GTL Draw

Nagyprogram Fejlesztői Dokumentáció

Készítette:

Vatai Emil

ELTE Informatikai Kar

Programtervező Matematikus

Konzulens:

Dr. Porkoláb Zoltán, egyetemi docens

ELTE Informatikai Kar

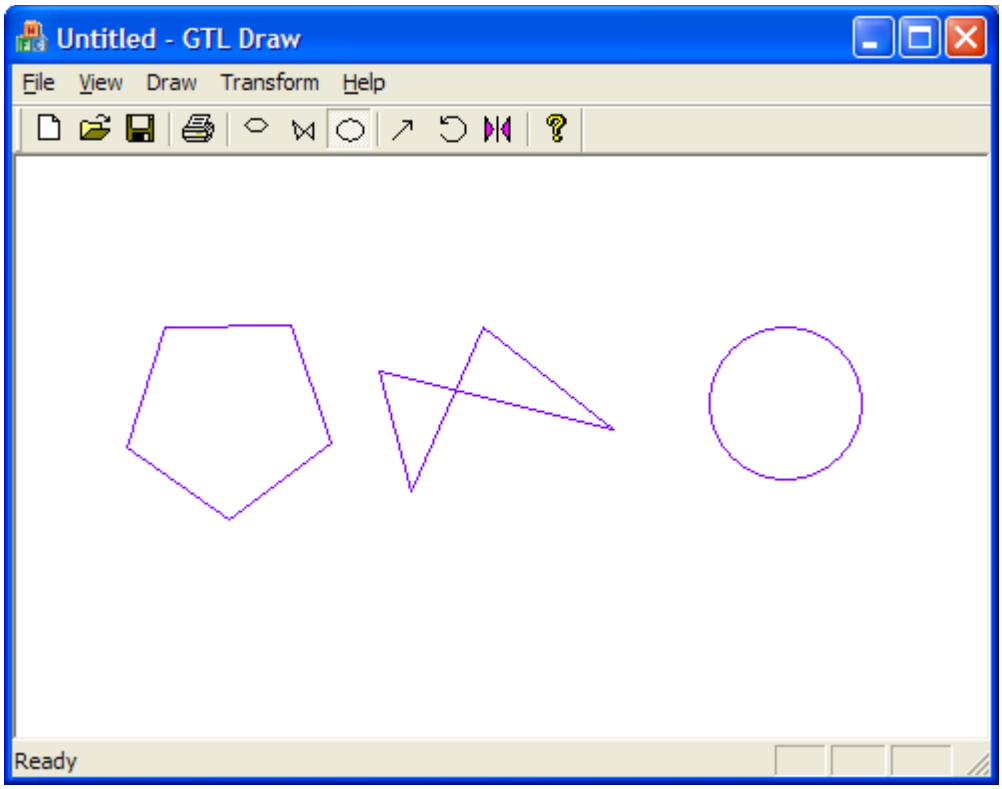
Programozási Nyelvek és Fordítóprogramok Tanszék

Budapest, 2007. május 21.

Tartalomjegyzék

GTL Draw

GTL a „Graphical Template Library” rövidítése, és a GTL Draw egy példaprogram, amely ezen a „grafikus sablon könyvtáron” alapszik. A GTL Draw egy egyszerű rajzoló program, amely a GTL alakzatait képes kirajzolni és rajtuk a GTL algoritmusait végrehajtani. A program célja nem kifejezetten egy minden szempontból hatékony és felhasználó barát rajzprogram elkészítése, hiszen ilyet könnyedén lehet, akár ingyen az Internetről, szerezni, hanem a GTL könyvtár alkalmazásának, lehetőségeinek és korlátinak bemutatása.



Fejlesztési eszközök

Fejlesztői környezet:	Microsoft Visual Studio .NET 2003
Program nyelv:	C++ [1]
Keretrendszer:	MFC – Microsoft Foundation Classes [5]
Könyvtár:	GTL – Graphical Template Library [3]
	GDI+ [7]

A GTL könyvtár szabványos ISO C++ nyelven van írva. Teljes mértékben platform és környezet független, bármilyen C++ programmal használható.

MFC keretrendszer

Az MFC, azaz Microsoft Foundation Classes (Library), eredetileg Application Framework eXtention (AFX), a Microsoft által fejlesztett könyvtára, amely a Windows API-t burkolja C++ osztályokba. Az MFC-re úgy is gondolhatunk, mint egy Windows specifikus C++ bővítés, amely a virtuális függvények kezelését próbálja megoldani üzenetekkel [6]. Az MFC egyik fő előnye a .NET-tel szemben az, hogy hatékonyabb kódot generál, azaz gyorsabb programokat amely egy rajzprogramnál kulcsszerepet játszhat.

Document/View architektúra

Az MFC alapból három osztályt ad a kiindulási pontban, a program felépítéséhez:

```
class CGTLDrawApp : public CWinApp;
class CGTLDrawDoc : public CDoc;
class CGTLDrawView : public CView;
```

Ezek az osztályok alkotják a Documentum/View (dokumentum/nézet) architektúrát. A `CGTLDrawApp` osztály az alkalmazást reprezentálja, a `CGTLDrawDoc` osztály a rajz dokumentumokat reprezentálja, míg a `CGTLDrawView` osztály a megjelenítést és a kezelői felületét kezeli. A GTL Draw program elkészítéséhez a `CGTLDrawDoc` és `CGTLDrawView` osztályokat kellett módosítani, azaz bővíteni.

