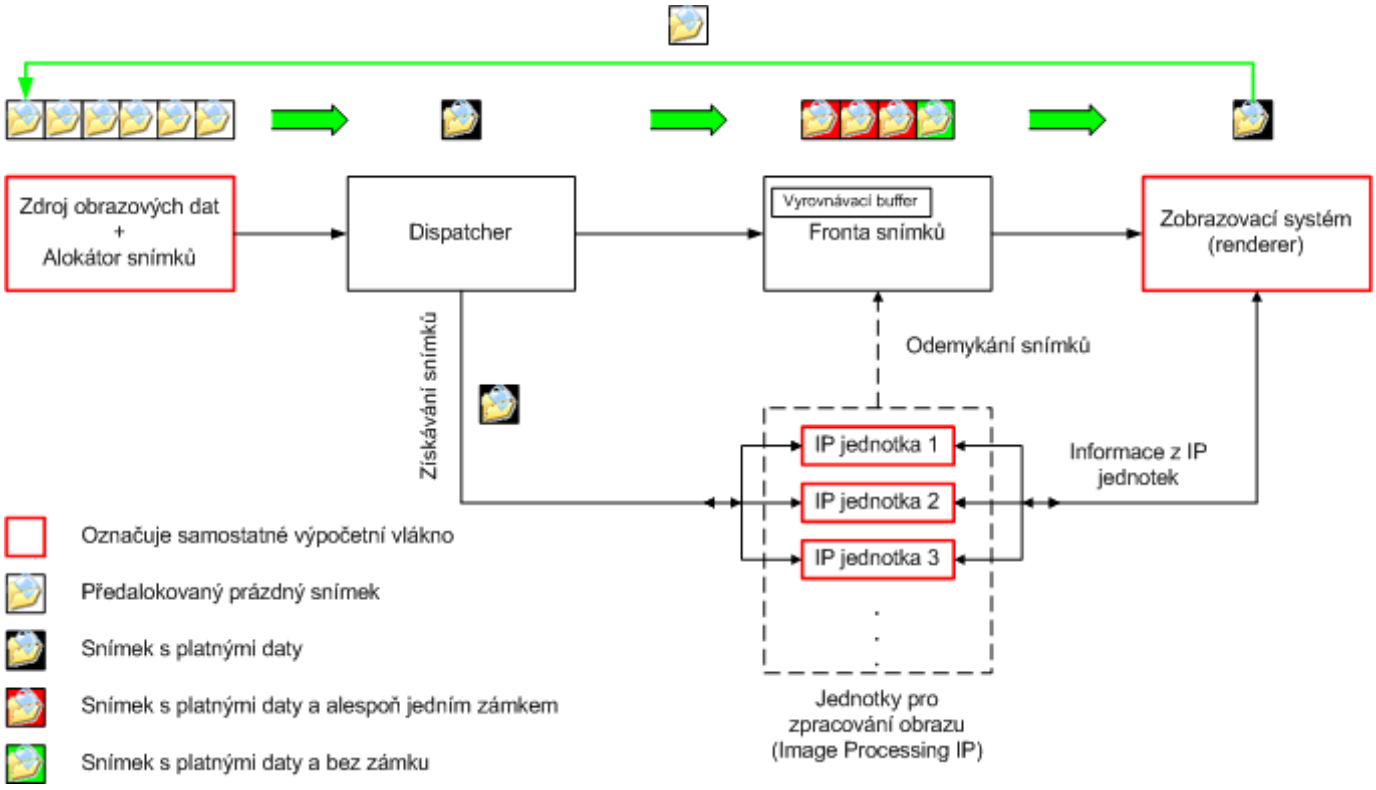
IPP – Image Processing Pipeline

Tomáš Mrkvička

Základní popis

Projekt obsahuje implementaci řetězce pro zpracování obrazu v reálném čase. Základní myšlenkou celého řetězce je implementace jednotlivých částí nezávisle na sobě, takže každá část může pracovat autonomně ve svém výpočetním vlákně (vláknech). Základní schéma řetězce je na následujícím obrázku:



Řetězec se skládá z několika povinných částí a výpočetních jednotek libovolného počtu, které se k řetězci připojují jako autonomní objekty.

Popis povinných částí řetězce

Zdroj obrazových dat a alokátor snímků

Související třídy:

- ***`TCameraAbstract`*** (CameraAbstract.h)
- ***`TCameraLoader`*** (CameraLoader.h)

- ***TCameraThread*** (CameraThread.h)

V této části se v samostatném vlákně získávají vstupní snímky, plní se obrazovými daty ze zdroje obrazu a posílají se dále do řetězce.

Důležitým rysem této části je efektivní alokace snímků. Jak je patrné podle schématu řetězce, snímky se po průchodu řetězcem automaticky vrací zpět do místa vytvoření a jsou znovu k dispozici pro aplikaci. Aplikace při spuštění musí určit maximální množství vytvořených snímků, pokud je tento počet překročen, tj. pokud aplikace aktuálně využívá všechny snímky, pak nejsou další snímky posílány do řetězce do doby než se dříve vyslané snímky znovu nevrátí zpět do alokátoru snímků.

