, …

- BLEU: bilingual evaluation understudy

- Cosine similarity

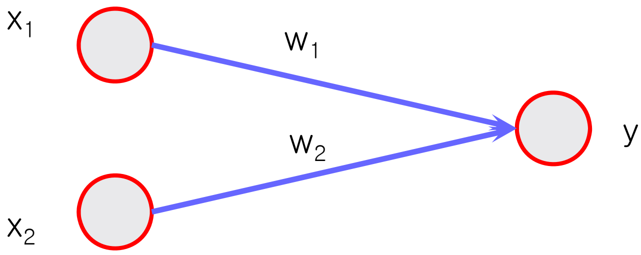
Project II

Data Analysis

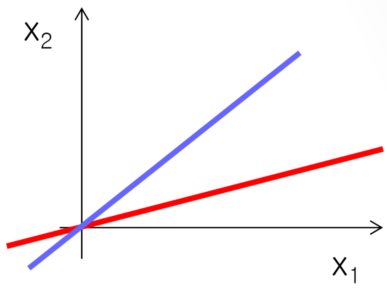
Data Analysis

- Correlation
- Clustering
- k-NN
- Parameter estimation
- Sound/image processing

10. Perceptron



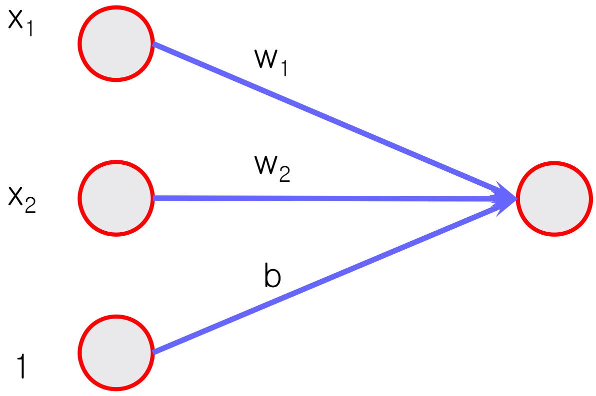
$$y = w_1 x_1 + w_2 x_2$$
$$= \begin{bmatrix} w_1 & w_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

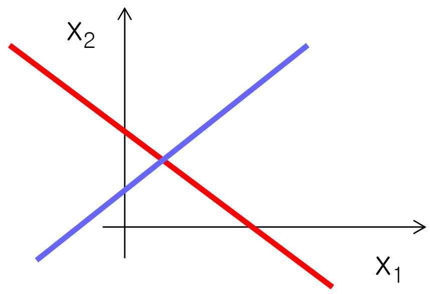


$$c = \begin{cases} c_1 & if \;\; y \ge 0 \\ c_2 & if \;\; y < 0 \end{cases}$$

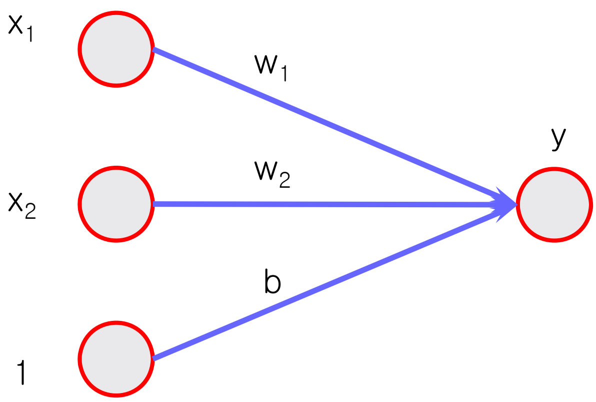
$$0 = w_1 x_1 + w_2 x_2$$
$$w_2 x_2 = -w_1 x_1$$
$$x_2 = -\frac{w_1}{w_2} x_1$$

$$y = w_1 x_1 + w_2 x_2 + b$$
$$= \begin{bmatrix} w_1 & w_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + b$$

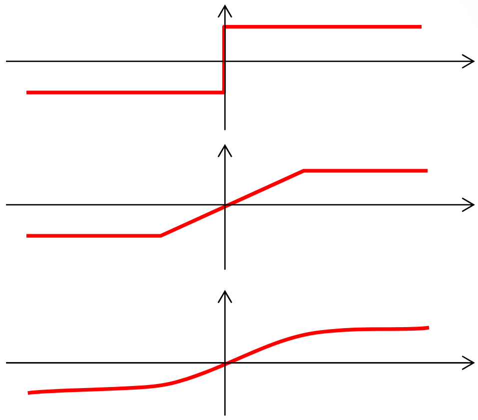




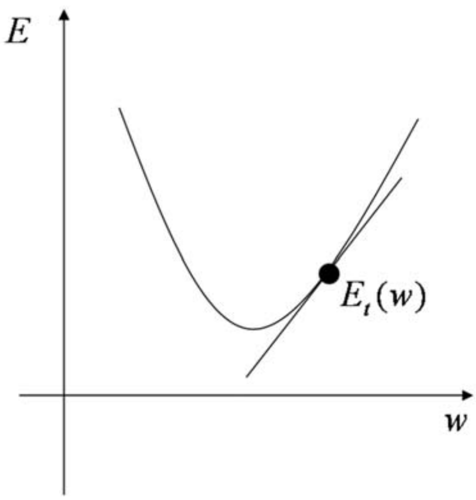
$$y = f(w_1 x_1 + w_2 x_2 + b)$$
$$= f\left(\begin{bmatrix} w_1 & w_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + b \right)$$



Activation function
(활성함수)



$$w_i(t+1) = w_i(t) + \eta\big(d(t) - y(t)\big)x_i(t)$$



Gradient descent

$$E = \frac{1}{2}\big(d(t) - y(t)\big)^2$$

$$\frac{\partial E}{\partial w_i} = -\big(d(t) - y(t)\big)\frac{\partial y(t)}{\partial w_i}$$
$$= -\big(d(t) - y(t)\big)x_i(t)$$

KhuDaNet.h (1)

```
#include "KhuDaNetLayer.h"
#include <vector>
#define MAX_INFORMATION_STRING_SIZE  1000

class CKhuDaNet {
public:
  CKhuDaNet();
  virtual ~CKhuDaNet();

  std::vector<CKhuDaNetLayer*> m_Layers;

  int m_nInputSize, m_nOutputSize;
  char *m_Information;

  char *GetInformation();
  bool IsNetwork();
  void ClearAllLayers();
  void AddLayer(CKhuDaNetLayer *pLayer);
  void AddLayer(CKhuDaNetLayerOption LayerOptionInput);
  void AllocDeltaWeight();
  void InitWeight();
  int Forward(double *Input, double *Probability = 0);
  int TrainBatch(double **Input, double **Output, int nBatchSize, double *pLoss);
```


KhuDaNet.h (1)

```
  void SaveKhuDaNet(char *Filename);
  void LoadKhuDaNet(char *Filename);

  static int ArgMax(double *List, int nCnt);
  static double **dmatrix(int nH, int nW);
  static void free_dmatrix(double **Image, int nH, int nW);
  static double **dmatrix1d(int nH, int nW);
  static void free_dmatrix1d(double **Image, int nH, int nW);
  static double Identify(double x);
  static double DifferentialIdentify(double x);
  static double BinaryStep(double x);
  static double DifferentialBinaryStep(double x);
  static double Sigmoid(double x);
  static double DifferentialSigmoid(double x);
};
```

KhuDaNet.cpp (1)

```
…
CKhuDaNet::CKhuDaNet() {
  m_nInputSize = m_nOutputSize = 0;

  m_Information = new char[MAX_INFORMATION_STRING_SIZE];
}

CKhuDaNet::~CKhuDaNet() {
  ClearAllLayers();

  delete [] m_Information;
}

bool CKhuDaNet::IsNetwork() {
  if(m_Layers.size() < 2) return false;
  return true;
}
```


KhuDaNet.cpp (2)

```
char *CKhuDaNet::GetInformation() {
  memset(m_Information, 32, MAX_INFORMATION_STRING_SIZE);

  m_Information[MAX_INFORMATION_STRING_SIZE-1] = 0;

  int nPos = 0;
  for(auto &Layer : m_Layers) {
    if(Layer->m_LayerOption.nLayerType & KDN_LT_FC) {
      sprintf(m_Information+nPos, "%s(%5d)  ", "FC",
        Layer->m_LayerOption.nNodeCnt);
      nPos += 10;
    }
  }

  return m_Information;
}
```

KhuDaNet.cpp (3)

```
void CKhuDaNet::ClearAllLayers() {
  for(std::vector<CKhuDaNetLayer*>::reverse_iterator Iter = m_Layers.rbegin();
    Iter != m_Layers.rend(); ++Iter) {
    delete [] *Iter;
    *Iter = 0;
  }
  m_Layers.clear();
}

void CKhuDaNet::AddLayer(CKhuDaNetLayer *pLayer) {
  if(m_Layers.size() == 0)
    m_nInputSize = pLayer->m_LayerOption.nNodeCnt;

  m_nOutputSize = pLayer->m_LayerOption.nNodeCnt;

  m_Layers.push_back(pLayer);
}
```


KhuDaNet.cpp (4)

```
void CKhuDaNet::AddLayer(CKhuDaNetLayerOption LayerOptionInput) {
  CKhuDaNetLayer *pLayer;

  if(m_Layers.size() == 0) {
    pLayer = new CKhuDaNetLayer(LayerOptionInput, nullptr);
    m_nInputSize = pLayer->m_LayerOption.nNodeCnt;
  }
  else
    pLayer = new CKhuDaNetLayer(LayerOptionInput, m_Layers[m_Layers.size()-1]);

    m_nOutputSize = pLayer->m_LayerOption.nNodeCnt;

    m_Layers.push_back(pLayer);
}

void CKhuDaNet::InitWeight() {
  for(auto &Layer : m_Layers)
    Layer->InitWeight();
}
```

KhuDaNet.cpp (5)

```
void CKhuDaNet::AllocDeltaWeight() {
  for(auto &Layer : m_Layers)
    Layer->AllocDeltaWeight();
}

int CKhuDaNet::Forward(double *Input, double *Probability) {
  int MaxPos;

  for(auto &Layer : m_Layers) {
    if(Layer->m_LayerOption.nLayerType & KDN_LT_INPUT) {
      if(Layer->m_LayerOption.nLayerType & KDN_LT_FC)
        memcpy(Layer->m_Node, Input,
          Layer->m_LayerOption.nNodeCnt*sizeof(double));
    }
    else if(Layer->m_LayerOption.nLayerType & KDN_LT_OUTPUT)
      MaxPos = Layer->ComputeLayer(Probability);
    else
      Layer->ComputeLayer();
  }

  return MaxPos;
}
```


KhuDaNet.cpp (6)

```
int CKhuDaNet::TrainBatch(double **Input, double **Output, int nBatchSize,
double *pLoss) {
  int nTP = 0;
  *pLoss = 0;

  AllocDeltaWeight();

  for(int i = 0 ; i < nBatchSize ; ++i) {
    int MaxPos = Forward(Input[i]);

    if(m_nOutputSize == 1) {
      if(MaxPos == 1 && Output[i][0] > 0.5)  nTP++;
      else if(MaxPos == 0 && Output[i][0] < 0.5) nTP++;
    }
    else {
      if(MaxPos == ArgMax(Output[i], m_nOutputSize)) nTP++;
    }
```

KhuDaNet.cpp (7)

```
    for(std::vector<CKhuDaNetLayer*>::reverse_iterator Iter = m_Layers.rbegin();
      Iter != m_Layers.rend(); ++Iter) {
      (*Iter)->ComputeDelta(Output[i]);
      (*Iter)->ComputeDeltaWeight(i==0?true:false);

      if(Iter == m_Layers.rbegin())
        *pLoss += (*Iter)->GetLoss();
    }
  }

  *pLoss /= nBatchSize;

  for(auto &Layer : m_Layers)
    Layer->UpdateWeight(nBatchSize);

  return nTP;
}
```


