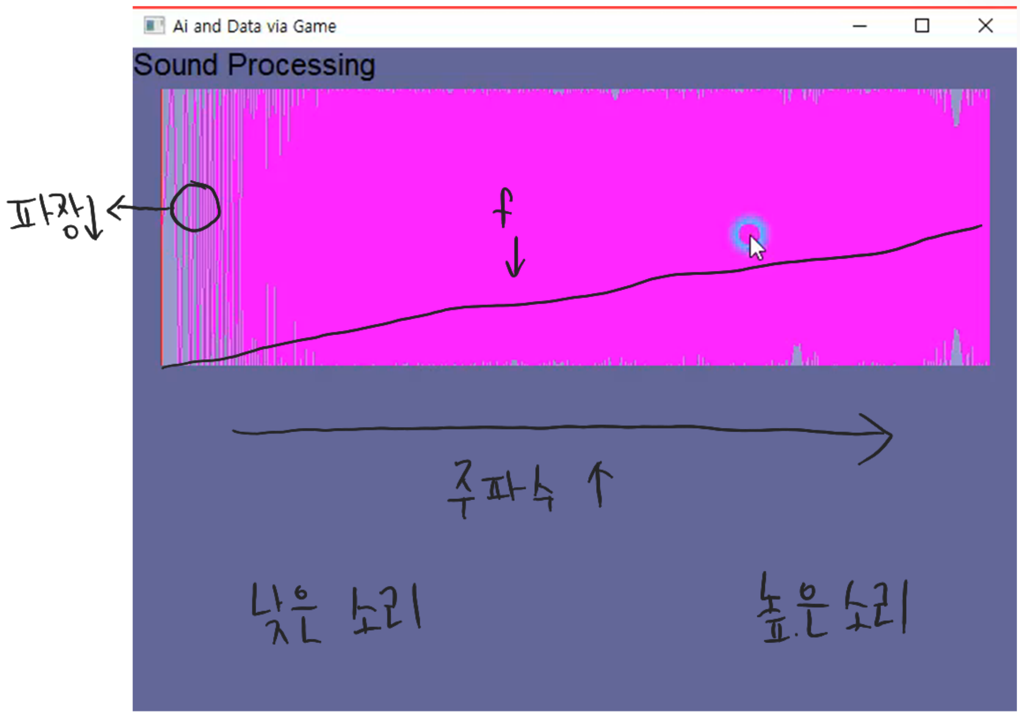
5. Sound Processing



Time and frequency domain

- 정현파(sinusoids), 주기(period), 주파수(frequency)

$$x(t) = X_M \sin \omega t$$

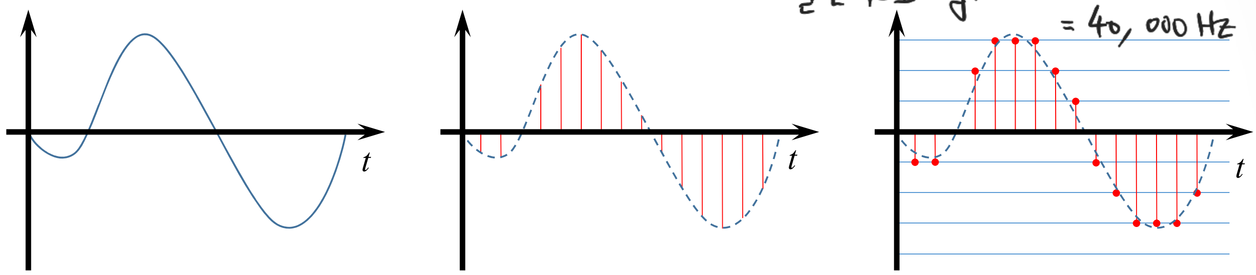
$$x\big(\omega(t+T)\big) = x(\omega t)$$

$$f = \frac{1}{T}$$

Hz 1/s

- 표본화 주파수(sampling frequency) 샘플링 진동수 사람의 가청 주파수 20,000 Hz
  - 표본화 (sampling) → 양자화 (quantization) → 부호화 (encoding)