Zdroj obrazových dat v této části je reprezentován abstraktní třídou ***TAbstractCamera***, která podle implementace vybírá fyzický zdroj obrazových dat. Kamera vždy vrací obrázek v datovém formátu 24-bitů RGB (uložení v paměti je B G R …) bez zarovnávacích bajtů mezi řádky (pixelová data jsou tedy uložena spojitě). Pro snazší změnu používané kamery je pro implementaci každé jednotlivé kamery určena DLL knihovna, která poskytuje požadovanou implementaci. Každá taková knihovna exportuje metodu ***CreateCamera***, která vrací implementaci zadané kamery. Aktuálně jsou k dispozici následující implementace kamery, každá se specifikovanou funkcí ***CreateCamera***:

- **video soubor** – obrázky jsou získávány ze zadaného videosouboru. Podporované videoformáty jsou závislé na kodecích dostupných v systému.

  TCameraAbstract * CreateCamera(const char * videofile);

- **lokální kamera** – obrázky jsou získávány z kamery fyzicky připojené k aktuálnímu PC např. přes USB nebo Firewire rozhraní. Dostupnost kamery je závislá na systému DirectShow, přes který je ke kameře přistupováno.

  TCameraAbstract * CreateCamera(void);

- **soubor s obrázkem** – v některých situacích lze využít kameru, která vždy vrací snímek shodný se zadaným obrázkem.

  TCameraAbstract * CreateCamera(const char * imagefile);

- **„null“ kamera** – tato kamera slouží pouze k testovacím účelům. Vždy vrací platný obrázek o velikosti 640x480 se všemi pixely nastavenými na černou barvu.

  TCameraAbstract * CreateCamera(void);

Pro snadné načítání kamery lze použít třídu ***TCameraLoader***. Po načtení kamery lze použít třídu ***TCameraThread***, která vytvoří celou tuto část pomocí načtené kamery. Tato třída dále zajišťuje načítání snímků. V pravidelných intervalech (zadaných uživatelem) načítá snímek z načtené kamery ve formátu RGB, následně tento snímek převede na formáty ARGB a monochromatický a z těchto tří snímků a časové značky vytvoří finální snímek, který je zaslán do další části řetězce – dispatcheru (viz dále). Do aplikace je tedy zasílán snímek, který obsahuje tři obrázky stejné velikosti ve formátech RGB, ARGB a odstínech šedi.

Snímek

Související třídy:

- ***TFrame*** (ImageAbstract.h)
- ***TImage*** (ImageAbstract.h)
- ***TImageSet*** (ImageAbstract.h)
- ***TTimeStamp*** (TimeStamp.h)

Snímek je datová struktura vytvářená zdrojem obrazových dat, naplněná obrazem ze třídy ***TCameraAbstract*** a putující celým řetězcem. Je reprezentován abstraktní třídou ***TFrame***. Snímky jsou vytvářeny třídou ***TCameraThread*** a postupně prochází celým řetězcem. Protože se řetězec skládá z více paralelně pracujících vláken, je každý snímek synchronizován pro vícenásobný přístup.

Snímek po vytvoření obsahuje časovou značku a tři varianty obrazových dat. Časová značka je reprezentována třídou ***TTimeStamp***, která obsahuje datum vytvoření snímku (v milisekundách od startu systému) a dále jedinečný 32-bitový identifikátor snímku, pomocí něhož lze lépe určovat identitu dvou snímků. Tento identifikátor je zvyšován o 1 s každým snímkem a tedy určuje i pořadí jednotlivých snímků.

Obrazová data jsou uložena ve snímku pomocí abstraktní třídy ***TImageSet***, která pomocí svých metod zpřístupňuje jednotlivé varianty snímku.

Kvůli nutnosti návratu snímků do alokátoru snímků byl pro snímky zaveden mechanismus počítání referencí známý například z COM. Každý snímek má tedy metodu ***AddRefs()*** a ***Release()***, které musí být řádně použité při práci s referencemi. Jakmile je počet referencí roven nule, je snímek automaticky vrácen do alokátoru snímků a je tak znovu k dispozici pro další obrazová data.

Pro snímky je k dispozici ještě jeden podobný mechanismus, který slouží k zamykání snímků. Jedná se mechanismus, který umožňuje zamknout snímek tak, aby jej nebylo možné zobrazit dokud

není řádně zpracován (více v sekci o výpočetních jednotkách). K dispozici jsou metody ***AddLock()*** a ***ReleaseLock()*** pro práci se zámky na snímku.

Dispatcher

Související třídy:

- ***TDispatcherInterface*** (ImageAbstract.h)
- ***TDispatcher*** (Dispatcher.h)

Dispatcher je objekt bez vlastního aktivního řízení (nemá vlastní vlákno), který se stará o distribuci snímků přicházejích ze zdroje obrazových dat k výpočetním jednotkách a zobrazovacímu systému. Dispatcher je jediným spojením mezi řetězcem a připojovanými jednotkami (více v sekci o výpočetních jednotkách).

