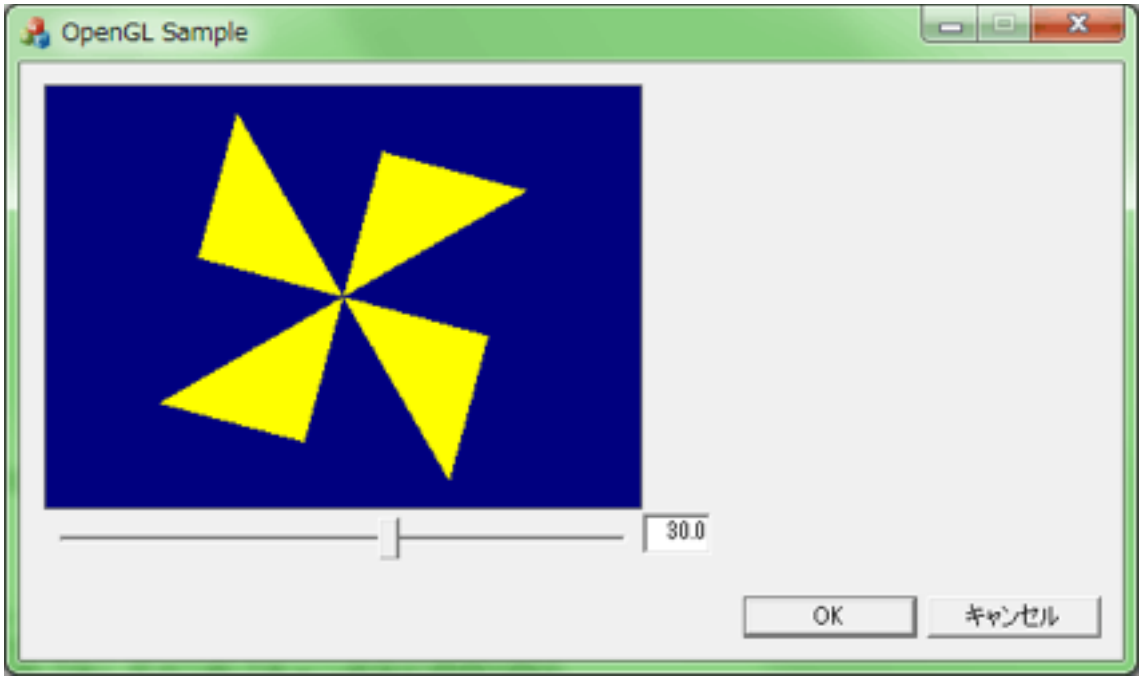
MFCによる
ダイアログベースの
OpenGLアプリケーション

MFC を初めて使ったマカーによる信頼性のないメモ

床井浩平
和歌山大学・システム工学部・デザイン情報学科・2014年10月18日



MFCによる
ダイアログベースの
OpenGLアプリケーション

MFC を初めて使ったマカーによる信頼性のないメモ

床井浩平
和歌山大学・システム工学部・デザイン情報学科・2014年10月18日

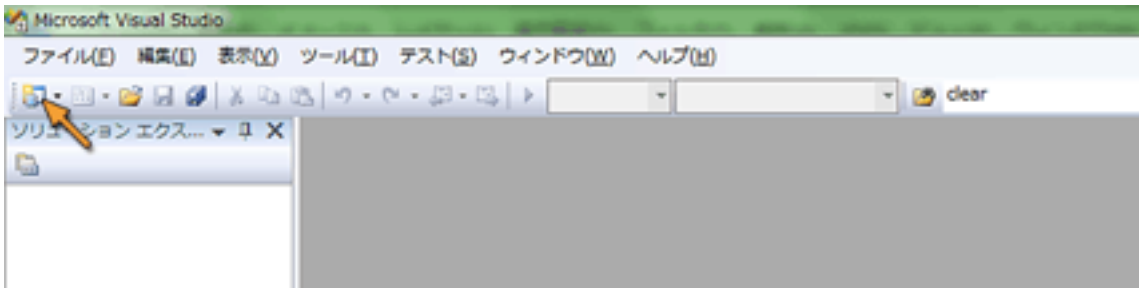
はじめに

MFC って実は私，使ったことないんです．それに，いまさら MFC というのもなんだかなあとは思います．実際，某所で UI 用のツールキットって何がいいかなって聞いてみたら Qt 一択みたいに教えてもらいました．でも私，自分の Linux マシンから KDE 関係排除しようとして Qt も全部消しちゃったくらい Qt には近づかんとこうとしてました．んで，今回なぜか MFC を使っちゃったんですね．いろいろ事情もあったんですけど（でもその目論見は失敗したのですけど)．で，そのプログラムを以前に私自身が FLTK 使えばってアドバイスした学生さんに送っちゃったから，罪滅ぼしに勉強した MFC の使い方をメモっておきます．もちろん，間違ってるところがあると思います．

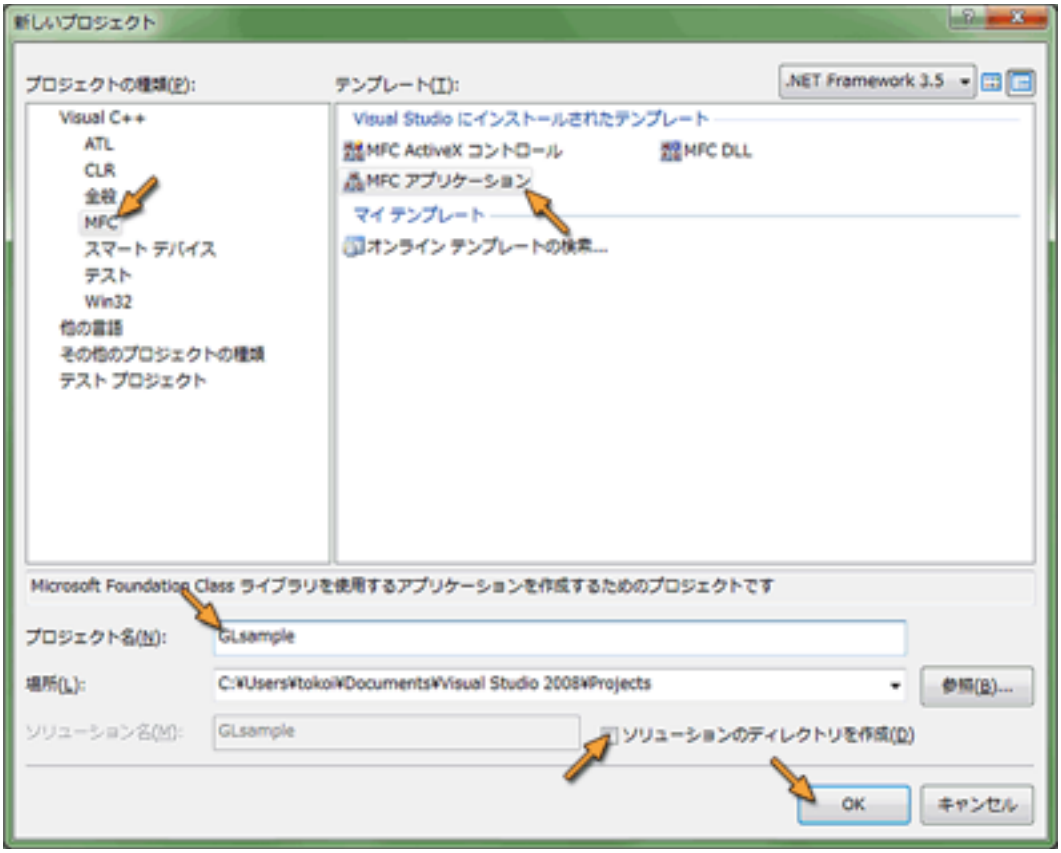
このメモを書く際に「数学と計算」さま (http://mail2.nara-edu.ac.jp/%7Easait/) と「【G.Ishihara流 】Visual C++ (MFC)超入門」さま (http://www.g-ishihara.com/) さまを参考にさせていただきました．ありがとうございました．本当に勉強したい場合は，上記のサイトをご覧頂くことを強くお勧めします．

プロジェクトの作成

Visual Studio を起動して, 新しいプロジェクトを作成します.



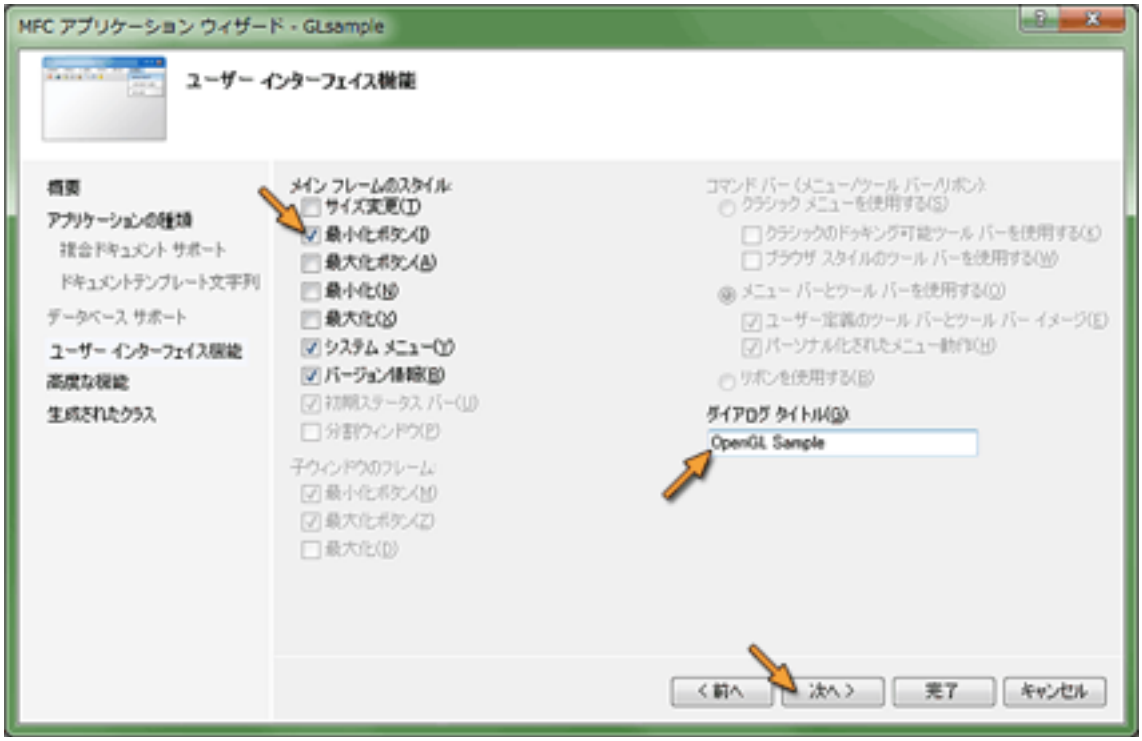
プロジェクトの種類は MFC アプリケーションにします. 「プロジェクト名」を設定した後「OK」をクリックしてください. 「ソリューションのディレクトリ」は作成するまでもないでしょう.



MFC アプリケーションウィザードが起動したら「次へ >」をクリックして先に進みます.

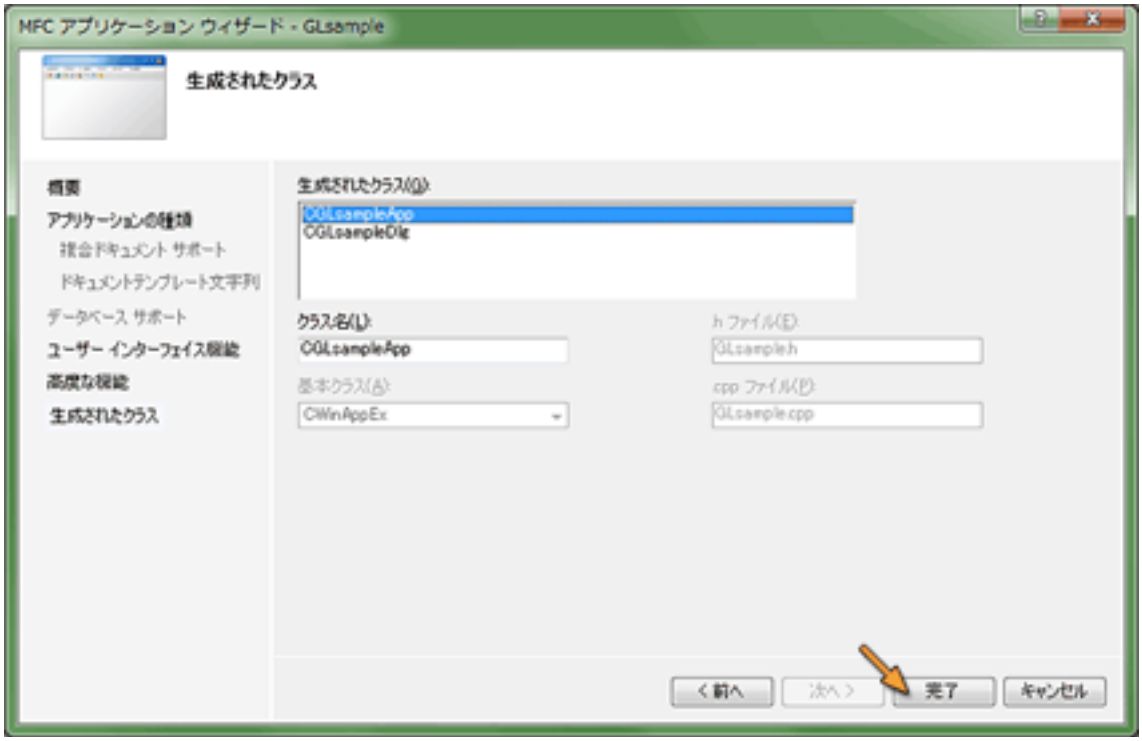
アプリケーションの種類は「ダイアログベース」にします．これが一番簡単みたいです．

ユーザインタフェース機能は「最小化ボタン」だけ付けておくことにします． サイズ変更や最大化ができるようにすると，後の処理が少し面倒になります．例によって手抜きですね．ウィンドウを隠すために「最小化ボタン」は付けておきます．あと，ウィンドウのタイトルバーなどに表示する「ダイアログタイトル」も設定します．



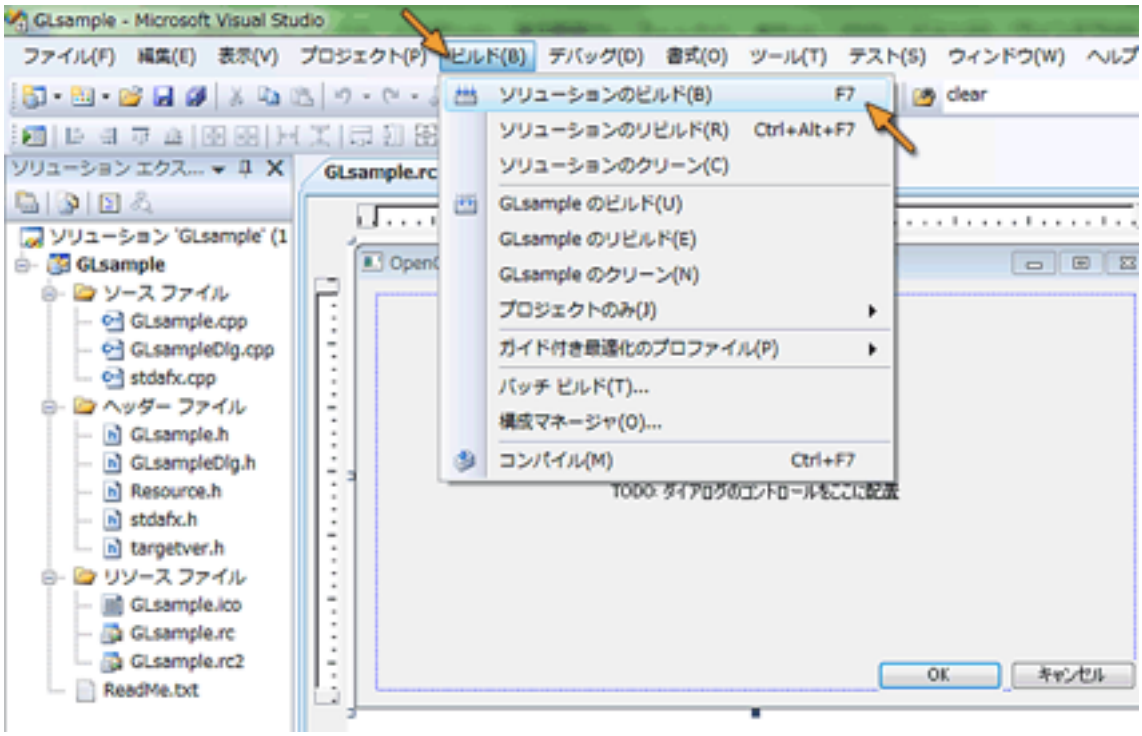
「高度な機能」は一切使わないので, チェックボックスを全部外します.

これで終わりです. 「Cプロジェクト名App」と「Cプロジェクト名Dlg」の二つのクラスができていると思います. 「完了」をクリックします.

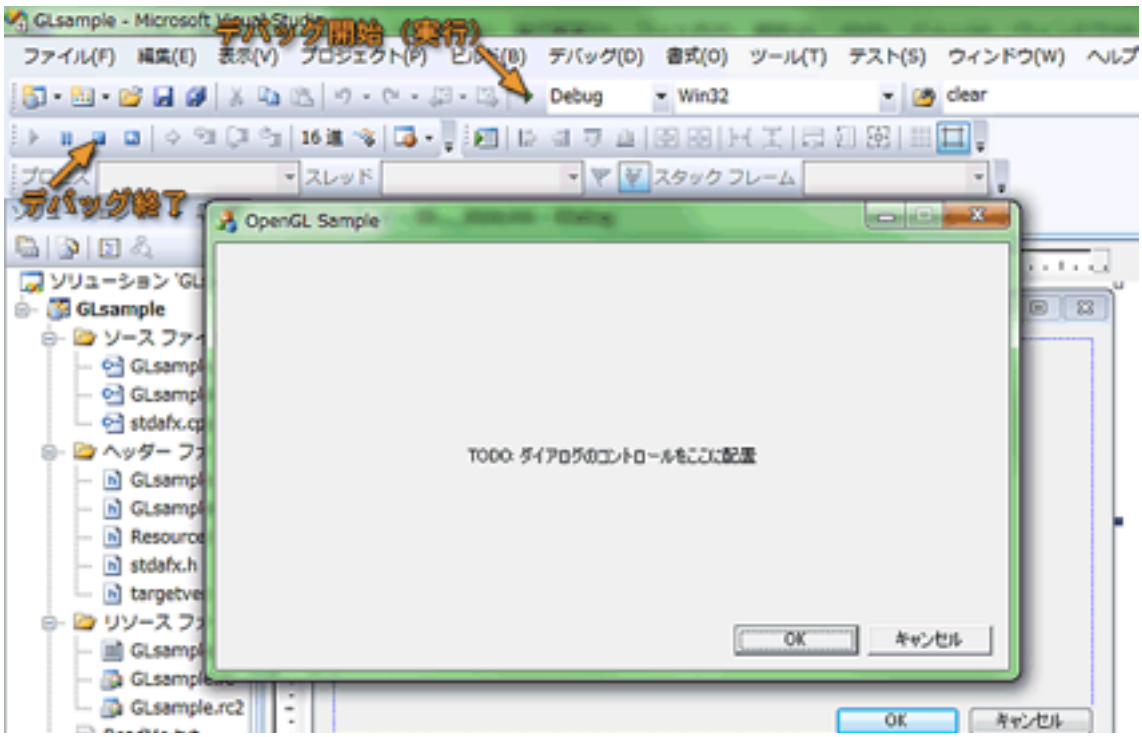

MFC アプリケーション ウィザード - GLsample
生成されたクラス
概要
アプリケーションの種類
複合ドキュメント サポート
ドキュメントテンプレート文字列
データベース サポート
ユーザー インターフェイス機能
高度な機能
生成されたクラス
生成されたクラス(G):
CGLsampleApp
CGLsampleDlg
クラス名(L):
CGLsampleApp
基本クラス(A):
CWinAppEx
h ファイル(E):
GLsample.h
cpp ファイル(P):
GLsample.cpp
< 前へ
次へ >
完了
キャンセル

ソリューションのビルド

試しに, ここで一度アプリケーションをビルドしてみます. 「ビルド」メニューから「ソリューションのビルド」を選ぶか [F7] をタイプします.

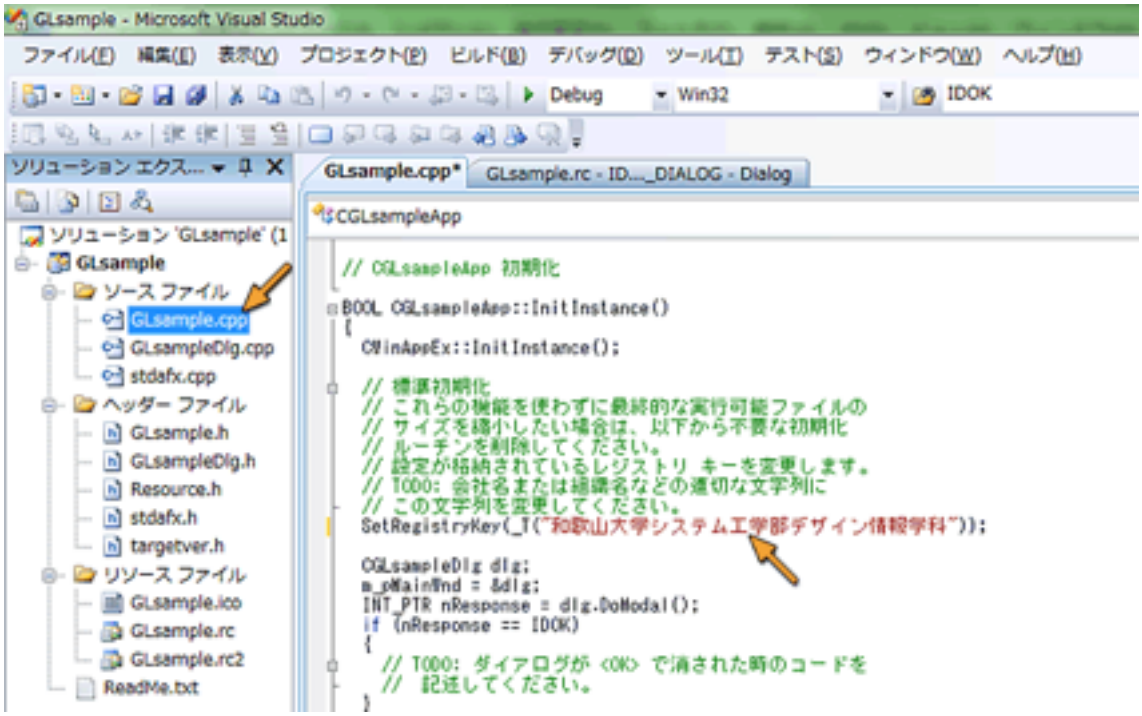


コンパイルエラーが出なければ, [F5] をタイプするなどして実行してみます.