일반적으로 signed int 샘플링 진동수 = 40,000 Hz



Time Domain 에서의 주기 신호를 (signal) 기본적인 주기 함수의 합으로 나타냄 sin , cos

Fourier Series

주기함수를 기본 주기 함수의 합으로 나타냄

$\frac{\pi}{2}$ 만큼 차이 남

$e^{j\theta} = \cos\theta + j\sin\theta$

실수부 허수부

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

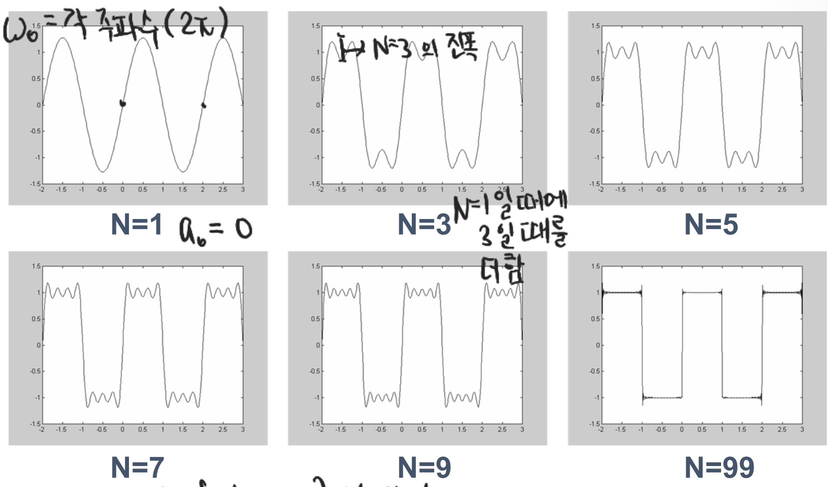
$$a_k = \frac{1}{T_0}\int_{T_0} f(t)e^{-jk\omega_0 t}dt$$

→ 주기함수의 성분(얘가 계수로 들어감)

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

$k\omega_0$ = f(t)의 주기 K = {R | R = {-∞, ∞}} ↳ 성분이 얼마만큼 포함되었는지 나타냄

ω_0 = 각 주파수(2π)



N = 2n {n | n ∈ N} 인 경우는 0.

→ 결론적으로 디지털화된 주기 함수를 구할 수 있음

f(t)가 주기함수일 때로 한정되어있음

Fourier Transform

$\lim_{T_0\to\infty} a_K$ 하면

일반적인 함수도 표현 가능

Continuous Time F.T

변환

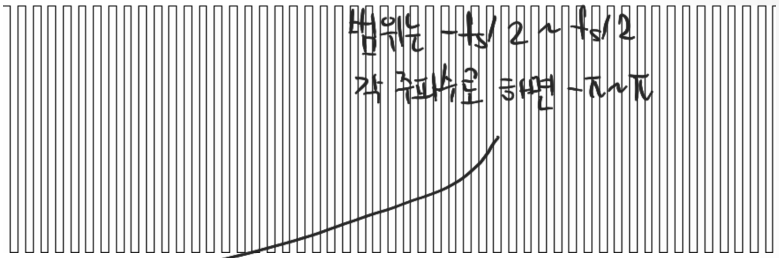
F Domain　Time Domain

$$F(\omega)=\int_{-\infty}^{\infty} f(t)e^{-j\omega t}dt$$ → Discrete면 시그마로 바뀜 F(ω)는 continuous

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

범위는 −fs/2 ~ fs/2

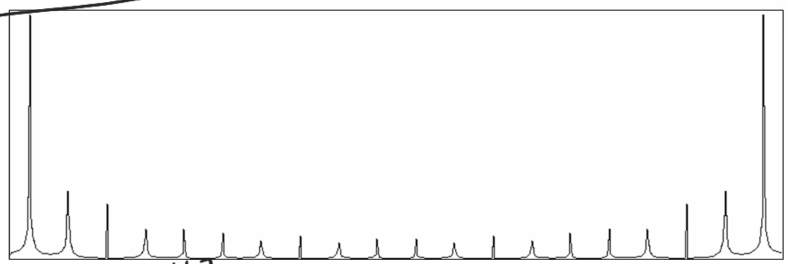
각 주파수로 하면 −π~π



→ f(n)

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

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



→ F(n)