Reprezentován je abstraktní třídou ***TDispatcherInterface***,která zároveň představuje komunikační rozhraní pro výpočetní jednotky.

V dispatcheru je vždy uložen poslední snímek zaslaný ze zdroje obrazových dat. Kvůli tomu musí zdroj obrazových dat získat po vytvoření rozhraní dispatcheru, kterému bude zasílat vytvářené snímky. Starý snímek uložený v dispatcheru je vždy nahrazen novým snímkem a poté poslán dále do řetězce (do fronty snímků, viz dále). Protože k dispatcheru může přistupovat více vláken, je synchronizován pro vícenásobný přístup.

Z dispatcheru mohou výpočetní jednotky získávat poslední vytvořený snímek pomocí metod ***GetFrame()*** a ***GetLockedFrame()*** podle toho, zda daná jednotka zamyká nebo nezamyká snímky. Důležitou vlastností řetězce je, že jednotka smí snímek získat pouze přes dispatcher, jakmile snímek opustí dispatcher pak už jej žádná jednotka získat nemůže (což ovšem neznamená, že jej nějaká jednotka ještě nevlastní!).

Fronta snímků

Související třídy:

- ***TFrameQueue*** (FrameQueue.h)

Tato třída zajišťuje přesun snímků z dispatcheru do zobrazovací části. Fronta zajišťuje plynulost zobrazování snímků svým vnitřním vyrovnávacím bufferem snímků. Jehož velikost lze libovolně měnit za běhu aplikace. Fronta také zajišťuje, že snímky se zámkem nebudou moci být zobrazeny do doby než jsou všechny zámky uvolněny.

Vnitřně je fronta rozdělena na tři logické části. Po vstupu do fronty nejprve snímek vstoupí do vyrovnávacího bufferu, čímž můžou být některé snímky z tohoto bufferu vytlačeny do druhé logické části a tou je fronta potencionálních výstupních snímků. Za ní následuje jednoprvková fronta pro aktuální výstupní snímek. To je snímek, který nemá žádné zámky a může tedy být zobrazen. Po odebrání tohoto snímku z fronty je na jeho místo dosazen další snímek čekající ve frontě potencionálních výstupních snímků.

Fronta není autonomní objekt (nemá řídící vlákno), její metody jsou však synchronizovány pro vícenásobný přístup. Zobrazitelný snímek lze získat pomocí metody ***GetRenderableFrame()***, nastavení velikosti vyrovnávacího bufferu lze změnit pomocí metody ***SetInputBufferLength(DWORD)***. Pokud je nová velikost vnitřního bufferu menší než původní velikost pak jsou přebytečné snímky přesunuty do druhé fronty v této třídě.

Fronta musí být po vytvoření předána dispatcheru, který musí vědět o frontě, do které má zasílat snímky. Pokud je dispatcheru nastaven místo fronty ukazatel NULL, pak je to pro dispatcher signál, že má snímky pouze uvolnit a že uživatel nepotřebuje jejich zobrazení. Takové chování se hodí např. při testování výpočetních jednotek.

Zobrazovací systém

Zobrazovací systém není součástí řetězce, na jeho místo musí uživatel dosadit libovolný mechanismus, který bude číst snímky z fronty ***TFrameQueue***, zpracovávat je a následně uvolňovat tak, aby mohly být znovu zařazeny do řetězce. Typicky běží zobrazovací systém v hlavním vláknu procesu, zatímco ostatní objekty a vlákna jsou tímto vláknem vytvářena.

Pro zobrazovač je důležité, aby stíhal zobrazovat snímky přicházející řetězcem. V ideálním případě by měl pracovat s trochu vyšší rychlostí než je rychlost generování snímků vláknem zdrojové kamery.

Popis výpočetních jednotek

Výpočetní jednotka je vždy autonomní objekt (má vlastní výpočetní vlákno/vlákna). Jednotka je vytvořena hlavním vláknem aplikace, je jí předáno rozhraní dispatcheru, který využívá pro získávání snímků a následně je spuštěno vlastní výpočetní vlákno jednotky.