KhuDaNet.cpp (8)

```
void CKhuDaNet::SaveKhuDaNet(char *Filename) {
  FILE *fp = fopen(Filename, "wb");
  if(!fp) return;

  fwrite("KhuDaNet", sizeof(char), 8, fp);

  int Cnt = m_Layers.size();
  fwrite(&Cnt, sizeof(int), 1, fp);

  for(auto &Layer : m_Layers) {
    fwrite(&(Layer->m_LayerOption), sizeof(CKhuDaNetLayerOption), 1, fp);
  }
```

KhuDaNet.cpp (9)

```
  CKhuDaNetLayer *pBackwardLayer = nullptr;
  for(auto &Layer : m_Layers) {
    if(pBackwardLayer) {
      if(Layer->m_LayerOption.nLayerType & KDN_LT_FC) {
        if(pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_FC)
          for(int i = 0 ; i < Layer->m_LayerOption.nNodeCnt ; ++i)
            fwrite(Layer->m_Weight[i], sizeof(double),
              pBackwardLayer->m_LayerOption.nNodeCnt, fp);

        fwrite(Layer->m_Bias, sizeof(double),
          Layer->m_LayerOption.nNodeCnt, fp);
      }
    }

    pBackwardLayer = Layer;
  }

  fclose(fp);
}
```


KhuDaNet.cpp (10)

```
void CKhuDaNet::LoadKhuDaNet(char *Filename) {
  ClearAllLayers();
  FILE *fp = fopen(Filename, "rb");
  if(!fp) return;

  char Buf[10];
  int nLayerCnt;

  fread(Buf, sizeof(char), 8, fp);
  fread(&nLayerCnt, sizeof(int), 1, fp);

  for(int s = 0 ; s < nLayerCnt ; ++s) {
    CKhuDaNetLayer *pLayer;
    char *pRawLayerOption = new char[sizeof(CKhuDaNetLayerOption)];

    fread(pRawLayerOption, sizeof(CKhuDaNetLayerOption), 1, fp);

    CKhuDaNetLayerOption
      KhuDaNetLayerOption(*(CKhuDaNetLayerOption *)pRawLayerOption);
```

KhuDaNet.cpp (11)

```
    if(s == 0)  {
      pLayer = new CKhuDaNetLayer(KhuDaNetLayerOption, nullptr);
      m_nInputSize = pLayer->m_LayerOption.nNodeCnt;
    }
    else
      pLayer = new CKhuDaNetLayer(KhuDaNetLayerOption,
                    m_Layers[m_Layers.size()-1]);

    m_Layers.push_back(pLayer);
    m_nOutputSize = pLayer->m_LayerOption.nNodeCnt;

    delete [] pRawLayerOption;
  }
```


KhuDaNet.cpp (12)

```
  CKhuDaNetLayer *pBackwardLayer = nullptr;
  for(int s = 0 ; s < nLayerCnt ; ++s) {
    if(pBackwardLayer) {
     if(m_Layers[s]->m_LayerOption.nLayerType & KDN_LT_FC) {
       if(pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_FC)
         for(int i = 0 ; i < m_Layers[s]->m_LayerOption.nNodeCnt ; ++i)
           fread(m_Layers[s]->m_Weight[i], sizeof(double),
             pBackwardLayer->m_LayerOption.nNodeCnt, fp);

         fread(m_Layers[s]->m_Bias, sizeof(double),
           m_Layers[s]->m_LayerOption.nNodeCnt, fp);
      }
    }

    pBackwardLayer = m_Layers[s];
  }

  fclose(fp);
}
```

KhuDaNet.cpp (13)

```
int CKhuDaNet::ArgMax(double *List, int nCnt) {
  int MaxPos = 0;

  for(int i = 0 ; i < nCnt ; ++i)
    if(List[i] > List[MaxPos]) MaxPos = i;

  return MaxPos;
}

double **CKhuDaNet::dmatrix(int nH, int nW) {…}
void CKhuDaNet::free_dmatrix(double **Image, int nH, int nW) {…}
double **CKhuDaNet::dmatrix1d(int nH, int nW) {…}
void CKhuDaNet::free_dmatrix1d(double **Image, int nH, int nW) {…}
```


KhuDaNet.cpp (14)

```
double CKhuDaNet::Identify(double x) {
  return x;
}

double CKhuDaNet::DifferentialIdentify(double x) {
  return 1;
}

double CKhuDaNet::BinaryStep(double x) {
  return (x>0)?1:0;
}

double CKhuDaNet::DifferentialBinaryStep(double x) {
  return 0;
}

double CKhuDaNet::Sigmoid(double x) {
  return 1./(1.+exp(-1. * x));
}

double CKhuDaNet::DifferentialSigmoid(double x) {
  return x*(1.-x);
}
```

KhuDaNetLayer.h (1)

```
#define KDN_LT_FC          0x0001

#define KDN_LT_INPUT       0x0100
#define KDN_LT_OUTPUT      0x0400

#define KDN_AF_NONE        0
#define KDN_AF_IDENTIFY    1
#define KDN_AF_BINARY_STEP         2
#define KDN_AF_SIGMOID     3
```


KhuDaNetLayer.h (2)

```
struct CKhuDaNetLayerOption{
  CKhuDaNetLayerOption(unsigned int nLayerTypeIntput, int nImageCntInput,
    int nNodeCntIput, int nWidthInput, int nHeightInput, int nKernelSizeInput,
    int nActicationFnInput, double dLearningRateInput);

  unsigned int nLayerType;
  int nImageCnt;
  int nNodeCnt;
  int nW, nH;
  int nKernelSize;
  int nActicationFn;

  double dLearningRate;
};
```

KhuDaNetLayer.h (3)

```
class CKhuDaNetLayer {
public:
  CKhuDaNetLayerOption m_LayerOption;
  CKhuDaNetLayer *m_pBackwardLayer;

  bool m_bTrained;

  double *m_Node;
  double **m_Weight;

  double *m_Bias;
```


KhuDaNetLayer.h (3)

```
  double *m_Loss;

  double *m_DeltaNode;
  double **m_DeltaWeight;
  double *m_DeltaBias;

  double (*Activation)(double);
  double (*DifferentialActivation)(double);

  CKhuDaNetLayer(CKhuDaNetLayerOption m_LayerOptionInput,
    CKhuDaNetLayer *pBackwardLayerInput);
  virtual ~CKhuDaNetLayer();

  void AllocDeltaWeight();
  void InitWeight();
  int ComputeLayer(double *Probability = 0);
  void ComputeDelta(double *Output);
  void ComputeDeltaWeight(bool bReset);
  void UpdateWeight(int nBatchSize);
  double GetLoss();
};
```

KhuDaNetLayer.cpp (1)

```
…
CKhuDaNetLayerOption::CKhuDaNetLayerOption(unsigned int nLayerTypeIntput,
  int nImageCntInput, int nNodeCntIput,
  int nWidthInput, int nHeightInput, int nKernelSizeInput,
  int nActicationFnInput, double dLearningRateInput)  {

  nLayerType = nLayerTypeIntput;
  nImageCnt = nImageCntInput;
  nNodeCnt = nNodeCntIput;

  nW = nWidthInput;
  nH = nHeightInput;

  nKernelSize = nKernelSizeInput;
  nActicationFn = nActicationFnInput;

  dLearningRate = dLearningRateInput;
}
```


KhuDaNetLayer.cpp (2)

```
CKhuDaNetLayer::CKhuDaNetLayer(CKhuDaNetLayerOption m_LayerOptionInput,
  CKhuDaNetLayer *pBackwardLayerInput)
  : m_LayerOption(m_LayerOptionInput), m_bTrained(false) {

  if(m_LayerOption.nActicationFn == KDN_AF_IDENTIFY) {
    Activation = CKhuDaNet::Identify;
    DifferentialActivation = CKhuDaNet::DifferentialIdentify;
  }
  else if(m_LayerOption.nActicationFn == KDN_AF_BINARY_STEP) {
    Activation = CKhuDaNet::BinaryStep;
    DifferentialActivation = CKhuDaNet::DifferentialBinaryStep;
  }
  else if(m_LayerOption.nActicationFn == KDN_AF_SIGMOID) {
    Activation = CKhuDaNet::Sigmoid;
    DifferentialActivation = CKhuDaNet::DifferentialSigmoid;
  }
```

KhuDaNetLayer.cpp (3)

```
  m_pBackwardLayer = pBackwardLayerInput;

  if((m_LayerOption.nLayerType & KDN_LT_OUTPUT) &&
     (m_LayerOption.nLayerType & KDN_LT_FC)){
    m_Loss = new double [m_LayerOption.nNodeCnt];
  }

  if((m_LayerOption.nLayerType & KDN_LT_INPUT) &&
     (m_LayerOption.nLayerType & KDN_LT_FC)) {
    m_Node = new double [m_LayerOption.nNodeCnt];
  }
  else if(m_LayerOption.nLayerType & KDN_LT_FC) {
    m_Node = new double [m_LayerOption.nNodeCnt];
    if(m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_FC)
      m_Weight = CKhuDaNet::dmatrix1d(m_LayerOption.nNodeCnt,
                             m_pBackwardLayer->m_LayerOption.nNodeCnt);

    m_Bias = new double[m_LayerOption.nNodeCnt];
  }
}
```


KhuDaNetLayer.cpp (4)

```
CKhuDaNetLayer::~CKhuDaNetLayer() {
  if((m_LayerOption.nLayerType & KDN_LT_OUTPUT) &&
     (m_LayerOption.nLayerType & KDN_LT_FC)) {
    delete [] m_Loss;
  }
  if((m_LayerOption.nLayerType & KDN_LT_INPUT) &&
     (m_LayerOption.nLayerType & KDN_LT_FC)) {
    delete [] m_Node;
  }
```

KhuDaNetLayer.cpp (5)

```
  else if(m_LayerOption.nLayerType & KDN_LT_FC) {
    delete [] m_Node;
    if(m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_FC)
      CKhuDaNet::free_dmatrix1d(m_Weight, m_LayerOption.nNodeCnt,
                                m_pBackwardLayer->m_LayerOption.nNodeCnt);
    delete [] m_Bias;

    if(m_bTrained) {
      delete [] m_DeltaNode;
      if(m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_FC)
        CKhuDaNet::free_dmatrix1d(m_DeltaWeight,
           m_LayerOption.nNodeCnt, m_pBackwardLayer->m_LayerOption.nNodeCnt);

      delete [] m_DeltaBias;
    }
  }
}
```


KhuDaNetLayer.cpp (6)

```
void CKhuDaNetLayer::AllocDeltaWeight() {
  if((m_LayerOption.nLayerType & KDN_LT_OUTPUT) &&
     (m_LayerOption.nLayerType & KDN_LT_FC))
  {
  }
  if((m_LayerOption.nLayerType & KDN_LT_INPUT) &&
     (m_LayerOption.nLayerType & KDN_LT_FC)) {
  }
  else if(m_LayerOption.nLayerType & KDN_LT_FC) {
    if(!m_bTrained) {
      m_DeltaNode = new double [m_LayerOption.nNodeCnt];
      if(m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_FC)
        m_DeltaWeight = CKhuDaNet::dmatrix1d(m_LayerOption.nNodeCnt,
                        m_pBackwardLayer->m_LayerOption.nNodeCnt);

      m_DeltaBias = new double[m_LayerOption.nNodeCnt];
    }
  }

  m_bTrained = true;
}
```

KhuDaNetLayer.cpp (7)

```
void CKhuDaNetLayer::InitWeight() {
  static unsigned int seed = (unsigned int)std::chrono::
    system_clock::now().time_since_epoch().count();
  static std::default_random_engine generator(seed);
  if(m_LayerOption.nLayerType & KDN_LT_INPUT) return;
  if(m_LayerOption.nLayerType & KDN_LT_FC) {
    double var = 1;
    if(m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_FC)
      var = sqrt(2./
        (m_pBackwardLayer->m_LayerOption.nNodeCnt + m_LayerOption.nNodeCnt));

    std::normal_distribution<double> distribution(0., var);

    for(int i = 0 ; i < m_LayerOption.nNodeCnt ; ++i) {
      if(m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_FC) {
        for(int j = 0 ; j < m_pBackwardLayer->m_LayerOption.nNodeCnt ; ++j)
          m_Weight[i][j] = distribution(generator);
      }
      m_Bias[i] = 0;
    }
  }
}
```


