ゲームプログラミング

▶　DirectDraw－第１０回　クリッパー　◀

DirectDrawにはクリッパーという機能があり、設定した領域の外にはみ出た部分を自動的に切り取ることができます。この機能を使うと、サーフェイスの特定の部分だけ描画するようにすることができます。

クリッパー

BltメソッドやBltFastメソッドで画像を転送したとき、サーフェイスの外の座標や領域を指定するとエラーになり、転送されません。これを防ぐには、座標や領域を調べて範囲外にならないように計算してから転送する必要があります。
DirectDrawにはクリッパーという機能があります。クリッパーを設定すると、指定した領域の外にはみ出た画像を自動的に切り取って転送する「クリッピング機能」が働くようになります。サーフェイスのサイズと同じ領域をクリッパーに設定しておけば、サーフェイスの外に出た部分を自動的に切り取るようになり、エラーにならなくなります。クリッピング領域は、複数の領域を設定することもできます。

クリッパーの生成と解放

クリッパーはDirectDrawClipperオブジェクトで管理します。このオブジェクトは、DirectDrawオブジェクトのCreateClipperメソッドで生成します。このメソッドは１つ目の引数に０、２つ目の引数に生成されるクリッパーオブジェクトのインタフェースを受け取る変数(LPDIRECTDRAWCLIPPER型)のアドレス、３つ目の引数にNULLを指定します。

```
LPDIRECTDRAW7        lpDDraw7;              // DirectDrawオブジェクト(初期化済みとする)
LPDIRECTDRAWCLIPPER  lpDDClipper = NULL;    // DirectDrawクリッパーオブジェクト
lpDDraw7->CreateClipper(0, &lpDDClipper, NULL);
```

このオブジェクトの解放は、ほかのオブジェクトと同じようにReleaseメソッドで行います。

クリッピング領域の設定

クリッピング領域の設定は、DirectDrawClipperオブジェクトのSetClipListメソッドで行います。

SetClipListメソッド
－説明－
SetClipListメソッドは、クリップを行う領域(クリッピングリスト)の設定または解除を行います。

－パラメータ－
１つ目の引数は、クリッピング情報を格納したRGNDATA構造体変数のアドレスです。NULLの場合はクリッピングリストを解除し、クリッピング機能を無効にします。
２つ目の引数は、現時点では使われていないので０にします。

－戻り値－
成功した場合はDD_OK、それ以外はエラーの原因をエラーコードとして返します。

クリッピングリストの情報はRGNDATA構造体で指定します。この構造体は、RGNDATAHEADER構造体を先頭に、クリッピングを行う領域の数だけRECT構造体が続く構成となっています。

▲RGNDATA構造体の構成

RGNDATAHEADER構造体は以下のメンバで構成されます。

名前	データ型	機　能
dwSize	DWORD	この構造体のサイズ(バイト数)
iType	DWORD	領域の種類。RDH_RECTANGLESを指定
nCount	DWORD	クリッピングする矩形領域の数
nRgnSize	DWORD	領域１つのサイズを指定。sizeof(RECT)
rcBound	RECT	クリッピング領域のいちばん外側を指定

```
RGNDATA   rdClipList;      // クリップリスト(設定済みとする)
lpDDClipper->SetClipList(&rdClipList, 0);
```

SetClipListメソッドにより、クリッパーオブジェクトにクリッピング領域が設定されます。しかし、このままでは効果が現れません。クリッパーは、サーフェイスに接続することによってはじめて有効になります。
サーフェイスにクリッパーを接続するには、サーフェイスオブジェクトのSetClipperメソッドで行います。引数はDirectDrawClipperオブジェクトを指定します。

```
LPDIRECTDRAWSURFACE7   lpDDSBackBuffer;      // バックバッファオブジェクト(初期化済みとする)
LPDIRECTDRAWCLIPPER    lpDDClipper;          // DirectDrawクリッパーオブジェクト(初期化済みとする)
lpDDSBackBuffer->SetClipper(lpDDClipper);    // バックバッファにクリッパーを設定
```

クリッパーを設定したサーフェイスには、Bltメソッドでしか転送ができなくなります。BltFastメソッドは必ずエラーになります。また、環境によっては、クリッパーを設定すると転送速度が落ちる場合があります。