A program szerkezetének informális leírása

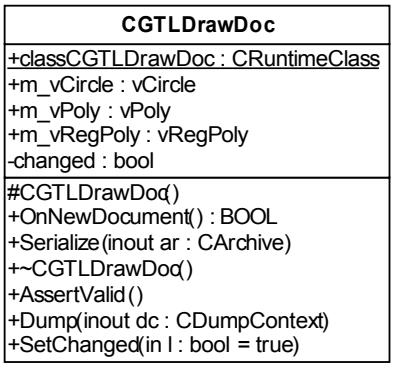
A program fontosabb funkciói: új rajz létrehozása, rajz elmentése, rajz betöltése, rajz nyomtatása, alakzatok rajzolása (hozzáadása a rajzhoz), a rajz módosítása. Három fajta alakzatot tudunk rajzolni, (szabálytalan) sokszöget, szabályos sokszöget, kört és három féle módosítást tudunk végrehajtani ezeken az alakzatokon, eltolást, forgatást és tükrözést.

Mivel a sablon könyvtár nem teszi a polimorfizmust lehetővé, mind három féle alakzatot, külön-külön vektorban tároljuk a CGTLDrawDoc osztály tagjai ként (ld. CGTLDrawDoc osztály a 6. oldalon). Az MFC és a Visual Studio elég sok funkciót már a projekt létrehozásánál rendelkezésünkre bocsát, mint például a nyomtatás és ezzel kapcsolatos parancsok. Ezeket nem is kell implementálnunk, esetleg minimális módosításokkal a kívánt eredményt kapjuk. Amit módosítani kellet az a OnNewDocument() (ld. 7. oldal) és a Serialize() (ld. 7. oldal). Az első függvény az új rajz létrehozásért felelős, itt értelemszerűen kitöröltük az összes alakzatot a rajzról. A második a rajzok mentése és betöltésekor hívódik meg.

A program viselkedését a CGTLDrawView osztály (ld. 10. oldal) határozza meg. Mivel az MFC szoros kapcsolatban van a Windows API-val, eme osztály metódusai többnyire bizonyos üzenetek kezelésekor hívódnak meg, azaz esemény kezelőként gondolhatunk rájuk, vagyis a megfelelő gomb vagy menüpont kiválasztásakor hívódnak meg. Alapvetően a program kilenc állapot közül, mindig pontosan az egyikben van (ez az

m_state változóban tároljuk). Három állapot felel meg annak az esetnek amikor a három alakzat egyikét akarjuk rajzolni (de még nem kezdtük el, ezeket a programban CIRCLE, POLI és a REGPOLY-val jelöljük). Három állapot annak felel meg amikor már elkezdtük rajzolni a kívánt alakzatot (ezeket CIRCLE_, POLI_ és REGPOLY_ állapotokkal jelöljük). Végül három állapot felel meg a három műveletnek (ezek a MOVE, ROTATE és MIRROR állapotok). A bal egérgomb lenyomásánál az OnLButtonDown() metódus hívódik meg, és az állapottól függően megfelelően cselekszik. Hasonlóan a bal felengedése és a jobb egérgomb lenyomása is le van kezelve a megfelelő metódusokkal, míg a parancsok kiválasztása a megfelelő állapot beálítását eredményezi. Az MFC keretrendszer lehetőséget add, hogy a menüpontok és az eszközsor gombjai bizonyos feltételek teljesülésekor, bekapcsolva, vagyis lenyomva maradjanak. Ezeket az OnUpdate kezdetű metódusok kezelik az aktuális állapot alapján. Ha transzformációt hajtunk végre, akkor a kattintásnál, a program megnézi, hogy mire kattintottunk és ezt a FindHit metódus segíti. Továbbá az OnDraw metódusban van a kirajzoló kód, ez a Draw sablonfüggvény segítségével rajzolgatja ki az alakzatokat.

CGTLDrawDoc osztály




```
public:
	// typedefs
	typedef double scalar;
	typedef gtl::vect<scalar> Vect;

	typedef gtl::circle<scalar>   Circle;
	typedef gtl::poly<scalar>     Poly;
	typedef gtl::reg_poly<scalar> RegPoly;

	typedef std::vector<Circle>   vCircle;
	typedef std::vector<Poly>     vPoly;
	typedef std::vector<RegPoly>  vRegPoly;
```

A `CGTLDrawDoc` osztály tartalmazza a rajz dokumentumok reprezentációját. Elsősorban rögzíti milyen típusú objektumokkal reprezentáljuk majd a rajzot. A GTL sablonokat nyújt az alakzatok reprezentálásához, melyeket tetszőleges számszerű, skaláris, típusokkal paraméterezhettük. Számunkra a `double` típus lesz a legalkalmasabb.