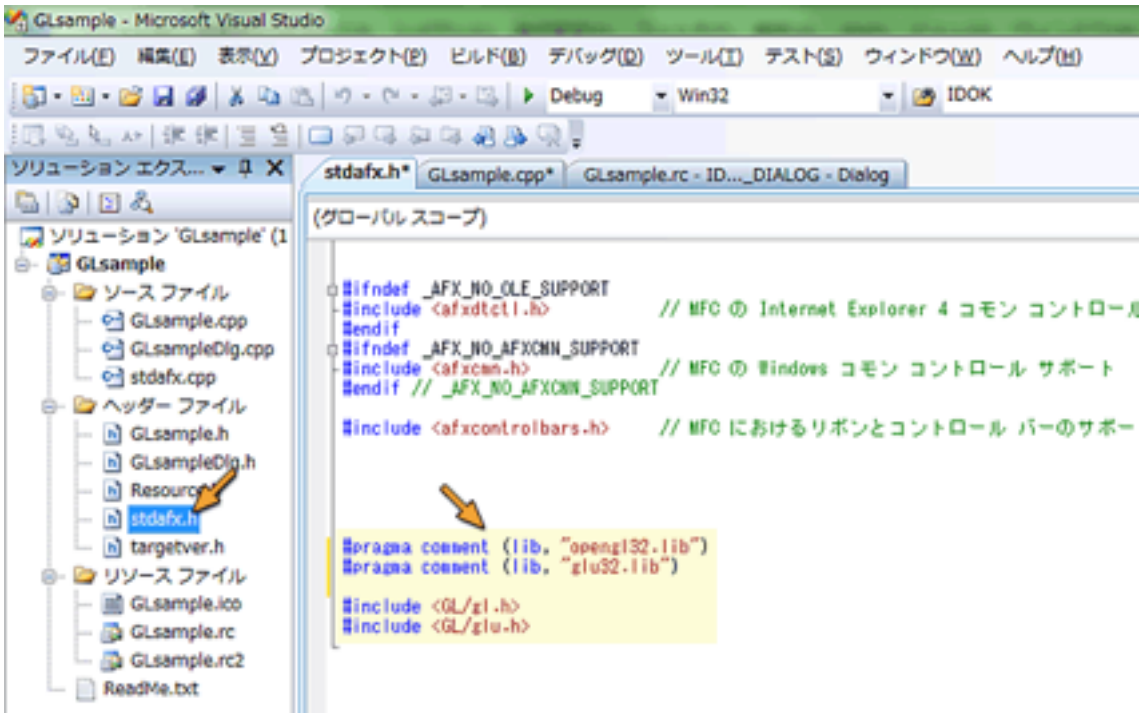


プログラムの変更

プログラムが正常に実行できるようなら，このコードに変更を加えていきます．最初に「プロジェクト名Dlg.cpp」ファイル（ここでは GLsampleDlg.cpp）の初期化のところで SetRegistryKey() の引数を変更します．これは別にどうでもいいんでしょうね．



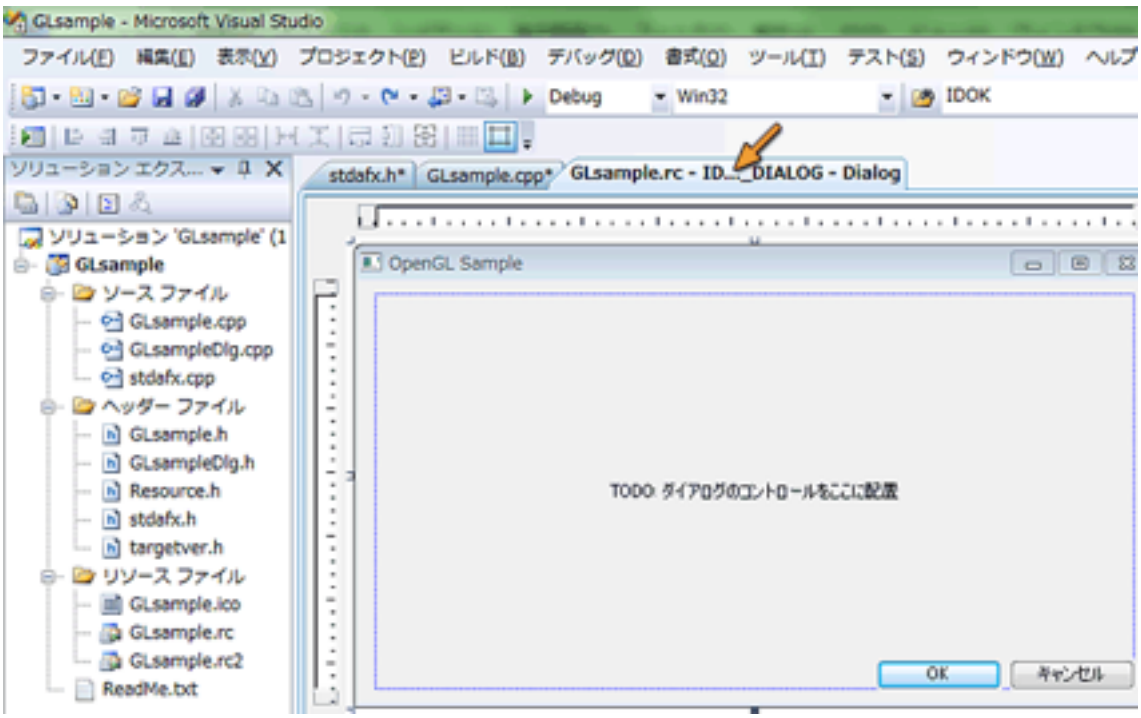
OpenGL を使うので，ヘッダファイルやライブラリの指定を stdafx.h の最後に追加します．



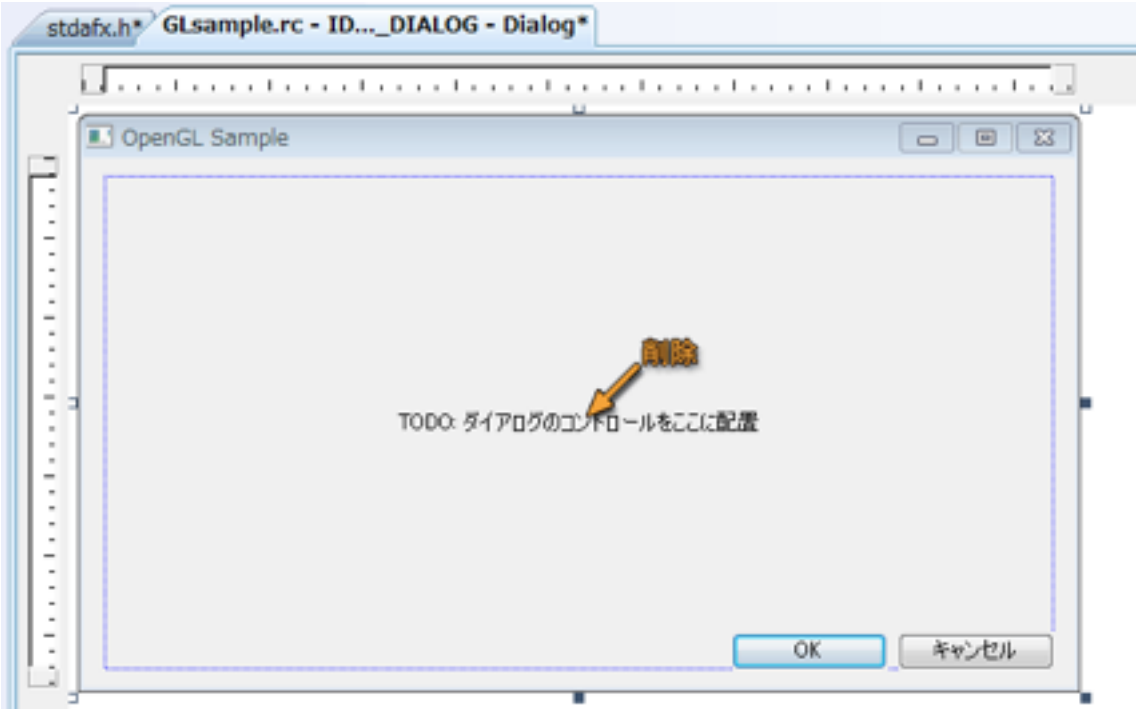
GLEW やそのほかの標準ライブラリのヘッダファイルを #include する場合も，ここに書けばいいんじゃないでしょうか．

```
#pragma comment (lib, "opengl32.lib")
#pragma comment (lib, "glu32.lib")

#include <GL/gl.h>
#include <GL/glu.h>
```

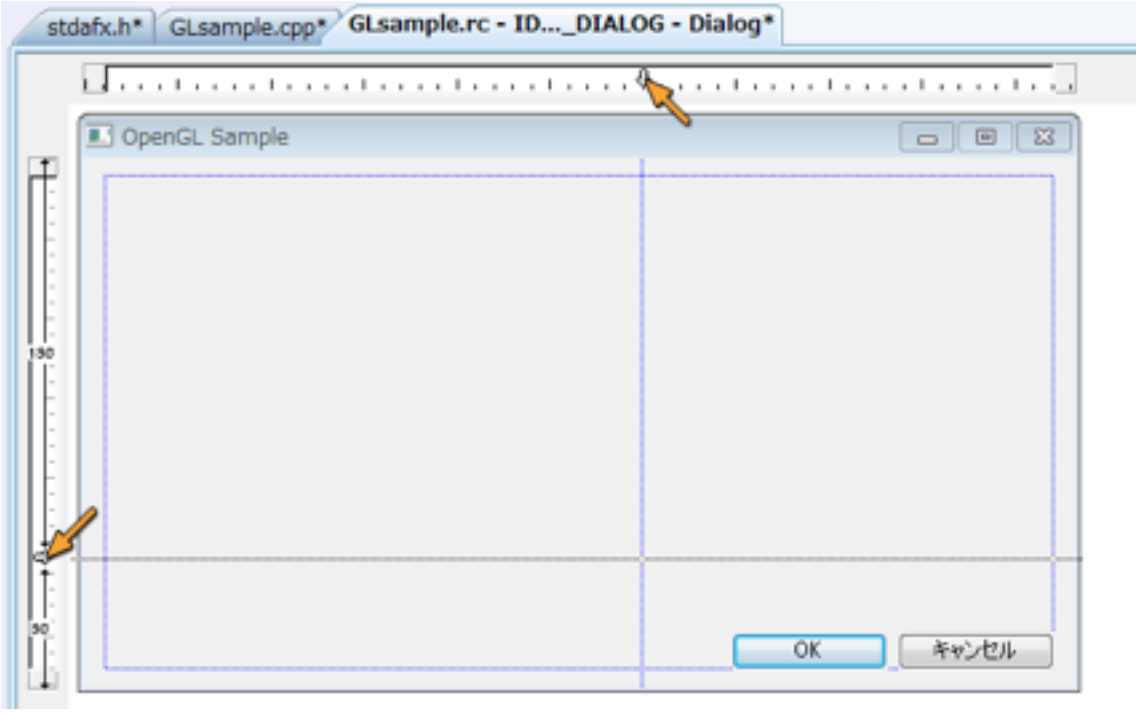
ソースコードの上のタブをクリックして, ダイアログエディタに切り替えます.



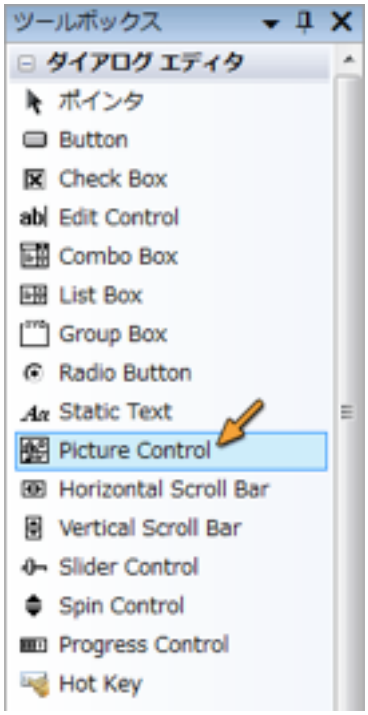
ウィンドウの中央にある「TODO: ダイアログのコントロールをここに配置」という文字を選択し，[Delete] キーをタイプするなどして削除します. 「OK」や「キャンセル」も使わないので削除してしまいたいところですが, なんかもったいない気がする？ので今は置いておくことにします.



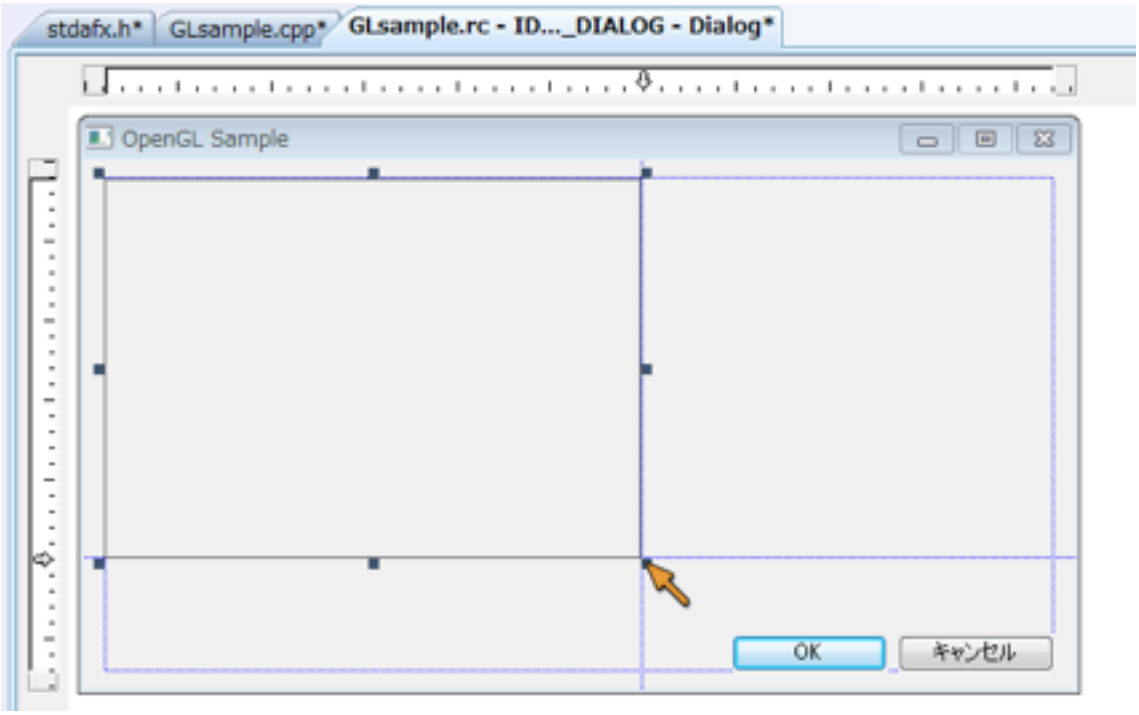
ダイアログエディタの上と左にあるルーラをマウスでクリック・ドラッグして，ガイドを設定します．グリッドシステムは画面設計の基本ですね．



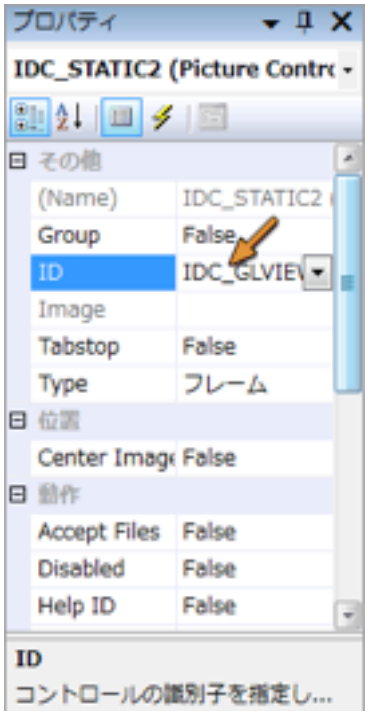
「ツールボックス」のウィンドウから「Picture Control」を選びます．



ガイドに沿ってマウスをドラッグして，「Picture Control」を配置します．

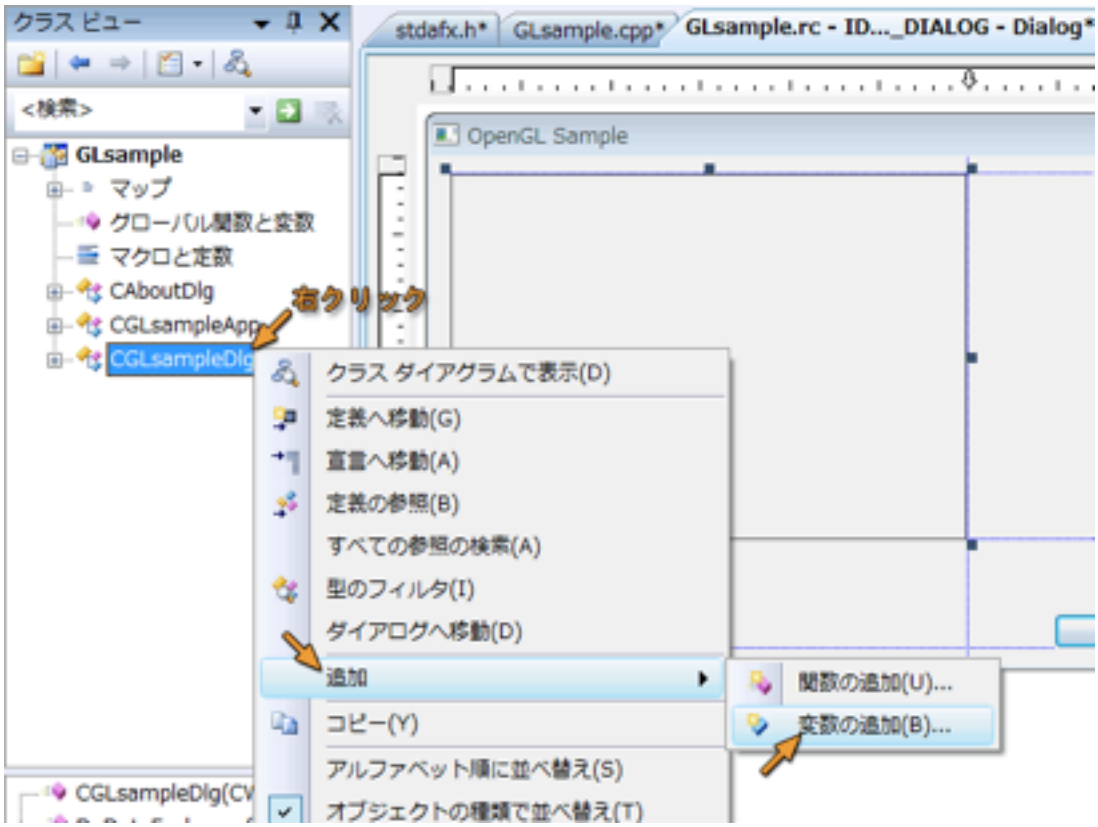


「プロパティ」のウィンドウの「ID」を「IDC_STATIC」から「IDC_GLVIEW」などほかの名前に変更します．「IDC_STATIC」は内容が変更されないコントロールの ID だそうです．

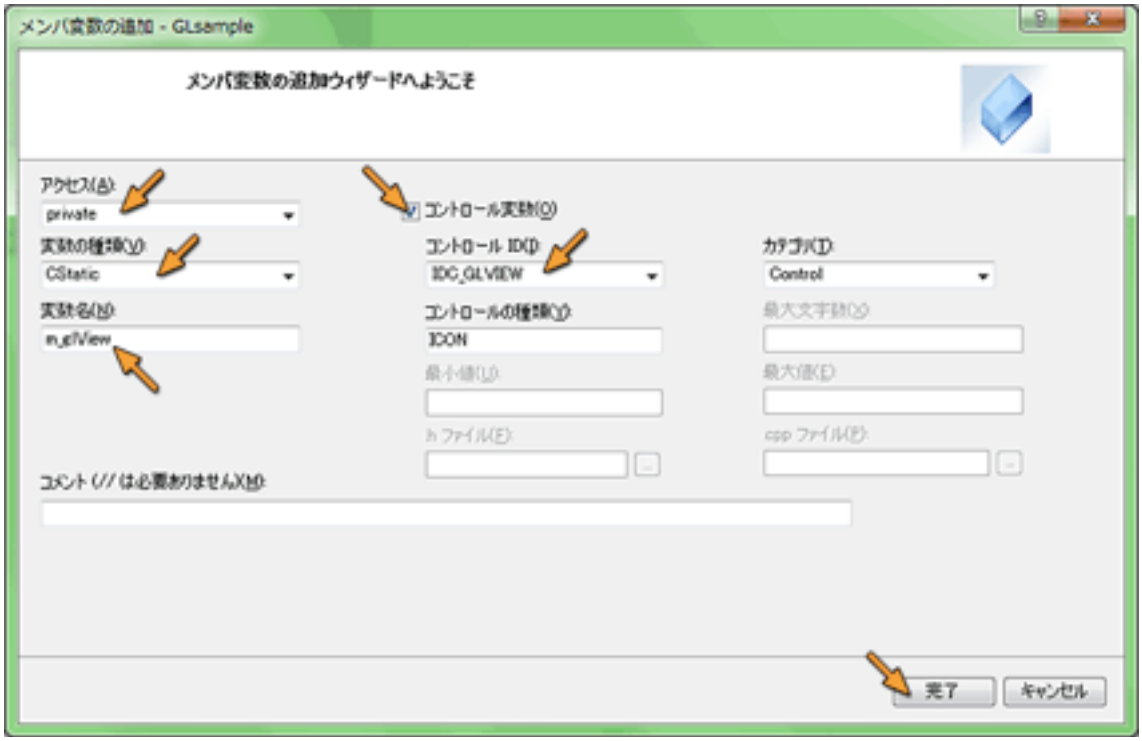

プロパティ
IDC_STATIC2 (Picture Contr
その他
(Name)
IDC_STATIC2
Group
False
ID
IDC_GLVIEW
Image
Tabstop
False
Type
フレーム
位置
Center Image
False
動作
Accept Files
False
Disabled
False
Help ID
False
ID
コントロールの識別子を指定し...