○ 課題 ○

クリッピング機能を追加しましょう。

(１)以下のプログラムを適切な場所に追加しましょう。

－追加１－
```
static LPDIRECTDRAWCLIPPER   g_lpDDClipper = NULL;
```

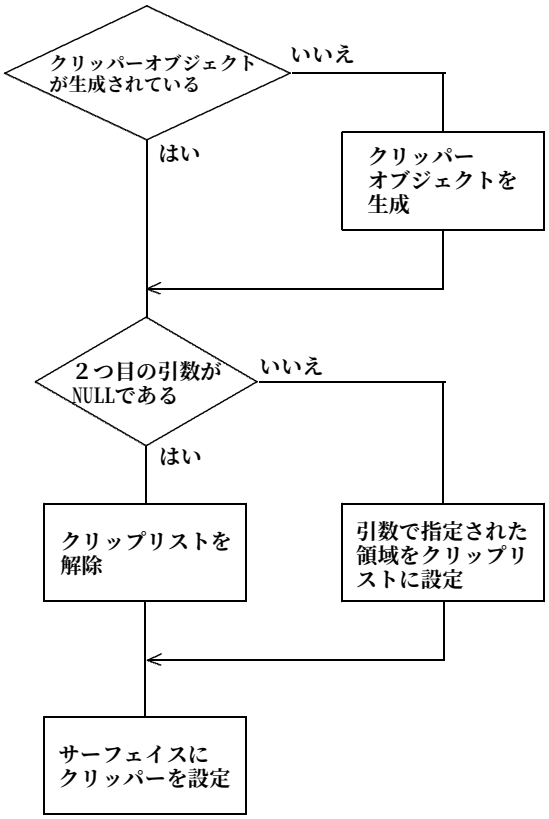
(２)以下のプログラムは、クリッピング領域の設定を行うDDSetClipper関数です。関数の仕様とフローチャートをよく読んで完成させ、適切な場所に追加しましょう。

DDSetClipper関数
－説明－
DDSetClipper関数は、クリッピング領域の設定または解除を行います。

－パラメータ－
DDSRFC dds…クリッパーを有効にするサーフェイス。DDSRFC列挙型を指定
LPRECT lprcClip…クリッピング領域を設定したRECT構造体配列の先頭アドレス。NULLの場合はクリッピングを解除します
const int nClipCnt…クリッピング領域の数。２つ目の引数がNULLの場合は無視されます

－戻り値－
関数が成功した場合はtrue，それ以外はfalseを返す。

－フローチャート－



－追加２－

```
/*****************************************************************************************************/
/*                                          クリッパー設定                                           */
/*****************************************************************************************************/
bool DDSetClipper(DDSRFC dds, LPRECT lprcClip, const int nClipCnt)
{
#ifdef _DEBUG
    if(NULL == g_lpDDraw7 || NULL == g_lpDDSurface7[dds]) {
        OutputDebugString("*** Error - DirectDrawまたは対象サーフェイス未初期化(DDSetClipper)\n");
        return false;
    }
#endif

    // DirectDrawClipperオブジェクト生成
    if(NULL == g_lpDDClipper) {
        if(DD_OK != ※ここは各自考えましょう) {
            OutputDebugString("*** Error - Clipperオブジェクト生成失敗(DDSetClipper)\n");
            return false;
        }
    }

    if(NULL != lprcClip) {
        // クリッピング情報(RGNDATA)設定
        // RGNDATA構造体メモリ確保
        LPBYTE   lpbyClipList = new BYTE[sizeof(RGNDATAHEADER) + sizeof(RECT) * nClipCnt];
        if(NULL == lpbyClipList) {
            OutputDebugString("*** Error - メモリ確保失敗(DDSetClipper)\n");
            return false;
        }

        // サーフェイス情報取得
        DDSURFACEDESC2   ddsd2;
        ddsd2.dwSize = sizeof(ddsd2);
        g_lpDDSurface7[dds]->GetSurfaceDesc(&ddsd2);

        // RGNDATAHEADER設定
```

```
        RGNDATAHEADER*   lprdh = (RGNDATAHEADER*)lpbyClipList;
        lprdh->dwSize          = sizeof(RGNDATAHEADER);
        lprdh->iType           = RDH_RECTANGLES;
        lprdh->nCount          = nClipCnt;
        lprdh->nRgnSize        = sizeof(RECT);
        lprdh->rcBound.left    = 0;
        lprdh->rcBound.top     = 0;
        lprdh->rcBound.right   = ddsd2.dwWidth;
        lprdh->rcBound.bottom  = ddsd2.dwHeight;

        // クリッピング領域(RECT配列)設定
        CopyMemory(lpbyClipList + sizeof(RGNDATAHEADER), lprcClip, sizeof(RECT) * nClipCnt);

        // クリップリスト設定
        const HRESULT   hr = g_lpDDClipper->???????????((LPRGNDATA)lpbyClipList, 0);
        delete[] lpbyClipList;
        if(DD_OK != hr) {
            OutputDebugString("*** Error - クリップリスト設定失敗(DDSetClipper)\n");
            return false;
        }
    } else {
        // クリップリスト解除
        if(DD_OK != ※ここは各自考えましょう) {
            OutputDebugString("*** Error - クリップリスト解除失敗(DDSetClipper)\n");
            return false;
        }
    }

    // サーフェイスにクリッパーを設定
    if(DD_OK != ※ここは各自考えましょう) {
        OutputDebugString("*** Error - クリッパー設定失敗(DDSetClipper)\n");
        return false;
    }

    return true;
}
```