KhuDaNetLayer.cpp (8)

```
int CKhuDaNetLayer::ComputeLayer(double *Probability) {
  if(m_LayerOption.nLayerType & KDN_LT_INPUT) return 0;

  if((m_LayerOption.nLayerType & KDN_LT_FC) &&
     (m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_FC)) {
    for(int i = 0 ; i < m_LayerOption.nNodeCnt ; ++i) {
      m_Node[i] = 0;
      for(int j = 0 ; j < m_pBackwardLayer->m_LayerOption.nNodeCnt ; ++j )
        m_Node[i] += m_pBackwardLayer->m_Node[j] * m_Weight[i][j];

      m_Node[i] = Activation(m_Node[i] + m_Bias[i]);
    }
  }
  int nMaxNode;
  if((m_LayerOption.nLayerType & KDN_LT_OUTPUT) &&
     (m_LayerOption.nLayerType & KDN_LT_FC)) {
    if(m_Node[0] < 0.5) nMaxNode = 0;
    else nMaxNode = 1;
  }
  if(Probability) *Probability = 0;
  return nMaxNode;
}
```

KhuDaNetLayer.cpp (9)

```
void CKhuDaNetLayer::ComputeDelta(double *Output) {
  if(m_LayerOption.nLayerType & KDN_LT_INPUT) return;

  if(m_LayerOption.nLayerType & KDN_LT_OUTPUT) {
    if(m_LayerOption.nLayerType & KDN_LT_FC) {
      for(int i = 0 ; i < m_LayerOption.nNodeCnt ; ++i)
        m_DeltaNode[i]
            = (Output[i]-m_Node[i]) * DifferentialActivation(m_Node[i]);
      for(int i = 0 ; i < m_LayerOption.nNodeCnt ; ++i)
        m_Loss[i] = (Output[i]-m_Node[i])*(Output[i]-m_Node[i]);
    }
  }

  if(m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_INPUT)
  return;
}
```


KhuDaNetLayer.cpp (10)

```
void CKhuDaNetLayer::ComputeDeltaWeight(bool bReset) {
  if(m_LayerOption.nLayerType & KDN_LT_INPUT) return;
  if(bReset) {
    if(m_LayerOption.nLayerType & KDN_LT_FC) {
      if(m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_FC) {
        for(int i = 0 ; i < m_LayerOption.nNodeCnt ; ++i) {
          for(int j = 0 ; j < m_pBackwardLayer->m_LayerOption.nNodeCnt ; ++j )
            m_DeltaWeight[i][j] = 0;
          m_DeltaBias[i] = 0;
        }
      }
    }
  }
  if(m_LayerOption.nLayerType & KDN_LT_FC) {
    if(m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_FC) {
      for(int i = 0 ; i < m_LayerOption.nNodeCnt ; ++i) {
        for(int j = 0 ; j < m_pBackwardLayer->m_LayerOption.nNodeCnt ; ++j )
          m_DeltaWeight[i][j] += m_DeltaNode[i] * m_pBackwardLayer->m_Node[j];
        m_DeltaBias[i] += m_DeltaNode[i];
      }
    }
  }
}
```

KhuDaNetLayer.cpp (11)

```
void CKhuDaNetLayer::UpdateWeight(int nBatchSize) {
  if(m_LayerOption.nLayerType & KDN_LT_INPUT) return;

  if(m_LayerOption.nLayerType & KDN_LT_FC) {
    if(m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_FC) {
      for(int i = 0 ; i < m_LayerOption.nNodeCnt ; ++i) {
        for(int j = 0 ; j < m_pBackwardLayer->m_LayerOption.nNodeCnt ; ++j )
          m_Weight[i][j]
            += m_LayerOption.dLearningRate * m_DeltaWeight[i][j]/nBatchSize;

        m_Bias[i] += m_LayerOption.dLearningRate * m_DeltaBias[i]/nBatchSize;
      }
    }
  }
}
```


KhuDaNetLayer.cpp (12)

```
double CKhuDaNetLayer::GetLoss() {
  double Loss = 0;
  if((m_LayerOption.nLayerType & KDN_LT_OUTPUT) &&
     (m_LayerOption.nLayerType & KDN_LT_FC)) {
    for(int i = 0 ; i < m_LayerOption.nNodeCnt ; ++i)
      Loss += m_Loss[i];
  }
  return Loss;
}
```

MNIST (1)

- MNIST (modified national institute of standards and technology) database
- The MNIST Database of handwritten digits
  - http://yann.lecun.com/exdb/mnist/
  - Training set: 60,000, Test set: 10,000
  - Image file
    - [0]: 2051(0x00000803 – 08: unsigned char, 00000011: 2D),
    - [4]: 60000/10000, [8]: 28(rows), [12]: 28(columns),
    - [16]: pixel, raw data, row-wise, [0-255], unsigned char
  - Label file
    - [0]: 2049(0x00000801 – 08: unsigned char, 00000001: 1D),
    - [4]: 60000/10000,
    - [8]: label, [0-9], unsigned char

MNIST (2)

```
void CPerceptronTest::LoadMnistTrain() {
  char TrainImagePath[MAX_PATH], TrainLabelPath[MAX_PATH];

  sprintf(TrainImagePath, "%s\\train-images.idx3-ubyte", m_ExePath);
  sprintf(TrainLabelPath, "%s\\train-labels.idx1-ubyte", m_ExePath);

  FILE *fp = fopen(TrainImagePath, "rb");
  if(fp) {
    unsigned char Buf[28*28];
    fread(Buf, 1, 16, fp);
    int nCnt = 0;
    for(int i = 0 ; i < m_nMnistTrainTotal ; ++i) {
      fread(Buf, 1, 28*28, fp);

      for(int k = 0 ; k < 28*28 ; k++)
        m_MnistTrainInput[nCnt][k] = (double)Buf[k]/255.;
      nCnt++;
    }
    fclose(fp);
  }
```

MNIST (3)

```
  fp = fopen(TrainLabelPath, "rb");
  if(fp) {
    unsigned char Buf[32];
    fread(Buf, 1, 8, fp);

    int nCnt = 0;
    for(int i = 0 ; i < m_nMnistTrainTotal ; ++i) {
      fread(Buf, 1, 1, fp);
      m_MnistTrainOutput[nCnt] = Buf[0];
      nCnt++;
    }
    fclose(fp);
  }
}
```


CkhuGleGraphLayer

Main.cpp (1)

```
class CKhuGleGraphLayer : public CKhuGleLayer {
public:
  // double m_TrainAccuacy, m_TrainLoss;
};
```

Main.cpp (2)

```
class CPerceptronTest : public CKhuGleWin {
public:
  CKhuGleGraphLayer *m_pTrainGraphLayer, *m_pTestGraphLayer;
  CKhuDaNet m_Perceptron;
  bool m_bTrainingRun;

  char m_ExePath[MAX_PATH];
  int m_nBatchCnt, m_nEpochCnt, m_nBatch;
  int m_nMnistTrainTotal, m_nMnistTestTotal;

  double **m_MnistTrainInput, **m_MnistTestInput;
  int *m_MnistTrainOutput, *m_MnistTestOutput;

  CPerceptronTest(int nW, int nH, char *ExePath);
  ~CPerceptronTest();
  void LoadMnistTrain();
  void LoadMnistTest();
  void Update();
};
```


Main.cpp (3)

```
CPerceptronTest::CPerceptronTest(int nW, int nH, char *ExePath)
  : CKhuGleWin(nW, nH) {
  strcpy(m_ExePath, ExePath);

  m_pScene = new CKhuGleScene(640, 480, KG_COLOR_24_RGB(100, 100, 150));
  m_pTrainGraphLayer = new CKhuGleGraphLayer(600, 200,
    KG_COLOR_24_RGB(150, 150, 200),2,  CKgPoint(20, 30));
  m_pTrainGraphLayer->SetMaxData(0, 100.);
  m_pTrainGraphLayer->SetMaxData(1, 2.5);
  m_pScene->AddChild(m_pTrainGraphLayer);

  m_pTestGraphLayer = new CKhuGleGraphLayer(600, 200,
    KG_COLOR_24_RGB(150, 150, 200), 1, CKgPoint(20, 260));
  m_pTestGraphLayer->SetMaxData(0, 100.);
  m_pScene->AddChild(m_pTestGraphLayer);

  m_Perceptron.AddLayer(CKhuDaNetLayerOption(KDN_LT_INPUT | KDN_LT_FC,
    0, 28*28, 0, 0, 0, 0, 0.15));
  m_Perceptron.AddLayer(CKhuDaNetLayerOption(KDN_LT_FC | KDN_LT_OUTPUT,
    0, 1, 0, 0, 0, KDN_AF_SIGMOID, 0.15));

  m_Perceptron.InitWeight();
```

Main.cpp (4)

```
  m_nBatchCnt = 0;
  m_nEpochCnt = 0;
  m_nBatch = 100;
  m_nMnistTrainTotal = 60000;
  m_nMnistTestTotal = 10000;

  m_MnistTrainInput = m_MnistTestInput = nullptr;
  m_MnistTrainOutput = m_MnistTestOutput = nullptr;

  std::cout << m_Perceptron.GetInformation() << std::endl;
  int i;
  if(!m_MnistTrainInput){
    m_MnistTrainInput = new double *[m_nMnistTrainTotal];

    for(i = 0 ; i < m_nMnistTrainTotal ; i++)
      m_MnistTrainInput[i] = new double[28*28];
  }

  if(!m_MnistTrainOutput)
    m_MnistTrainOutput = new int [m_nMnistTrainTotal];
```


Main.cpp (5)

```
  if(!m_MnistTestInput){
    m_MnistTestInput = new double *[m_nMnistTestTotal];

    for(i = 0 ; i < m_nMnistTestTotal ; i++)
      m_MnistTestInput[i] = new double[28*28];
  }

  if(!m_MnistTestOutput)
    m_MnistTestOutput = new int [m_nMnistTestTotal];

  LoadMnistTrain();
  LoadMnistTest();

  m_bTrainingRun = false;
}
```

Main.cpp (6)

```
CPerceptronTest::~CPerceptronTest() {
  int i;
  if(m_MnistTrainInput){
    for(i = 0 ; i < m_nMnistTrainTotal ; i++)
      delete [] m_MnistTrainInput[i];

    delete [] m_MnistTrainInput;
  }
  if(m_MnistTrainOutput)
    delete [] m_MnistTrainOutput;

  if(m_MnistTestInput){
    for(i = 0 ; i < m_nMnistTestTotal ; i++)
      delete [] m_MnistTestInput[i];

    delete [] m_MnistTestInput;
  }
  if(m_MnistTestOutput)
    delete [] m_MnistTestOutput;
}
```


Main.cpp (7)

```
void CPerceptronTest::Update() {
  if(m_bKeyPressed['S']) {
    m_bTrainingRun = !m_bTrainingRun;
    m_bKeyPressed['S'] = false;
  }
  if(!m_bTrainingRun) {
    m_pScene->Render();
    DrawSceneTextPos("Perceptron Test", CKgPoint(0, 0));

    CKhuGleWin::Update();
    return;
  }
```

Main.cpp (8)

```
int nIndex = (m_nBatchCnt*m_nBatch)%m_nMnistTrainTotal;
if(nIndex+m_nBatch >= m_nMnistTrainTotal)
  nIndex = m_nMnistTrainTotal-m_nBatch;

int nOutputCnt = 1;
double **OutputList = new double*[m_nBatch];
for(int i = 0 ; i < m_nBatch ; ++i)
  OutputList[i] = new double[nOutputCnt];
for(int i = 0 ; i < m_nBatch ; ++i) {
  for(int j = 0 ; j < nOutputCnt ; ++j) {
    OutputList[i][j] = 0;
    if(m_MnistTrainOutput[nIndex+i] > 4) OutputList[i][j] = 1;
  }
}
```