Majd utána definiáljuk `typedef`-vel (a skaláris típus alapján) a felparaméterezett alakzat típusokat, mint a `Circle`, `Poly`, `RegPoly`. Ezek segítségével még bevezetünk egy-egy szinonimát a megfelelő alakzat típusú objektumokat tartalmazó vektorok típusára, mint a `vCircle`, `vPoly` és a `vRegPoly`.

```
	// members
	vCircle     m_vCircle;
	vPoly       m_vPoly;
	vRegPoly    m_vRegPoly;
	void SetChanged(bool l=true){ changed = l; }
private:
	bool changed;
```

Az előbb bevezetett típusok segítségével definiáljuk a megfelelő alakzatokat tároló vektorokat az `m_vCircle`, `m_vPoly`, `m_vRegPoly`, amelyekben, sorban a köröket, a (szabálytalan) sokszögeket és a szabályos sokszögeket tároljuk.

Továbbá bevezetünk egy logikai `bool changed` változót, amely akkor igaz, ha a dokumentumot módosították. Mivel a `changed` változó nem publikus, ezért a `SetChanged(bool)` tagfüggvénnyel tudjuk azt módosítani.

Meg kell jegyezni, hogy itt a GTL alapvető korlátjába ütközünk. A GTL egy sablon (template) könyvtár, melyben a típusok paraméterezhetjük, de nincs öröklődés, azaz polimorfizmus. Ebből adódóan nincs közös (például egy shape nevű) bázisosztály, hogy minden alakzat, sokszögek, körök is egy alakzatokat tároló (például vShape nevű) vektorban tárolunk. Az eredeti elképzelés ennek a problémának a megoldása volt, de egy „jól" megírt rajzoló programban valahol elkerülhetetlen, ugyanis az, hogy melyik alakzatot fogjuk rajzolni, futás időben dől el, a program felhasználója dönti el, mit rajzol és ezt, sablonokkal nem oldható meg, mivel a sablonok, úgymond, fordítási időben működnek.

OnNewDocument()

```
BOOL CGTLDrawDoc::OnNewDocument()
{
	if (!CDocument::OnNewDocument())
		return FALSE;

	if( changed && (AfxMessageBox( "Discard changes?" )!=IDOK) )
		return FALSE;
	changed = false;
	m_vCircle.clear();
	m_vPoly.clear();
	m_vRegPoly.clear();

	return TRUE;
}
```

A New parancs végrehajtásakor, a `changed` változót hamisra állítjuk és az alakzatokat tároló vektorokat kiürítjük, feltéve, ha a felhasználó elfogadja, hogy a rajz módosításai elvesznek.

Serialize()

```
void CGTLDrawDoc::Serialize(CArchive& ar)
{
	if (ar.IsStoring())
	{
		// TODO: add storing code here
		ar << m_vCircle.size();
		for( size_t i=0; i<m_vCircle.size(); i++)
		{
			ar << m_vCircle[i].r;
			ar << m_vCircle[i].o.x;
			ar << m_vCircle[i].o.y;
		}


		ar << m_vPoly.size();
		for( size_t i=0; i<m_vPoly.size(); i++ )
		{
			Poly& poly = m_vPoly[i];
			size_t size = poly.size();
```

```
			ar << size;

			Poly::input_iterator iter = poly.get_input_iter();
			Poly::input_iterator save = iter;
			do {
				Vect &v = *iter++;
				ar << v.x;
				ar << v.y;
			} while( save != iter );

		}

		ar << m_vRegPoly.size();
		for( size_t i=0; i<m_vRegPoly.size(); i++ )
		{
			ar << m_vRegPoly[i].n;
			ar << m_vRegPoly[i].o.x;
			ar << m_vRegPoly[i].o.y;
			ar << m_vRegPoly[i].p.x;
			ar << m_vRegPoly[i].p.y;
		}
	}
	else
	{
		// TODO: add loading code here
		OnNewDocument();
		size_t size;
		scalar r, x, y;
		ar >> size; m_vCircle.reserve(size);
		for( size_t i=0; i<size; i++)
		{
			ar >> r;
			ar >> x;
			ar >> y;
			m_vCircle.push_back( Circle( Vect(x,y),r) );
		}

		ar >> size;
		m_vPoly.reserve(size);
		for( size_t i=0; i<size; i++ )
		{
			Poly poly;
			size_t size;
			ar >> size;
			for( size_t j=0; j<size; j++ )
			{
				scalar x,y;
				ar >> x; ar >> y;
				poly.add( Vect(x,y) );
			}
			m_vPoly.push_back( poly );
		}

		ar >> size;
		m_vRegPoly.reserve(size);
		for( size_t i=0; i<size; i++ )
		{
			int n;
			scalar ox, oy, px, py;
			ar >> n;
```

```
                    ar >> ox;
                    ar >> oy;
                    ar >> px;
                    ar >> py;
                    m_vRegPoly.push_back( RegPoly( Vect(ox,oy),
                          Vect(px,py), n ) );
              }
        }
        changed  = false;
}
```

A `Serialize()` függvény hívódik meg a Save és a Load parancsok meghívásánál, azaz a rajzok mentésénél és betöltésénél. Ha `ar.IsStoring()` igaz, akkor mentésről van szó, vagyis az if első ága fut le, míg ha hamis, akkor betöltésről van szó és a másik ág fut, le majd a `changed` hamisra állítjuk. Mentéskor sorba elmentjük hány eleme van az alakzatokat tároló vektornak (sorba m_vCircle, m_vPoly, m_vRegPoly), majd sorba a az alakzatokat leíró adatokat tárolja el a megfelelő módon. Betöltésnél először kiürítjük a dokumentumot (azaz a vektorokat) majd a ahogy a mentésnél is, csak most fordítva, kiolvassuk a megfelelő adatokat.

