TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta mechatroniky a mezioborových inženýrských studií

Projekt předmětu GDA

PIŠKVORKY

Liberec 2008

Viktor Bubla

Obsah

1 Zadání

Naprogramujte hru „PIŠKVORKY“

1. Hrací pole vytvořte dynamicky jako pole $M \times N$ (čtvercových, obdélníkových) komponent.
2. Po události kliknutí myší na dané komponentě, změňte tvar na kruh a barvu změňte podle barvy hráče na tahu.
3. Hlídejte, aby bylo možné pole nastavit pouze jednou (zamezte „švindlování“).
4. Kontrolujete, zda vznikl piškvorek (vodorovně, svisle a diagonálně). V případě, že ano, vypište o tom zprávu.
5. Nová hra — nastavte opět počáteční stav.
6. Pro zadávání konstant používejte pojmenované proměnné.

2 Řešení

Po rozvaze, jak řešit úlohu, jsem zvolil maximálně dynamické řešení, které mě napadlo. Hrací pole sestává z plochy zaplněné objekty, jejichž popisy jsou uloženy v poli pomocí indexů v horizontálním a vertikálním směru a integer property pojmenované *Typ*, která nabývá hodnot 0, 1 či 2. 0 odpovídá „neposkvrněnému“ čtverečku hrací plochy, 1 znamená, že konkrétní pole zaškrtl první hráč a políčko typu 2 náleží jeho soupeři. Při každé změně (klik hráče či změna velikosti okna...) je spouštěna procedura *Paint* formu, ve které se ve dvojitém for cyklu prochází celé pole a na pozice vypočítávané jako float hodnoty z velikosti herní plochy a počtu políček v obou směrech jsou vždy vykreslovány obdélníky, v případě typu 1 nebo 2 je navíc zakreslena elipsa či křížek složený ze dvou úseček. Aby překreslování nebylo rušivé, je povolen *DoubleBuffering*. Vzhledem k tomu, že překreslování spouští klikem hráči, není potřeba proces překreslování nikterak optimalizovat. Na svém stroji jsem zaznamenal nepříjemné překreslování při velikosti herního pole vetším, než cca 200×200 políček, kdy procházené pole obsahuje 40000 položek. Na(ne)štěstí jsou políčka při rozlišení 1680×1050 a fullscreen piškvorkách téměř neviditelná, takže se tímto není třeba zdržovat.

Velikost pole je dána dvěma *#define* hodnotami. Třetí *#define* hodnotou je délka spojité linie políček, při níž hráč vyhrává. Je tedy možné hrát nejen na klasických „5 v řadě“, ale třeba jen na 4 či 3. Z velikosti pole a rozměrů *ClientRectangle* je vypočítávána (a aktualizována při změně velikosti okna) float hodnota velikosti jednoho dílku, která je později využívána při veškerém vykreslování a detekci pozice kliku.