メンバ変数 (クラス変数) の追加

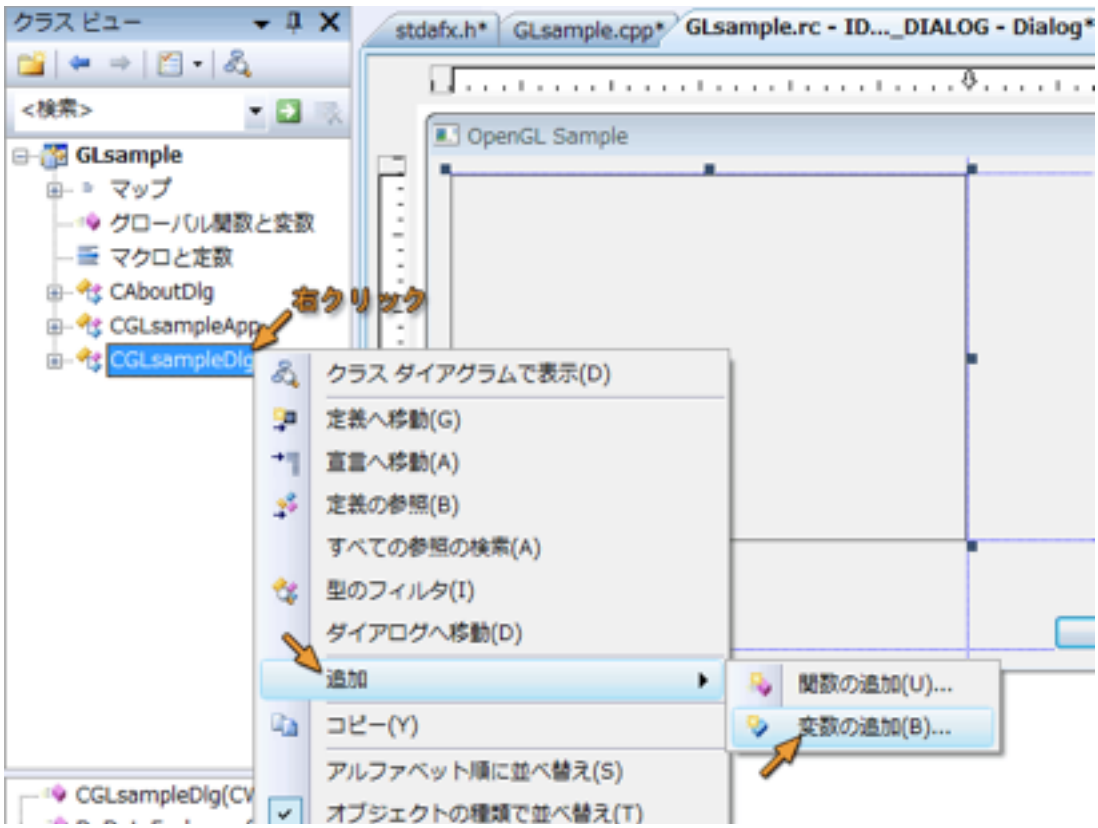
「クラスビュー」の中にある「Cプロジェクト名Dlg」クラス (ここでは CGLsampleDlg) を右クリックして，「追加」から「変数の追加」を選びます．



このクラスのメンバ変数を追加します．この変数の「アクセス」は "private"，「変数の種類」は "CStatic" とします．「変数名」は，ここでは "m_glView" にすることにします．その後「コントロール変数」にチェックを入れ，「コントロールID」に "IDC_GLVIEW" (Picture Control に設定した ID) を選択して「完了」をクリックします．

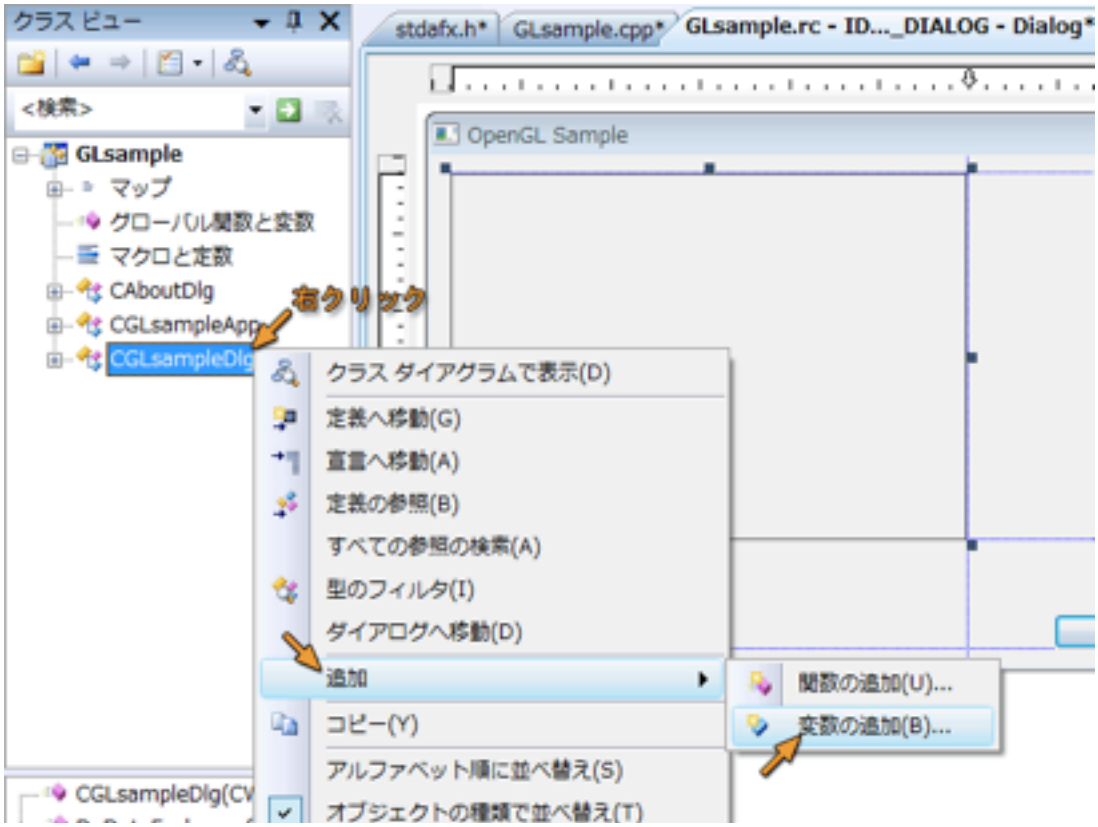


「Cプロジェクト名Dlg」クラス (ここでは CGLsampleDlg) にもう一つメンバ変数を追加します. これに Picture Control のデバイスコンテキストを保持します.



この変数の「アクセス」は "private", 「変数の種類」は "CDC *" にします. 「変数名」は "m_pDC" とかにするのが習わしなのでしょうか. 最後に「完了」をクリックします.

「Cプロジェクト名Dlg」クラス (ここでは CGLsampleDlg) に更にもう一つメンバ変数を追加します. これに Picture Control 上に表示する OpenGL の領域のレンダリングコンテキストを保持します.

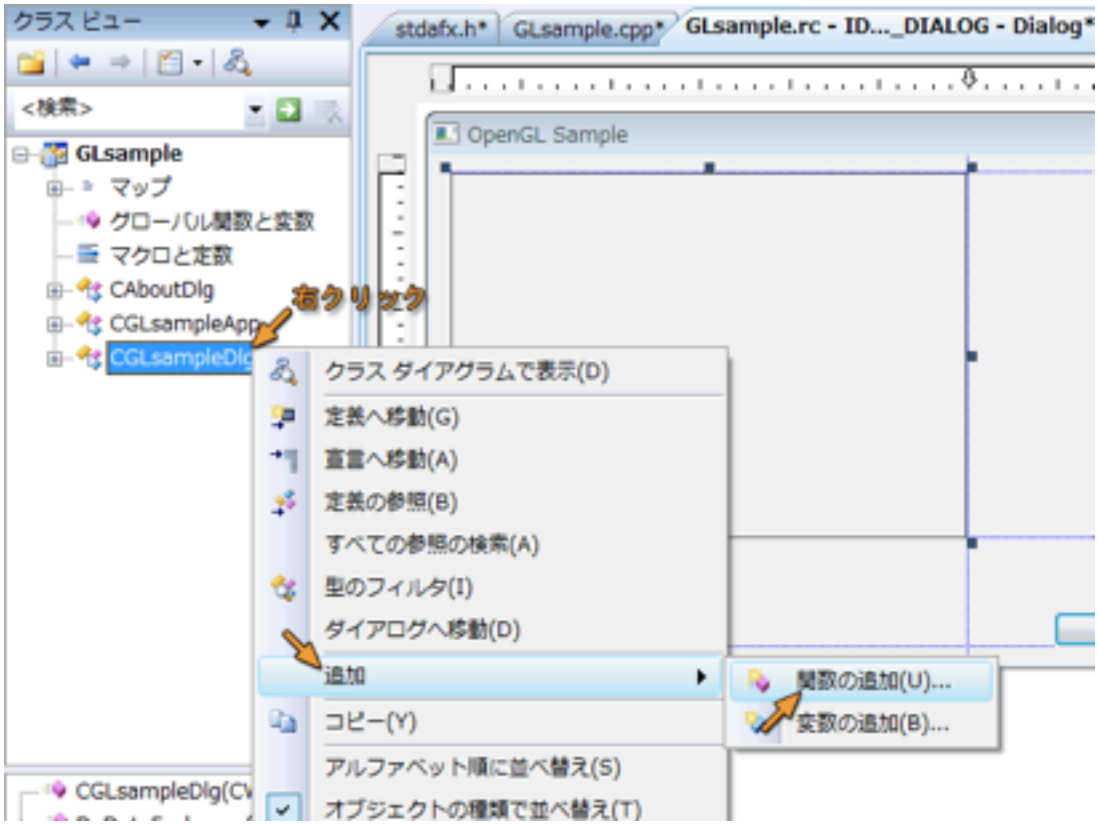


この変数の「アクセス」は "private", 「変数の種類」は "HGLRC" とします. 「変数名」は "m_GLRC" ということにします. 最後に「完了」をクリックします.


メンバ変数の追加 - GLsample
メンバ変数の追加ウィザードへようこそ
アクセス(A):
private
変数の種類(V):
HGLRC
変数名(N):
m_GLRC
コメント (// は必要ありません)(M):
コントロール変数(C)
コントロール ID(I):
IDCANCEL
カテゴリ(T):
Control
コントロールの種類(Y):
BUTTON
最大文字数(X):
最小値(U):
最大値(E):
h ファイル(F):
cpp ファイル(P):
完了
キャンセル

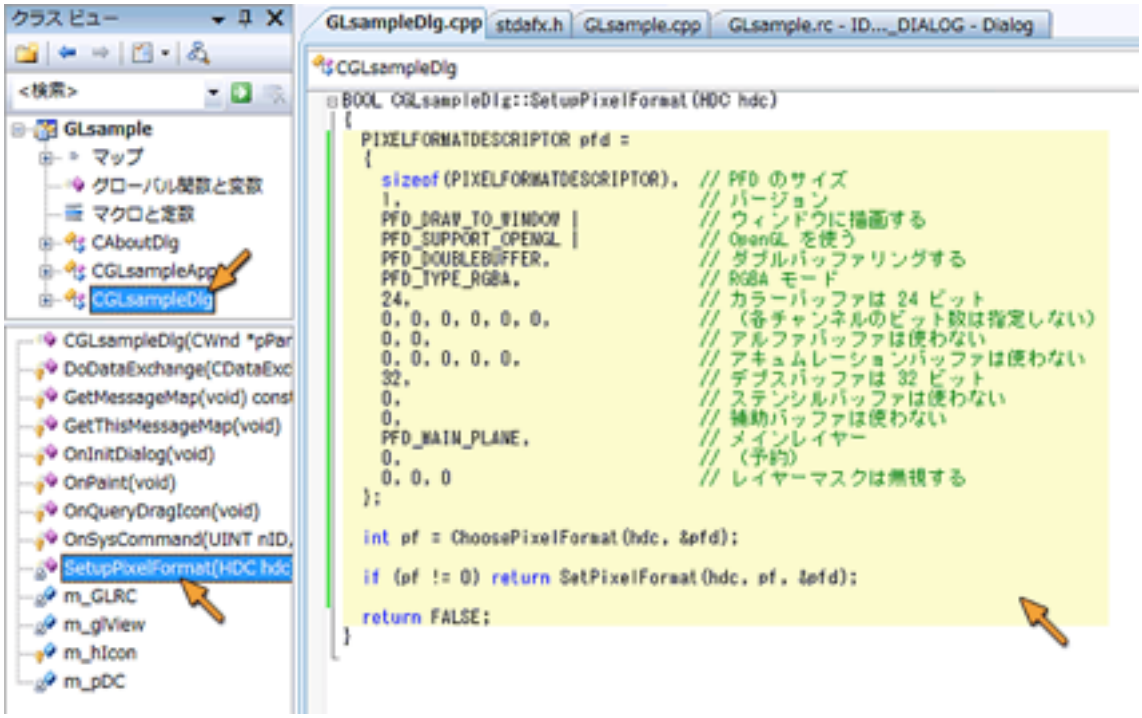
メンバ関数 (メソッド) の追加

同じようにして，メンバ関数も追加します．「クラスビュー」の中にある「Cプロジェクト名Dlg」クラス（ここでは CGLsampleDlg）を右クリックして，「追加」から「関数の追加」を選びます．



この関数の「戻り値の型」は "BOOL" とし，「関数名」は "SetUpPixelFormat" にすることにします．次に「パラメータの型」に "HDC" を設定し，「パラメータ名」に "hdc" を設定した後，「追加」を忘れずにクリックします．また，「アクセス」は "private" にします．最後に「完了」をクリックします．

すると「プロジェクト名Dlg.cpp」ファイル (ここでは GLsampleDlg.cpp) にメンバ関数 (ここでは SetupPixelFormat()) が追加され，ソースコードを編集する状態になります．ここで SetupPixelFormat() の内容を実装します．



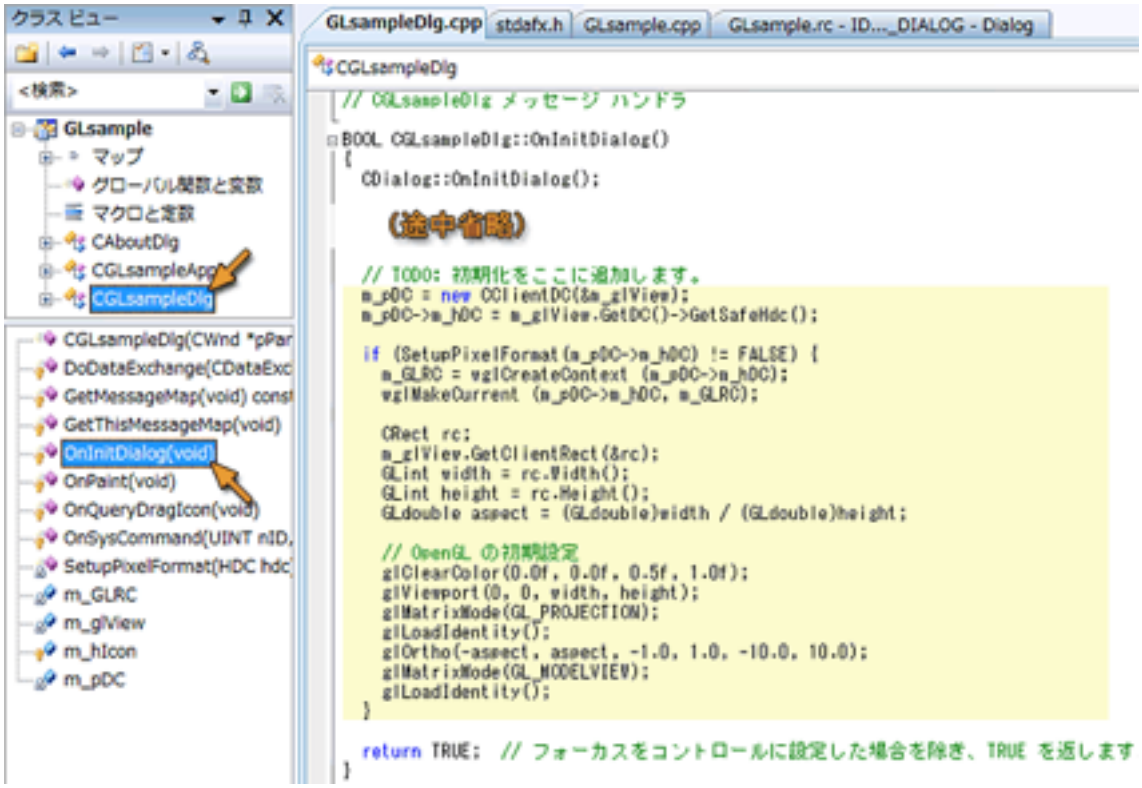

```
BOOL CGLsampleDlg::SetupPixelFormat(HDC hdc)
{
  PIXELFORMATDESCRIPTOR pfd = {
    sizeof(PIXELFORMATDESCRIPTOR),  // PFD のサイズ
    1,                              // バージョン
    PFD_DRAW_TO_WINDOW |            // ウィンドウに描画する
    PFD_SUPPORT_OPENGL |            // OpenGL を使う
```

```
        PFD_DOUBLEBUFFER,               // ダブルバッファリングする
        PFD_TYPE_RGBA,                  // RGBA モード
        24,                             // カラーバッファは 24 ビット
        0, 0, 0, 0, 0, 0,               //  (各チャンネルのビット数は指定しない)
        0, 0,                           // アルファバッファは使わない
        0, 0, 0, 0, 0,                  // アキュムレーションバッファは使わない
        32,                             // デプスバッファは 32 ビット
        0,                              // ステンシルバッファは使わない
        0,                              // 補助バッファは使わない
        PFD_MAIN_PLANE,                 // メインレイヤー
        0,                              //  (予約)
        0, 0, 0                         // レイヤーマスクは無視する
      };

      int pf = ChoosePixelFormat(hdc, &pfd);

      if (pf != 0) return SetPixelFormat(hdc, pf, &pfd);

      return FALSE;
    }
```

この関数では, コンピュータが備える OpenGL のサブシステムが用意しているピクセルフォーマットをの中から, 変数 pfd に設定した仕様を満たすものを ChoosePixelFormat() を使って選択し, 見つかったものの番号 pf を SetPixelFormat() で現在のデバイスコンテキスト hdc に設定します. 見つからなければ, pf は 0 になります.

イベントハンドラの修正

次に，ダイアログウィンドウが開かれるときに呼ばれるメンバ関数 OnInitDialog() に処理内容を追加します．クラスビューの「Cプロジェクト名Dlg」クラス (ここでは CGLsampleDlg) を選択し，その下のメンバ一覧にある OnInitDialog(void) をダブルクリックします．関数の本体が表示されますから，「// TODO: 初期化をここに追加します。」の後に下記の内容を追加します．

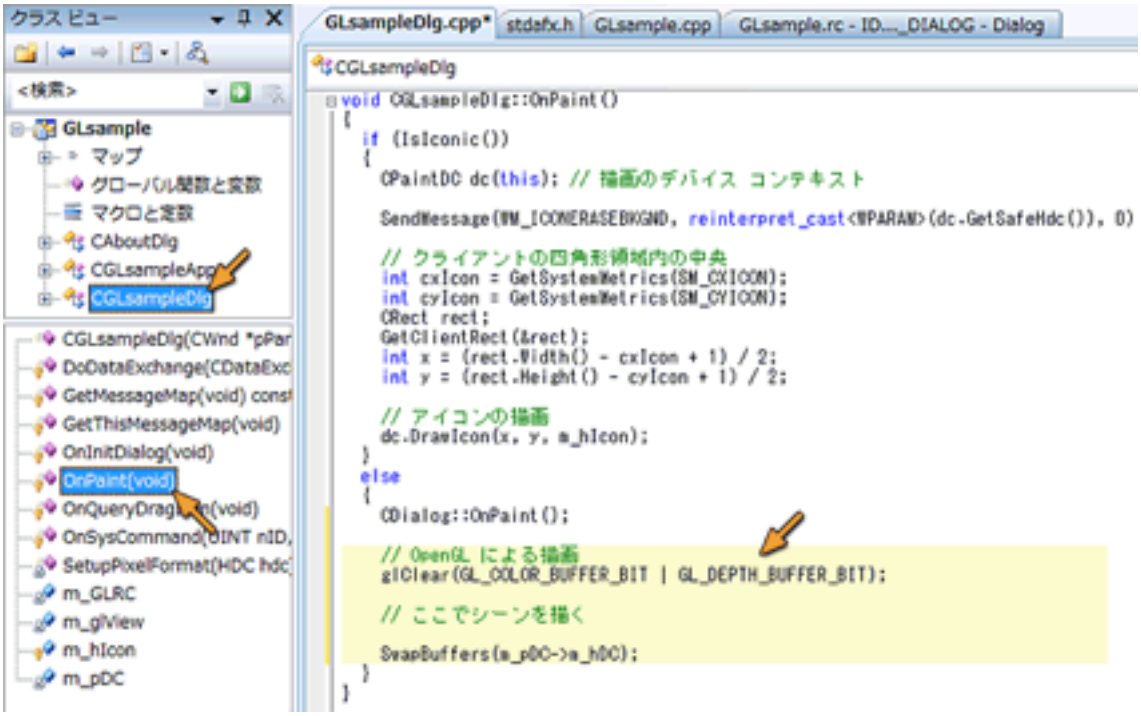


Picture Control に OpenGL による描画を行うので，そのクライアント領域にアクセスする CClientDC オブジェクトを，それを制御するメンバ変数 (ここでは m_glView) 使って生成します．これでいいんでしょうか？ 自信がアリマセン．OpenGL の初期設定も，ここでやってしまいます．多分 OpenGL の機能の呼び出しは別の関数にまとめて，それをここで呼び出すようにしたほうがいいんでしょうけど，ここに書く手順が増えるので手を抜きます．

```
BOOL CGLsampleDlg::OnInitDialog()
{
  CDialog::OnInitDialog();

  ...

  // TODO: 初期化をここに追加します。
  m_pDC = new CClientDC(&m_glView);
  if (SetupPixelFormat(m_pDC->m_hDC) != FALSE) {
    m_GLRC = wglCreateContext(m_pDC->m_hDC);
    wglMakeCurrent(m_pDC->m_hDC, m_GLRC);

    CRect rc;
    m_glView.GetClientRect(&rc);
    GLint width = rc.Width();
    GLint height = rc.Height();
    GLdouble aspect = (GLdouble)width / (GLdouble)height;
```

```
      // OpenGL の初期設定
      glClearColor(0.0f, 0.0f, 0.5f, 1.0f);
      glViewport(0, 0, width, height);
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      glOrtho(-aspect, aspect, -1.0, 1.0, -10.0, 10.0);
      glMatrixMode(GL_MODELVIEW);
      glLoadIdentity();
    }

    return TRUE;  // フォーカスをコントロールに設定した場合を除き、TRUE を返します。
}
```

同様に，ウィンドウの描画が必要になった時に呼ばれるメンバ関数 OnPaint() を変更します．クラスビューの「Cプロジェクト名Dlg」クラス (ここでは CGLsampleDlg) を選択し，その下のメンバ一覧にある OnPaint(void) をダブルクリックします．関数の本体が表示されますから，その中にある if 文の else 節に下記の内容を追加します．