CGTLDrawView osztály

CGTLDrawView
+classCGTLDrawView : CRuntimeClass -m_state : state -m_N : int -m_Bitmap : Bitmap * -m_VectBuf : Vect -m_index : size_t -m_selected : state
+GetThisClass() : CRuntimeClass * #CGTLDrawView() +GetDocument() : CGTLDrawDoc* +OnDraw(in pDC : CDC*) +PreCreateWindow(inout cs : CREATESTRUCT) : BOOL #OnPreparePrinting(in pInfo : CPrintInfo*) : BOOL #OnBeginPrinting(in Parameter1 : CDC*, in Parameter2 : CPrintInfo*) #OnEndPrinting(in Parameter1 : CDC*, in Parameter2 : CPrintInfo*) +~CGTLDrawView() +AssertValid() +Dump(inout dc : CDumpContext) +OnLButtonDown(in nFlags : UINT, in point : CPoint) : void +OnLButtonUp(in nFlags : UINT, in point : CPoint) : void +OnRButtonDown(in nFlags : UINT, in point : CPoint) : void +OnDrawRegularpolygon() : void +OnUpdateDrawRegularpolygon(in pCmdUI : CCmdUI*) : void +OnDrawPolygon() : void +OnUpdateDrawPolygon(in pCmdUI : CCmdUI*) : void +OnDrawCircle() : void +OnUpdateDrawCircle(in pCmdUI : CCmdUI*) : void +OnTransformMirror() : void +OnUpdateTransformMirror(in pCmdUI : CCmdUI*) : void +OnTransformMove() : void +OnUpdateTransformMove(in pCmdUI : CCmdUI*) : void +OnTransformRotate() : void +OnUpdateTransformRotate(in pCmdUI : CCmdUI*) : void -ToPoint(inout v : const Vect) : Point -ToVect(inout p : const CPoint) : Vect -VectDist(inout v1 : const Vect, inout v2 : const Vect) : scalar +Draw(inout t : Circle, inout g : Graphics, inout p : Pen) +FindHit(inout v : vCircle, in s : const state, inout point : CPoint)

A program működése, többnyire a CGTLDrawView osztály implementációjával van meghatározva.

```
private:
	// states
	enum state{ POLY, POLY_, REGPOLY, REGPOLY_, CIRCLE, CIRCLE_,
MOVE, ROTATE, MIRROR } m_state;

	int m_N; // number of sides of reg_poly
	Bitmap *m_Bitmap; // screen buffer
	CGTLDrawDoc::Vect m_VectBuf;  // a vector buffer,
	                              // for storing the 1st click

	size_t m_index;   // index of the selected shape
	state m_selected; // the kind of selected shape
	                  //(CIRCLE,POLY or REGPOLY; MOVE if none)
```

Egy state nevű felsoroló típusú `m_state` változó tárolja a program állapotát. A `CIRCLE`, `POLY`, `REGPOLY`, `MOVE`, `ROTATE`, `MIRROR` jelöli, hogy melyik alakzatot rajzoló parancsot vagy melyik transzformációt választottuk ki. A `CIRCLE_`, `POLY_`, `REGPOLY_` állapotba akkor kerül a program, amikor már megkezdtük a megfelelő alakzat rajzolását, de még nem fejeztük be (ld. GTL Draw Felhasználói Dokumentáció).

Az `m_N` változóba mentjük el, szabályos sokszög rajzolása esetén, hogy hány oldalú sokszöget szeretnénk majd rajzolni. Az `m_Bitmap` egy puffer képre mutató pointer, amelyiket kirajzolunk, ez segíti a villogás mentes (flicker free) kirajzolást.

Az `m_VectBuf`-ban tároljuk az a pontot, ahova először kattintottunk, (a kör vagy szabályos sokszög középpontját, ld. GTL Draw, Felhasználói Dokumentáció).

Az `m_index` és az `m_selected` változók határozzák meg, hogy módosító műveleteknél (move, rotate, mirror) melyik alakzaton hajtjuk végre a műveletet. Az `m_selected` határozza meg, hogy kör, sokszög vagy szabályos sokszögről van szó, és ennek megfelelően az `m_vCircle`, `m_vPoly` vagy `m_vRegPoly` vektorban található-e, míg az `m_index` mondja meg hányadik. Ha `m_selected` `MOVE`-ra van állítva, akkor egy objektum sincs kiválasztva, vagyis akkor nem hajtjuk végre a műveletet.