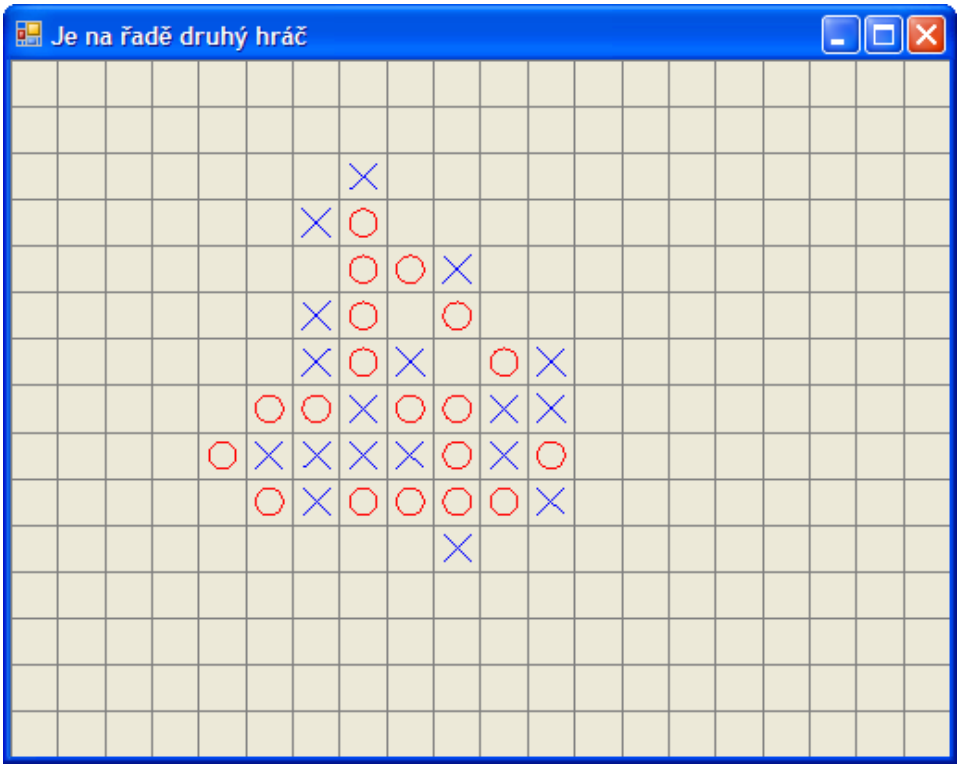
Po spuštění hry je v záhlaví formu informace o hráči, který je na řadě. Hrající hráč je držen v integer proměnné, která sice nabývá jen dvou hodnot, ale pro zjednodušující využití dále v programu není použita proměnná bool. Při kliku do prostoru herního pole je vyvolána událost *OnClick* formu a v ní je, bohužel trochu krkolomně, vypočten horizontální a vertikální index odpovídající políčku ležícímu pod kurzorem myši. Procedura, která obdrží získané indexy, provede kontrolu, zda již políčko nebylo zaškrtnuto a případně do pole na dané indexy uloží proměnnou aktivního hráče (zde výše zmíněné zjednodušení).

Následuje zavolání další procedury, jež má na starost zjistit, zda posledním tahem hráč nezvítězil. Opět jsou předány indexy pole. Od místa kliku se na všechny strany postupně kontroluje *DelkaPiskvorku* políček. Tato procedura je spolu se správným vykreslováním stěžejní částí programu. Je potřeba zajistit několik maličkostí. Kromě správné algoritmizace zjištění spojité řady políček jednoho hráče je nutné omezit kontrolu pole na indexech mimo herní plochu. Zde se program docela zásadně znepřehledňuje, protože bylo třeba dodržet dynamičnost v délce výherní piškvorky. Využil jsem hromadu podmínek a cyklů for, v nichž je nejdříve (a zvlášť pro každý diagonální směr) zjištěno, od kterého okraje je políčko méně vzdálené a následně je vypočítána, pokud je potřeba, korekční hodnota, která je odečtena od koncové hodnoty v cyklu for a tím je zamezeno testování neexistujících (tedy spíše cizích) položek mimo pole. Ve zkratce algoritmus počítání délky spojité řady políček jednoho

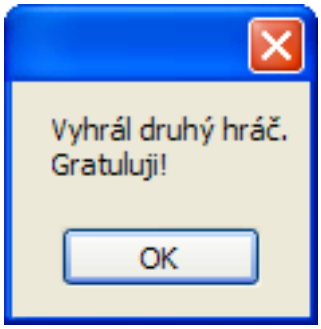
hráče funguje tak, že je inicializována proměnná *DelkaRady* držící počet souvislých políček na 1 a následně je v cyklu testováno maximálně *DelkaPiskvorku* políček jedním směrem, dokud políčka náleží aktuálnímu hráči nebo dokud nenarazíme díky hodnotě korekce odečtené od konstanty na okraj pole. Při každém otestovaném platném políčku je inkrementována *DelkaRady*. Následuje test v opačném směru.