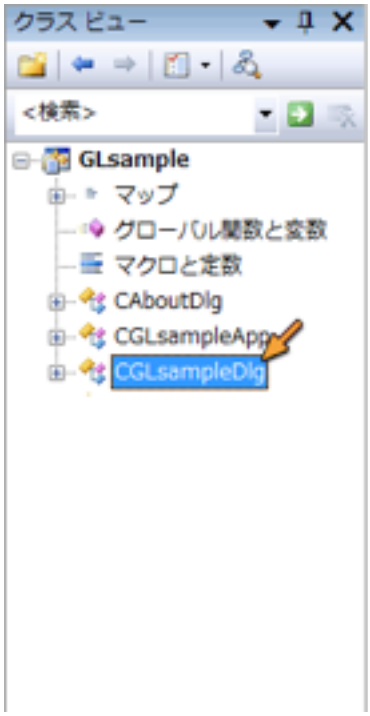
シーンの描画はこの部分で行います．今のところは，画面クリアだけしておきます．これも別の関数にまとめておいた方がいいでしょうね．

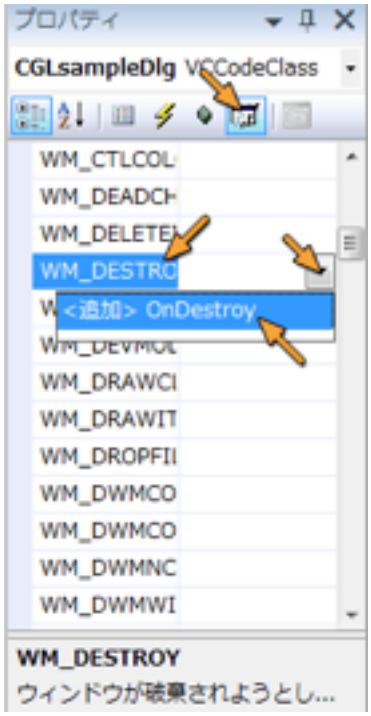
```
void CGLsampleDlg::OnPaint()
{
  if (IsIconic())
  {
    ...
  }
  else
  {
    CDialog::OnPaint();

    // OpenGL による描画
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // ここでシーンを描く
```

```
        SwapBuffers(m_pDC->m_hDC);
      }
    }
```

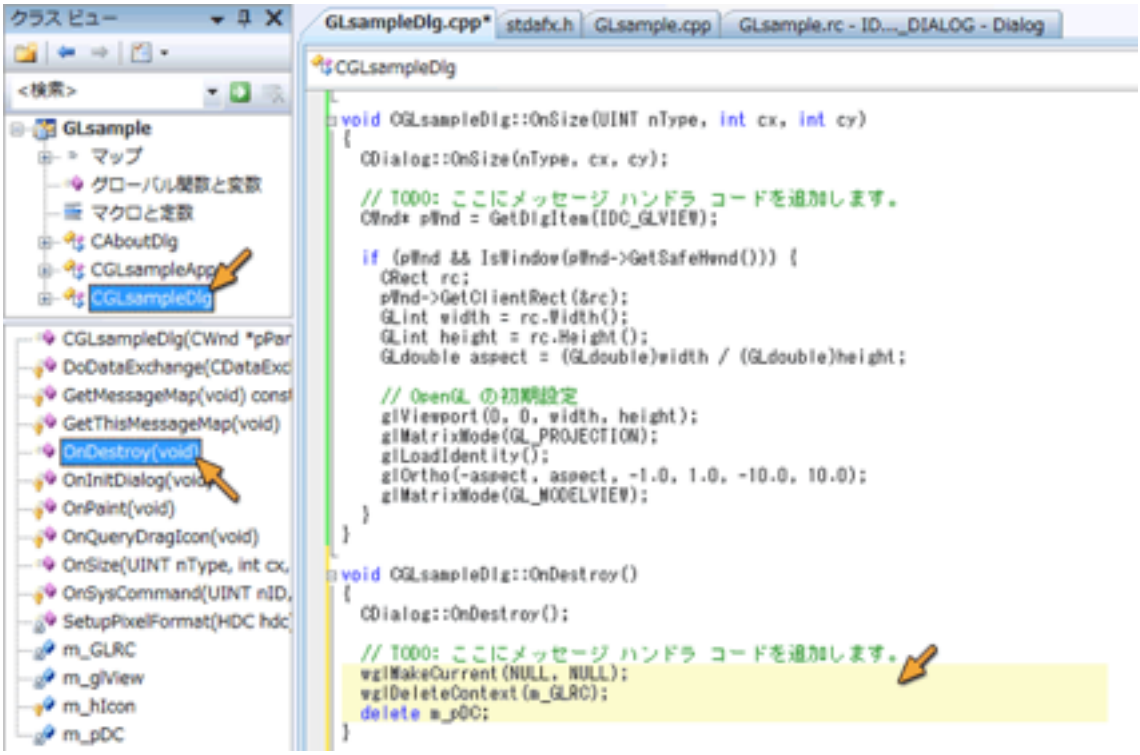
イベントハンドラの追加

ウィンドウを閉じた時に発生するイベント　WM_DESTROY　に対するハンドラを追加します．クラスビューの「Cプロジェクト名Dlg」クラス (ここでは CGLsampleDlg) を選択し，「プロパティ」ウィンドウの「メッセージ」のボタン　(左から 6 個目のボタン)　をクリックします．イベントのリストの中から WM_DESTORY を選び，その右側の▼のボタンをクリックして，「<追加>> OnDestroy」を選びます．





すると「プロジェクト名Dlg.cpp」ファイル　(ここでは　GLsampleDlg.cpp)　にメンバ関数 OnDestroy() が追加され，ソースコードを編集する状態になります．ここで OnDestroy() の内容を実装します．

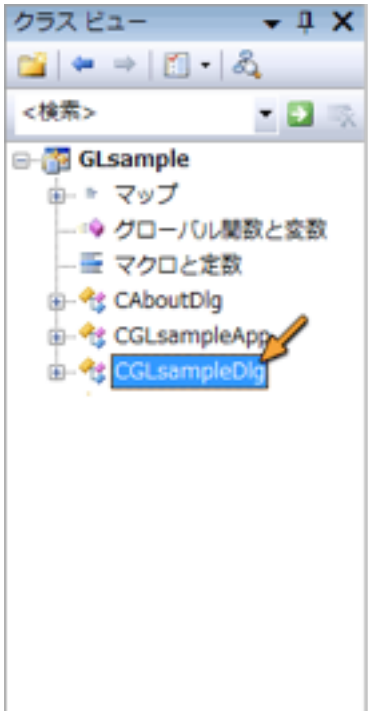


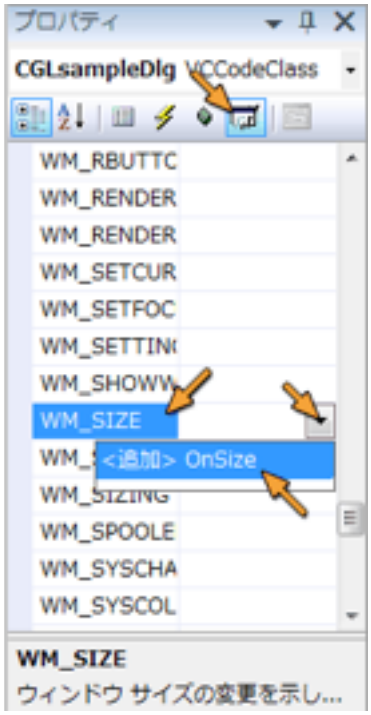
レンダリングコンテキストをウィンドウのデバイスコンテキストから結合解除し，そのレンダリングコンテキストを削除します．またクライアント領域のデバイスコンテキストを保持している CClientDC オブジェクトを削除します．

```
void CGLsampleDlg::OnDestroy()
{
  CDialog::OnDestroy();

  // TODO: ここにメッセージ ハンドラ コードを追加します。
  wglMakeCurrent(NULL, NULL);
  wglDeleteContext(m_GLRC);
  delete m_pDC;
}
```

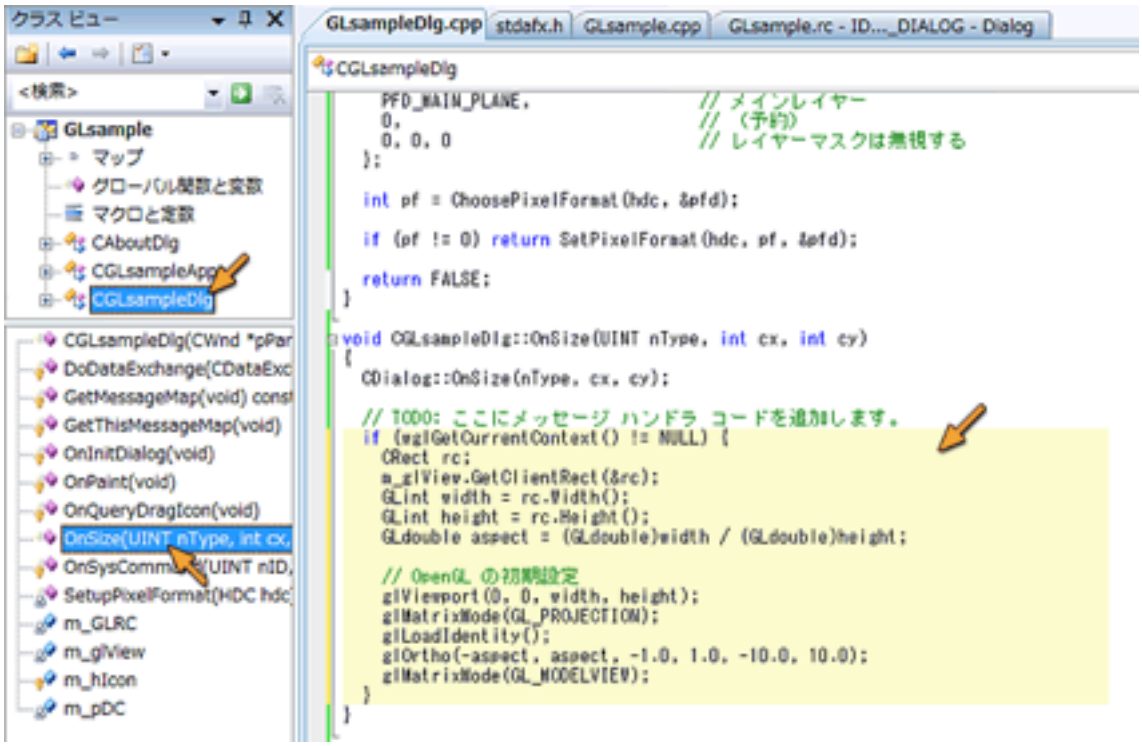
最後に，ウィンドウのサイズを変えた時に発生するイベント WM_SIZE に対するハンドラを追加します．ですが，今作っているプログラムはウィンドウサイズの変更や最大化ができないようにしているので，現時点ではこのイベントは発生しません．なので，ここは省略してかまいません．

クラスビューの「Cプロジェクト名Dlg」クラス (ここでは CGLsampleDlg) を選択し，「プロパティ」ウィンドウの「メッセージ」のボタン (左から５個目のボタン) をクリックします．イベントのリストの中から WM_SIZE を選び，その右側の▼のボタンをクリックして，「<追加> OnSize」を選びます．





すると「プロジェクト名Dlg.cpp」ファイル（ここでは GLsampleDlg.cpp）にメンバ関数 OnSize() が追加され，ソースコードを編集する状態になります．ここで OnSize() の内容を実装します．



OnSize() は OnInitDialog() の実行より前に一度実行されます．しかし，その時点ではまだ OpenGL が初期化されていないため，ここで OpenGL の機能を使おうとするとエラーになってしまいます（はまりました）．そこで wglGetCurrentContext() を使って OpenGL が使えるかどうか確かめていますが，このやり方がいいのかどうかは知りません．

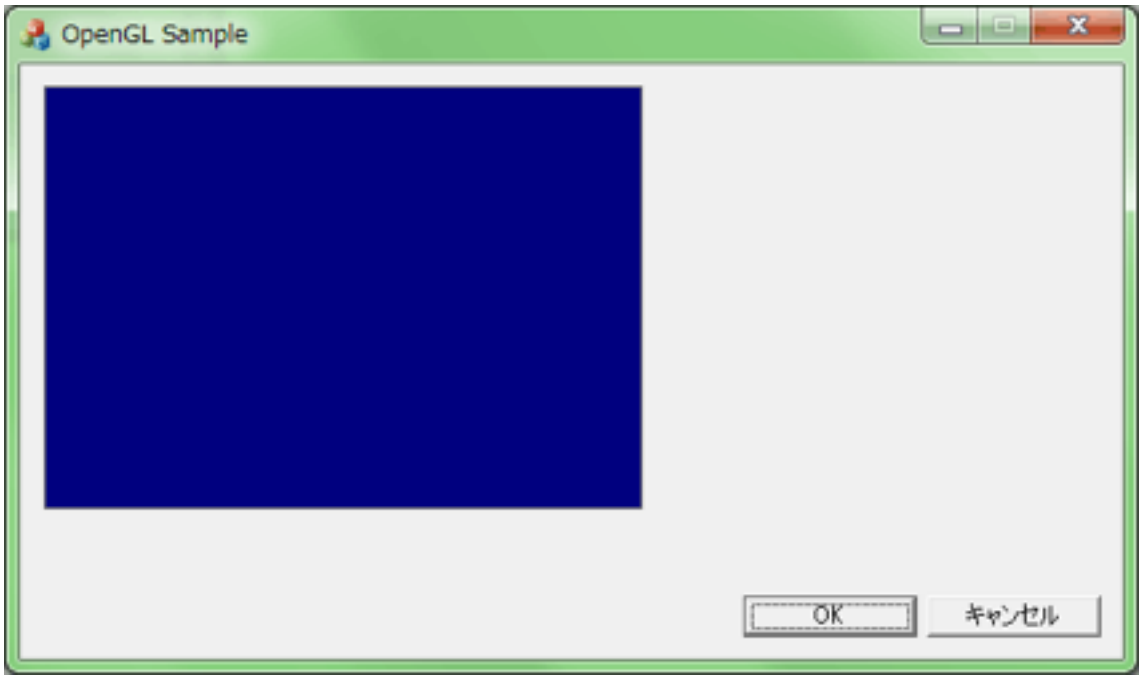
```
void CGLsampleDlg::OnSize(UINT nType, int cx, int cy)
{
  CDialog::OnSize(nType, cx, cy);

  // TODO: ここにメッセージ ハンドラ コードを追加します。
  if (wglGetCurrentContext() != NULL) {
    CRect rc;
    m_glView.GetClientRect(&rc);
    GLint width = rc.Width();
    GLint height = rc.Height();
    GLdouble aspect = (GLdouble)width / (GLdouble)height;

    // OpenGL の初期設定
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-aspect, aspect, -1.0, 1.0, -10.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
  }
}
```

しかし，繰り返しになりますが，このプログラムでは WM_SIZE イベントが OnInitDialog() 実行後に呼び出されることはありません．もし，ウィンドウのサイズを変更できるようにした場合は，OnSize() の引数 cx, cy を使って MoveWindow() などにより OpenGL の表示を行っている Picture Control のサイズを変更することになります．その場合は，変更するサイズを使って width や height を求めればよいので，わざわざ GetClientRect() を使う必要はありません．

でも，ウィンドウのサイズを変更できるようにするには，すべてのコントロールの配置を計算し直さないといけないんですかね．もう考えるのはめんどくさいので，これは考えないことにします．

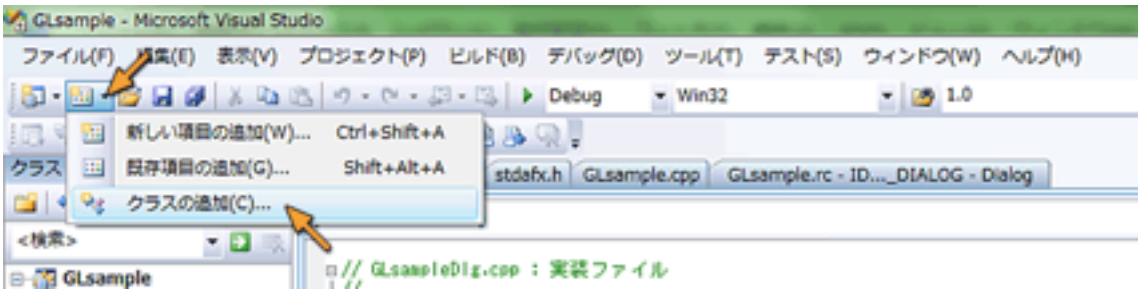
プログラムの実行

ここで一旦プログラムをビルドし, 実行してみます. 動くかな.

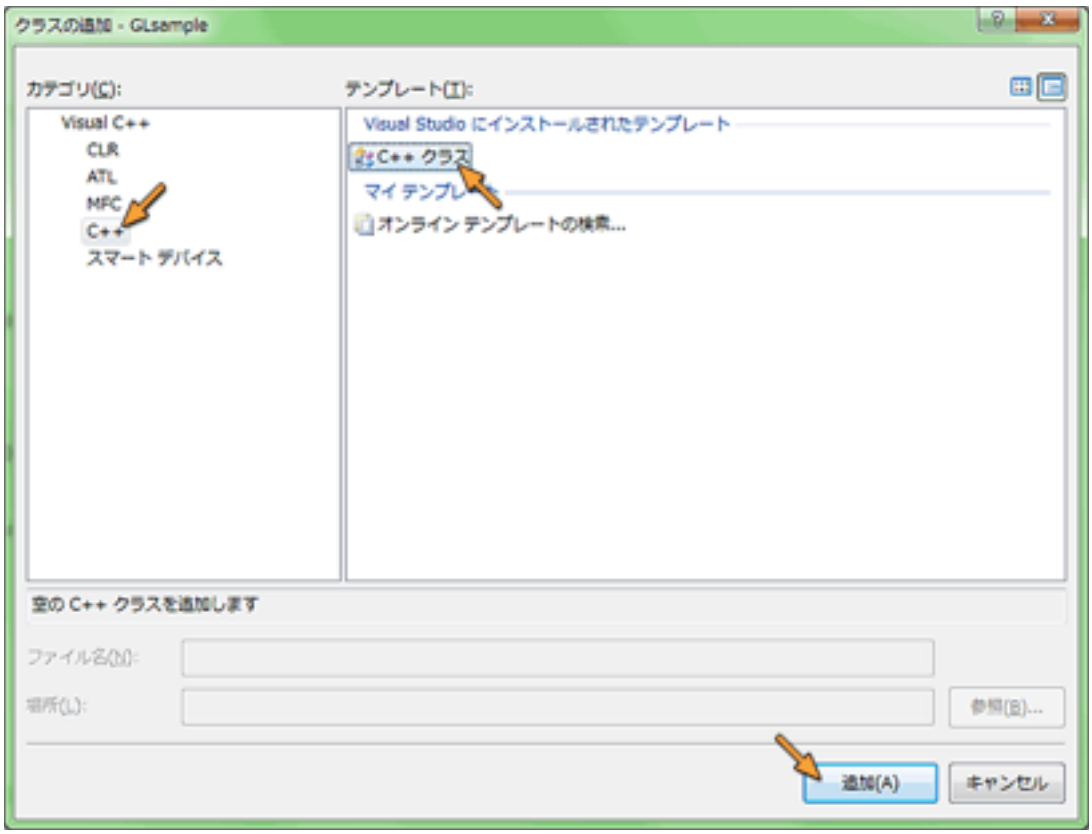


図形を描画するクラスを作る

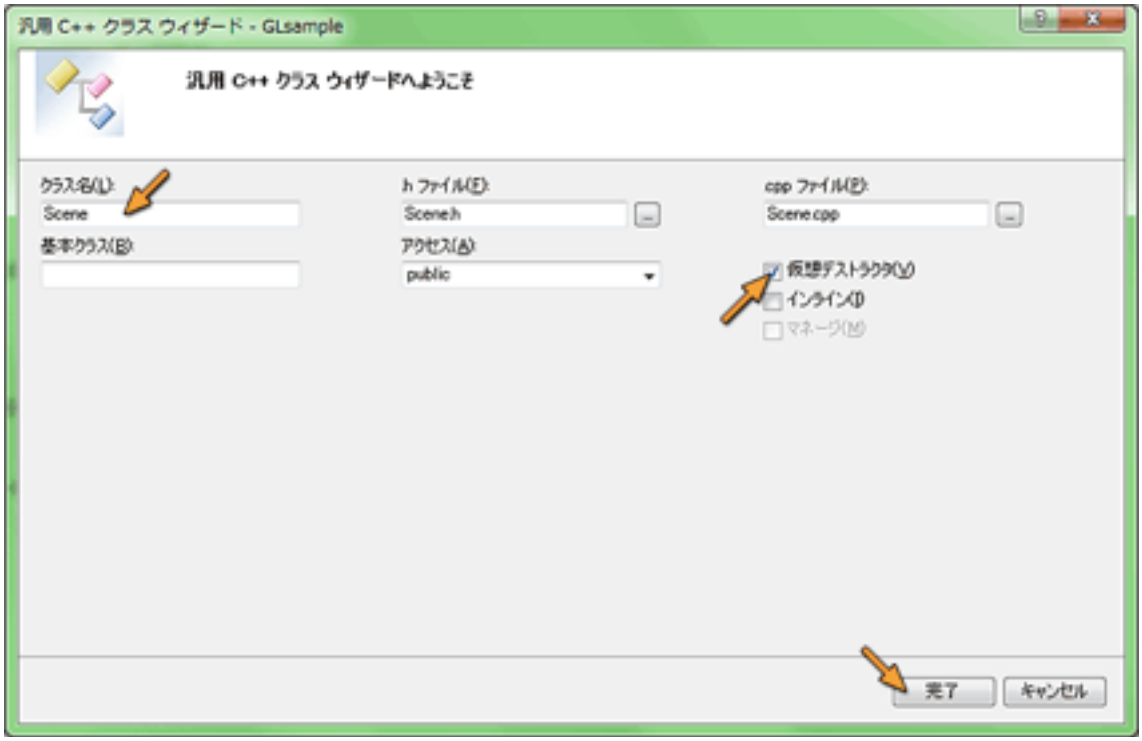
Picture Control に結びつけた OpenGL の描画領域に描くシーンのクラスを作成します. 「クラスの追加」を選びます. 別にクラスにする必要はなんにもない気がするんですけど.



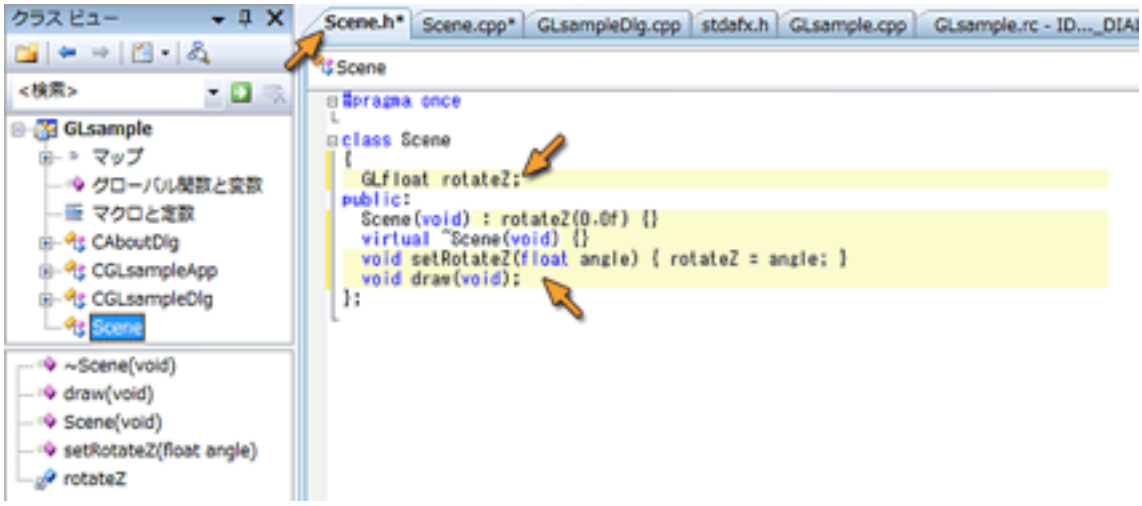
「カテゴリ」から「C++」を選び, 「C++クラス」のテンプレート鵜を選んで「追加」をクリックします. OpenGL のクラスなので, MFC なんかの流儀は気にしないことにします.