주파수 커질수록 → $f_s/2$

Discrete 일때 010101···

↳ 최대 주파수 = Sampling Frequency/2

Fourier Transform Properties

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

Convolution

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

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

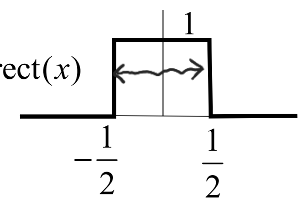
Time　Frequency

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

Convolution

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

→ 여기만 살아남음



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

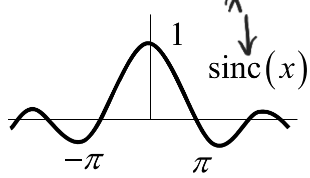
$$1\leftrightarrow 2\pi\delta(\omega)$$

F.T

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

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

$\frac{\sin x}{x}$

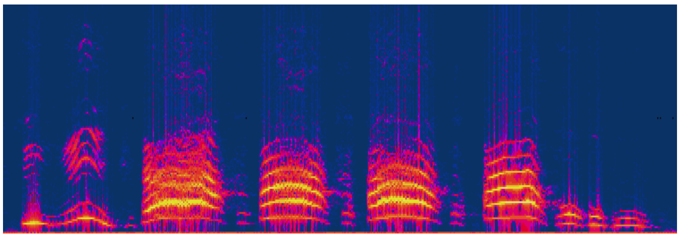


Spectrogram

= STFT

Short-time Fourier transform
Windowed Fourier transform

└> 부분적인 F.T 복소수를 크기와 각도 값으로 표현



└> 가짜 색상

FFT I

그냥 F.T는 너무 오래 걸림.

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

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

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

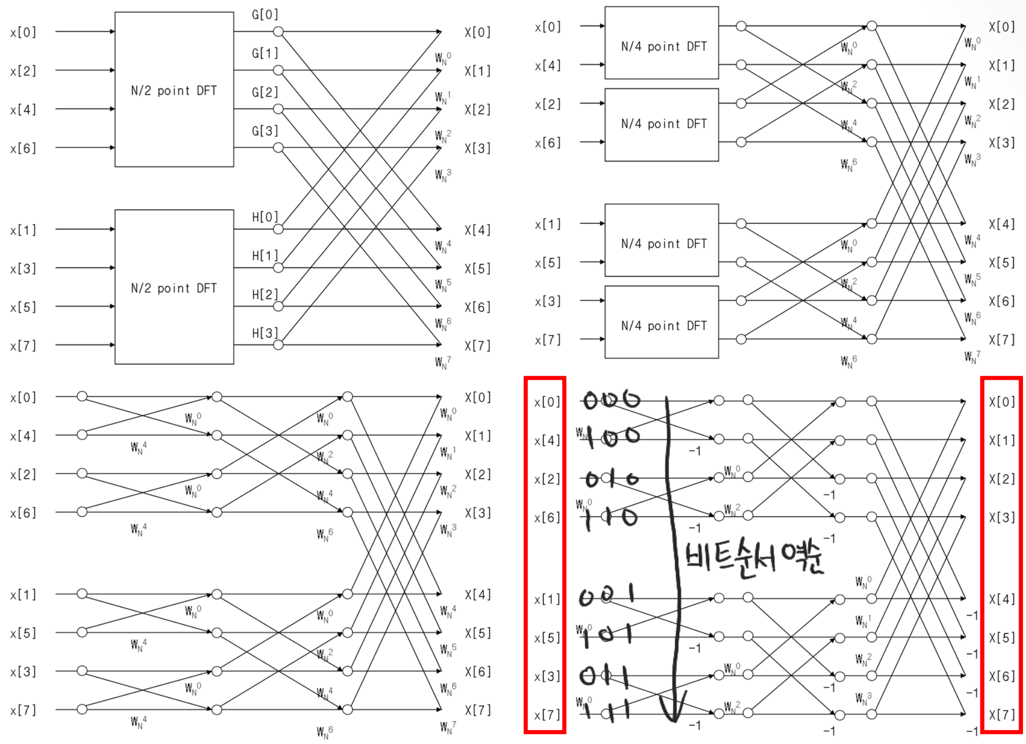
$$=\sum_{r=0}^{N/2-1} f(2r)\left(W_N^2\right)^{ru}+W_N^u\sum_{r=0}^{N/2-1} f(2r+1)\left(W_N^2\right)^{ru}$$

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

짝수 데이터 FT 홀수 데이터 FT

한번만 쪼개는게 아니라 계속 쪼갬

FFT II



$N^2 \rightarrow N\log_2 N$

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;
```

└→ DFT or IDFT

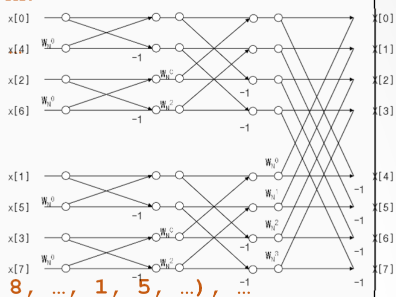
역순

16/2=8 <= j

0000				
1000	16/2			
0100	8-8	+0100		
1100	+1000			
0010	12-8	4-4	+0010	
1010	+1000			
0110	10-8	+0100		
1110	+1000			
0001	14-8	6-4	2-2	+0001
1001	+1000			
0101	9-8	+0100		
1101	+1000			
0011	13-8	5-4	+0010	
1011	+1000			
0111	11-8	+0100		
1111	+1000			

FFT IV