Segéd függvények

```
private:
	// helper functions
	// point <-> Vect conversion
	Point ToPoint( const CGTLDrawDoc::Vect& v )
	{ return Point(v.x,v.y); }
	CGTLDrawDoc::Vect ToVect( const CPoint& p )
	{ return CGTLDrawDoc::Vect( p.x, p.y ); }

	// vect vect distance calculation
	CGTLDrawDoc::scalar VectDist( const CGTLDrawDoc::Vect& v1,
		const CGTLDrawDoc::Vect& v2 )
	{ return (sqrt( double((v2.x-v1.x)*(v2.x-v1.x) +
		(v2.y-v1.y)*(v2.y-v1.y) ) ) ); }
```

Két segéd függvény könnyíti meg a `gtl::vect<T>` és a `CPoint` közötti konverziót, míg a harmadik két `gtl::vect<T>` távolságát adják meg.

Sablonfüggvények

```
public:
	// Draw shape
	template<typename T> void Draw( T& t, Graphics& g, Pen& p )
	{
		if( t.size() < 2 ) return;

		T::input_iterator iter, save;
		save = iter = t.get_input_iter();

		Point p1,p2;
		p1 = p2 = ToPoint( *iter );
```

```
            do
            {
                  iter++;
                  p2 = ToPoint( *iter );
                  g.DrawLine(&p, p1, p2);
                  p1 = p2;
            } while(iter!=save);
      }
```

A `Draw()` függvény végig iterál a `t` alakzat csúcsain és kirajzolja azt. Ez a `Draw()` függvény egyértelműen nem lehet a GTL implementáció része, ugyanis az alakzatok kirajzolása teljes mértékben a környezettől, vagyis a keretrendszertől függ [9].

```
      // Draw circle (specialization)
      template<> void Draw<CGTLDrawDoc::Circle>( CGTLDrawDoc::Circle&
t,
            Graphics& g, Pen& p )
      { g.DrawEllipse( &p, Rect(t.o.x-t.r,t.o.y-t.r,2*t.r,2*t.r) ); }
```

Elvileg a `Draw()` függvény alkalmas lenne az összes alakzat kirajzolására, még a kör kirajzolására is [11]. A kör `get_input_iter(int n=3)` függvény n paraméterével adhatjuk meg a kör felbontását és elég nagy n választásával, a kívánt eredményt kapnánk, de az valószínűleg nem lenne túl hatékony. A C++ viszont lehetővé teszi a sablonok specializációját, így a kört, az arra alkalmas függvénnyel rajzolhatjuk ki, a középpontja és a sugara alapján [4].

```
      // Point p on shape t
      template<typename T> bool Hit( T& t, CPoint &p )
      {
            BOOL hit = FALSE;

            Pen pen(Color(255,0,255),3);
            GraphicsPath path;
            T::input_iterator iter, save;
            save = iter = t.get_input_iter();

            Point p1,p2;
            p1 = p2 = ToPoint( *iter );

            do{
                  iter++;
                  p2 = ToPoint( *iter );
                  path.AddLine( p1, p2 );
                  hit = path.IsOutlineVisible(p.x, p.y, &pen);
                  p1 = p2;
            } while( !hit && iter!=save);

            return hit;
      }
```

A `Hit()` függvény akkor ad vissza igaz értéket, ha a `p` pont a `t` alakzaton van [12]. A GDI+ `GraphicsPath path` objektumhoz, a rajzolásnál látott módon, hozzáadjuk a `t` alakzat oldalait az `AddLine()` tagfüggvénnyel, majd megnéz, hogy a `p` pont benne van-e az így kapott, nem kirajzolt, alakzatban. Hogy megkönnyítsük az alakzat kiválasztását, a `path` objektumhoz, hármas vastagságú vonalakat adunk (ld. `Pen pen` inicializálása).

```
template<typename T> void FindHit( T& v, const state s,
      CPoint &point )
{
      for( size_t i=0; i<v.size(); i++ )
            if( Hit(v[i],point ) )
            {
                  m_index = i;
                  m_selected = s;
                  m_N = v[i].size();
                  return;
            }
}
```

A `FindHit()` függvény a `v` tömböt nézi át, hogy van-e benne alakzat, amely széle a `p` ponton halad át. Ezt az előző `Hit()` sablonfüggvény segítségével oldja meg. Viszont mivel nem tudjuk, hogy milyen típusú vektorról van szó (`vCircle`, `vPoly` vagy `vRegPoly`), ezért az `s` paraméterben adjuk át és az `m_selected` tagváltozóban tároljuk ezt az információt. Továbbá az is el kell tárolni, hogy hány oldalú alakzatról van szó, konkrétan a szabályos sokszögnél (ez a kirajzolásnál lesz fontos).

```
template<> void FindHit( CGTLDrawDoc::vCircle &v,
      const state s, CPoint &point )
{
      for( size_t i=0; i<v.size(); i++)
      {
            if(abs(VectDist(v[i].o, ToVect(point))-v[i].r )< 3)
            {
                  m_index = i;
                  m_selected = CIRCLE;
                  m_N = 0;
                  return;
            }
      }
}
```

Mivel a `Circle`-nek nincs `size()` tagfüggvénye, ezért nem tudjuk a Hit() sablonfüggvényt specializálni, hanem a `FindHit()` szintjén történik a specializáció.

```
template<typename T> void Move( T* pc,
      const CGTLDrawDoc::Vect& v )
{ if(pc) move(pc->get_input_iter(),pc->get_output_iter(),v ); }
```