クラス名は「Scene」とします. 「仮想デストラクタ」にするのはデフォだという話ですが, これについてはいろいろ議論もあるようです. でも, ここでは日和ります.



Scene クラスの宣言を行います. Scene.h を開きます.



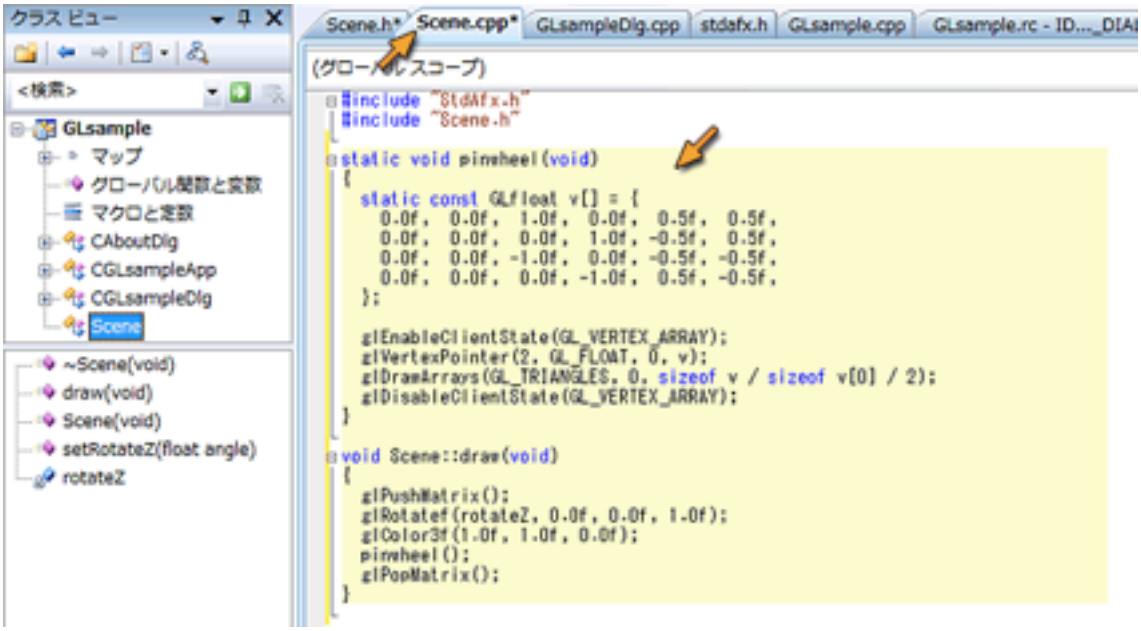
ユーザインタフェース機能はクラス変数として GLfloat 型の rotateZ を 1 個だけ宣言します. クラス変数って，メンバ関数にとってのグローバル変数なんだなって MFC を使ってると意識させられます. コンストラクタとデストラクタは，ここではあまり仕事がないので，インラインで定義してしまいます. あと，コンストラクタで rotateZ を初期化するようにします. メンバ関数には rotateZ に値を設定する setRotateZ() とシーンを描画する draw() を用意することにします.

個人的には #pragma once じゃなくて #ifndef 〜 #define 〜 #endif を使いたいところですが, ここでは Visual Studio の流儀に従います.

```
#pragma once

class Scene {
```

```
  GLfloat rotateZ;
public:
  Scene(void) : rotateZ(0.0f) {}
  virtual ~Scene(void) {}
  void setRotateZ(float angle) { rotateZ = angle; }
  void draw(void);
};
```

Scene.cpp を開いて Scene クラスの実装を行います．コンストラクタとデストラクタは自動生成されていますが，Scene.h でインラインにしちゃったので消してしまいます．ここでは draw() の実装だけを行います．



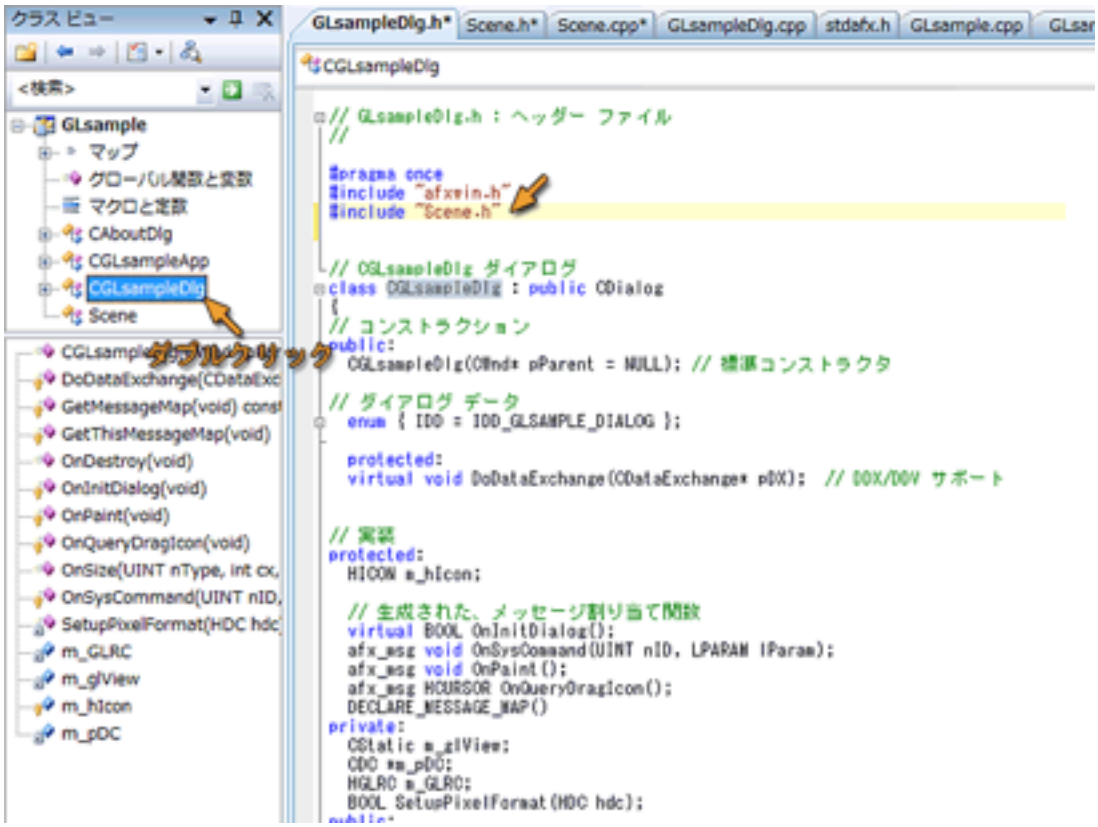
描画する図形はなんでもいいんですけど，GLUT も AUX も使ってないので (GLU は使えますけど) 自分で定義することにします．もうちょっとこだわってシーングラフっぽくしたくなるのですが，そんなことをしていたら終わらないので我慢します．

```
#include "StdAfx.h"
#include "Scene.h"

static void pinwheel(void)
{
  static const GLfloat v[] = {
    0.0f,  0.0f,  1.0f,  0.0f,  0.5f,  0.5f,
    0.0f,  0.0f,  0.0f,  1.0f, -0.5f,  0.5f,
    0.0f,  0.0f, -1.0f,  0.0f, -0.5f, -0.5f,
    0.0f,  0.0f,  0.0f, -1.0f,  0.5f, -0.5f,
  };

  glEnableClientState(GL_VERTEX_ARRAY);
  glVertexPointer(2, GL_FLOAT, 0, v);
  glDrawArrays(GL_TRIANGLES, 0, sizeof v / sizeof v[0] / 2);
  glDisableClientState(GL_VERTEX_ARRAY);
}

void Scene::draw(void)
{
  glPushMatrix();
  glRotatef(rotateZ, 0.0f, 0.0f, 1.0f);
  glColor3f(1.0f, 1.0f, 0.0f);
  pinwheel();
```

```
        glPopMatrix();
    }
```

図形の描画を組み込む

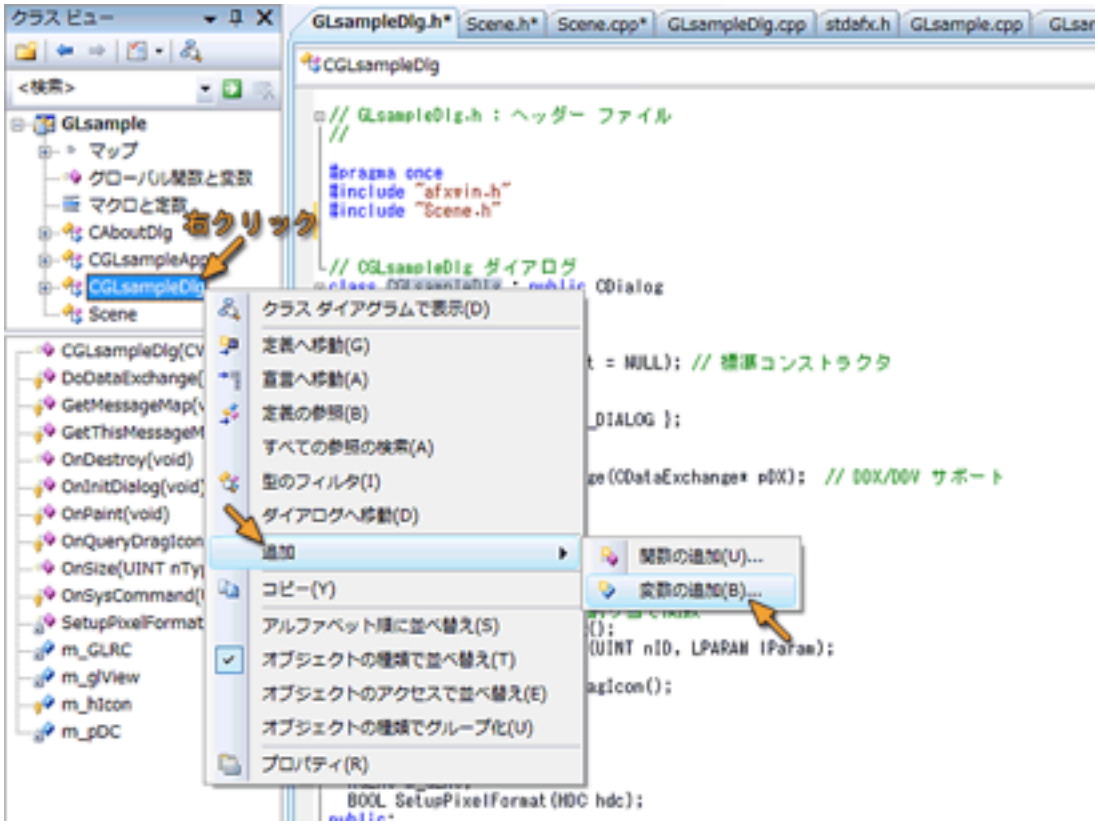
Scene クラスをダイアログウィンドウで使うので，クラスビューで Cプロジェクト名Dlg クラス (ここでは CGLsampleDlg) をダブルクリックして定義を開きます．



Scene.h を #include します．

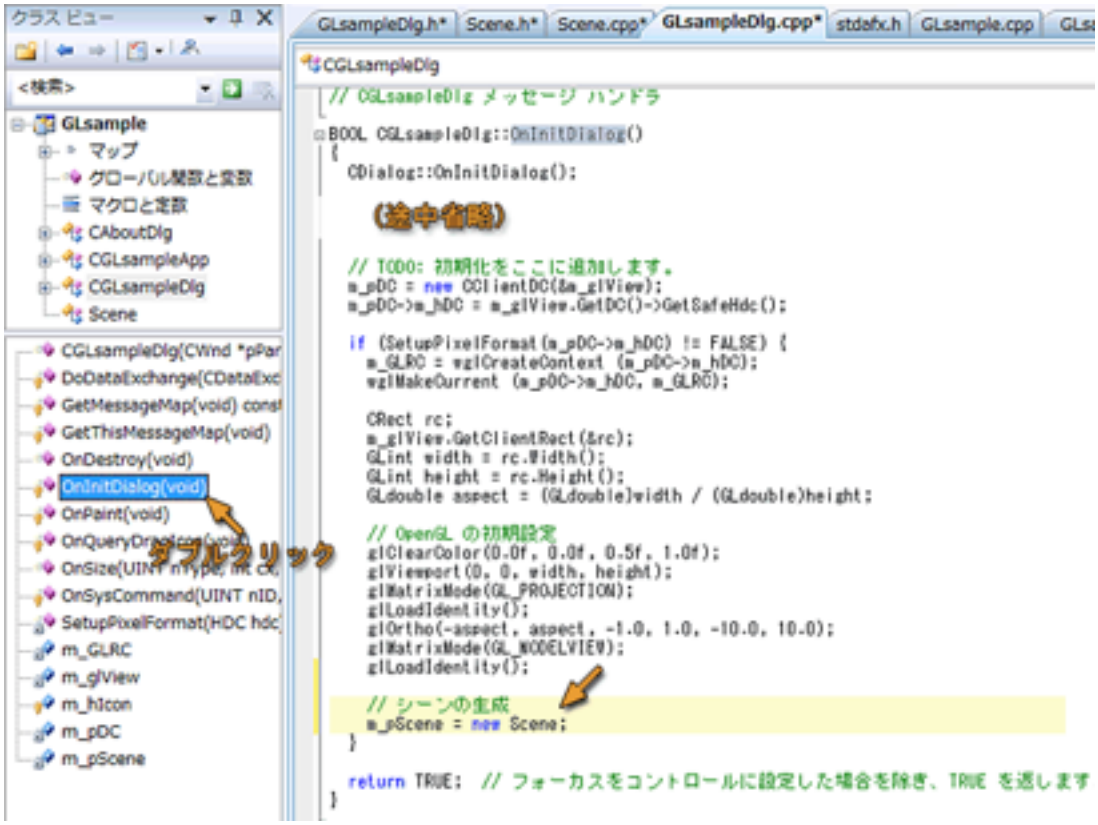
```
// GLsampleDlg.h : ヘッダー ファイル
//

#pragma once
#include "afxwin.h"
#include "Scene.h"


// CGLsampleDlg ダイアログ
class CGLsampleDlg : public CDialog
{
  ...
};
```

クラスビューで Cプロジェクト名Dlg クラス (ここでは CGLsampleDlg) を右クリックして「追加」から「変数の追加」を選びます．



「アクセス」は "private", 変数の種類は "Scene *"として, "m_pScene" という変数を追加します. "m_" はメンバ変数, "p" はポインタを表すそうです. このように変数名で変数の「立場」を表しておくことを「ハンガリアン記法」というそうです. これもいろいろ議論があるみたいですが, MFC の流儀に従うことにします. 最後に「完了」をクリックします.

クラスビューで OnInitDialog(void) をダブルクリックして, その定義を変更します.

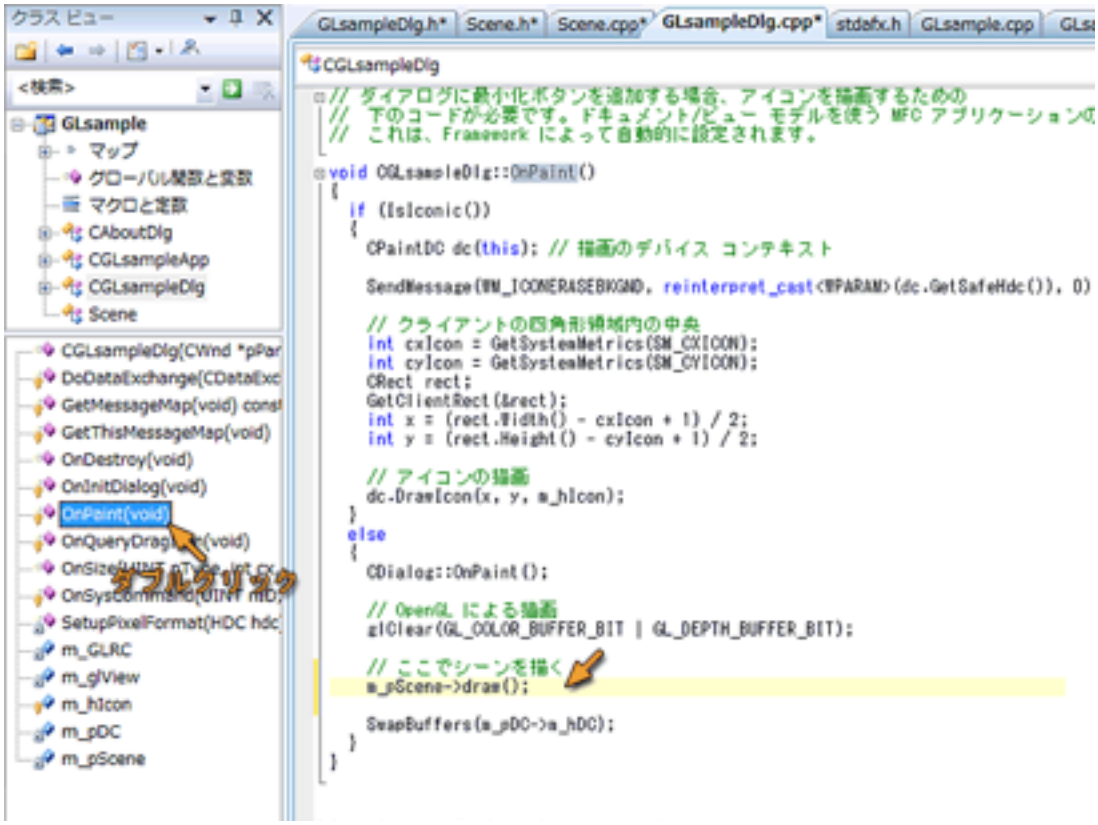


OpenGL の初期化が終わった後で Scene クラスのインスタンスを生成して，m_pScene に代入します．Scene のコンストラクタは OpenGL 的な処理を何もしていないので，実はこのインスタンスはどこで生成しても構わないのですが，気分的な問題と将来の拡張（あるのか）に備えて，ここで生成することにします．

```
BOOL CGLsampleDlg::OnInitDialog()
{
  CDialog::OnInitDialog();

  ...

  // TODO: 初期化をここに追加します。
  m_pDC = new CClientDC(&m_glView);
  if (SetupPixelFormat(m_pDC->m_hDC) != FALSE) {
    m_GLRC = wglCreateContext (m_pDC->m_hDC);
    wglMakeCurrent (m_pDC->m_hDC, m_GLRC);

    CRect rc;
    m_glView.GetClientRect(&rc);
    GLint width = rc.Width();
    GLint height = rc.Height();
    GLdouble aspect = (GLdouble)width / (GLdouble)height;

    // OpenGL の初期設定
    glClearColor(0.0f, 0.0f, 0.5f, 1.0f);
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-aspect, aspect, -1.0, 1.0, -10.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    // シーンの生成
    m_pScene = new Scene;
  }
```

```
    return TRUE;  // フォーカスをコントロールに設定した場合を除き、TRUE を返します。
}
```

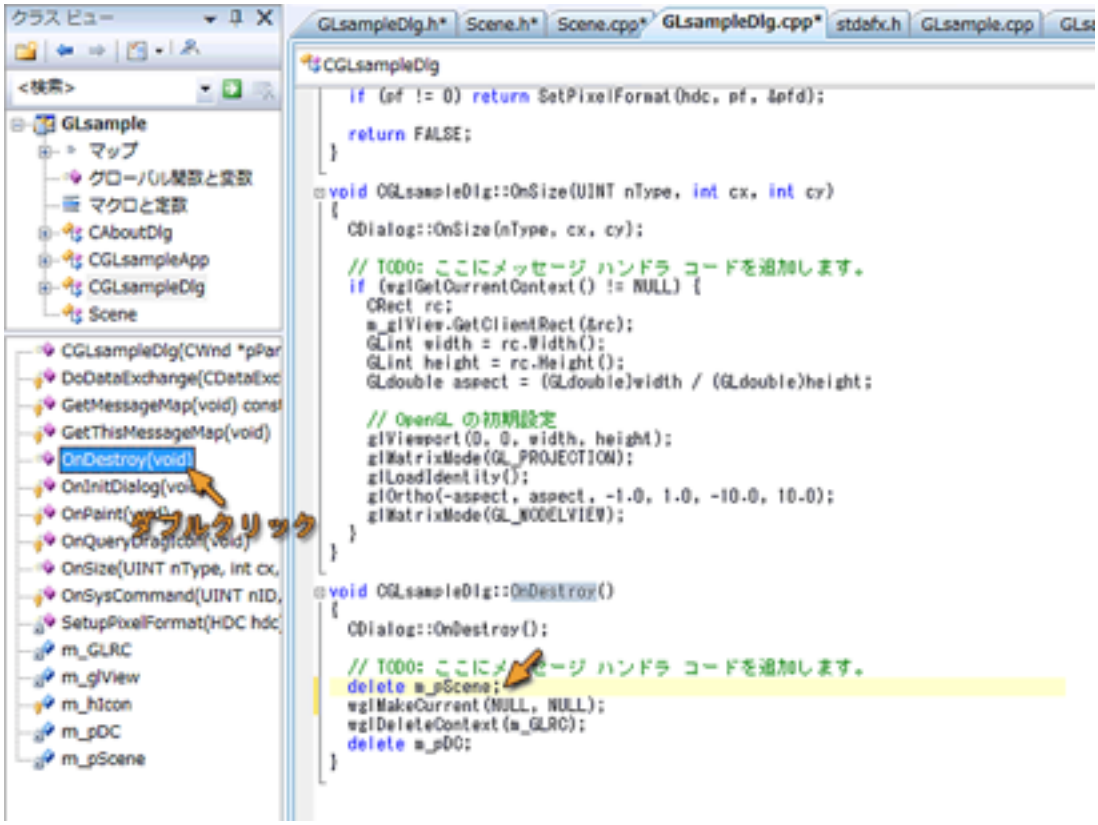
クラスビューで OnPaint(void) をダブルクリックして, その定義を変更します.