```
while(k > 0) {                              ①// iter: (1), (2), (3), …
  j = 0;
  j2 = 0;
  for(i = 0 ; i < nN >> 1 ; i++) {③
    Wr = cos(2.*Pi*(j2*k)/nN);
    if(bInverse == 0)
      Wi = -sin(2.*Pi*(j2*k)/nN); ⑤// j2: (0, 0, 0, …), (0, 0, 0, …, 1, 1, 1, …), …
    else
      Wi = sin(2.*Pi*(j2*k)/nN);    // j2*k/nN = j2*nN/2^iter/nN

    pos = j+(1 << (iter-1));        // (j+1), (j+2), (j+4), …
    Tr = Yr[pos] * Wr - Yi[pos] * Wi;         // Tr
    Ti = Yr[pos] * Wi + Yi[pos] * Wr;         // Ti

    Yr[pos] = Yr[j] - Tr;           // x[j] - T
    Yi[pos] = Yi[j] - Ti;

    Yr[j] += Tr;                    // x[j] = x[j] + T
    Yi[j] += Ti;
    j += 1 << iter;                ④// (0, 2, 4, …), (0, 4, 8, …, 1, 5, …), …
    if(j >= nN) j = ++j2;           //                              +j2
  }
  k >>= 1;                         ②// nN/2, nN/4, nN/8, nN/16
  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;
}
```


Wave & bitmap files

KhuGleSignal.h (1)

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

KhuGleSignal.h (2)

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

  double **m_Real, **m_Imaginary;
  int m_nWindowSize;
  int m_nFrequencySampleLength;    // Spectrogram에서 계산한 sample 수
                                  m_Real = dmatrix(m_nFrequencySampleLength, m_nWindowSize);
  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 (1)

```
#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 (2)

```
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(){}
```


Playing wave

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);

…
```


CKhuGleSoundLayer

Main.cpp (1)

```
class CKhuGleSoundLayer : public CKhuGleLayer {
public:
  CKhuGleSignal m_Sound;
  int m_nViewType;                  // 0: time, 1: time-frequency,
                                    // 2: time-frequency (log scale)

  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;
}
```

Main.cpp (2)

```
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;
    }
  }
```


Main.cpp (3)

```
  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;
      }
```

Main.cpp (4)

```
  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];
    }
}
```


Main.cpp (5)

```
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;
    }
```

Main.cpp (6)

```
    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];
      }
  }
}
```


Main.cpp (7)

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

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

Main.cpp (8)

```
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);
}
```


Main.cpp (9)

```
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;
  }
```

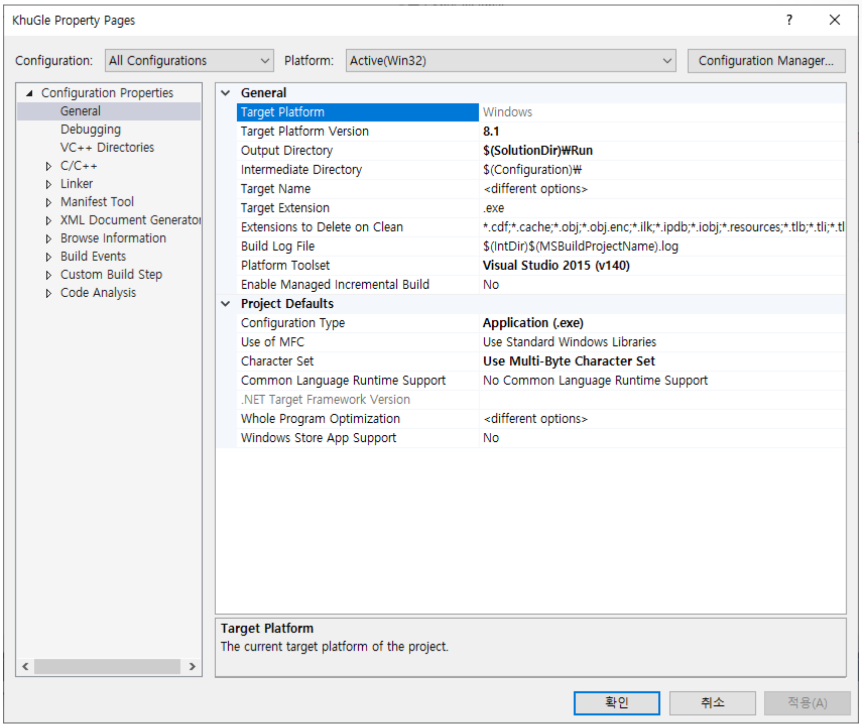
Main.cpp (10)