Základní rozhraní jednotky

Související třídy:

- ***TUnitInterface*** (Unit.h)

Každá jednotka musí být odvozena od abstraktní třídy ***TUnitInterface***, která definuje rozhraní pro výpočetní jednotku následujícím způsobem:

```
class TUnitInterface
{
public:
      virtual EnumUnitType                GetType(void) = 0;
      virtual TUnitRetTypeInterface*      GetResult(DWORD id) = 0;
      virtual DWORD                       GetFrameInterval() = 0;
      virtual BOOL                        Start(void) = 0;
      virtual BOOL                        Stop(void) = 0;
      virtual void                        Release(void) = 0;
};
```

Význam metod je následující:

- ***GetType*** – vrací typ návratové hodnoty této jednotky (tedy typ výsledku, které jednotka počítá). U této metody se předpokládá, že vrací konstantní hodnotu a tedy nemusí být synchronizována pro vícenásobný přístup.
- ***GetResult*** – vrací výsledek vypočítaný jednotkou pro snímek se zadaným identifikátorem získaným z třídy ***TTimeStamp*** daného snímku. Jednotka může vrátit NULL pokud pro daný snímek nemá nebo pokud nevrací žádné výsledky. Pokud je výsledek vrácen, pak vždy musí být vrácena nová reference na výsledek a tato reference je dále spravována volajícím (více informací v sekci o návratových hodnotách). Tato metoda by měla být synchronizována pro vícenásobný přístup, neboť uvnitř jednotky dochází zároveň k zápisu výsledků a zároveň se výsledky získávají touto metodou.
- ***GetFrameInterval*** - tato metoda vrací přibližnou informaci o počtu snímků, které jednotka nestačila zpracovat během zpracovávání jednoho snímků. Typicky jednotka může počítat tuto hodnotu jako rozdíl časových identifikátorů dvou posledních zpracovaných snímků. Předpokládá se, že se tato hodnota bude měnit – tedy jednotka tuto hodnotu pořád aktualizuje. Její výpočet může být založen buď na prostém rozdílu identifikátorů, nebo např. na průměrných hodnotách apod. Typicky aplikace tuto hodnotu využívá pro výpočet délky vyrovnávacího bufferu ve frontě snímků. Aplikace typicky nastavuje délku bufferu jako maximum těchto hodnot

vrácených všemi jednotkami. Metoda by měla být synchronizována pro vícenásobný přístup.

- ***Start*** - tato metoda spustí výpočetní jednotku. Musí vrátit TRUE pokud již byla jednotka spuštěna nebo pokud se spuštění podařilo. Metoda vrací FALSE při kritické chybě. Metoda nemusí být nutně synchronizována pro vícenásobný přístup, protože je volána typicky z hlavního vlákna aplikace, ale je to vhodné.
- ***Stop*** - tato metoda zastaví spuštěnou jednotku. Při zastavení je důležité, aby jednotka uvolnila referenci na všechny snímky, které právě vlastní!!! Metoda vrací TRUE pokud se jednotku podařilo zastavit nebo pokud je již jednotka zastavená. Vrací FALSE při kritické chybě. Metoda nemusí být nutně synchronizována pro vícenásobný přístup, protože je volána typicky z hlavního vlákna aplikace, ale je to vhodné.
- ***Release*** – tato metoda odstraní jednotku. Nejde o uvolnění reference, ale kompletní odstranění jednotky!!! Před zavoláním této metody by měla být jednotka zastavena. Tato metoda by se v případě nutnosti měla o toto zastavení pokusit. Tato metoda nemusí být synchronizována pro vícenásobný přístup, protože v okamžiku zničení se předpokládá, že žádné další vlákno už s jednotkou nepracuje.

Napojení jednotek na řetězec, zamykání snímků

Související třídy:

- ***TDispatcherInterface*** (ImageAbstract.h)

Jediná možnost komunikace jednotky s výpočetním řetězcem je skrze rozhraní dispatcheru ***TDispatcherInterface***, přes který může jednotka získávat snímky vstupující do řetězce. Toto rozhraní musí být jednotce předáno při jejím vytváření. Rozhraní poskytuje jednotce při vytvoření další cenné informace jakými jsou např. velikost snímků procházejících řetězcem.

Jednotky se obecně dělí na dvě skupiny podle toho jakým způsobem pracují se získaným snímkem. Jednotka může získat snímek z dispatcheru buď jako snímek s referencí nebo jako snímek s referencí a zámkem. Pokud jednotka získá snímek i se zámkem (tj. zamyká snímky), pak to pro snímek znamená, že nemůže být zobrazen (tedy vystoupit z fronty ***TFrameQueue***) dokud není snímek jednotkou uvolněn. Takové chování je vhodné v případech, kdy zobrazení daného snímku vyžaduje hodnoty vypočítané danou jednotkou – tj. jednotka zámkem zajišťuje, že výsledky budou k dispozici v okamžiku, kdy bude snímek zobrazen.

Typy výpočetních jednotek a návratové hodnoty

Související třídy:

- ***TUnitInterface*** (Unit.h)
- ***TUnitRetTypeInterface*** (UnitTypes.h)

Protože jednotky obecně počítají velmi různorodá data a zároveň jsou typicky implementovány např. v DLL knihovnách, byl vytvořen speciální mechanismus pro získávání výsledků z jednotek. Výsledky jednotek jsou vždy vraceny metodou

```
virtual TUnitRetTypeInterface * TUnitInterface::GetResult(DWORD id) = 0;
```

která vrací výsledek pro snímek se zadaným identifikátorem. Každý typ výsledku musí být odvozen od abstraktní třídy ***TUnitRetTypeInterface***:

```
class TUnitRetTypeInterface
{
public:
      virtual EnumUnitType          GetType(void) const = 0;
      virtual void                  Release(void) = 0;
};
```

Význam metod je následující:

- ***GetType*** – vrací typ výsledku. Vrácená hodnota by měla být shodná s typem jednotky, která tento výsledek vytvořila.
- ***Release*** – uvolní výsledek zpět do jednotky. Tato metoda je důležitá vzhledem k tomu, že výsledek může být vytvořen např. v DLL knihovně a tedy na něj nelze aplikovat např. operátor delete.