画面クリアの後でシーンを描画する draw() メソッドを呼び出します.

```
void CGLsampleDlg::OnPaint()
{
  if (IsIconic())
  {
    ...
  }
  else
  {     CDialog::OnPaint();

    // OpenGL による描画
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // ここでシーンを描く
    m_pScene->draw();

    SwapBuffers(m_pDC->m_hDC);
  }
}
```

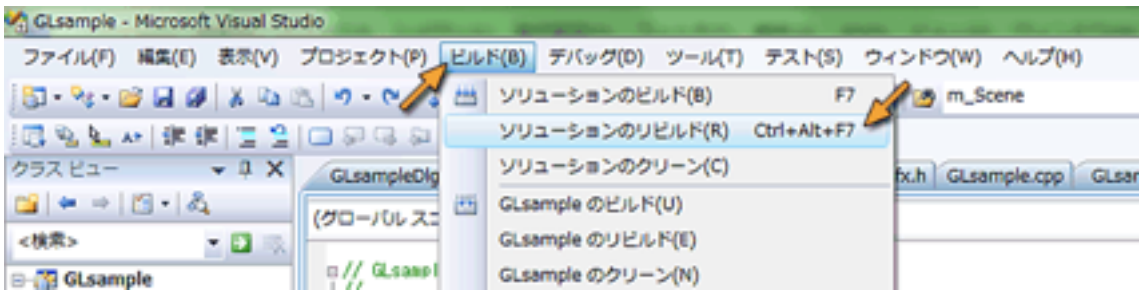
クラスビューで OnDestroy(void) をダブルクリックして, その定義を変更します.



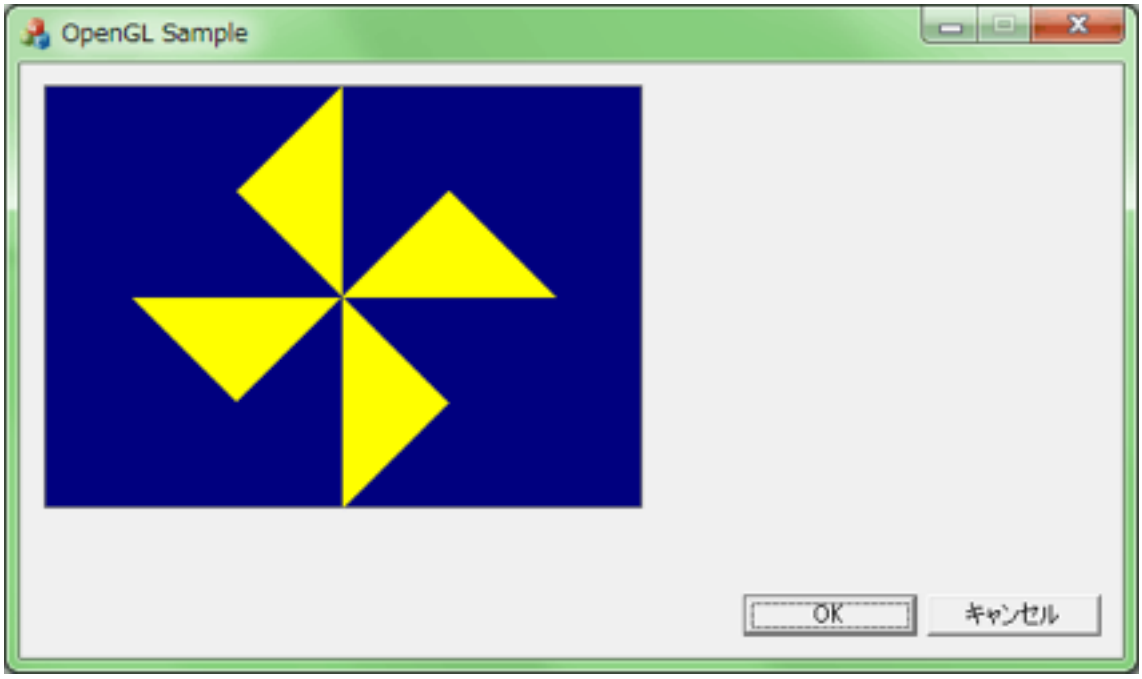
OnDestroy() はウィンドウを閉じるときに呼び出されるので，OnInitDialog() で生成したインスタンスをここで削除します．

```
void CGLsampleDlg::OnDestroy()
{
  CDialog::OnDestroy();

  // TODO: ここにメッセージ ハンドラ コードを追加します。
  delete m_pScene;
  wglMakeCurrent(NULL, NULL);
  wglDeleteContext(m_GLRC);
  delete m_pDC;
}
```

ここでプロジェクトを一旦ビルドして，プログラムが正常に動作するか確認します．

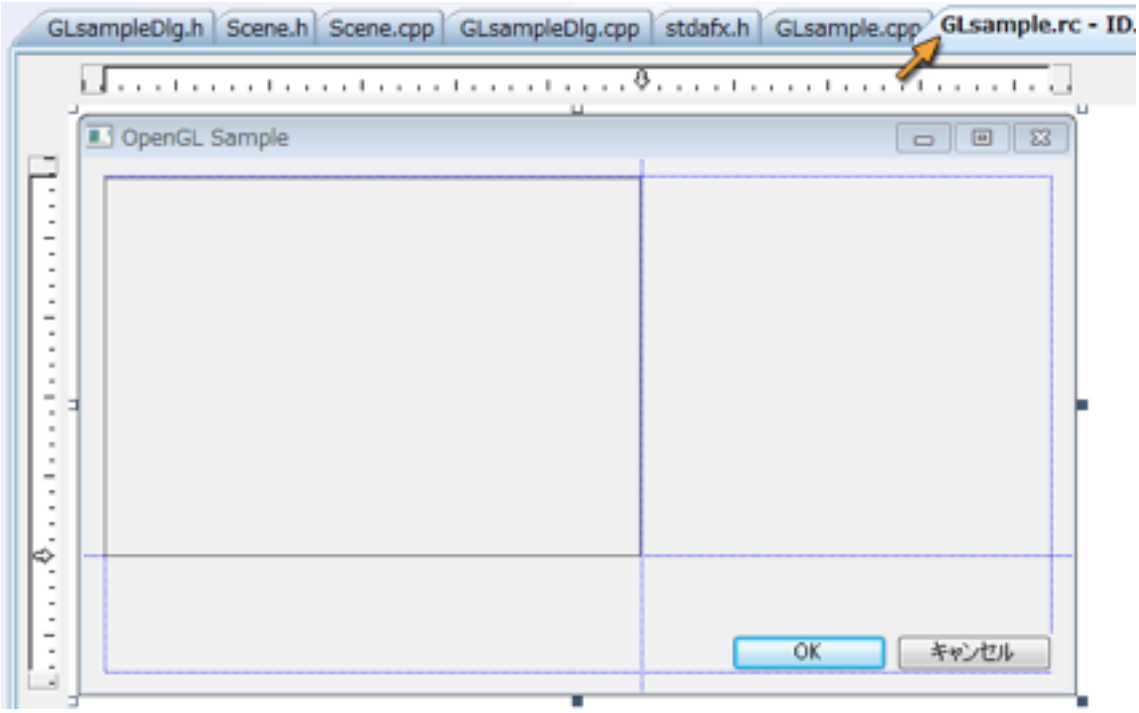


Picture Contol 上の OpenGL の描画領域に図形が表示されると思います．

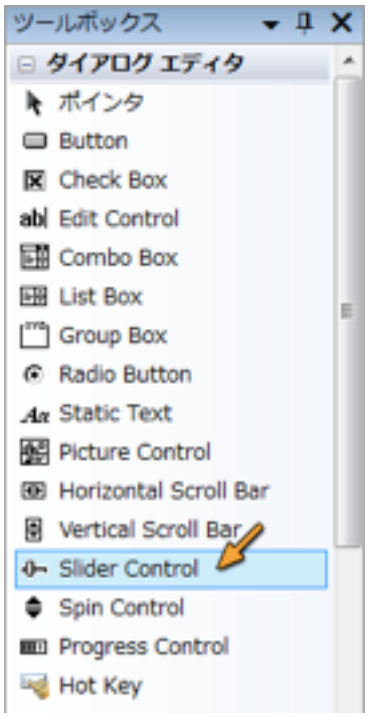

OpenGL Sample
OK
キャンセル

「SLIDER CONTROL」を追加する

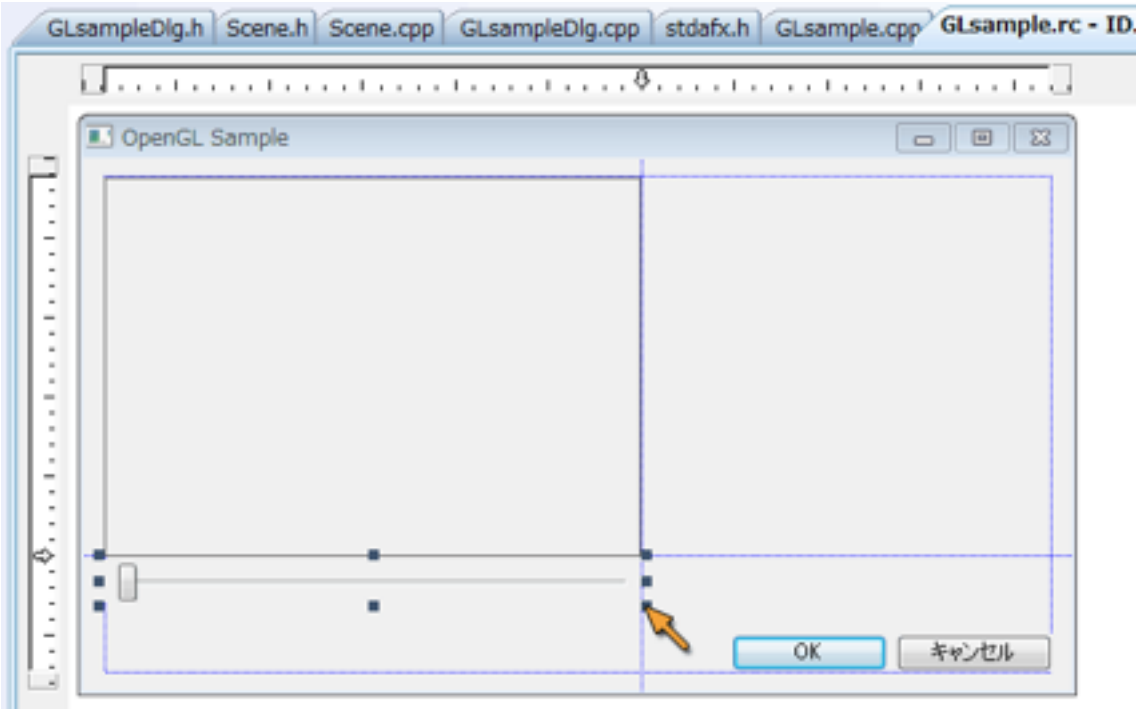
ダイアログエディタに切り替えます.



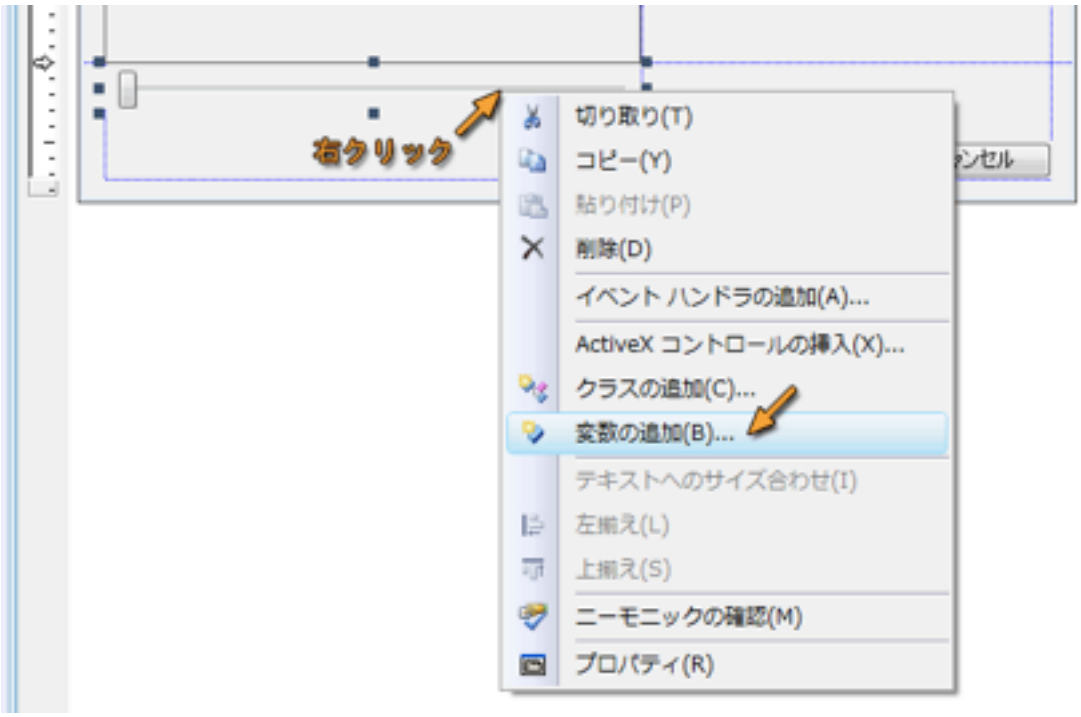
「ツールボックス」のウィンドウから「Slider Control」を選びます.



マウスを使って「Slider Control」を配置します. ちなみに, 縦方向の「Slider Control」を配置するには, 「プロパティ」の "Orientation" に「垂直方向」を設定します.

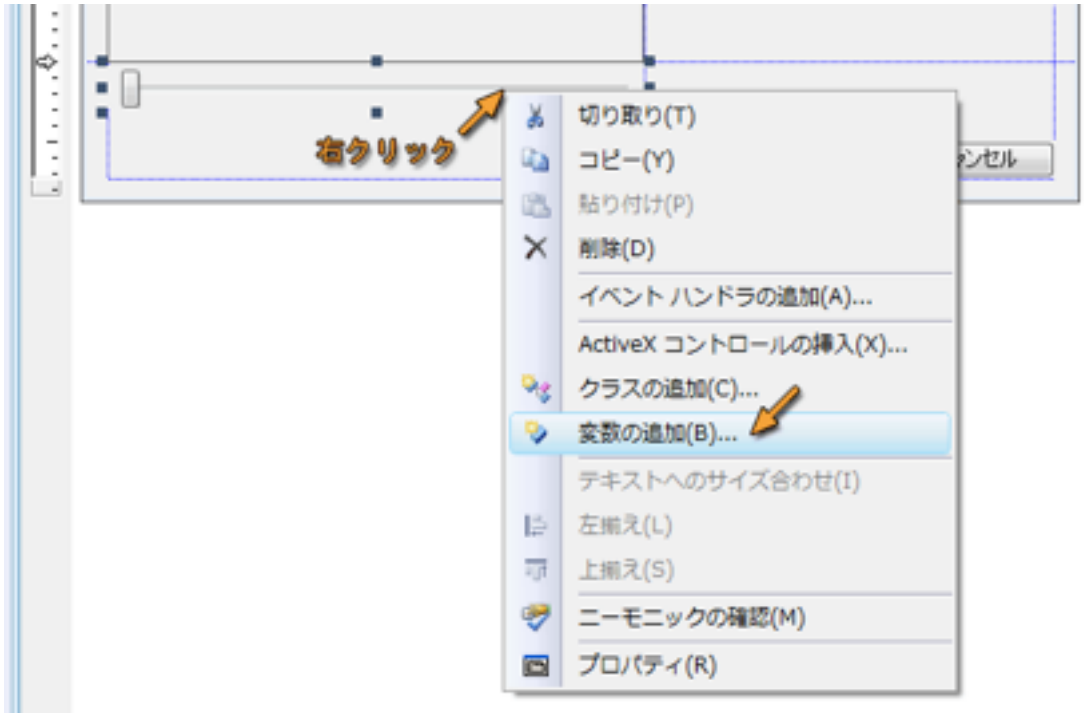


配置した「Slider Control」を右クリックして，「変数の追加」を選びます．



「アクセス」は "private"，「変数の種類」は最初から設定されている "CSliderCtrl" として，m_xcRotateZ という変数を追加します．コントロールと変数とのデータのやり取りに DDX というメカニズムを使うので，変数名に "x" を付けています．また "c" はこの変数のカテゴリが「Control」であることを示すんじゃないかと思います．最後に「完了」をクリックします．

再び「Slider Control」を右クリックして，「変数の追加」を選びます．

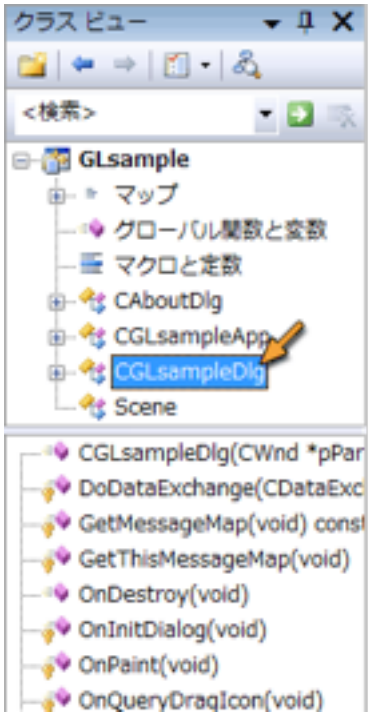


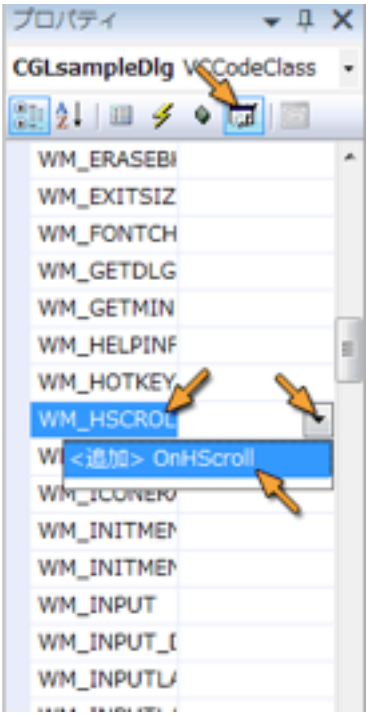
最初に「カテゴリ」から "Value" を選択します．そのあと「アクセス」に "private"，変数の種類に "int" を選びます．この変数を使って「Slider Control」と値をやり取りします．変数名は "m_xvRotateZ" とします．"v" は変数のカテゴリが "Value" ということを表すんじゃないかと思います．なお，この変数には -180 度〜 180 度の「角度」を入れるつもりなので，最大値と最小値をそれに設定します．が，意味あるのかな？


メンバ変数の追加 - GLsample
メンバ変数の追加ウィザードへようこそ
アクセス(A):
private
変数の種類(V):
int
変数名(N):
m_xvRotateZ
コントロール変数(C)
コントロール ID(I):
IDC_SLIDER1
コントロールの種類(Y):
msctls_trackbar32
最小値(U):
-180
最大値(E):
180
カテゴリ(T):
Value
最初にここを選択
コメント (// は必要ありません)(M):
完了
キャンセル

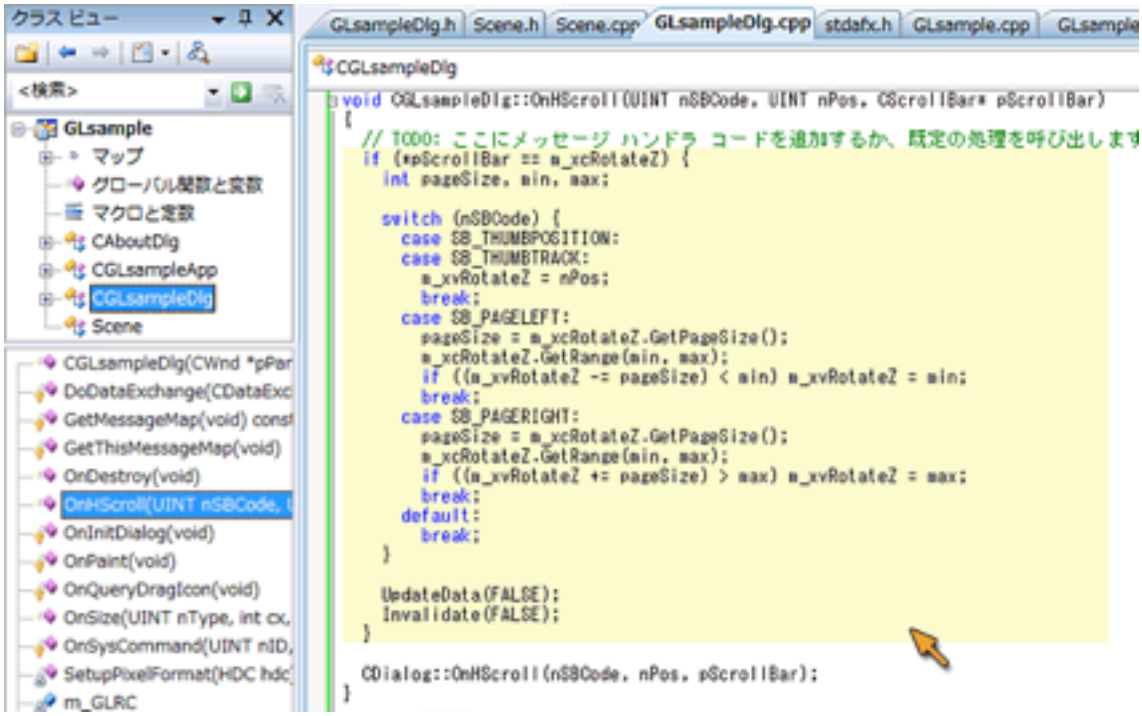
「SLIDER CONTROL」を使って図形を回転する

水平の「Slider Control」のツマミを動かすと WM_HSCROLL イベントが発生するので (垂直の「Slider Control」では WM_VSCROLL イベントが発生します), それを処理するハンドラを用意します. クラスビューで Cプロジェクト名Dlg クラス (ここでは CGLsampleDlg) を選択して, プロパティで WM_HSCROLL に OnHScroll を追加します.





OnHScroll() を実装します.