Main.cpp (9)

```
  double Loss;
  int nTP = m_Perceptron.TrainBatch(m_MnistTrainInput+nIndex, OutputList,
    m_nBatch, &Loss);

  for(int i = 0 ; i < m_nBatch ; ++i)
    delete [] OutputList[i];
  delete [] OutputList;

  m_nBatchCnt++;

  char Msg[256];
  sprintf(Msg, "Train accuracy: %6.2lf, %5.3lf(batch index: %5d, \
total : %6d(%5.1lf), ep(%2d)",
    (double)nTP/(double)m_nBatch*100, Loss, m_nBatchCnt, nIndex+m_nBatch,
    (double)(nIndex+m_nBatch)/m_nMnistTrainTotal*100, m_nEpochCnt+1);
  std::cout << Msg << std::endl;
```

Main.cpp (10)

```
  if(nIndex+m_nBatch == m_nMnistTrainTotal) {
    m_nEpochCnt++;

    int nTP = 0;
    int i;
    for(i = 0 ; i < m_nMnistTestTotal ; i++) {
      int nResult = m_Perceptron.Forward(m_MnistTestInput[i]);
      if((m_MnistTestOutput[i]>4?1:0) == nResult) nTP++;
    }

    sprintf(Msg, "Test accuracy: %7.3lf\n",
      (double)nTP/(double)m_nMnistTestTotal*100.);
    std::cout << Msg << std::endl;

    m_pTestGraphLayer->m_Data[0].push_back
      ((double)nTP/(double)m_nMnistTestTotal*100);
    m_pTestGraphLayer->m_nCurrentCnt++;
    m_pTestGraphLayer->DrawBackgroundImage();
  }
```


Main.cpp (11)

```
  m_pTrainGraphLayer->m_Data[0].push_back((double)nTP/(double)m_nBatch*100);
  m_pTrainGraphLayer->m_Data[1].push_back(Loss);
  m_pTrainGraphLayer->m_nCurrentCnt++;
  m_pTrainGraphLayer->DrawBackgroundImage();

  m_pScene->Render();
  DrawSceneTextPos("Perceptron Test", CKgPoint(0, 0));

  CKhuGleWin::Update();
}

void CPerceptronTest::LoadMnistTrain() {
…
}

void CPerceptronTest::LoadMnistTest() {
…
}
```

Main.cpp (12)

```
int main() {
  char ExePath[MAX_PATH];
  GetModuleFileName(NULL, ExePath, MAX_PATH);

  int i;
  int LastBackSlash = -1;
  int nLen = strlen(ExePath);
  for(i = nLen-1 ; i >= 0 ; i--) {
    if(ExePath[i] == '\\') {
      LastBackSlash = i;
      break;
    }
  }
  if(LastBackSlash >= 0)
    ExePath[LastBackSlash] = '\0';

  CPerceptronTest *pPerceptronTest = new CPerceptronTest(640, 480, ExePath);

  KhuGleWinInit(pPerceptronTest);

  return 0;
}
```


Practice VIII

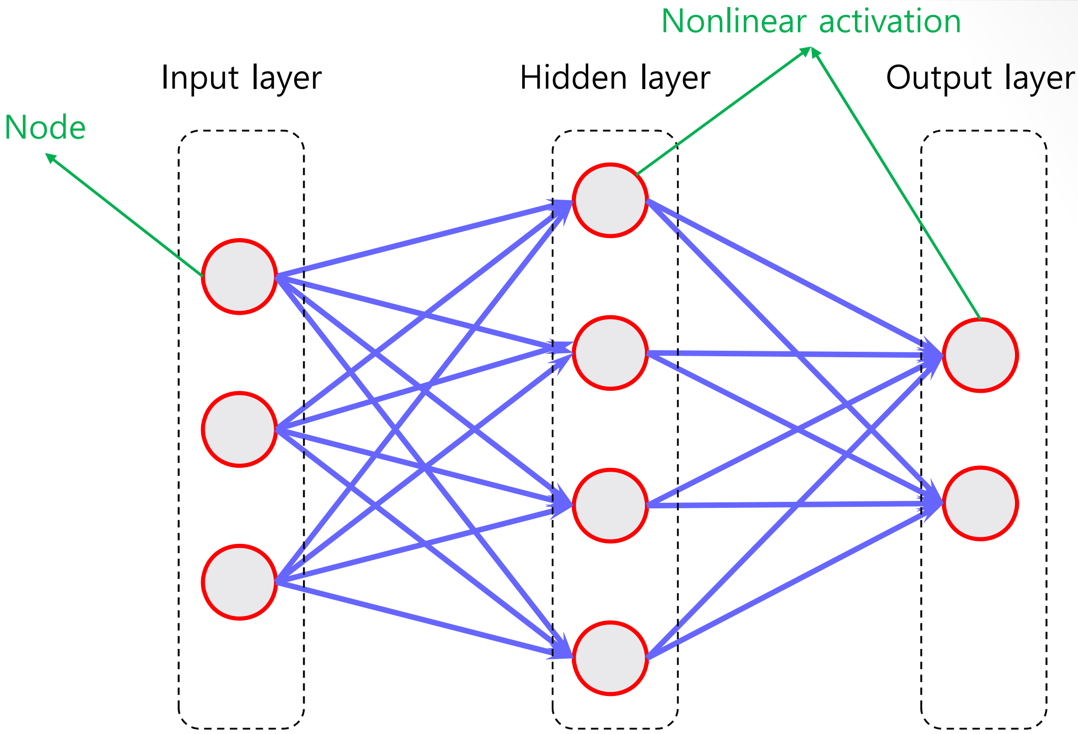
- Learning rate analysis

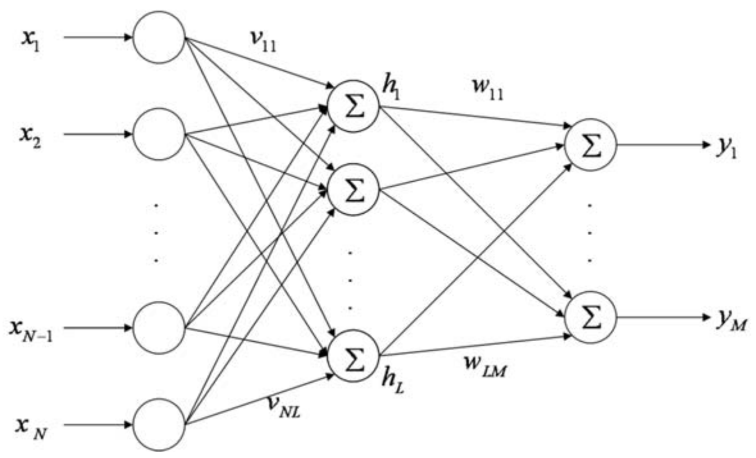
Advanced Courses

- Functional link network

11. MLP (DNN)


Ai and Data via Game
MLP (DNN) Test


Nonlinear activation
Input layer
Hidden layer
Output layer
Node



Loss function

- Function to be minimized
- Cost function, error function
- RMSE (root mean square error)
- Cross-entropy
  - Derivative
    - softmax – d
  - Minimizes the difference, d(x)

$$E(d,p) = -\sum_x d(x) \ln p(x)$$

Initial Weight Distribution

- Zero, (weight and bias)
- Uniform
- Xavier

$$N\left(0, \sqrt{\frac{2}{n_{in}+n_{out}}}\right)$$

$$U\left(-\sqrt{\frac{6}{n_{in}+n_{out}}}, \sqrt{\frac{6}{n_{in}+n_{out}}}\right)$$

- He
- Unsupervised learning

KhuDaNet.h

```
class CKhuDaNet {
  …
  static double TanH(double x);
  static double DifferentialTanH(double x);
  static double Relu(double x);
  static double DifferentialRelu(double x);
  static double LeakyRelu(double x);
  static double DifferentialLeakyRelu(double x);
  static double Softmax(double x);
  static double DifferentialSoftmax(double x);
};
```

KhuDaNet.cpp

```
double CKhuDaNet::TanH(double x) {
  return (exp(x)-exp(-x))/(exp(x)+exp(-x));
}
double CKhuDaNet::DifferentialTanH(double x) {
  return 1-x*x;
}
double CKhuDaNet::Relu(double x)  {
  return (x > 0)?x:0;
}
double CKhuDaNet::DifferentialRelu(double x) {
  return (x > 0)?1:0;
}
double CKhuDaNet::LeakyRelu(double x) {
  return (x > 0)?x:0.001*x;
}
double CKhuDaNet::DifferentialLeakyRelu(double x) {
  return (x > 0)?1:0.001;
}
double CKhuDaNet::Softmax(double x) {
  return x;
}
double CKhuDaNet::DifferentialSoftmax(double x) {
  return 1;
}
```


KhuDaNetLayer.h

```
#define KDN_LT_INPUT      0x0100
#define KDN_LT_HIDDEN     0x0200
#define KDN_LT_OUTPUT     0x0400

#define KDN_AF_NONE              0
#define KDN_AF_IDENTIFY          1
#define KDN_AF_BINARY_STEP       2
#define KDN_AF_SIGMOID           3
#define KDN_AF_TANH              4
#define KDN_AF_RELU              5
#define KDN_AF_LEAKY_RELU        6
#define KDN_AF_SOFTMAX           7
```

KhuDaNetLayer.cpp (1)

```
CKhuDaNetLayer::CKhuDaNetLayer(CKhuDaNetLayerOption m_LayerOptionInput,
  CKhuDaNetLayer *pBackwardLayerInput)
  : m_LayerOption(m_LayerOptionInput), m_bTrained(false) {
  …
  else if(m_LayerOption.nActicationFn == KDN_AF_TANH) {
    Activation = CKhuDaNet::TanH;
    DifferentialActivation = CKhuDaNet::DifferentialTanH;
  }
  else if(m_LayerOption.nActicationFn == KDN_AF_RELU) {
    Activation = CKhuDaNet::Relu;
    DifferentialActivation = CKhuDaNet::DifferentialRelu;
  }
  else if(m_LayerOption.nActicationFn == KDN_AF_LEAKY_RELU) {
    Activation = CKhuDaNet::LeakyRelu;
    DifferentialActivation = CKhuDaNet::DifferentialLeakyRelu;
  }
  else if(m_LayerOption.nActicationFn == KDN_AF_SOFTMAX) {
    Activation = CKhuDaNet::Softmax;
    DifferentialActivation = CKhuDaNet::DifferentialSoftmax;
  }
  …
```


KhuDaNetLayer.cpp (2)