Pro libovolný datový typ je nutné vytvořit odpovídající abstraktní třídu odvozenou od rozhraní ***TUnitRetTypeInterface*** a přidat nové metody pro získávání výsledků. Zároveň je nutné definovat nový datový typ do enumerátoru ***EnumUnitType*** (UnitTypes.h) tak, aby bylo možné nový návratový typ a typ jednotky identifikovat pomocí metod ***GetType()***. Pro odvozování nových tříd je vždy nutné použít pouze jednoduchou dědičnost (tj. žádná třída nesmí mít více bázových tříd)!!!

Nová odvozená abstraktní třída musí být v rámci souboru UnitTypes.h deklarována jako abstraktní, její implementaci vždy provádí výpočetní jednotka. Např. aktuálně je k dispozici abstraktní třída ***TUnitRetType_integer***, kterou lze použít pro jednotku, která vrací jako výsledek výpočtu číslo typu integer. Její deklarace vypadá takto:

```
class TUnitRetType_integer : public TUnitRetTypeInterface
{
public:
      virtual EnumUnitType    GetType(void) const { return ENUM_UNITTYPE_INTEGER;};
      virtual void            Release(void) = 0;

public:
      virtual int             GetValue(void) = 0;
};
```

Třída přímo definuje, že typ návratové hodnoty je ENUM_UNITTYPE_INTEGER (což je nový enumerátor zavedený do ***EnumUnitType***). Dále třída přidává abstraktní metodu ***GetValue()***, pomocí které lze získávat výsledek uložený v objektu.

Výpočetní jednotka musí implementovat tuto třídu a její metody ***Release()*** a ***GetValue()***. Obecně pro implementaci všech tříd pro návratové hodnoty platí, že musí být synchronizovány pro vícenásobný přístup.

Program pak jednotku používá tak, že získá výsledek pomocí metody ***TUnitInterface::GetResult()***, což je datový typ ***TUnitRetTypeInterface***. Dále podle ***TUnitInterface::GetType()*** nebo podle ***TUnitRetTypeInterface::GetType()*** získá informace o typu návratové hodnoty a přetypuje výsledek na správný datový typ. Vzhledem k jednoduššímu používání je nutné, aby byla pro odvozování výpočetních tříd použita pouze jednoduchá dědičnost a tím je zajištěno, že přetypování z bázové třídy na třídu potomka bude bezpečnější.

Jednotka může vrátit místo výsledku hodnotu NULL, která vyjadřuje, že daný výsledek není k dispozici. Pokud je vrácen platný ukazatel na výsledek, pak musí program zajistit, že výsledek bude pomocí metody ***TUnitRetTypeInterface::Release()*** uvolněn zpět do jednotky, kde byl výsledek vytvořen.

Ukázka implementace jednotky

Pro snadnější implementaci jednotek je k dispozici v adresáři IPP projekt SimpleUnitDLL (je součástí celého solution pro VS). V tomto projektu je ukázka implementace jednotky pro výpočet průměrné intenzity každého snímku. Taková jednotka vrací pro každý snímek celé číslo určující intenzitu. Jednotku lze vytvářet libovolně, použití třídy ***TSimpleUnit*** pouze zjednodušuje proces vytváření.

Implementace využívá třídu ***TSimpleUnit***, která je odvozená od třídy ***TUnitInterface*** a umožňuje snazší implementaci výpočetních jednotek. Celý proces implementace je založen na pomocné pracovní třídě ***TSimpleUnitProcessingInterface***, která definuje rozhraní pro

odvozenou třídu, která následně zajišťuje výpočet nad jednotlivými snímky v rámci třídy ***TSimpleUnit***.

Samotná třída ***TSimpleUnit*** je řídící třída představující výpočetní jednotku, která se stará o běh vlákna, načítání snímků, zastavení a spouštění jednotky. Třída odvozená od ***TSimpleUnitProcessingInterface*** pak slouží pouze jako objekt pro výpočet a ukládání výsledků. Její povinné rozhraní je následující.

```
class TSimpleUnitProcessingInterface
{
public:
      virtual void                  ProcessFrame( const TFrame * frame ) = 0;
      virtual                       ~TSimpleUnitProcessingInterface(void) = 0 {};

      virtual EnumUnitType          GetType(void) const
                                    { return ENUM_UNITTYPE_NODATA; };

      virtual TUnitRetTypeInterface*      GetResult( DWORD id )
                                          { return NULL; };
};
```

Význam metod je následující:

- ***ProcessFrame*** – Tato metoda je automaticky volána třídou ***TSimpleUnit*** pro každý nový snímek. Důležitou vlastností této metody je to, že nesmí v žádném případě způsobit selhání (např. v důsledku výjimky).
- ***destruktor*** – pro zděděné třídy je důležité správně definovat destruktor, který bude automaticky zavolán třídou ***TSimpleUnit*** v okamžiku, kdy bude tato třída zrušena.
- ***GetType*** - metoda vrací informace o typu výsledku této jednotky.
- ***GetResult -*** metoda vrací výsledek pro snímek se zadaným ID.