Mivel, újra a plimorfizmus hiányába ütközünk, azt a problémát, hogy megfelelő típusú objektumon hajtjuk-e végre a műveletet, mutatókkal oldhatjuk meg. A `pc`, egy `T` típusú alakzatra mutat, ha `T` típusú alakzaton kell végrehajtani a műveletet, különben nullpointer. Tehát az eltolást is csak akkor hajtja végre, ha, úgymond, a mutató mutat valamire.

```
template<typename T> void Rotate( T* pc,
      const CGTLDrawDoc::Vect& v1, const CGTLDrawDoc::Vect& v2)
{
      if(pc){
            CGTLDrawDoc::Vect wp =
                  weight_point( pc->get_input_iter() );
            CGTLDrawDoc::scalar dx1, dx2, dy1, dy2;
            double rad;
```

```
                dx1=wp.x-v1.x; dy1=wp.y-v1.y;
                dx2=wp.x-v2.x; dy2=wp.y-v2.y;
                rad=atan2(dy1,dx1);
                rad=atan2(dy2,dx2)-rad;
                inplace_rotate(pc->get_input_iter(),
                      pc->get_output_iter(), rad );
        }
}
```

Hasonlóan, mint az eltolásnál, az elforgatásnál is, csak akkor végezi el a program a műveletet, ha a `pc` mutató nem nulla. Az elforgatást, az objektum wp súlypontja körül történik. Az elforgatás szögét pedig a `rad` változóban mentjük el, és ez a `v1` pont és `wp` súlyponton és a `v2` pont és `wp` súlyponton áthaladó egyenesek által közbezárt szög.

```
template<typename T> void Mirror( T* pc,
      const CGTLDrawDoc::Vect& v )
{ if(pc) mirror(pc->get_input_iter(),pc->get_output_iter(), v); }
```

A tükrözés is az eltoláshoz hasonló módon történik.

OnDraw()

```
if( pDoc->m_vRegPoly.size() +
      pDoc->m_vCircle.size() +
      pDoc->m_vPoly.size() > 0 )
{
      CRect rect;
      GetClientRect( rect );
      Bitmap bmp( rect.Width(), rect.Height() );
      Graphics* graph = Graphics::FromImage(&bmp);

      Pen pen(Color(128,0,255));

      for( size_t i=0; i < pDoc->m_vCircle.size(); i++ )
            Draw( pDoc->m_vCircle[i], *graph, pen );
      // ...
```

Csak akkor rajzolunk, ha van mit rajzolni. Először a `bmp` bittérképre rajzolunk és majd ezt a bittérképet fogjuk kirajzolni a képernyőre. A `bmp` objektumra, sorba a `vCircle`, `vPoly` és `vRegPoly` vektorok elemein végigiterálva, egyenként rajzoljuk ki őket.

```
      Pen sp( Color( 255,0,0), 3);
      // sp is the pen for drawing the selected shape
      if( m_selected < MOVE )
      {
            try
            {
                  switch(m_selected)
                  {
                  case CIRCLE :
                        Draw(pDoc->m_vCircle.at(m_index),
                              *graph, sp );
                        break;
                  case POLY :
                        Draw(pDoc->m_vPoly.at(m_index),
                              *graph, sp );
```

```
                        break;
                    case REGPOLY :
                        Draw(pDoc->m_vRegPoly.at(m_index),
                            *graph, sp );
                        break;
                    }
                } catch(...){}
            }
```

Amennyiben ki van választva egy objektum, azaz `m_selected` nem egyenlő `MOVE`, akkor a kiválasztott alakzatot, más tollal (`Pen`-nel) rajzoljuk.

OnLButtonDown()

```
void CGTLDrawView::OnLButtonDown(UINT nFlags, CPoint point)
{

    CGTLDrawDoc* pDoc = GetDocument();
    CGTLDrawDoc::vCircle &vCircle = pDoc->m_vCircle;
    CGTLDrawDoc::vPoly &vPoly           = pDoc->m_vPoly;
    CGTLDrawDoc::vRegPoly &vRegPoly = pDoc->m_vRegPoly;
```

Az egyszerűség kedvéjért, a megfelelő referenciákkal könnyebben tudjuk elérni a `CGTLDrawDoc` osztály alakzatokat tartalmazó vektorait.

```
    if( vPoly.size() < 1 && POLY_==m_state ) m_state=POLY;
```

Elő fordulhat, hogy sokszög rajzolása közben, új dokumentumot nyitunk, és ekkor a program az utolsó sokszöget próbálná bővíteni. A fenti sor ezt akadályozza meg.

```
    switch(m_state)
    {
    case POLY:
        vPoly.push_back( CGTLDrawDoc::Poly());
        vPoly[vPoly.size()-1].add( ToVect(point) );
        m_state = POLY_;
        break;
    case POLY_:
        vPoly[vPoly.size()-1].add(ToVect(point));
        break;
```