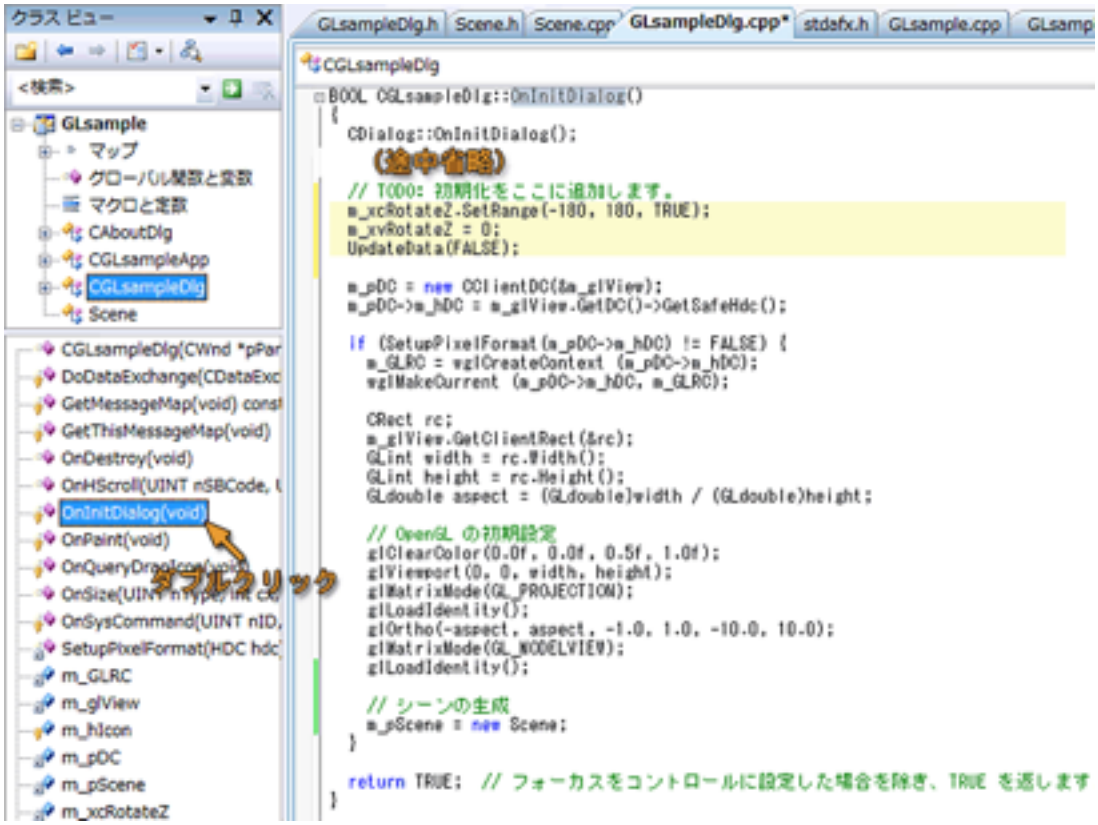
ウィンドウ上に (水平方向の)「Slider Control」や「Scroll Bar」が複数存在するとき, そのどれを動かしてもこの OnHScroll() が呼ばれます (Vista 以降であればコントロールごとにハンドラを指定できるみたいですけど). したがって, OnHScroll() ではどのコントロールが操作されたのかを判断する必要があります. pScrollBar にコントロール変数のポインタが入っているので, これを使ってコントロールを識別します.

「Slider Control」のツマミを動かしているときは nSBCode に SB_THUMBPOSITION か SB_THUMBTRACK が入っているので, この時は現在の位置 nPos を m_xvRotateZ に代入します. nSBCode が SB_PAGELEFT あるいは SB_PAGERIGHT は「Slider Control」上のツマミ以外の部分をクリックしたときなので, 「1 ページ分のジャンプ量 (「Slider Control」の実態はスクロールバーなので)」を求めて m_xvRotateZ に加算 / 減算します.

最後に UpdateData(FALSE) によりコントロールの設定値をコントロール自体に反映し, Invalidate(FALSE) で画面の再表示を行います. OpenGL の表示領域は OpenGL 自体で画面クリアを行いますので, Invalidate() の引数を FALSE にして, ここでは画面クリアを行わないようにします (ちらつくので).

```
void CGLsampleDlg::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
  // TODO: ここにメッセージ ハンドラ コードを追加するか、既定の処理を呼び出します。
  if (*pScrollBar == m_xcRotateZ) {
    int pageSize, min, max;

    switch (nSBCode) {
      case SB_THUMBPOSITION:
      case SB_THUMBTRACK:
        m_xvRotateZ = nPos;
        break;
      case SB_PAGELEFT:
        pageSize = m_xcRotateZ.GetPageSize();
        m_xcRotateZ.GetRange(min, max);
        if ((m_xvRotateZ -= pageSize) < min) m_xvRotateZ = min;
        break;
      case SB_PAGERIGHT:
        pageSize = m_xcRotateZ.GetPageSize();
        m_xcRotateZ.GetRange(min, max);
        if ((m_xvRotateZ += pageSize) > max) m_xvRotateZ = max;
        break;
      default:
        break;
    }

    UpdateData(FALSE);
    Invalidate(FALSE);
  }

  CDialog::OnHScroll(nSBCode, nPos, pScrollBar);
}
```

OnInitDialog(void) をダブルクリックして, その定義を変更します. ここでは m_xcRotateZ と m_xvRotateZ の初期設定を行います.

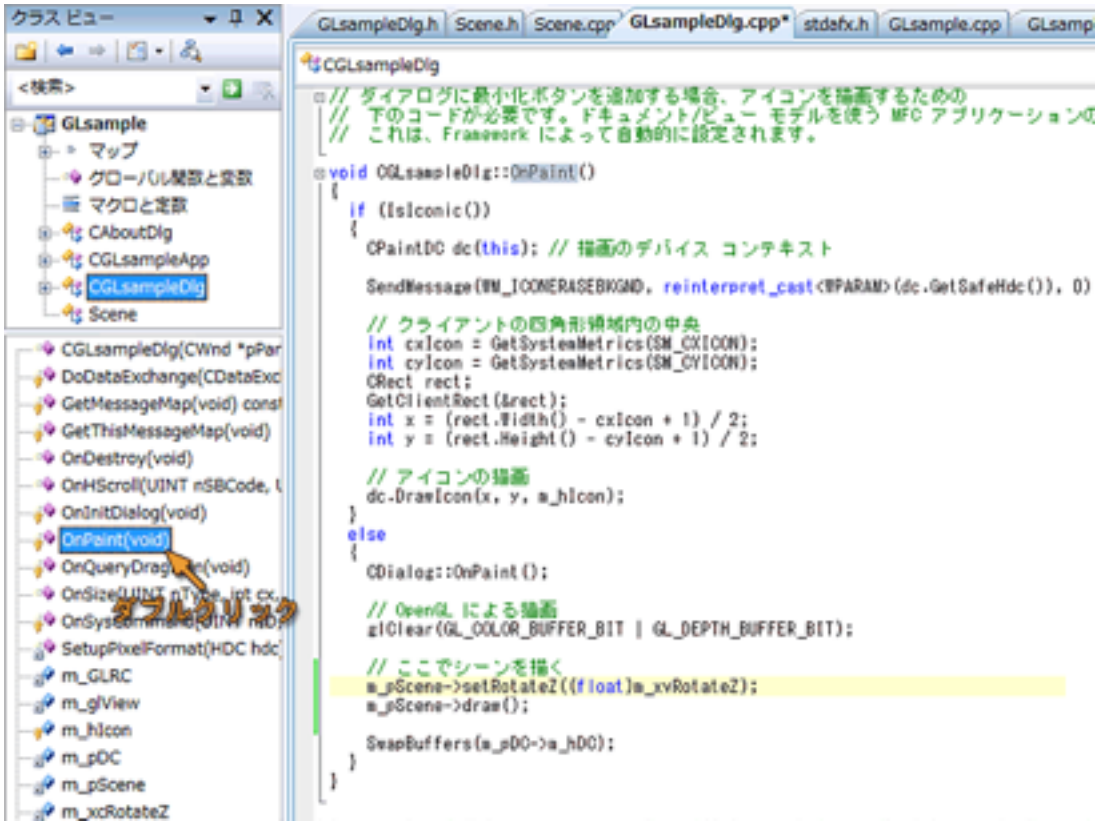


SetRange() メソッドはツマミの上限値と下限値を設定します. OnHScroll() の引数 nPos で得られる値はこの範囲を変化します.

```
BOOL CGLsampleDlg::OnInitDialog()
{
  CDialog::OnInitDialog();

  ...

  // TODO: 初期化をここに追加します。
  m_xcRotateZ.SetRange(-180, 180, TRUE);
  m_xvRotateZ = 0;
  UpdateData(FALSE);

  m_pDC = new CClientDC(&m_glView);
  if (SetupPixelFormat(m_pDC->m_hDC) != FALSE) {
    m_GLRC = wglCreateContext(m_pDC->m_hDC);
    wglMakeCurrent (m_pDC->m_hDC, m_GLRC);

    CRect rc;
    m_glView.GetClientRect(&rc);
    GLint width = rc.Width();
    GLint height = rc.Height();
    GLdouble aspect = (GLdouble)width / (GLdouble)height;

    // OpenGL の初期設定
    glClearColor(0.0f, 0.0f, 0.5f, 1.0f);
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-aspect, aspect, -1.0, 1.0, -10.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    // シーンの生成
    m_pScene = new Scene;
  }
```

```
    return TRUE;  // フォーカスをコントロールに設定した場合を除き、TRUE を返します。
}
```

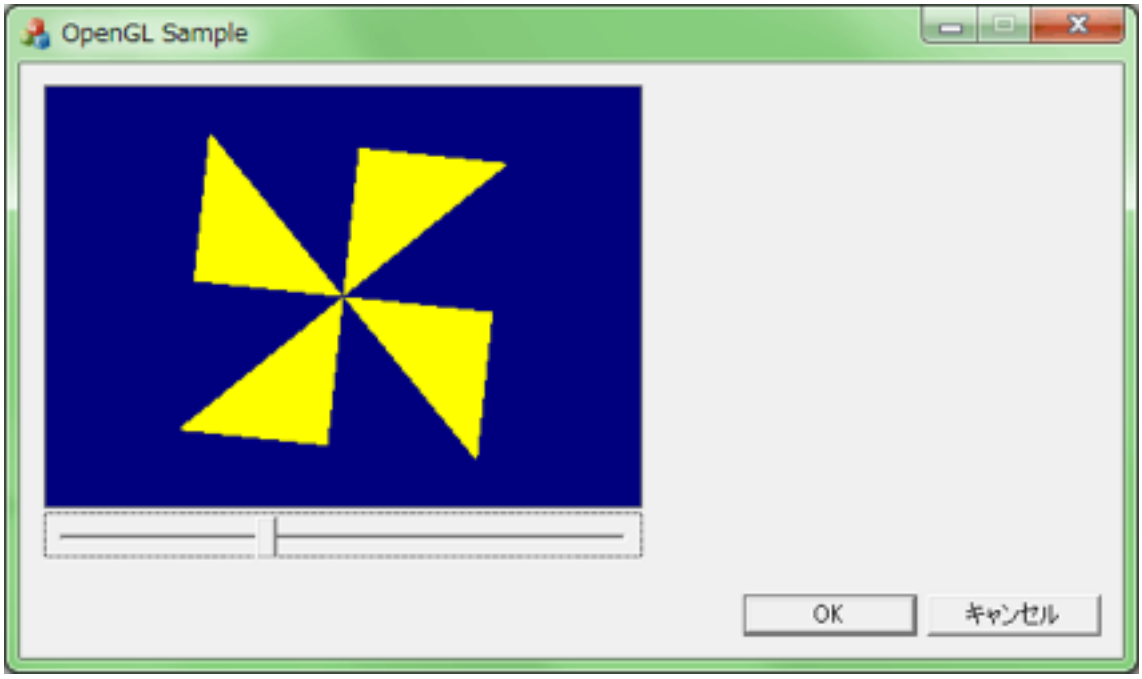
OnPaint(void) をダブルクリックして，その定義を変更します．



ここでは m_xvRotateZ の値を setRotateZ() メソッドの引数に与えて．シーンの回転角を設定します．

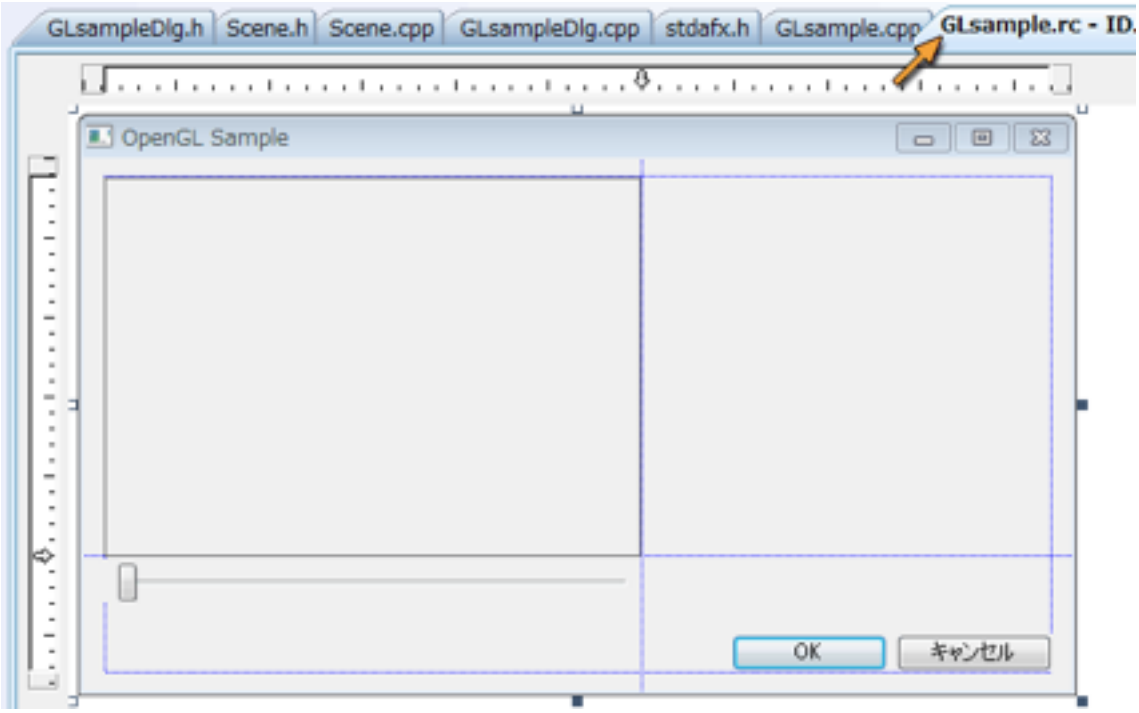
```
void CGLsampleDlg::OnPaint()
{
  if (IsIconic())
  {
    ...
  }
  else
  {
    CDialog::OnPaint();

    // OpenGL による描画
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // ここでシーンを描く
    m_pScene->setRotateZ((float)m_xvRotateZ);
    m_pScene->draw();

    SwapBuffers(m_pDC->m_hDC);
  }
}
```

ここまでできたら一旦ビルドして，プログラムを実行します．「Slider Control」のツマミを動かすと，図形が回転するでしょうか．

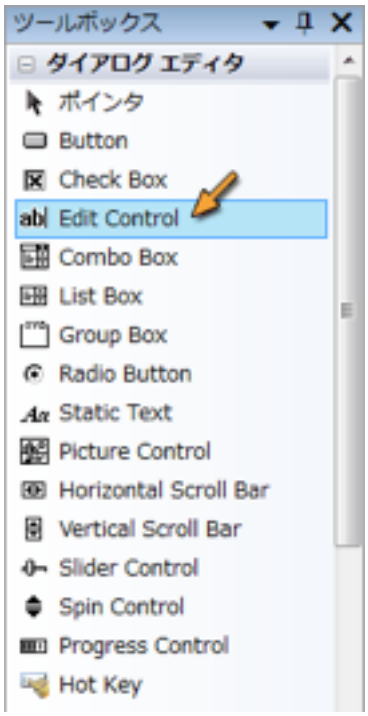

OpenGL Sample
OK
キャンセル

「EDIT CONTROL」を追加する

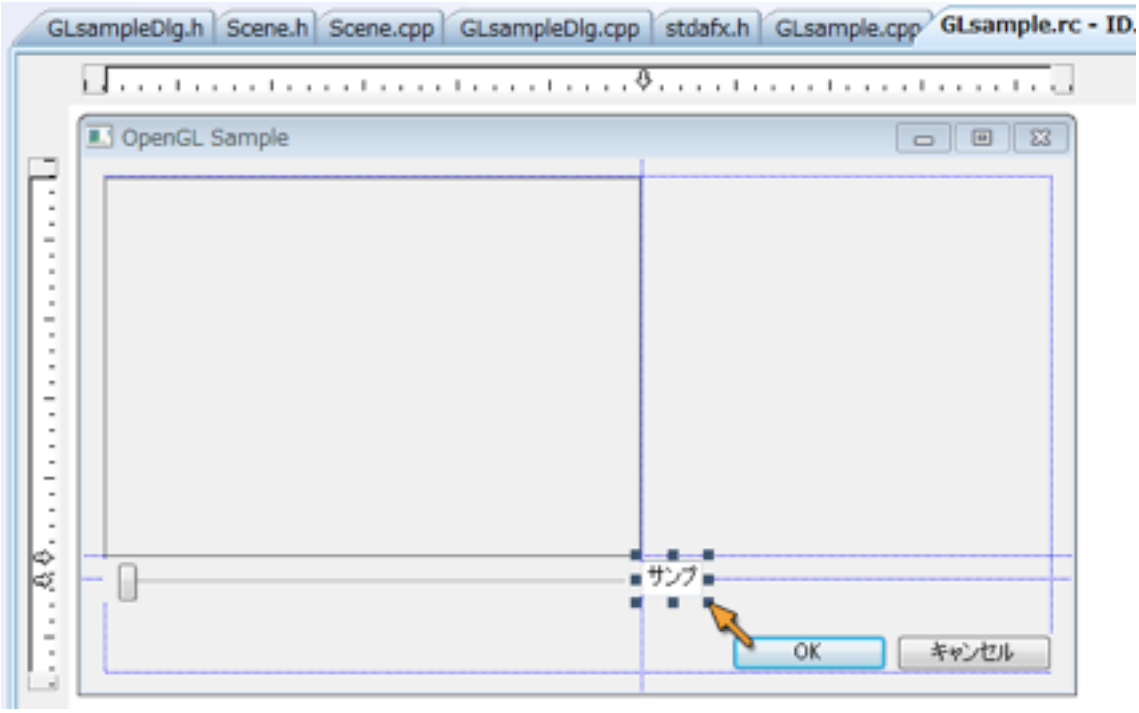
またダイアログエディタに切り替えます.



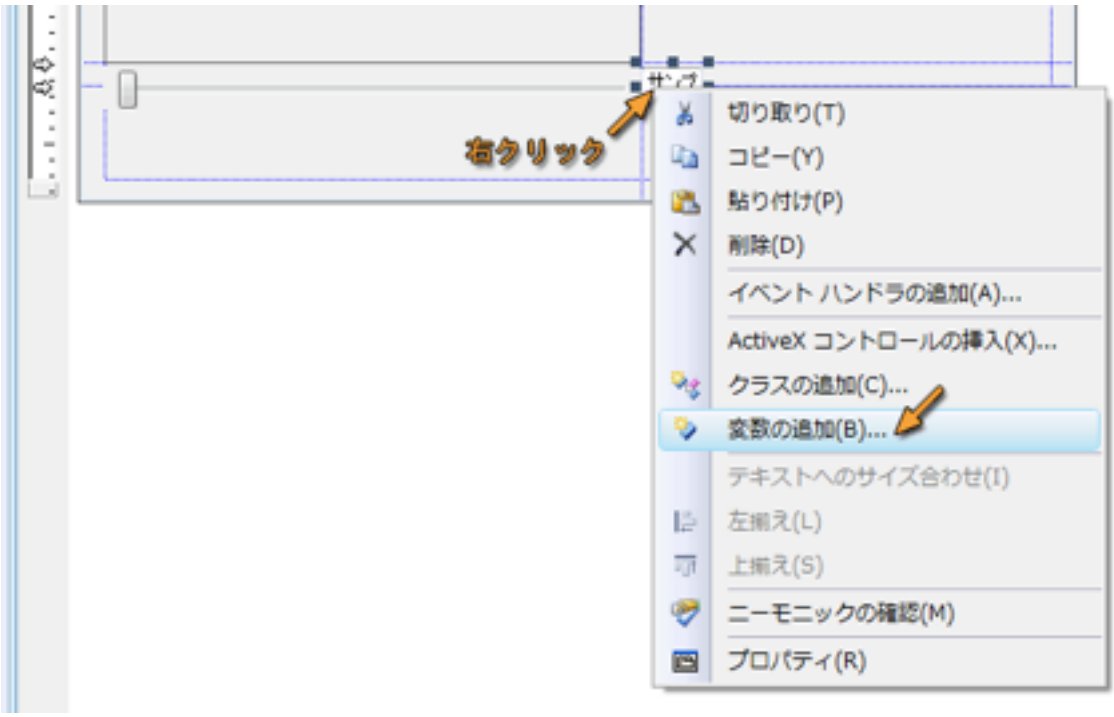
ツールボックスから，今度は「Edit Control」を選択します．「Edit Control」は文字の表示や入力を行います.



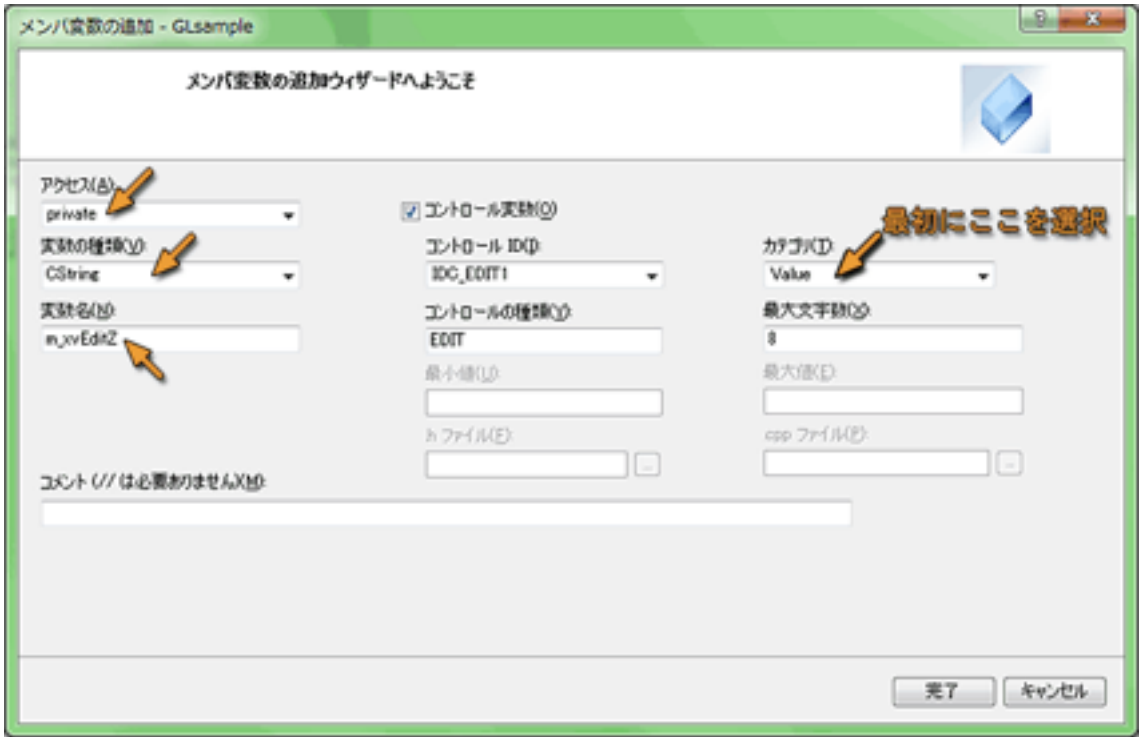
マウスを使って「Edit Control」を配置します.