Z rozhraní ***TSimpleUnitProcessingInterface*** je zřejmé, že pokud uživatel chce implementovat jednotku, která nevrací výsledky (např. je pouze vypisuje na standardní výstup), pak stačí odvodit novou třídu od ***TSimpleUnitProcessingInterface***, reimplementovat její metodu ***ProcessFrame*** pro požadovaný výpočet a objekt této nově vytvořené třídy následně použít v konstruktoru třídy ***TSimpleUnit***.

Pokud chce uživatel implementovat jednotku vracející nějakou hodnotu (v našem případě celé číslo), pak musí implementovat novou výpočetní třídu odvozenou od rozhraní ***TSimpleUnitProcessingInterface*** a v této třídě implementovat všechny potřebné metody. Zároveň musí tato nová třída umožňovat ukládání výsledků a jejich management. V našem případě je

k dispozici třída ***TSimpleUnitProcessingInterface_integer***, která poskytuje potřebné metody:

```
class TSimpleUnitProcessingInterface_integer :
                  public TSimpleUnitProcessingInterface
{
//PUBLIC OVERRIDEN METHODS
public:
      virtual void            ProcessFrame( const TFrame * frame ) = 0;

      virtual                 ~TSimpleUnitProcessingInterface_integer(void);

      virtual EnumUnitType    GetType(void) const
                              { return ENUM_UNITTYPE_INTEGER; };

      virtual TUnitRetTypeInterface*      GetResult( DWORD id );

//PUBLIC METHODS
public:

      TSimpleUnitProcessingInterface_integer(void);

      void  AddResult( DWORD id, TUnitRetType_integer_implemented * res );

      TUnitRetType_integer_implemented*   GetObject(void);

      void ReturnObject( TUnitRetType_integer_implemented * object );
};
```

K dispozici jsou nové metody:

- ***AddResult*** – vložení nového výsledku typu ***TUnitRetType_integer_implemented*** do objektu se zadaným ID, které reprezentuje snímek, jehož se výsledek týká.
- ***GetObject*** – touto metodou vrátí třída nový objekt typu TUnitRetType_integer_implemented pro uložení výsledku. Třída je zodpovědná za management výsledků, proto je nutné získat tento objekt právě přes ní.
- ***ReturnObject*** – navrácení objektu s výsledkem zpátky do objektu, kde vznikl.

Od třídy ***TSimpleUnitProcessingInterface_integer*** stačí následně odvodit již konkrétní výpočetní třídu, která bude provádět požadovaný výpočet. Např. v našem případě odvodíme od třídy ***TSimpleUnitProcessingInterface_integer*** novou třídu ***TTestUnit***, a neimplementujeme její metodu ***ProcessFrame***. Možná implementace metody vypadá např. takto:

```
/** Metoda pro zpracovani snimku.
*
*     \param      frame       [in] nove prichozi snimek
*                             zaslany jednotkou TSimpleUnit
*/
void TTestUnit::ProcessFrame( const TFrame * frame )
{
      // ukazkova uloha - vypocet prumerne hodnoty intenzity
      // v monochromatickem obraze

      //informace o snimku
      const DWORD f_id  = frame->GetTimestamp().GetID();
      const DWORD f_width  = frame->GetImageSet()->GetGray()->GetWidth();
      const DWORD f_height = frame->GetImageSet()->GetGray()->GetHeight();

      //data snimku v odstinech sedi
      const DWORD f_size = frame->GetImageSet()->GetGray()->GetDataSize();
      const unsigned char * f_data =
            (unsigned char *)frame->GetImageSet()->GetGray()->GetData();

      //soucet intenzit
      int sum = 0;
      for ( DWORD i = 0 ; i < f_size ; i++ )
      {
            sum += f_data[i];
      }

      //vysledna intenzita
      sum /= f_size;

      //ZAPIS VYSLEDKU
      //ziskame objekt pro ulozeni vysledku
      TUnitRetType_integer_implemented * res = this->GetObject();

      //nastaveni hodnoty vysledku do objektu pro ulozeni vysledku
      res->SetValue( sum );

      //ulozime vysledek pro tento snimek - je identifikovat jednoznacnym ID
      this->AddResult( frame->GetTimestamp().GetID(), res );
};
```

Tím je implementace výpočetního objektu typu ***TTestUnit*** dokončena. Tento objekt pak lze snadno použít pro vytvoření jednotky pomocí ***TSimpleUnit*** např. takto:

```
TTestUnit * worker      = new TTestUnit;
TSimpleUnit * unit      = new TSimpleUnit( dispatcher, worker, FALSE );
```

Nejprve se tedy vytvoří objekt ***„worker“***, který představuje vlastní výpočetní objekt. Ten se poté předá novému objektu typu ***TSimpleUnit***, který dále vyžaduje informaci o dispatcheru, od kterého bude získávat snímky a také informaci o tom, zda budou snímky touto jednotkou zamykány nebo ne (v tomto případě jednotka snímky nezamyká).