Ha sokszöget rajzolunk, akkor először egy, új egy csúcsot tartalmazó sokszöggel bővítjük a `vPoly` vektort és `POLY_` állapotba helyezzük a programot. Amíg `POLY_` állapotban van a program, addig az `vPoly` utolsó elem csúcsait bővítjük az aktuális ponttal.

```
    case REGPOLY:
        m_VectBuf = ToVect(point);
        m_state = REGPOLY_;
        break;
    case REGPOLY_:
        vRegPoly.push_back(CGTLDrawDoc::RegPoly( m_VectBuf,
            ToVect(point), m_N));
        m_state = REGPOLY;
        m_VectBuf = CGTLDrawDoc::Vect();
        break;
```

A szabályos sokszögnél könnyebb dolgunk van, első kattintásra elmentjük az aktuális pontot az `m_VectBuf` tagváltozóba (és `m_state` változót `REGPOLY_`-ra változtatjuk), majd a második kattintásnál, a `vRegPoly` vektorba, egy új `RegPoly`-t teszünk, amelyet az `m_VectBuf` és a `ToVect(point)` vektorokkal konstruálunk.

```
	case CIRCLE:
		m_VectBuf = ToVect(point);
		m_state = CIRCLE_;
		break;
	case CIRCLE_:
		vCircle.push_back(CGTLDrawDoc::Circle( m_VectBuf,
			VectDist( m_VectBuf, ToVect(point) )));
		m_state = CIRCLE;
		m_VectBuf = CGTLDrawDoc::Vect();
		break;
	default:
		break;
	}
```

A szabályos sokszöghöz hasonlóan, a körnél is első kattintásra elmentjük a középpontot, majd a második kattintás és `m_VectBuf` alapján meghatározzuk a kör sugarát.

```
	if( m_state >= MOVE )
	{
		m_selected = MOVE;
		FindHit( vCircle, CIRCLE, point );
		FindHit( vPoly, POLY, point );
		FindHit( vRegPoly, REGPOLY, point );
		if( m_selected < MOVE ) // found a hit
			m_VectBuf = ToVect(point);
	}
```

Ha m_state nagyobb egyenlő MOVE, akkor nem új alakzatot kell létrehozni, hanem transzformációt kell majd végrehajtani. Az egérgomb lenyomásánál csak azt határozzuk meg, melyik alakzatra lett kattintva a FindHit sablonfüggvények segítségével.

```
	pDoc->SetChanged();

	Invalidate();

	CView::OnLButtonDown(nFlags, point);
}
```

A `SetChanged()` metódussal jelezzük, hogy a rajzot módosítottuk és az `Invalidate()` függvény hívására az egész ablak újrarajzolódik, majd a bázisosztály `OnLButtonDown()` függvénye hívódik meg.

OnLButtonUp()

```
void CGTLDrawView::OnLButtonUp(UINT nFlags, CPoint point)
{
	if( m_state < MOVE && m_selected < MOVE) return;
	// drag&drop only used with transformations
```

Csak transzformációknál használjuk ezt az eseményt és akkor is csak ha valamelyik alakzatra rákattintottunk.

```
CGTLDrawDoc::Vect vec = ToVect( point );
CGTLDrawDoc::Circle *pc =0;
CGTLDrawDoc::Poly *pp =0;
CGTLDrawDoc::RegPoly *pr =0;
```

Ezek a pointerek közül pontosan az egyik fog a kiválasztott objektumra mutatni. (ld. `Move()`, `Rotate()` és `Mirror()` sablonfüggvények)

```
try{
	switch(m_selected)
	{
	case CIRCLE:
		pc = &(GetDocument()->m_vCircle.at(m_index));
		break;
	case POLY:
		pp = &(GetDocument()->m_vPoly.at(m_index));
		break;
	case REGPOLY:
		pr = &(GetDocument()->m_vRegPoly.at(m_index));
		break;
	}
}catch(...){}
```

Attól függően, milyen alakzat van kiválasztva, a megfelelő pointert beállítjuk, hogy a kiválasztott alakzatra mutasson. Mind ezt biztonságosan tesszük, a vektor `at()` tagfüggvényével és az esetleges kivételek elkapásával.

```
switch(m_state)
{
case MOVE:
	Move( pc, vec-m_VectBuf );
	Move( pp, vec-m_VectBuf );
	Move( pr, vec-m_VectBuf );
	break;
case ROTATE:
	Rotate( pp, vec, m_VectBuf );
	Rotate( pr, vec, m_VectBuf );
	// no need to rotate a circle!
	break;
case MIRROR:
	Mirror( pc, vec );
	Mirror( pp, vec );
	Mirror( pr, vec );
	break;
}
```

Attól függően melyik transzformációt szeretnénk végrehajtani, mindegyik pointerre meghívjuk a megfelelő függvényt és ez majd csak a nem nulla pointerre lesz végrehajtva.

```
m_selected = MOVE;
Invalidate();
```

```
	CView::OnLButtonUp(nFlags, point);
}
```

Miután megtörtént a transzformáció, frissítjük az ablakot.

OnRButtonDown()

```
void CGTLDrawView::OnRButtonDown(UINT nFlags, CPoint point)
{
	if(m_state==POLY_) m_state=POLY;
	CView::OnRButtonDown(nFlags, point);
}
```

A jobb egérgombot csak arra használjuk, hogy megszakítsuk a sokszög rajzolását.