```
 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();
}
```


Main.cpp (11)

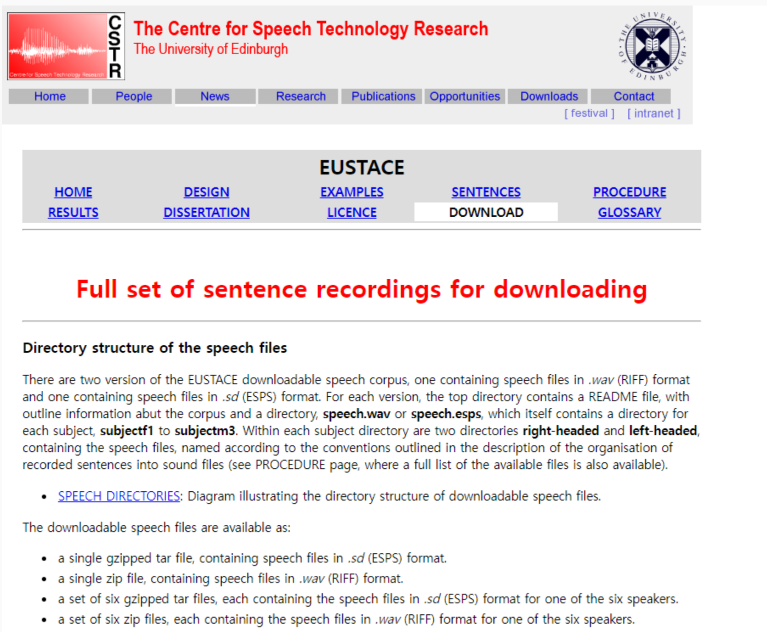
```
int main() {
  char ExePath[MAX_PATH], SoundPath[MAX_PATH];
  GetModuleFileName(NULL, ExePath, MAX_PATH);

  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;
}
```

Project setting



Speech data

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



FIR/IIR

- FIR
  - Finite impulse response

$$y[n] = b[0]x[n] + b[1]x[n-1] + \cdots + b[N-1]x[n-N+1]$$

- IIR
  - Infinite impulse response

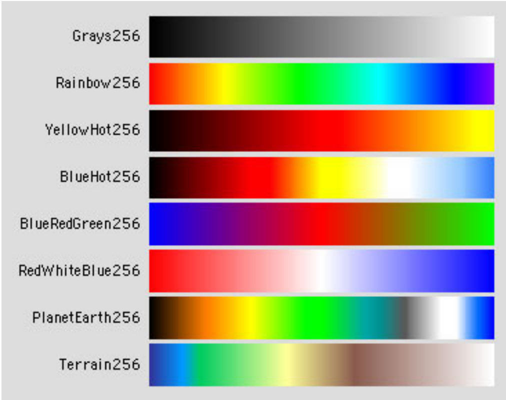
$$\begin{aligned} y[n] &= b[0]x[n] + b[1]x[n-1] + \cdots + b[N-1]x[n-N+1] \\ &\quad -a[1]y[n-1] + a[2]y[n-2] + \cdots + a[M]y[n-M] \end{aligned}$$

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

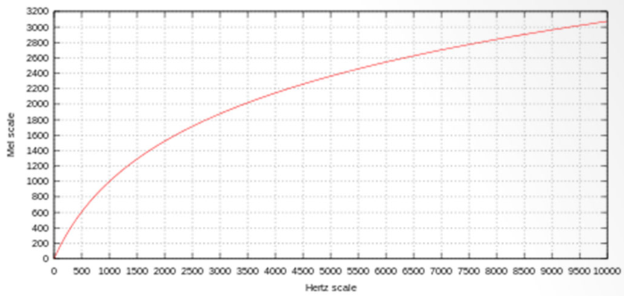
- Window types
  - Hamming window

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

Advanced Courses (2)

- Sound features
  - Short time energy

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



https://upload.wikimedia.org/wikipedia/commons/thumb/a/aa/Mel-Hz_plot.svg/450px-Mel-Hz_plot.svg.png

  - 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