Rozhraní DLL knihovny s výpočetní jednotkou

Vzhledem k tomu, že celý systém je navržen velmi modulárně, je vhodné implementovat výpočetní jednotky uvnitř samostatných DLL knihoven a exportovat pouze základní rozhraní, takže používání těchto jednotek bude skryto od složitosti celé implementace. Implementace uvnitř knihovny samozřejmě není podmínkou, obzvláště v případě počáteční implementace není nutné se s ní zabývat. Po dokončení jednotky je však vhodné tuto přesunout do knihovny s následujícími exportovanými funkcemi:

```
extern "C" __declspec(dllexport)
EnumDLLType       GetType(void);

extern "C" __declspec(dllexport)
TUnitInterface*   CreateUnit(TDispatcherInterface * dispatcher );

extern "C" __declspec(dllexport)
EnumUnitType      GetUnitResultType(void);

extern "C" __declspec(dllexport)
const char *      GetUnitCreateInfo(void);

extern "C" __declspec(dllexport)
const char *      GetUnitResultInfo(void);
```

Význam funkcí je následující:

- ***GetType*** – určení typu následující funkce ***CreateUnit***. Tato funkce se může obecně lišit podle potřebných parametrů pro danou jednotku, typicky každá funkce ***CreateUnit*** požaduje jako první parametr rozhraní na dispatcher.
- ***CreateUnit*** - tato funkce vytvoří a vrátí jednotku. Může vrátit NULL pokud se jednotku nepodařilo vytvořit. Tato funkce má obecně libovolné parametry, typ funkce lze rozpoznat podle návratové hodnoty funkce ***GetType()***.
- ***GetUnitResultType*** – funkce vrací návratový typ této jednotky. Uživatel jednotky tam může ještě před načtením jednotky zjistit datový typ, který jednotka vrací.
- ***GetUnitCreateInfo*** - tato funkce vrací textový řetězec s popisem funkce ***CreateUnit***. Tento řetězec může obsahovat např. deklaraci funkce a popis parametrů. Uživatel tak může snadno zjistit, jak zadanou jednotku vytvořit. Řetězec není nijak omezen, může tedy být víceřádkový, obsahovat jednoduché formátování apod.
- ***GetUnitResultInfo –*** tato funkce vrací textový řetězec s popisem návratové hodnoty jednotky. Řetězec může podobně jako u funkce ***GetUnitCreateInfo*** obsahovat obsáhlý popis, který například informuje o skutečném návratovém typu, o

interpretaci výsledků apod. Dále může tento popis obsahovat jednoduchou charakteristiku jednotka, která popisuje co vlastně jednotka dělá.

Ukázka použití řetězce

Inicializace řetězce

Jako první je nutné správně inicializovat ty části řetězce, které nemají vlastní výpočetní vlákno. Těmi jsou fronta a dispatcher, vytvoříme tyto dva prvky a propojíme.

```
// nejprve vytvorime staticke prvky (neobsahuji sve vypocetni vlakno)
// celeho retezce
TFrameQueue * pipeline_queue  = new TFrameQueue;
TDispatcher * pipeline_dispatcher = new TDispatcher( pipeline_queue );
```

Nyní vytvoříme zdroj obrazových dat. Musíme tedy nejprve získat nějaký fyzický zdroj obrázků, dále nastavit do dispatcheru informace o velikosti obrázků a poté vytvořit samostatné vlákno, které bude obrázky z kamery načítat a posílat přes dispatcher do řetězce.

```
// nyni musime vybrat zdroj obrazovych dat - nacteme tedy kameru ze zvolene DLL
TCameraLoader * pipeline_cameraLoader = new TCameraLoader;
TCameraAbstract * pipeline_camera =
      pipeline_cameraLoader->GetCamera_DirectShow( "dll_pro_kameru.dll" );

// nastavime informace o vlastnostech snimku do dispatcheru
// ten bude tyto informace poskytovat vypocetnim jednotkam
pipeline_dispatcher->SetFramesInfo(
      pipeline_camera->GetWidth(), pipeline_camera->GetHeight() );

// vytvorime vlakno pro kameru, ktery bude ziskavat snimky
// kazdych 40 milisekund (25 FPS)
// a bude disponovat maximalne 100 snimky, ktere smi poslat do retezce
TCameraThread * pipeline_cameraThread =
      new TCameraThread( pipeline_dispatcher, pipeline_camera, 40, 100 );
```

Nyní je celý řetězec inicializován a můžeme k němu připojit výpočetní jednotky.