A többi eseménykezelő

```
void CGTLDrawView::OnDrawRegularpolygon()
{
	m_state = REGPOLY;
	GetNDlg d;
	d.m_Nval = 5;
	if( d.DoModal() == IDOK )
		m_N = d.m_Nval;
	else return;
}
```

A `RegPoly` gombra kattintva, `m_state` beállítása után, egy megnyílik a csúcsok számát kérdező ablak. A többi parancs eseménykezelője sokkal egyszerűbb, ugyanis csak az `m_state` beállítását hajtja végre.

```
void CGTLDrawView::OnUpdateDrawRegularpolygon(CCmdUI *pCmdUI)
{
	pCmdUI->SetCheck( m_state == REGPOLY || m_state == REGPOLY_ );
}
```

Az `OnUpdate` esemény kezelők mindig a parancshoz kapcsolódó kezelő felület elemeit bekapcsoltra állítják, ha megfelelő állapotban van a program.

GDI+ könyvtár

Graphics Device Interface, azaz GDI, a kernel és az ablakkezelő mellet, egyike a Windows operációs rendszer három fő komponensének [8]. A GDI egy Microsoft Windows szabvány, grafikus objektumok ábrázolására és kimeneti egységekre való átvitelére, mint például monitorokra vagy nyomtatókra.

A Windows XP operációs rendszer bemutatkozásával, a GDI helyét átvette az C++ alapú utódja, a GDI+. A .NET környezetben, a `System.Drawing` névtéren keresztül lehet elérni, de ez nem működik az MFC keretrendszerben, így pár módosítást kell még végrehajtani.

Az stdafx.h fejállományba, ahova a többi `#include` direktíva után, a következőket kell bevinni:

```
#include <gdiplus.h>
#include <Gdiplusinit.h>
using namespace Gdiplus;
#pragma comment(lib, "gdiplus.lib")
```

Továbbá, a GTL Draw.h állományban, a `CGTLDrawApp` osztályt a következő adattagokkal kell bővíteni:

```
GdiplusStartupInput     m_gdiplusStartupInput;
ULONG_PTR               m_gdiplusToken;
```

Utána a GTL Draw.cpp állományban, a `CGTLDrawApp::InitInstance()` függvény a visszatérése elé a következő parancsot kell beszúrni:

```
GdiplusStartup(&m_gdiplusToken, &m_gdiplusStartupInput, NULL);
```

Végül, a `CGTLDrawApp::ExitInstance()` függvénybe kell a következő sort beírni:

```
GdiplusShutdown(m_gdiplusToken);
```

Így már készek is vagyunk és használhatjuk a GDI+ szolgáltatásait.

GTL könyvtár

A GTL, azaz Graphical Template Library, egy olyan kísérleti generikus C++ sablon könyvtár, ami a C++ Standard Template Library alapötletét próbálja átvinni a grafikus feladatok körébe. A GTL-t 1999-ben készítette el Kisteleki Róbert (ELTE Programtervező matematikus hallgató) és Dr. Porkoláb Zoltán.

A GTL-ben, a konténerek szerepét az alakzatok veszik át, az algoritmusok szerepét meg a transzformációs műveletek. Az alakzatok iterátorai az alakzat csúcsain lépked végig, ciklikusan.

Irodalomjegyzék

[1] Bjarne Stroustrup: A C++ programozási nyelv, Addison-Wesley,
ISBN 963-9301-18-3 és 963-9301-17-5 ö

[2] Bjarne Stroustrup's Home Page
http://www.research.att.com/~bs/homepage.html
Elérés dátuma: 2007-05-20

[3] Zoltán Porkoláb, Róbert Kisteleki: "Alternative Generic Libraries"
ICAI'99 4th International Conference on Applied Informatics,
Ed: Emőd Kovács et al., Eger-Noszvaj, 1999. pp. 79-86.
http://gsd.web.elte.hu/contents/articles/icai99.pdf

[4] Prokoláb Zoltán: Advanced C++ Lessons,
http://aszt.inf.elte.hu/~gsd/halado_cpp/

[5] MFC Library Reference, The Framework
http://msdn2.microsoft.com/en-us/library/k9kb0kba(VS.80).aspx
Elérés dátuma: 2007-05-20

[6] Microsoft Foundation Class Library
http://en.wikipedia.org/wiki/Microsoft_Foundation_Class_Library
Elérés dátuma: 2007-05-20

[7] GDI+
http://msdn2.microsoft.com/en-us/library/ms533798.aspx
Elérés dátuma: 2007-05-20

[8] Graphics Device Interface
http://en.wikipedia.org/wiki/Graphics_Device_Interface
Elérés dátuma: 2007-05-20

[9] Fekete István: Algoritmusok és adatszerkezetek I, II jegyzet
http://people.inf.elte.hu/fekete/

[10] Boost dokumentáció,
http://www.boost.org/libs/libraries.htm
Elérés dátuma: 2007-05-20

[11] Krzysztof Czarnecki, Ulrich W. Eisenecker, Robert Glück, David Vandevoorde,
Todd L. Veldhuizen, Generative Programming and Active Libraries, Springer-Verlag,
2000

[12] Husni Che Ngah, GDI+ Line/Curve Drawing and Hit Test
http://www.codeproject.com/vcpp/gdiplus/HitTester.asp
Elérés dátuma: 2007-05-20