Pokud je nyní *DelkaRady* větší, než *DelkaPiskvorku*, spouští se procedura *Vyhra*, případně pokračuje kontrola v dalších směrech. Hra je zastavena vyskočením *MessageBoxu* s gratulací a informací, kdo vyhrál. Reset hry je řešen velice jednoduše a pohotově. Při kliku na tlačítko *OK* vyskočeného blahopřání je rovnou spuštěna procedura *NovaHra*, která jednoduše projde pole a naplní ho položkami s *Typem* 0, překreslí herní plochu a nastaví prvního hráče.

Na obrázcích (1) a (2) je ukázka printscreenu aplikace při jednom konkrétním rozumném nastavení a ukázka *MessageBoxu*.



Obrázek 1: Ukázka běžící aplikace.



Obrázek 2: MessageBox zobrazivší se při výhře jednoho z hráčů.

3 Zdrojový kód

V této kapitole uvádím zajímavé či důležité části programu. Ne vždy se jedná přímo o algoritmy, avšak pro dodržení jednotného vzhledu... Kompletní program zašlu v případě potřeby například mailem. *bublaviktor@seznam.cz*

Algorithm 1 Define na začátku programu.

```
#define VelikostX 40
#define VelikostY 30
#define DelkaPiskvorek 5
```

Algorithm 2 Třída políčka.

```
ref class Policko
{
public:
property int X;
property int Y;
property int Typ;
Policko(int X, int Y, int Typ)
{
this->X = X;
this->Y = Y;
this->Typ = Typ;
}
};
```

Algorithm 3 Inicializace a výpočet velikosti dílků.

```
Form1::DoubleBuffered = true;
DilX = (float) ClientRectangle.Width / VelikostX;
DilY = (float) ClientRectangle.Height / VelikostY;
Policka = gcnew array<Policko^,2>(VelikostX,VelikostY);
Hrac = 1;
```

Algorithm 4 Inicializace pole při spuštění programu či nové hře.

```
for (int a = 0; a < VelikostX; a++)
{
for (int b = 0; b < VelikostY; b++)
{
Policka[a,b] = gcnew Policko(a,b,0);
}
}
```

Algorithm 5 Procedura při kliknutí do *ClientRectangle*.

```
void Kliknuto(int KlikX, int KlikY)
{
if (Policka[KlikX,KlikY]->Typ == 1 || Policka[KlikX,KlikY]->Typ == 2)
return;
Policka[KlikX,KlikY]->Typ = Hrac;
Form1::Invalidate();
ProzkoumatStav(KlikX,KlikY);
if (Hrac == 1) Hrac = 2;
else Hrac = 1;
return;
};
```

Algorithm 6 Část procedury testu okolních políček.

```
void ProzkoumatStav(int KlikX, int KlikY)
{
int i;
int DelkaRady;
int Okraj;
//vodorovne
DelkaRady = 1;
if (VelikostX - KlikX < DelkaPiskvorek) Okraj = DelkaPiskvorek - (VelikostX - KlikX);
else Okraj = 0;
for (i = 1; i < DelkaPiskvorek - Okraj; i++)
{
if (Hrac == Policka[KlikX+i,KlikY]->Typ) DelkaRady++;
else break;
}
if (KlikX < DelkaPiskvorek - 1) Okraj = DelkaPiskvorek - KlikX;
else Okraj = 0;
for (i = 1; i < DelkaPiskvorek - Okraj; i++)
{
if (Hrac == Policka[KlikX-i,KlikY]->Typ) DelkaRady++;
else break;
}
if (DelkaRady >= DelkaPiskvorek)
{
Vitez(Hrac);
return;
}
... Zde svislé a diagonální testy
```

Algorithm 7 Procedura provedená při „pozitivním“ průchodu předchozím algoritmem.

```
void Vitez(int Hrac)
{
if (Hrac == 1)
{
MessageBox::Show("Vyhrál první hráč.\nGratuluji!");
NovaHra();
}
else
{
MessageBox::Show("Vyhrál druhý hráč.\nGratuluji!");
NovaHra();
}
return;
};
```

Algorithm 8 Procedura pro novou hru.

```
void NovaHra(void)
{
for (int a = 0; a < VelikostX; a++)
{
for (int b = 0; b < VelikostY; b++)
{
Policka[a,b] = gcnew Policko(a,b,0);
}
}
Hrac = 2;
Form1::Invalidate();
return;
};
```