Napojení jednotek

Pro napojení jednotek je vhodné použít pomocnou třídu ***TUnitLoader***. Každý objekt této třídy se stará o načtení jedné jednotky. To v sobě zahrnuje načtení příslušné DLL knihovny a vytvoření jednotky z této knihovny.

```
TUnitLoader pipeline_unit01_loader;

//budeme nacitat jednotku, ktera neprijima zadne dodatecne parametry
TUnitInterface * pipeline_unit01 =
      pipeline_unit01_loader.GetUnit_BASIC(
            "soubor_dll_s_jednotkou", pipeline_dispatcher );
```

V tomto případě jsme načetli jednotku, která nepřijímá žádné dodatečné parametry při vytváření. Vyžaduje pouze rozhraní dispatcheru. Po vytvoření je možné jednotku okamžitě spustit. Jednotka poběží naprázdno, protože stále není spuštěn zdroj obrazových dat. Protože už další jednotky přidávat nebudeme, spustíme zároveň celý řetězec a tím se snímky začnou dostávat do výpočetní jednotky.

Běh řetězec

Běh řetězce by měl být kontrolován hlavním vláknem aplikace, které se typicky stará o výsledky získávané z jednotek a případné zobrazování výstupů. Typická smyčka může vypadat např. takto:

```
while ( pokracovat ) // dokud má aplikace běžet
{
      // získání zobrazitelného snímku z fronty
      TFrameReal * frame = pipeline_queue->GetRenderableFrame();

      if ( frame )
      {
            // mame snimek
            // ziskame jeho jedinecne ID
            DWORD frameID = frame->GetTimestamp().GetID();

            // pokusime se k nemu ziskat vysledek
            TUnitRetTypeInterface * res = pipeline_unit01->GetResult( frameID );
            if ( res )
            {
                  // mame vysledek
                  // pretypujeme na spravny typ a pouzijeme ziskanou hodnotu

                  //musime vratit vysledek zpet
                  res->Release();
            }
            //zde napr. snimek zobrazime

            //musime uvolnit referenci snimku
            frame->Release();
      }
}
```

Odstranění řetězce

Po dokončení hlavní smyčky programu musí být řetězec a všechny jednotky správně ukončeny. Především je důležité dodržovat základní pravidla. Typicky před zničením jednotky se nesmí v aplikaci vyskytovat žádná reference na výsledek vrácený jednotkou. Podobně před zničením zdroje obrazových dat musí být všechny reference na snímky z řetězce odstraněny. Proto je dobré dodržovat následující postup.

Nejprve je důležité uvolnit všechny výsledky všech výpočetních jednotek. V našem případě jsou výsledky uvolňovány automaticky v hlavní smyčce aplikace. Následně je nutné zastavit a zrušit všechny výpočetní jednotky.

```
// zastavime nasi vypocetni jednotku
pipeline_unit01->Stop();
pipeline_unit01->Release();
pipeline_unit01 = NULL;
```

Dále je nutné zastavit zasílaní snímků do řetězce. Zdroj ale není možné ihned zrušit, protože fronta snímků nebo disptacher stále mohou obsahovat reference na některé snímky. Proto je nutné tyto reference z těchto prvků odstranit ihned v tomto kroku.

```
// nejprve zastavime generovani novych snimku
pipeline_cameraThread->Stop();

// nyni musime uvolnit vsechny snimky z retezce
pipeline_queue->Free();
pipeline_dispatcher->Free();
```

V tomto okamžiku už můžeme odstranit zdroj obrazových dat a zároveň i statické prvky řetězce.

```
// odstranime cely system kamery
delete pipeline_cameraThread;
pipeline_camera->Destroy();
pipeline_camera = NULL;
delete pipeline_cameraLoader;

//odstranime staticke slozky retezce
delete pipeline_dispatcher;
delete pipeline_queue;
```

Tím je řetězec úspěšně odstraněn a aplikace může být řádně ukončena.

Tipy na závěr

Vzhledem k možnosti modulárního vývoje aplikací se velice doporučuje implementovat jednotky do DLL knihoven. Implementace jednotek by měla být maximálně bezchybná a to takovým způsobem,

aby se např. v případě pádu jednotky tato jednotka sama uvedla do nějakého speciálního stavu, ve kterém nebude obsahovat reference na žádné snímky a pouze bude čekat na ukončení aplikace. Takový přístup implementace jednotek chrání hlavní aplikaci i ostatní jednotky od nečekaného pádu aplikace.

Vzhledem k tomu, že jednotky jsou často výpočetně náročné, je často vhodné implementovat výpočetní vlákna těchto jednotek s nižší prioritou vzhledem k aktuálnímu procesu. Vždy by mělo platit, že hlavní vlákno musí stihnout zobrazovat výsledky přicházející řetězcem.