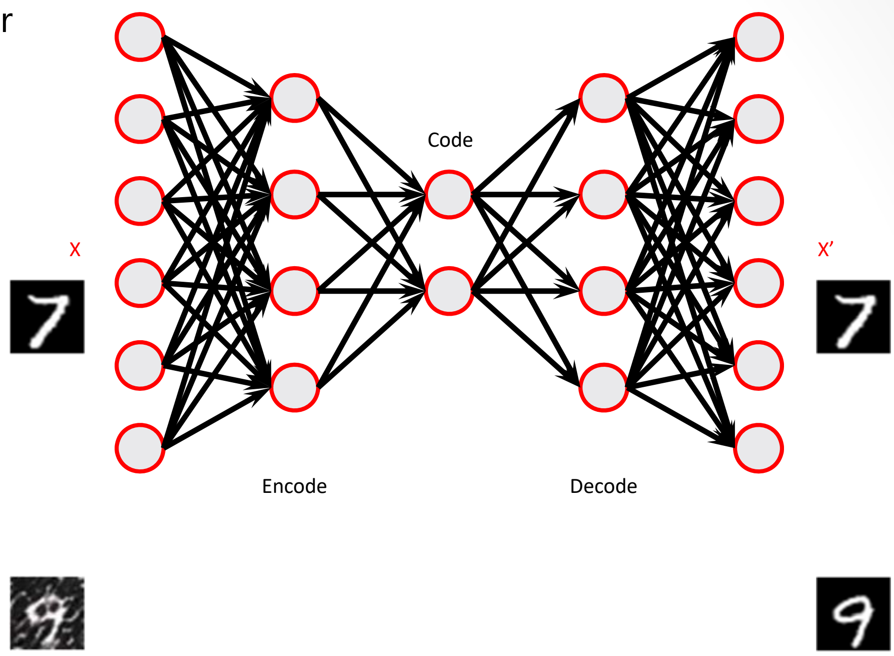
```
void CKhuDaNetLayer::ComputeDelta(double *Output) {
  if(m_LayerOption.nLayerType & KDN_LT_INPUT) return;
  if(m_LayerOption.nLayerType & KDN_LT_OUTPUT) {
    if(m_LayerOption.nLayerType & KDN_LT_FC) {
      if(m_LayerOption.nActicationFn == KDN_AF_SOFTMAX) {
        double Sum = 0;
        for(int i = 0 ; i < m_LayerOption.nNodeCnt ; ++i)
          Sum += m_Loss[i] = exp(m_Node[i]);
        for(int i = 0 ; i < m_LayerOption.nNodeCnt ; ++i)
          m_Loss[i] /= Sum;
        for(int i = 0 ; i < m_LayerOption.nNodeCnt ; ++i)
          m_DeltaNode[i]
            = (Output[i] - m_Loss[i]) * DifferentialActivation(m_Node[i]);
        for(int i = 0 ; i < m_LayerOption.nNodeCnt ; ++i)
          m_Loss[i] = -log(m_Loss[i])*Output[i];
      }
      else {
        …
     }
    }
  }
```

KhuDaNetLayer.cpp (2)

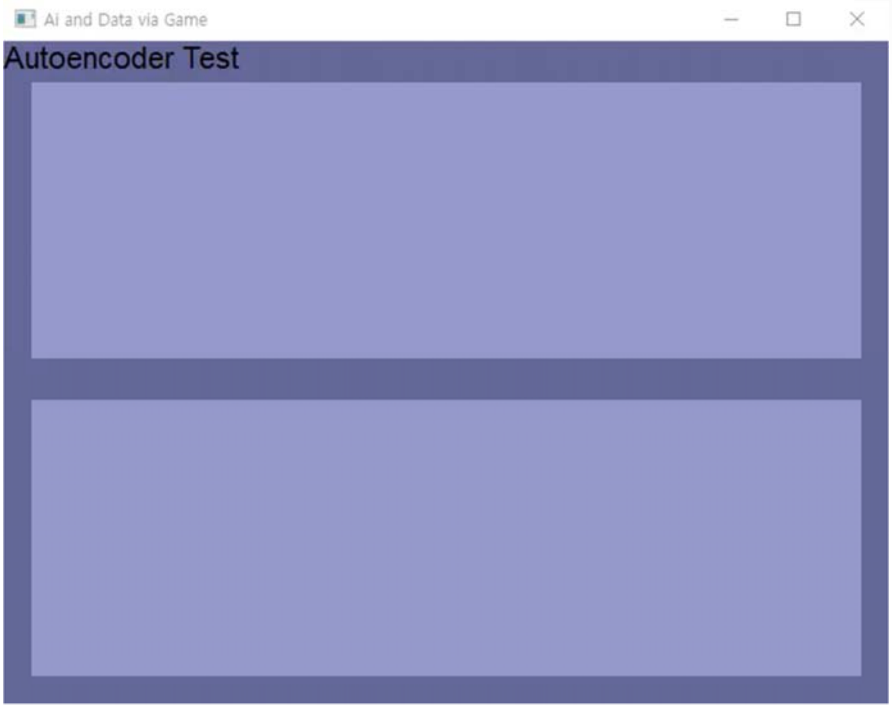
```
  if(m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_INPUT) return;

  if(m_LayerOption.nLayerType & KDN_LT_FC) {
    if(m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_FC) {
      for(int j = 0 ; j < m_pBackwardLayer->m_LayerOption.nNodeCnt ; ++j ) {
        m_pBackwardLayer->m_DeltaNode[j] = 0;
        for(int i = 0 ; i < m_LayerOption.nNodeCnt ; ++i)
          m_pBackwardLayer->m_DeltaNode[j] += m_DeltaNode[i] * m_Weight[i][j];

        m_pBackwardLayer->m_DeltaNode[j]
           *= m_pBackwardLayer->DifferentialActivation
                                       (m_pBackwardLayer->m_Node[j]);
      }
    }
  }
}
```


Practice IX (1)

- Auto encoder



Practice IX (2)



Advanced Courses (1)

- SIMD (single instruction multiple data)
- Streaming SIMD extensions (SSE)
- Advanced vector extensions (AVX)
  - AVX2: 256bit
  - AVX-512

Advanced Courses (2)

```
  double sum = 0;
#ifdef KHUDANET_AVX2
  double sum_4[4] = {0};
  __m256d acc;

  acc = _mm256_loadu_pd(sum_4);
  for(int j = 0 ; j < m_pBackwardLayer->m_LayerOption.nNodeCnt ; j+=4) {
    const __m256d a = _mm256_loadu_pd(m_pBackwardLayer->m_Node + j);
    const __m256d b = _mm256_loadu_pd(m_Weight[i]+j);
    acc = _mm256_fmadd_pd(a, b, acc);
  }
  _mm256_storeu_pd(sum_4, acc);
  sum = sum_4[0] + sum_4[1] + sum_4[2] + sum_4[3];
#elif
  for(int j = 0 ; j < m_pBackwardLayer->m_LayerOption.nNodeCnt ; ++j)
    sum += m_pBackwardLayer->m_Node[j] * m_Weight[i][j];
#endif
  m_Node[i] = Activation(sum + m_Bias[i]);
```


12. CNN

CNN (1)

- Convolutional neural networks (CNNs)
  - Convolution

188	170	195	205	215	234
168	192	208	215	210	211
196	186	220	193	195	193
201	235	192	187	201	198

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

	191				
				200	

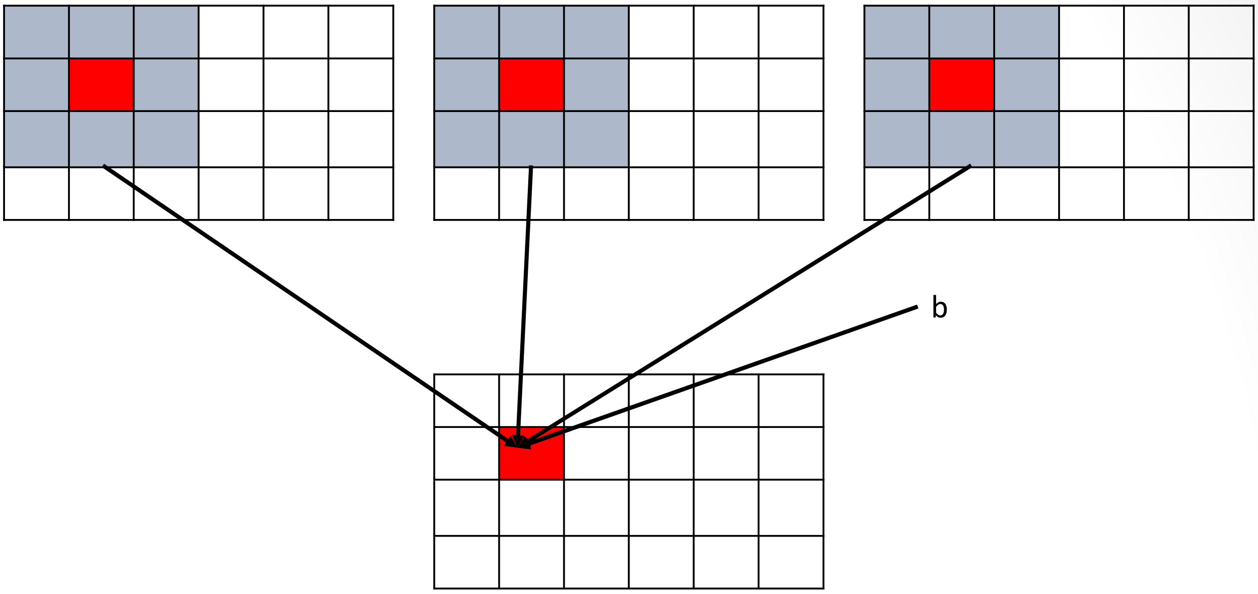
  - Pooling
    - Maximum value, mean value, …

188	170	195	205	215	234
168	192	208	215	210	211
196	186	220	193	195	193
201	235	192	187	201	198

192		

CNN (2)

- Convolutional neural networks (CNNs)
  - Convolution



CNN (3)

- Padding

188	170	195	205	215	234
168	192	208	215	210	211
196	186	220	193	195	193
201	235	192	187	201	198

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

191			
			200

0	0	0	0	0	0	0	0
0	188	170	195	205	215	234	0
0	168	192	208	215	210	211	0
0	196	186	220	193	195	193	0
0	201	235	192	187	201	198	0
0	0	0	0	0	0	0	0

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

	191				
				200	

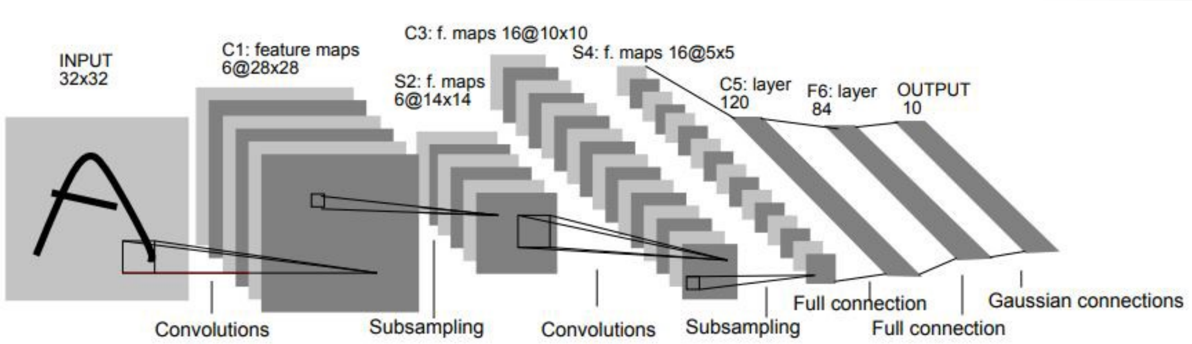
CNN (4)

- Stride

$$W_o = \frac{W_i + 2P - W_f}{S} + 1$$

Type	Size	Parameters
Input	32x32x3	0
Conv(5x5, s=1, p=0)	28x28x8	608 (5x5x3x8)+8
Pool	14x14x8	0
Conv(5x5, s=1, p=0)	10x10x16	3216 (5x5x8x16)+16
Pool	5x5x16	0
Dense	120x1	48120 (5x5x16x120+120)
Dense	84x1	10164 (120x84+84)
Output	10x1	850 (84x10+10)

LeNet-5



https://miro.medium.com/max/700/1*lvvWF48t7cyRWqct13eU0w.jpeg

KhuDaNet.cpp (1)

```
char *CKhuDaNet::GetInformation() {
  memset(m_Information, 32, MAX_INFORMATION_STRING_SIZE);

  m_Information[MAX_INFORMATION_STRING_SIZE-1] = 0;

  int nPos = 0;
  for(auto &Layer : m_Layers) {
    if(Layer->m_LayerOption.nLayerType & KDN_LT_FC) {
      sprintf(m_Information+nPos, "%s(%5d)  ", "FC",
        Layer->m_LayerOption.nNodeCnt);
      nPos += 10;
    }
    else if((Layer->m_LayerOption.nLayerType & KDN_LT_IMG) == KDN_LT_IMG) {
      sprintf(m_Information+nPos, "%s(%5d)  ", "IMG",
        Layer->m_LayerOption.nImageCnt);
      nPos += 11;
    }
```