Algorithm 9 Poněkud málo přehledná, avšak velice mocná procedura vykreslování.

```
private: System::Void Form1_Paint(System::Objectˆ sender, System::Windows::Forms::PaintEventArgsˆ e) {
if (Hrac == 1) Form1::Text = "Je na řadě první hráč";
else if (Hrac == 2) Form1::Text = "Je na řadě druhý hráč";
for (int a = 0; a < VelikostX; a++)
{
for (int b = 0; b < VelikostY; b++)
{
switch (Policka[a,b]->Typ)
{
case 0:
e->Graphics->DrawRectangle(System::Drawing::Pens::Gray,a*DilX,b*DilY,DilX,DilY);
break;
case 1:
e->Graphics->DrawRectangle(System::Drawing::Pens::Gray,a*DilX,b*DilY,DilX,DilY);
e->Graphics->DrawEllipse(System::Drawing::Pens::Red,
(float)((a*DilX)+(0.2*DilX)),(float)((b*DilY)+(0.2*DilY)),
(float)(DilX/1.7),(float)(DilY/1.7));
break;
case 2:
e->Graphics->DrawRectangle(System::Drawing::Pens::Gray, a*DilX,b*DilY,DilX,DilY);
e->Graphics->DrawLine(System::Drawing::Pens::Blue,
(float)((a*DilX)+(0.2*DilX)),(float)((b*DilY)+(0.2*DilY)),
(a*DilX)+(0.2*DilX)+(float)(DilX/1.7),(b*DilY)+(0.2*DilY)+(float)(DilY/1.7));
e->Graphics->DrawLine(System::Drawing::Pens::Blue,
(float)((a*DilX)+(0.2*DilX))+(float)(DilX/1.7),(float)((b*DilY)+(0.2*DilY)),
(float)((a*DilX)+(0.2*DilX)),(b*DilY)+(0.2*DilY)+(float)(DilY/1.7));
break;
}
}
}
}
```

Algorithm 10 Ošetření události změny velikosti okna.

```
private: System::Void Form1_Resize(System::Object^ sender, System::EventArgs^ e) {
DilX = (float) ClientRectangle.Width / VelikostX;
DilY = (float) ClientRectangle.Height / VelikostY;
Form1::Invalidate();
}
```

Algorithm 11 Událost kliku do formu a její řešení.

```
private: System::Void Form1_Click(System::Object^ sender, System::EventArgs^ e) {
int KlikX = (Cursor->Position.X - Form1::Left - 4) / DilX;
int KlikY = (Cursor->Position.Y - Form1::Top - 30) / DilY;
if (KlikX < VelikostX && KlikY < VelikostY)
Kliknuto(KlikX,KlikY);
}
```

4 Závěr

Krom jedné maličkosti jsem splnil všechno, co bylo požadováno v zadání úlohy. Tou maličkostí je spíše slovíčkaření — totiž že místo cituji: „Po události kliknutí myší na dané komponentě, změňte tvar na kruh a barvu změňte podle barvy hráče na tahu.“ jsem použil tvar kroužku a křížku, jež zaprvé považuji za hezčí řešení a pro piškvorky typičtější, a zadruhé podle toho, o čem se zmiňoval vyučující, jsem vydedukoval, že úloha je vytvořena původně pro Pascal (prostředí Delphi) a tedy ve Visual Studiu mohu improvizovat. Druhou věcí, kterou bych rád v závěru zmínil, je fakt, že již nyní při psaní zprávy jsem narazil na několik míst programu, která by šla řešit jednodušeji, elegantněji, úsporněji či rychleji a něco je dokonce i zbytečné a navíc, avšak program ve své podobě je plně funkční a optimalizace v tomto případě zbytečná. Koneckonců platí programátorské přísloví, že každý program obsahuje alespoň jednu chybu a že každý program se dá zjednodušit. Tedy i tento program by se dal postupně zjednodušit na jeden chybný příkaz :-).