(３)クリッパーオブジェクトの解放を行う以下のプログラムを完成させ、適切な場所に追加しましょう。

－追加３－

```
    // クリッパー解放
    if(NULL != g_lpDDClipper) {
        g_lpDDClipper->???????();
        g_lpDDClipper = NULL;
    }
```

(４)Test関数に以下のプログラムを追加し、クリッピング機能が働くことを確認しましょう。

```
    // クリッパー設定
    // 転送可能領域設定
    RECT   rcClipList[5] = {{0,   0, 214, 160}, {426,   0, 640, 160}, {214, 160, 426, 320},
                            {0, 320, 214, 480}, {426, 320, 640, 480}};
    DDSetClipper(DDS_BACKBUF, rcClipList, 5);
```

※このプログラムの実行後は、バックバッファに対してBltFastメソッドが使用できなくなります。

資料－newとdelete－

C++のnew演算子とdelete演算子は、おもにメモリの管理を行う演算子です。newはメモリを確保し、deleteは反対に確保した領域を解放します。この演算子によって変数を生成することができ、この場合、プログラマが変数の寿命を管理することができます。また、C言語のmalloc関数とfree関数と違い、領域確保時にコンストラクタ、領域解放時にデストラクタが呼び出されます。

new演算子は、以下の形式で使われます。

```
new 型
new 型 初期化子
new 型 [要素数]
```

いずれの場合も、指定された型を記憶するのに適した容量のメモリがＯＳから割り当てられます。そして、その変数のアドレスが返されます。メモリが空いていない場合、newはNULLを返すか、例外を発生させます。

```
int*   p;
int*   q;
int*   r;
p = new int;          // intと同じサイズの領域を割り当てる
q = new int(5);       // 割り当てと初期化
r = new int[10];      // r[0]からr[9]を取得、q = &r[0]になる
```

上のコードでは、intへのポインタ変数ｐに、int型の割り当てで獲得したメモリのアドレスを代入します。同じくintへのポインタ変数ｑは、int型の割り当てで獲得したメモリのアドレスを代入し、さらに値５で初期化されます。しかし、このような用法は一般的ではありません。new演算子での領域の割り当ては、要素の配列をポインタｒに割り当てるような場合に使うのが一般的です。特に、大きな領域が必要なときに使用します。

delete演算子はnewで生成した領域を解放し、割り当てられたメモリを再利用できるようにＯＳに返します。delete演算子には、以下の２つの形式があります。

```
delete     newで割り当てた要素
delete []  newで割り当てた要素
```

最初の形式は、対応するnewの式が配列以外の割り当ての場合に使います。２つ目の形式は割り当てが配列の場合に使います。

```
delete   p;      // pの解放
delete   q;      // qの解放
delete[] r;      // rの解放
```

newで割り当てた領域は、deleteで解放するまで寿命が続きます。解放しないでプログラムが終了した場合でも、メモリにそのまま残ってしまう場合があります。

newとdeleteの使用例

ある要素の集合を表現するとき、配列がよく用いられます。

```
int    array[10];     // intを10個
```

配列はとても便利ですが、欠点の１つは、要素個数が固定であることです。プログラムを実行しないと必要とする要素の個数が明らかにならない場合、配列を使うことができないので、必要な領域を未使用のメモリから動的に確保しなければなりません。このような場合にnewとdeleteを使います。

```
const int   N = 必要な個数;
// 領域確保
int*   array = new int[N];
```

こうすると、arrayは配列のように使うことができます。

```
int   sum = 0;
for(int i = 0; i < N; i++) {
    sum += array[i];
}
```

確保した領域は、必要がなくなったら必ず解放します。

```
// 領域解放
delete[] array;
array = NULL;
```