KhuDaNet.cpp (2)

```
  else if(Layer->m_LayerOption.nLayerType & KDN_LT_CON) {
      sprintf(m_Information+nPos, "%s(%5d)  ", "CON",
        Layer->m_LayerOption.nImageCnt);
      nPos += 11;
    }
    else if(Layer->m_LayerOption.nLayerType & KDN_LT_POOL) {
      sprintf(m_Information+nPos, "%s(%5d)  ", "POOL",
        Layer->m_LayerOption.nImageCnt);
      nPos += 12;
    }
  }

  return m_Information;
}
```

KhuDaNet.cpp (3)

```
int CKhuDaNet::Forward(double *Input, double *Probability) {
  int MaxPos;

  for(auto &Layer : m_Layers) {
    if(Layer->m_LayerOption.nLayerType & KDN_LT_INPUT) {
      if(Layer->m_LayerOption.nLayerType & KDN_LT_FC)
        memcpy(Layer->m_Node, Input,
          Layer->m_LayerOption.nNodeCnt*sizeof(double));
```


KhuDaNet.cpp (4)

```
    else if((Layer->m_LayerOption.nLayerType & KDN_LT_CON) ||
      (Layer->m_LayerOption.nLayerType & KDN_LT_POOL)) {
      int nPadding = m_Layers[1]->m_LayerOption.nKernelSize/2;
      int nSequenceIndex = 0;
        for(int i = 0 ; i < Layer->m_LayerOption.nImageCnt ; ++i)
          for(int y = 0 ; y < Layer->m_LayerOption.nH ; ++y)
            for(int x = 0 ; x < Layer->m_LayerOption.nW ; ++x) {
              if(x < nPadding || x >= Layer->m_LayerOption.nW - nPadding ||
                y < nPadding || y >= Layer->m_LayerOption.nH - nPadding)
                Layer->m_NodeCnnImage[i][y][x] = 0;
              else
                Layer->m_NodeCnnImage[i][y][x] = Input[nSequenceIndex++];
            }
      }
    }
```

KhuDaNet.cpp (5)

```
    else if(Layer->m_LayerOption.nLayerType & KDN_LT_OUTPUT)
      MaxPos = Layer->ComputeLayer(Probability);
    else
      Layer->ComputeLayer();
  }

  return MaxPos;
}
```


KhuDaNet.cpp (6)

```
void CKhuDaNet::SaveKhuDaNet(char *Filename) {
  /* … */ CKhuDaNetLayer *pBackwardLayer = nullptr;
  for(auto &Layer : m_Layers) {
    if(pBackwardLayer) {
      if(Layer->m_LayerOption.nLayerType & KDN_LT_FC) {  …  }
      else if(Layer->m_LayerOption.nLayerType & KDN_LT_CON) {
        if((pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_CON) ||
          (pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_POOL))
          for(int i = 0 ; i < Layer->m_LayerOption.nImageCnt ; ++i)
            for(int j = 0 ; j < pBackwardLayer->m_LayerOption.nImageCnt ; ++j)
              fwrite(Layer->m_CnnWeight[i][j][0], sizeof(double),
                Layer->m_LayerOption.nKernelSize
                *Layer->m_LayerOption.nKernelSize,fp);
        fwrite(Layer->m_Bias, sizeof(double),Layer->m_LayerOption.nImageCnt,fp);
      }
    }
    pBackwardLayer = Layer;
  }
  fclose(fp);
}
```

KhuDaNet.cpp (7)

```
void CKhuDaNet::LoadKhuDaNet(char *Filename) {
  /* … */ CKhuDaNetLayer *pBackwardLayer = nullptr;
  for(int s = 0 ; s < nLayerCnt ; ++s) {
    if(pBackwardLayer) {
      if(m_Layers[s]->m_LayerOption.nLayerType & KDN_LT_FC) {  …   }
      else if(m_Layers[s]->m_LayerOption.nLayerType & KDN_LT_CON) {
        if((pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_CON) ||
          (pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_POOL))
          for(int i = 0 ; i < m_Layers[s]->m_LayerOption.nImageCnt ; ++i)
            for(int j = 0 ; j < pBackwardLayer->m_LayerOption.nImageCnt ; ++j)
              fread(m_Layers[s]->m_CnnWeight[i][j][0], sizeof(double),
                m_Layers[s]->m_LayerOption.nKernelSize
                *m_Layers[s]->m_LayerOption.nKernelSize, fp);
        fread(m_Layers[s]->m_Bias,
          sizeof(double), m_Layers[s]->m_LayerOption.nImageCnt, fp);
      }
    }
    pBackwardLayer = m_Layers[s];
  }
  fclose(fp);
}
```


KhuDaNetLayer.h

```
#define KDN_LT_FC           0x0001
#define KDN_LT_CON          0x0002
#define KDN_LT_POOL         0x0004
#define KDN_LT_IMG          0x0006
```

KhuDaNetLayer.cpp (1)

```
CKhuDaNetLayer::CKhuDaNetLayer(CKhuDaNetLayerOption m_LayerOptionInput,
  CKhuDaNetLayer *pBackwardLayerInput)
  : m_LayerOption(m_LayerOptionInput), m_bTrained(false) {
  …
  if((m_LayerOption.nLayerType & KDN_LT_INPUT) &&
    (m_LayerOption.nLayerType & KDN_LT_CON||
    m_LayerOption.nLayerType & KDN_LT_POOL))
    m_LayerOption.nNodeCnt =
      m_LayerOption.nW*m_LayerOption.nH*m_LayerOption.nImageCnt;

  if((m_LayerOption.nLayerType & KDN_LT_INPUT) &&
    (m_LayerOption.nLayerType & KDN_LT_FC)) {
    m_Node = new double [m_LayerOption.nNodeCnt];
  }
  else if((m_LayerOption.nLayerType & KDN_LT_INPUT) &&
    (m_LayerOption.nLayerType & KDN_LT_CON||
    m_LayerOption.nLayerType & KDN_LT_POOL)) {
    m_NodeCnnImage = new double **[m_LayerOption.nImageCnt];
    for(int i = 0 ; i < m_LayerOption.nImageCnt ; ++i)
      m_NodeCnnImage[i]
        = CKhuDaNet::dmatrix1d(m_LayerOption.nH, m_LayerOption.nW);
  }
```


KhuDaNetLayer.cpp (2)

```
  else if(m_LayerOption.nLayerType & KDN_LT_FC) {
    m_Node = new double [m_LayerOption.nNodeCnt];
    if(m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_FC)
      m_Weight = CKhuDaNet::dmatrix1d(m_LayerOption.nNodeCnt,
        m_pBackwardLayer->m_LayerOption.nNodeCnt);
    else if((m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_CON) ||
      (m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_POOL))
      m_Weight = CKhuDaNet::dmatrix1d(m_LayerOption.nNodeCnt,
        m_pBackwardLayer->m_LayerOption.nImageCnt
       *m_pBackwardLayer->m_LayerOption.nW
       *m_pBackwardLayer->m_LayerOption.nH);
        m_Bias = new double[m_LayerOption.nNodeCnt];
  }
```

KhuDaNetLayer.cpp (3)

```
  else if(m_LayerOption.nLayerType & KDN_LT_CON) {
    m_NodeCnnImage = new double **[m_LayerOption.nImageCnt];
    for(int i = 0 ; i < m_LayerOption.nImageCnt ; ++i)
      m_NodeCnnImage[i]=CKhuDaNet::dmatrix1d(m_LayerOption.nH,m_LayerOption.nW);

    m_CnnWeight = new double ***[m_LayerOption.nImageCnt];
    for(int i = 0 ; i < m_LayerOption.nImageCnt ; ++i) {
      m_CnnWeight[i] = new double **[m_pBackwardLayer->m_LayerOption.nImageCnt];
      for(int j = 0 ; j < m_pBackwardLayer->m_LayerOption.nImageCnt ; ++j)
        m_CnnWeight[i][j] = CKhuDaNet::dmatrix1d(m_LayerOption.nKernelSize,
          m_LayerOption.nKernelSize);
    }
    m_Bias = new double[m_LayerOption.nImageCnt];
  }
  else if(m_LayerOption.nLayerType & KDN_LT_POOL) {
    m_NodeCnnImage = new double **[m_LayerOption.nImageCnt];
    for(int i = 0 ; i < m_LayerOption.nImageCnt ; ++i)
      m_NodeCnnImage[i]=CKhuDaNet::dmatrix1d(m_LayerOption.nH,m_LayerOption.nW);
  }
}
```


KhuDaNetLayer.cpp (4)

```
CKhuDaNetLayer::~CKhuDaNetLayer() {
  …
  else if((m_LayerOption.nLayerType & KDN_LT_INPUT) &&
    (m_LayerOption.nLayerType & KDN_LT_CON||
    m_LayerOption.nLayerType & KDN_LT_POOL)) {
    for(int i = 0 ; i < m_LayerOption.nImageCnt ; ++i)
      CKhuDaNet::free_dmatrix1d(m_NodeCnnImage[i], m_LayerOption.nH,
        m_LayerOption.nW);
    delete [] m_NodeCnnImage;
  }
```

KhuDaNetLayer.cpp (5)

```
else if(m_LayerOption.nLayerType & KDN_LT_FC) {
  delete [] m_Node;
  if(m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_FC)
    CKhuDaNet::free_dmatrix1d(m_Weight, m_LayerOption.nNodeCnt,
      m_pBackwardLayer->m_LayerOption.nNodeCnt);
  else if((m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_CON) ||
         (m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_POOL))
    CKhuDaNet::free_dmatrix1d(m_Weight, m_LayerOption.nNodeCnt,
      m_pBackwardLayer->m_LayerOption.nImageCnt
      *m_pBackwardLayer->m_LayerOption.nW*m_pBackwardLayer->m_LayerOption.nH);
  delete [] m_Bias;
```


KhuDaNetLayer.cpp (6)

```
  if(m_bTrained) {
    delete [] m_DeltaNode;
    if(m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_FC)
      CKhuDaNet::free_dmatrix1d(m_DeltaWeight, m_LayerOption.nNodeCnt,
        m_pBackwardLayer->m_LayerOption.nNodeCnt);
    else if((m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_CON) ||
           (m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_POOL))
      CKhuDaNet::free_dmatrix1d(m_DeltaWeight, m_LayerOption.nNodeCnt,
        m_pBackwardLayer->m_LayerOption.nImageCnt*
        m_pBackwardLayer->m_LayerOption.nW*
        m_pBackwardLayer->m_LayerOption.nH);
    delete [] m_DeltaBias;
  }
}
```

KhuDaNetLayer.cpp (7)

```
else if(m_LayerOption.nLayerType & KDN_LT_CON) {
  for(int i = 0 ; i < m_LayerOption.nImageCnt ; ++i)
    CKhuDaNet::free_dmatrix1d(m_NodeCnnImage[i], m_LayerOption.nH,
      m_LayerOption.nW);
  delete [] m_NodeCnnImage;

  for(int i = 0 ; i < m_LayerOption.nImageCnt ; ++i) {
    for(int j = 0 ; j < m_pBackwardLayer->m_LayerOption.nImageCnt ; ++j)
      CKhuDaNet::free_dmatrix1d(m_CnnWeight[i][j], m_LayerOption.nKernelSize,
        m_LayerOption.nKernelSize);
    delete [] m_CnnWeight[i];
  }
  delete [] m_CnnWeight;

  delete [] m_Bias;
```