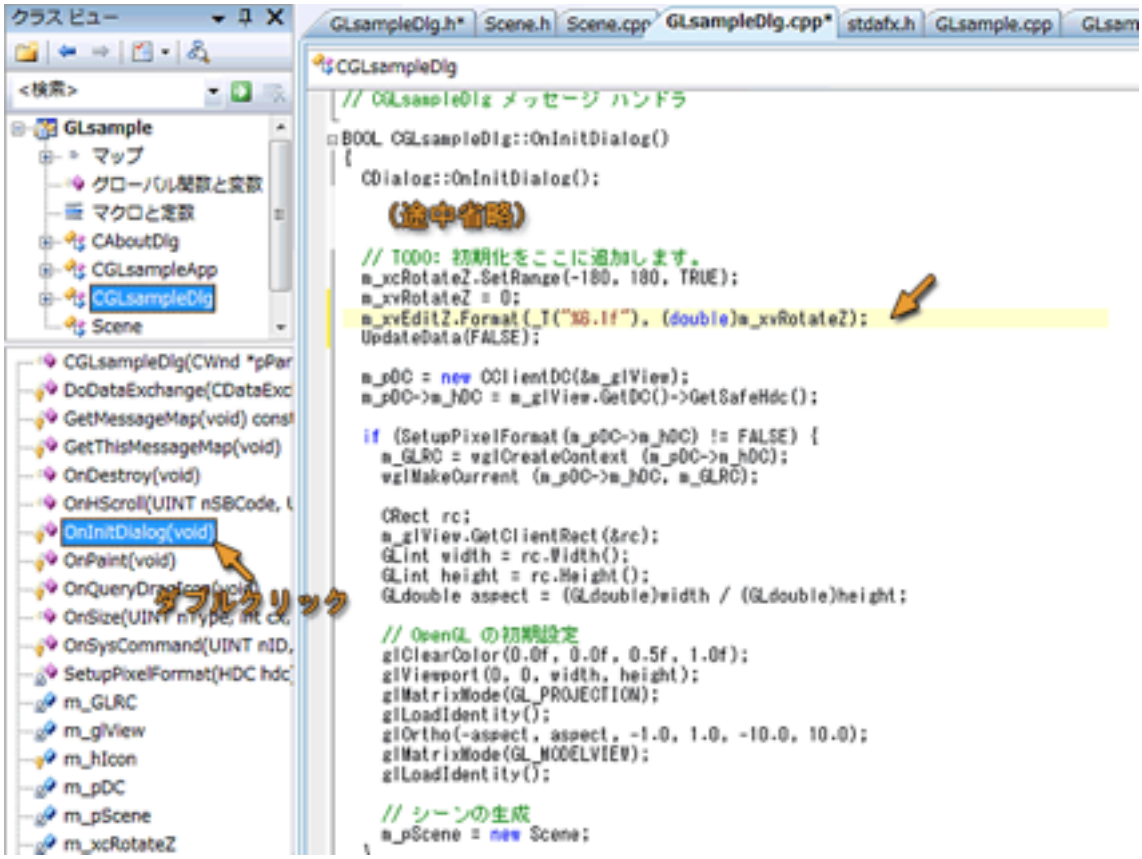
配置した「Edit Control」を右クリックして，「変数の追加」を選びます．



最初に「カテゴリ」から "Value" を選択します．そのあと「アクセス」に "private"，「変数の種類」に "CString" を設定して，m_xvEditZ という変数を追加します．この変数を使って「Edit Control」に文字列を表示したり文字列を取得したりします．最後に「完了」をクリックします．



OnInitDialog(void) をダブルクリックして，その定義を変更します．ここでは m_xvEditZ の初期設定を行います．



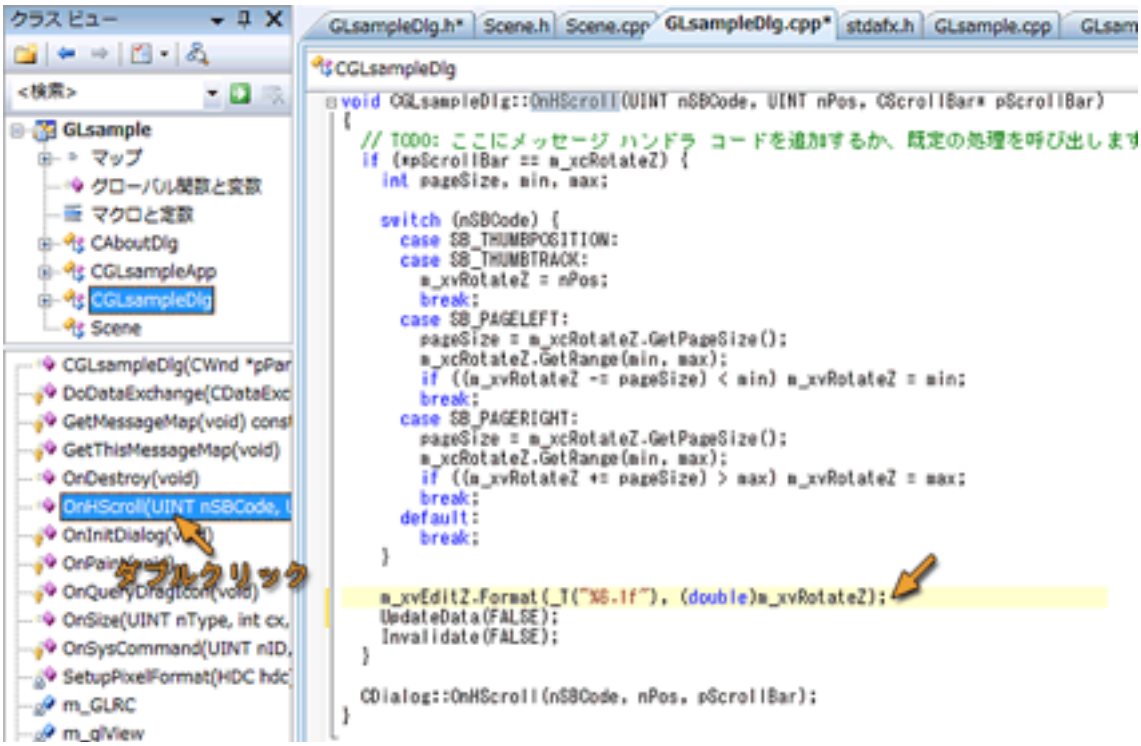
m_xvEditZ は CString 型すなわち文字列なので，角度を保持している m_xvRotateZ を文字列に変換してこれに格納します．Format() メソッドの第 1 引数は printf() と同様の書式文字列

です．このプロジェクトは文字集合として (デフォルトの) Unicode を使用する設定になっているので, _T() を使って変換しています．

```
BOOL CGLsampleDlg::OnInitDialog()
{
  CDialog::OnInitDialog();

  ...

  // TODO: 初期化をここに追加します。
  m_xcRotateZ.SetRange(-180, 180, TRUE);
  m_xvRotateZ = 0;
  m_xvEditZ.Format(_T("%6.1f"), (double)m_xvRotateZ);
  UpdateData(FALSE);

  m_pDC = new CClientDC(&m_glView);
  if (SetupPixelFormat(m_pDC->m_hDC) != FALSE) {
    m_GLRC = wglCreateContext (m_pDC->m_hDC);
    wglMakeCurrent (m_pDC->m_hDC, m_GLRC);

    CRect rc;
    m_glView.GetClientRect(&rc);
    GLint width = rc.Width();
    GLint height = rc.Height();
    GLdouble aspect = (GLdouble)width / (GLdouble)height;

    // OpenGL の初期設定
    glClearColor(0.0f, 0.0f, 0.5f, 1.0f);
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-aspect, aspect, -1.0, 1.0, -10.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    // シーンの生成
    m_pScene = new Scene;
  }

  return TRUE;  // フォーカスをコントロールに設定した場合を除き、TRUE を返します。
}
```

「SLIDER CONTROL」の設定値を「EDIT CONTROL」に表示する

OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar) をダブルクリックして，その定義を変更します．

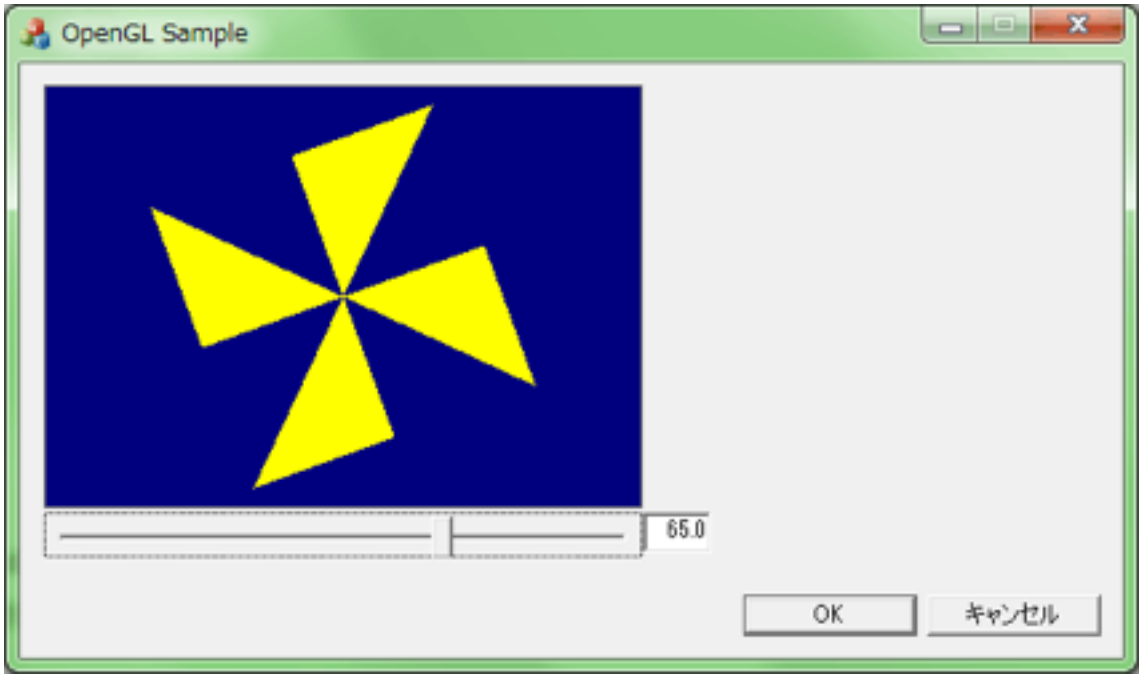


「Slider Control」のツマミを動かした位置 m_xvRotateZ を文字列に直して m_xvEditZ に設定します．これでツマミを動かしたときに，その値が「Edit Control」に表示されます．

```
void CGLsampleDlg::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
  // TODO: ここにメッセージ ハンドラ コードを追加するか、既定の処理を呼び出します。
  if (*pScrollBar == m_xcRotateZ) {
    int pageSize, min, max;

    switch (nSBCode) {
      case SB_THUMBPOSITION:
      case SB_THUMBTRACK:
        m_xvRotateZ = nPos;
        break;
      case SB_PAGELEFT:
        pageSize = m_xcRotateZ.GetPageSize();
        m_xcRotateZ.GetRange(min, max);
        if ((m_xvRotateZ -= pageSize) < min) m_xvRotateZ = min;
        break;
      case SB_PAGERIGHT:
        pageSize = m_xcRotateZ.GetPageSize();
        m_xcRotateZ.GetRange(min, max);
        if ((m_xvRotateZ += pageSize) > max) m_xvRotateZ = max;
        break;
      default:
        break;
    }

    m_xvEditZ.Format(_T("%6.1f"), (double)m_xvRotateZ);
    UpdateData(FALSE);
    Invalidate(FALSE);
```

```
        }

        CDialog::OnHScroll(nSBCode, nPos, pScrollBar);
    }
```

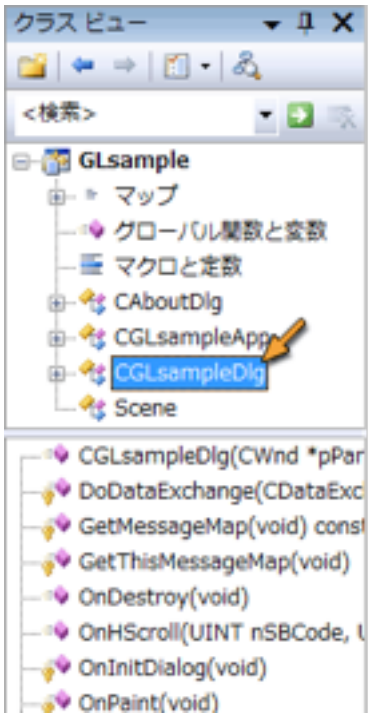
プログラムをビルドして，「Slider Control」のツマミを動かしてみます．図形が回転すると同時に，「Edit Control」の数値が変化すると思います．【8月26日追記】Invalidate(FALSE); の後で UpdateWindow(); を実行すれば，ツマミの操作がすぐに「Edit Control」に反映されます．処理が重くなる気がしますけど．

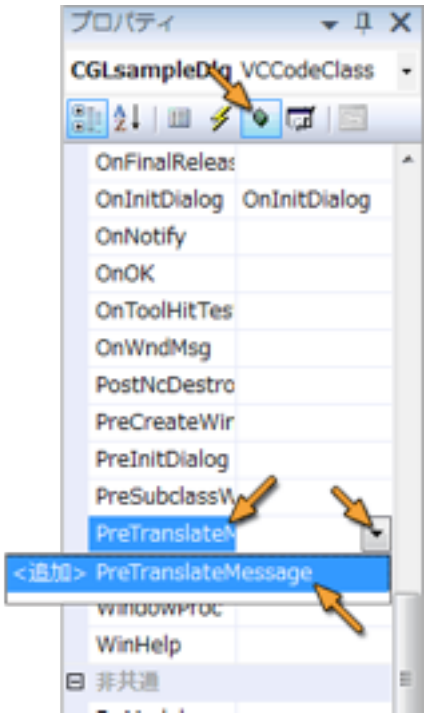


「EDIT CONTROL」に数値を入力する

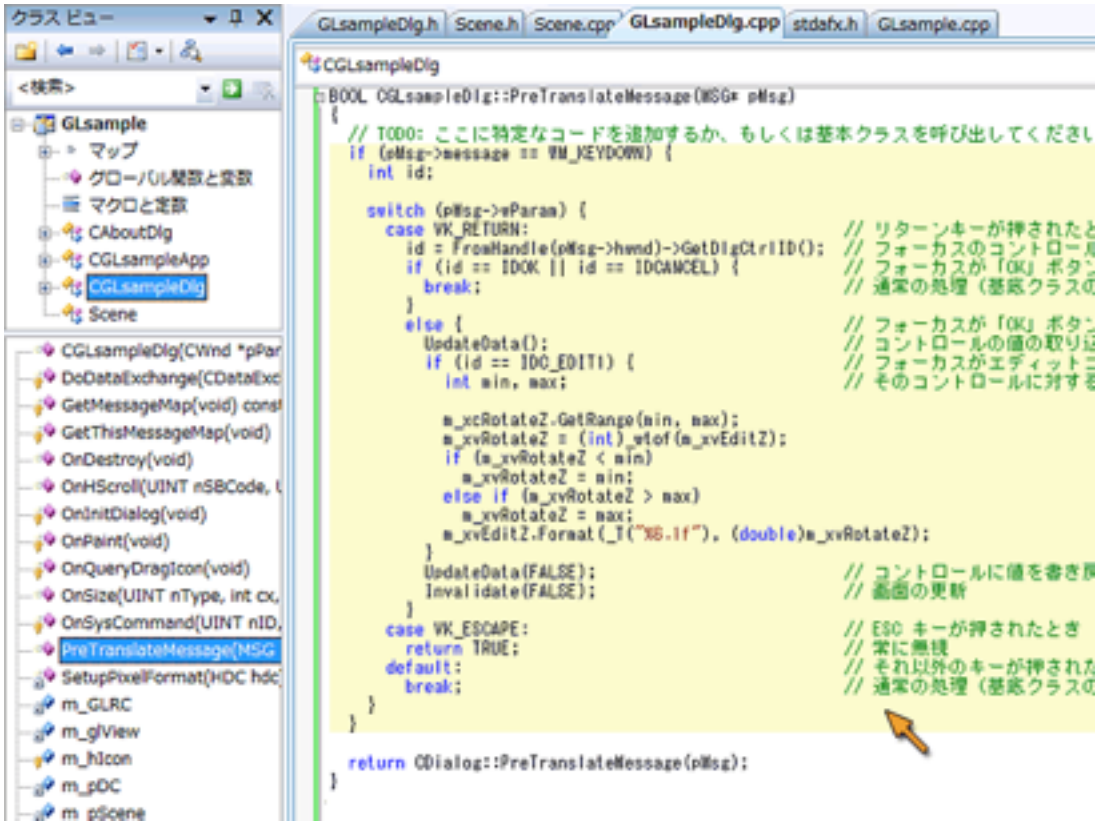
逆に，「Edit Control」に入力した数値が「Slider Control」や表示されている図形に反映されるようにします．ここで重要な問題があります．ダイアログアプリケーションの場合，リターン (Enter) キーや ESC キー，Tab キーなどは，フォーカスされている (操作の対象となっている) コントロールがどれであっても，そのキーに割り当てられた操作が実行されてしまいます．リターンキーには「OK」ボタンのクリック，ESC キーには「キャンセル」ボタンのクリック，Tab キーにはフォーカスの移動が割り当てられています．このため，特定のコントロールでリターンキーの入力を検出することができなかったり，ESC をタイプするといつでもダイアログウィンドウが閉じてしまったりします．

そこで，キー入力がコントロールで処理される前に呼び出される PreTranslateMessage() をオーバーライドして，キー入力を補足するようにします．まずクラスビューで Cプロジェクト名Dlg クラス (ここでは CGLsampleDlg) を選択して，プロパティの左から 5 つ目のボタンをクリックし，PreTranslateMassage を選択して右の▼から PreTranslateMessage を追加してください．





PreTranslateMessage() を実装します．



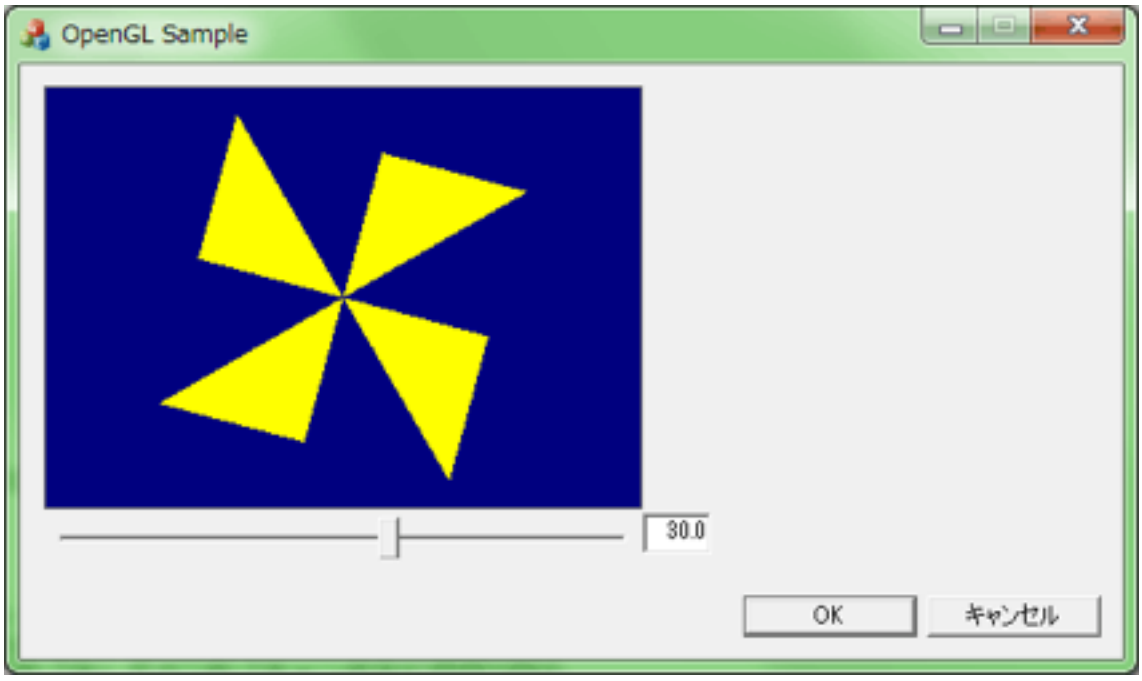
PreTranslateMessage(MSG* pMessage) がキータイプにより呼び出された場合は，pMsg->message が WM_KEYDOWN になっています. このとき pMsg->wParam にタイプされたキーが格納され, pMsg->hwnd にフォーカスされているコントロールの Window ハンドルが格納されています.

そこで pMsg->hwnd からそのコントロール ID を求め, それを使ってコントロールを識別します. 実はこの方法が正しいのかどうかも自信がアリマセン. FromHandle(pMsg->hwnd)->GetDlgCtrlID() とするより ::GetDlgCtrlID(pMsg->hwnd) とした方が手っ取り早い気がしますが, "::" が使いたくありませんでした.

```
BOOL CGLsampleDlg::PreTranslateMessage(MSG* pMsg)
{
  // TODO: ここに特定なコードを追加するか、もしくは基本クラスを呼び出してください。
  if (pMsg->message == WM_KEYDOWN) {
    int id;

    switch (pMsg->wParam) {
      case VK_RETURN:                                  // リターンキーが押されたとき
        id = FromHandle(pMsg->hwnd)->GetDlgCtrlID();   // フォーカスのコントロールID
        if (id == IDOK || id == IDCANCEL) {            // 「OK」か「キャンセル」なら
          break;                                       // 通常の処理
        }
        else {                                         //「OK」か「キャンセル」以外
          UpdateData();                                // コントロールの値の取り込み
          if (id == IDC_EDIT1) {                       // エディットコントロールなら
            int min, max;                              // そのコントロールに対する処理

            m_xcRotateZ.GetRange(min, max);
            m_xvRotateZ = (int)_wtof(m_xvEditZ);
```

```
            if (m_xvRotateZ < min)
              m_xvRotateZ = min;
            else if (m_xvRotateZ > max)
              m_xvRotateZ = max;
            m_xvEditZ.Format(_T("%6.1f"), (double)m_xvRotateZ);
          }
          UpdateData(FALSE);                          // コントロールに値を書き戻す
          Invalidate(FALSE);                          // 画面の更新

        }
      case VK_ESCAPE:                                 // ESC キーが押されたとき
        return TRUE;                                  // 常に無視
      default:                                        // それ以外のキーが押されたとき
        break;                                        // 通常の処理
    }
  }

  return CDialog::PreTranslateMessage(pMsg);
}
```

これでプログラムをビルドして実行します．「Edit Control」に数値を入力してリターンキーをタイプすれば，「Slider Control」と表示図形が変化するでしょうか．



おわりに

言い訳になってしまいますが, 初めて MFC って初めて Pages で EPUB 使ったので, とても不出来なものになってしまいました. ごめんなさい.