KhuDaNetLayer.cpp (8)

```
  if(m_bTrained) {
    for(int i = 0 ; i < m_LayerOption.nImageCnt ; ++i)
      CKhuDaNet::free_dmatrix1d(m_DeltaCnnImage[i], m_LayerOption.nH,
        m_LayerOption.nW);
    delete [] m_DeltaCnnImage;

    for(int i = 0 ; i < m_LayerOption.nImageCnt ; ++i) {
      for(int j = 0 ; j < m_pBackwardLayer->m_LayerOption.nImageCnt ; ++j)
        CKhuDaNet::free_dmatrix1d(m_DeltaCnnWeight[i][j],
          m_LayerOption.nKernelSize, m_LayerOption.nKernelSize);
      delete [] m_DeltaCnnWeight[i];
    }
    delete [] m_DeltaCnnWeight;

    delete [] m_DeltaBias;
  }
}
```

KhuDaNetLayer.cpp (9)

```
  else if(m_LayerOption.nLayerType & KDN_LT_POOL) {
    for(int i = 0 ; i < m_LayerOption.nImageCnt ; ++i)
      CKhuDaNet::free_dmatrix1d(m_NodeCnnImage[i], m_LayerOption.nH,
        m_LayerOption.nW);
    delete [] m_NodeCnnImage;

    if(m_bTrained) {
      for(int i = 0 ; i < m_LayerOption.nImageCnt ; ++i)
        CKhuDaNet::free_dmatrix1d(m_DeltaCnnImage[i], m_LayerOption.nH,
          m_LayerOption.nW);
      delete [] m_DeltaCnnImage;
    }
  }
}
```


KhuDaNetLayer.cpp (10)

```
void CKhuDaNetLayer::AllocDeltaWeight() {
  if((m_LayerOption.nLayerType & KDN_LT_OUTPUT) &&
    (m_LayerOption.nLayerType & KDN_LT_FC)){}
  if((m_LayerOption.nLayerType & KDN_LT_INPUT) &&
    (m_LayerOption.nLayerType & KDN_LT_FC)) {}
  else if((m_LayerOption.nLayerType & KDN_LT_INPUT) &&
    (m_LayerOption.nLayerType & KDN_LT_CON||
   m_LayerOption.nLayerType & KDN_LT_POOL)) {}
```

KhuDaNetLayer.cpp (11)

```
else if(m_LayerOption.nLayerType & KDN_LT_FC) {
  if(!m_bTrained) {
    m_DeltaNode = new double [m_LayerOption.nNodeCnt];
    if(m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_FC)
      m_DeltaWeight = CKhuDaNet::dmatrix1d(m_LayerOption.nNodeCnt,
                      m_pBackwardLayer->m_LayerOption.nNodeCnt);
    else if((m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_CON) ||
            (m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_POOL))
      m_DeltaWeight = CKhuDaNet::dmatrix1d(m_LayerOption.nNodeCnt,
         m_pBackwardLayer->m_LayerOption.nImageCnt*
         m_pBackwardLayer->m_LayerOption.nW*
         m_pBackwardLayer->m_LayerOption.nH);
    m_DeltaBias = new double[m_LayerOption.nNodeCnt];
  }
}
```


KhuDaNetLayer.cpp (12)

```
else if(m_LayerOption.nLayerType & KDN_LT_CON) {
  if(!m_bTrained) {
    m_DeltaCnnImage = new double **[m_LayerOption.nImageCnt];
    for(int i = 0 ; i < m_LayerOption.nImageCnt ; ++i)
      m_DeltaCnnImage[i]
        = CKhuDaNet::dmatrix1d(m_LayerOption.nH, m_LayerOption.nW);

    m_DeltaCnnWeight = new double ***[m_LayerOption.nImageCnt];
    for(int i = 0 ; i < m_LayerOption.nImageCnt ; ++i) {
      m_DeltaCnnWeight[i]
        = new double **[m_pBackwardLayer->m_LayerOption.nImageCnt];
      for(int j = 0 ; j < m_pBackwardLayer->m_LayerOption.nImageCnt ; ++j)
         m_DeltaCnnWeight[i][j] =
         CKhuDaNet::dmatrix1d(m_LayerOption.nKernelSize,
                        m_LayerOption.nKernelSize);
    }

    m_DeltaBias = new double[m_LayerOption.nImageCnt];
  }
}
```

KhuDaNetLayer.cpp (13)

```
  else if(m_LayerOption.nLayerType & KDN_LT_POOL) {
    if(!m_bTrained) {
      m_DeltaCnnImage = new double **[m_LayerOption.nImageCnt];
      for(int i = 0 ; i < m_LayerOption.nImageCnt ; ++i)
        m_DeltaCnnImage[i] = CKhuDaNet::dmatrix1d(m_LayerOption.nH,
                                                  m_LayerOption.nW);
    }
  }

  m_bTrained = true;
}
```


KhuDaNetLayer.cpp (14)

```
void CKhuDaNetLayer::InitWeight() {
  …

  if(m_LayerOption.nLayerType & KDN_LT_FC) {
    double var = 1;

    if(m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_FC)
      …
    else if((m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_CON) ||
           (m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_POOL))
      var = sqrt(2./(m_pBackwardLayer->m_LayerOption.nImageCnt*
        m_pBackwardLayer->m_LayerOption.nW*
        m_pBackwardLayer->m_LayerOption.nH + m_LayerOption.nNodeCnt));

    std::normal_distribution<double> distribution(0., var);
```

KhuDaNetLayer.cpp (15)

```
    for(int i = 0 ; i < m_LayerOption.nNodeCnt ; ++i) {
      if(m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_FC)
        …
      else if((m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_CON) ||
            (m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_POOL)) {
        int nSequenceIndex = 0;
        for(int j = 0 ; j < m_pBackwardLayer->m_LayerOption.nImageCnt ; ++j)
          for(int y = 0 ; y < m_pBackwardLayer->m_LayerOption.nH ; ++y)
            for(int x = 0 ; x < m_pBackwardLayer->m_LayerOption.nW ; ++x) {
              m_Weight[i][nSequenceIndex] = distribution(generator);

              ++nSequenceIndex;
             }
      }

      m_Bias[i] = 0;
    }
  }
```


KhuDaNetLayer.cpp (16)

```
  else if(m_LayerOption.nLayerType & KDN_LT_CON) {
    double var = sqrt(2./(m_LayerOption.nKernelSize *
      m_LayerOption.nKernelSize *
      (m_pBackwardLayer->m_LayerOption.nImageCnt + m_LayerOption.nImageCnt)));
    std::normal_distribution<double> distribution(0., var);

    for(int i = 0 ; i < m_LayerOption.nImageCnt ; ++i) {
      for(int j = 0 ; j < m_pBackwardLayer->m_LayerOption.nImageCnt ; ++j)
        for(int y = 0 ; y < m_LayerOption.nKernelSize ; ++y)
          for(int x = 0 ; x < m_LayerOption.nKernelSize ; ++x) {
            m_CnnWeight[i][j][y][x] = distribution(generator);
          }
      m_Bias[i] = 0;
    }
  }
}
```

KhuDaNetLayer.cpp (17)

```
int CKhuDaNetLayer::ComputeLayer(double *Probability) {
  …
  else if((m_LayerOption.nLayerType & KDN_LT_FC) &&
    ((m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_CON) ||
    (m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_POOL))) {
    for(int i = 0 ; i < m_LayerOption.nNodeCnt ; ++i) {
      double Sum = 0;

      int nSequenceIndex = 0;
      for(int j = 0 ; j < m_pBackwardLayer->m_LayerOption.nImageCnt ; ++j)
        for(int y = 0 ; y < m_pBackwardLayer->m_LayerOption.nH ; ++y)
          for(int x = 0 ; x < m_pBackwardLayer->m_LayerOption.nW ; ++x)
            Sum += m_pBackwardLayer->m_NodeCnnImage[j][y][x] *
                    m_Weight[i][nSequenceIndex++];

      m_Node[i] = Activation(Sum + m_Bias[i]);
    }
  }
```


KhuDaNetLayer.cpp (18)

```
  else if((m_LayerOption.nLayerType & KDN_LT_CON) &&
    ((m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_CON) ||
    (m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_POOL))) {
    for(int i = 0 ; i < m_LayerOption.nImageCnt ; ++i) {
      for(int y = 0 ; y < m_LayerOption.nH ; ++y)
        for(int x = 0 ; x < m_LayerOption.nW ; ++x) {
          double Sum = 0;
          for(int j = 0 ; j < m_pBackwardLayer->m_LayerOption.nImageCnt ; ++j)
            for(int dy = 0 ; dy < m_LayerOption.nKernelSize ; ++dy)
              for(int dx = 0 ; dx < m_LayerOption.nKernelSize ; ++dx)
                Sum +=
                  m_pBackwardLayer->m_NodeCnnImage[j][y+dy][x+dx]*
                  m_CnnWeight[i][j][dy][dx];

          m_NodeCnnImage[i][y][x]  = Activation(Sum + m_Bias[i]);
        }

    }
  }
```

KhuDaNetLayer.cpp (19)

```
  else if((m_LayerOption.nLayerType & KDN_LT_POOL) &&
    ((m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_CON) ||
    (m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_POOL))) {
    for(int i = 0 ; i < m_LayerOption.nImageCnt ; ++i) {
      for(int y = 0 ; y < m_LayerOption.nH ; ++y)
        for(int x = 0 ; x < m_LayerOption.nW ; ++x) {
          m_NodeCnnImage[i][y][x]
          = m_pBackwardLayer->m_NodeCnnImage[i][y*m_LayerOption.nKernelSize][x*m_LayerOption.nKernelSize];
          for(int py = y*m_LayerOption.nKernelSize ; py < (y+1)*m_LayerOption.nKernelSize ; ++py)
            for(int px = x*m_LayerOption.nKernelSize ; px < (x+1)*m_LayerOption.nKernelSize ; ++px) {
              if(m_NodeCnnImage[i][y][x] <
                   m_pBackwardLayer->m_NodeCnnImage[i][py][px])
                m_NodeCnnImage[i][y][x]
                  = m_pBackwardLayer->m_NodeCnnImage[i][py][px];
            }
        }
    }
  }
  …
}
```


KhuDaNetLayer.cpp (20)

```
void CKhuDaNetLayer::ComputeDelta(double *Output) {
  …
  if(m_LayerOption.nLayerType & KDN_LT_FC) {
    if(m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_FC) {  …   }
    else if((m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_CON) ||
      (m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_POOL)) {
      int nSequenceIndex = 0;
      for(int j = 0 ; j < m_pBackwardLayer->m_LayerOption.nImageCnt ; ++j)
        for(int y = 0 ; y < m_pBackwardLayer->m_LayerOption.nH ; ++y)
          for(int x = 0 ; x < m_pBackwardLayer->m_LayerOption.nW ; ++x) {
            double DeltaSum = 0;
            for(int i = 0 ; i < m_LayerOption.nNodeCnt ; i++)
              DeltaSum += m_DeltaNode[i] * m_Weight[i][nSequenceIndex];
            nSequenceIndex++;

            m_pBackwardLayer->m_DeltaCnnImage[j][y][x] =
               DeltaSum*m_pBackwardLayer->DifferentialActivation
                                    (m_pBackwardLayer->m_NodeCnnImage[j][y][x]);
          }
    }
  }
```

KhuDaNetLayer.cpp (21)

```
else if(m_LayerOption.nLayerType & KDN_LT_CON) {
  for(int j = 0 ; j < m_pBackwardLayer->m_LayerOption.nImageCnt ; ++j) {
    memset(m_pBackwardLayer->m_DeltaCnnImage[j][0], 0,
      m_pBackwardLayer->m_LayerOption.nW*
      m_pBackwardLayer->m_LayerOption.nH*sizeof(double));

    for(int i = 0 ; i < m_LayerOption.nImageCnt ; ++i)
      for(int y = 0 ; y < m_LayerOption.nH ; ++y)
        for(int x = 0 ; x < m_LayerOption.nW ; ++x)
          for(int dy = 0 ; dy < m_LayerOption.nKernelSize ; ++dy)
            for(int dx = 0 ; dx < m_LayerOption.nKernelSize ; ++dx) {
              m_pBackwardLayer->m_DeltaCnnImage[j][y+dy][x+dx] +=
              m_CnnWeight[i][j][dy][dx] * m_DeltaCnnImage[i][y][x];
            }

    for(int x = 0 ; x < m_pBackwardLayer->m_LayerOption.nW*m_pBackwardLayer->m_LayerOption.nH ; ++x)
      m_pBackwardLayer->m_DeltaCnnImage[j][0][x] *=
        DifferentialActivation(m_pBackwardLayer->m_NodeCnnImage[j][0][x]);
  }
}
```


KhuDaNetLayer.cpp (22)

```
else if(m_LayerOption.nLayerType & KDN_LT_POOL) {
  for(int j = 0 ; j < m_pBackwardLayer->m_LayerOption.nImageCnt ; ++j) {
    for(int y = 0 ; y < m_LayerOption.nH ; ++y)
      for(int x = 0 ; x < m_LayerOption.nW ; ++x) {
        int x0 = x*m_LayerOption.nKernelSize;
        int y0 = y*m_LayerOption.nKernelSize;
        for(int py = y*m_LayerOption.nKernelSize ; py < (y+1)*m_LayerOption.nKernelSize ; ++py)
          for(int px = x*m_LayerOption.nKernelSize ; px < (x+1)*m_LayerOption.nKernelSize ; ++px) {
            m_pBackwardLayer->m_DeltaCnnImage[j][py][px] = 0;
            if(m_pBackwardLayer->m_NodeCnnImage[j][py][px] >
               m_pBackwardLayer->m_NodeCnnImage[j][y0][x0]) {
              x0 = px;             y0 = py;
            }
          }
        m_pBackwardLayer->m_DeltaCnnImage[j][y0][x0]
          = m_DeltaCnnImage[j][y][x];
      }
  }
}
}
```

KhuDaNetLayer.cpp (23)

```
void CKhuDaNetLayer::ComputeDeltaWeight(bool bReset) {
  …
  if(bReset) {
    if(m_LayerOption.nLayerType & KDN_LT_FC) {
      if(m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_FC) {
        …
      }
      else if((m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_CON) ||
        (m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_POOL)) {
        for(int i = 0 ; i < m_LayerOption.nNodeCnt ; ++i) {
          int nSequenceIndex = 0;
          for(int j = 0 ; j < m_pBackwardLayer->m_LayerOption.nImageCnt ; ++j)
            for(int y = 0 ; y < m_pBackwardLayer->m_LayerOption.nH ; ++y)
              for(int x = 0 ; x < m_pBackwardLayer->m_LayerOption.nW ; ++x)
                m_DeltaWeight[i][nSequenceIndex++] = 0;

          m_DeltaBias[i] = 0;
        }
      }
    }
```


KhuDaNetLayer.cpp (24)

```
    else if(m_LayerOption.nLayerType & KDN_LT_CON) {
      for(int i = 0 ; i < m_LayerOption.nImageCnt ; ++i) {
        for(int j = 0 ; j < m_pBackwardLayer->m_LayerOption.nImageCnt ; ++j) {
          for(int dy = 0 ; dy < m_LayerOption.nKernelSize ; ++dy)
            for(int dx = 0 ; dx < m_LayerOption.nKernelSize ; ++dx)
              m_DeltaCnnWeight[i][j][dy][dx] = 0;
        }

        m_DeltaBias[i] = 0;
      }
    }
  }
```

KhuDaNetLayer.cpp (25)

```
  if(m_LayerOption.nLayerType & KDN_LT_FC) {
    if(m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_FC) {
      …
    }
    else if((m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_CON) ||
      (m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_POOL)) {
      for(int i = 0 ; i < m_LayerOption.nNodeCnt ; ++i) {
        int nSequenceIndex = 0;
        for(int j = 0 ; j < m_pBackwardLayer->m_LayerOption.nImageCnt ; ++j)
          for(int y = 0 ; y < m_pBackwardLayer->m_LayerOption.nH ; ++y)
            for(int x = 0 ; x < m_pBackwardLayer->m_LayerOption.nW ; ++x)
              m_DeltaWeight[i][nSequenceIndex++] += m_DeltaNode[i] *
                                  m_pBackwardLayer->m_NodeCnnImage[j][y][x];

        m_DeltaBias[i] += m_DeltaNode[i];
      }
    }
  }
```


KhuDaNetLayer.cpp (26)

```
  else if(m_LayerOption.nLayerType & KDN_LT_CON) {
    for(int i = 0 ; i < m_LayerOption.nImageCnt ; ++i) {
      for(int j = 0 ; j < m_pBackwardLayer->m_LayerOption.nImageCnt ; ++j)
        for(int dy = 0 ; dy < m_LayerOption.nKernelSize ; ++dy)
          for(int dx = 0 ; dx < m_LayerOption.nKernelSize ; ++dx)
            for(int y = 0 ; y < m_LayerOption.nH ; ++y)
              for(int x = 0 ; x < m_LayerOption.nW ; ++x) {
                m_DeltaCnnWeight[i][j][dy][dx] +=
                   m_pBackwardLayer->m_NodeCnnImage[j][dy+y][dx+x] *
                   m_DeltaCnnImage[i][y][x];
              }
    }

    for(int i = 0 ; i < m_LayerOption.nImageCnt ; ++i)
      for(int y = 0 ; y < m_LayerOption.nH ; ++y)
        for(int x = 0 ; x < m_LayerOption.nW ; ++x)
          m_DeltaBias[i] += m_DeltaCnnImage[i][y][x];
  }
}
```

KhuDaNetLayer.cpp (27)

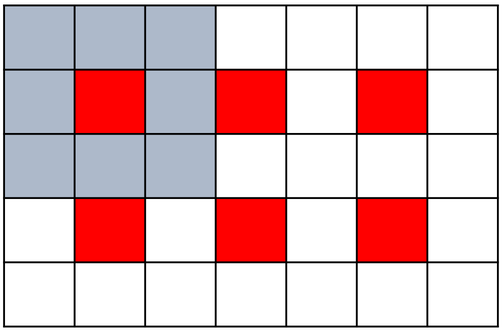
```
void CKhuDaNetLayer::UpdateWeight(int nBatchSize) {
  …
  if(m_LayerOption.nLayerType & KDN_LT_FC) {
    if(m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_FC) { … }
    else if((m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_CON) ||
      (m_pBackwardLayer->m_LayerOption.nLayerType & KDN_LT_POOL)) {
      for(int i = 0 ; i < m_LayerOption.nNodeCnt ; ++i) {
        for(int j = 0 ;
          j < m_pBackwardLayer->m_LayerOption.nImageCnt*
          m_pBackwardLayer->m_LayerOption.nW*
          m_pBackwardLayer->m_LayerOption.nH ; ++j)
           m_Weight[i][j] +=
              m_LayerOption.dLearningRate * m_DeltaWeight[i][j]/nBatchSize;

        m_Bias[i] += m_LayerOption.dLearningRate * m_DeltaBias[i]/nBatchSize;
      }
    }
  }
```


KhuDaNetLayer.cpp (28)

```
  else if(m_LayerOption.nLayerType & KDN_LT_CON) {
    for(int i = 0 ; i < m_LayerOption.nImageCnt ; ++i) {
      for(int j = 0 ; j < m_pBackwardLayer->m_LayerOption.nImageCnt ; ++j) {
        for(int dy = 0 ; dy < m_LayerOption.nKernelSize ; ++dy)
          for(int dx = 0 ; dx < m_LayerOption.nKernelSize ; ++dx) {
            m_CnnWeight[i][j][dy][dx] +=
              m_LayerOption.dLearningRate *
              m_DeltaCnnWeight[i][j][dy][dx]/nBatchSize;
          }
      }
    m_Bias[i] += m_LayerOption.dLearningRate * m_DeltaBias[i]/nBatchSize;
    }
  }
}
```

Practice X

- Convolutional layer
  - Stride



Type	Size	Parameters
Input	32x32x1	0
Conv(5x5, s=2, p=0)	14x14x6	156 (5x5x1x6)+6
Conv(5x5, s=2, p=0)	5x5x16	2416 (5x5x6x16)+16
Dense	120x1	48120 (5x5x16x120+120)
Dense	84x1	10164 (120x84+84)
Output	10x1	850 (84x10+10)

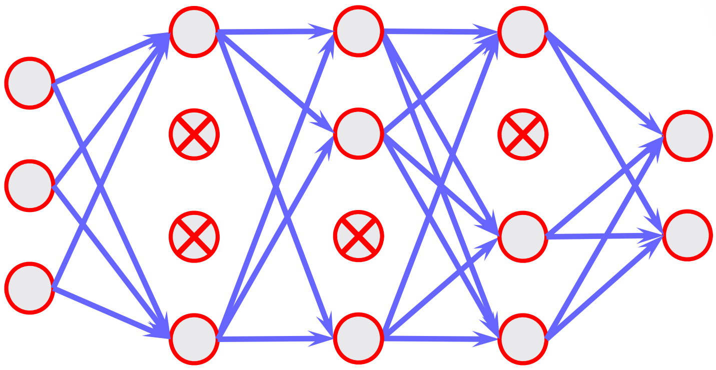
Advanced Courses (1)

- Regularization
  - Parameter regularization
    - Weights ↓ → Simple model -> regularization

$$L^{*} = L + \lambda \sum |w|$$

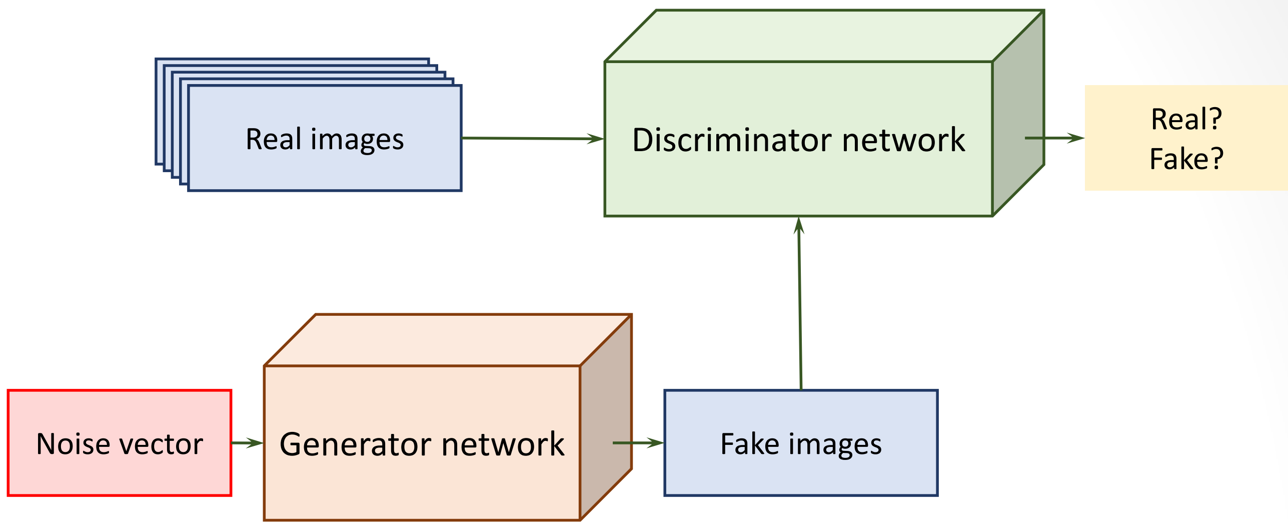
$$w \leftarrow (1-\lambda)w - \eta \frac{\partial L}{\partial w}$$

  - Data augmentation
  - Dropout



Advanced Courses (2)

- GAN (generative adversarial networks)



Project III

Deep learning analysis

Deep learning analysis

- Deep learning model design
  - DNN, CNN, GAN
- Initial weight analysis
- Optimizer analysis
- Regularization analysis