Pokročilé programování
v jazyce C pro chemiky
(C3220)

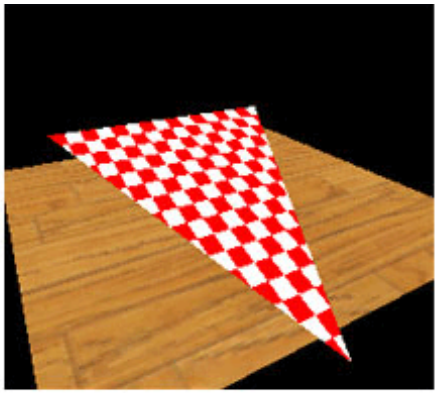
3D grafika v knihovně Qt

Rozhraní pro 3D grafiku

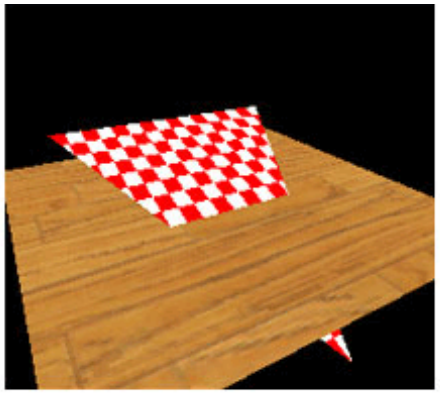
- Běžné grafické knihovny zpravidla podporují pouze 2D grafiku
- Pro 3D grafický výstup jsou využívány specializované knihovny OpenGL a DirectX, které obsahují funkce pro kreslení 3D grafických objektů
- Funkce těchto knihoven jsou schopny přímo komunikovat s příslušným hardwarem (grafickou kartou) a využít tak hardwarové 3D akcelerace
- OpenGL rozhraní je otevřený standard široce dostupný pro různé operační systémy a různé hardwarové platformy, je využíváno především pro profesionální 3D modelování
- DirectX rozhraní bylo vytvořeno firmou Microsoft a je dostupné pouze pro MS Windows, využívá se především pro počítačové hry
- Vulkan – nízkoúrovňové rozhraní, nevhodné pro běžnou práci
- V oblasti profesionální grafiky (včetně molekulového modelování) se používá hlavně rozhraní OpenGL
- Úvod do OpenGL: https://www.glprogramming.com/red/

Renderování ve 3D grafice

- Při vykreslování objektů (čar, geometrických tvarů) v 2D grafice se spočítá hodnota souřadnice *x* a *y* každého vykreslovaného pixelu a určí se jeho barva; hodnota barvy se zapíše do grafické paměti – tato paměť se nazývá color buffer
- V 3D grafice je potřeba navíc ukládat hodnotu souřadnice *z*, k tomu slouží depth buffer (též Z-buffer)
- Ve funkcích pro vykreslování grafických objektů je třeba v 3D grafických knihovnách specifikovat nejen souřadnice *x* a *y*, ale také souřadnici *z* (tj. souřadnice ve směru předo-zadním)
- Když je příslušný grafický objekt renderován, je spočítána barva a hodnota *x*, *y* a *z* každého pixelu
- Než dojde k zápisu hodnoty pixelu do color bufferu a depth bufferu, porovná se jeho souřadnice *z* s hodnotou v depth bufferu; pixel se do bufferů zapíše pouze tehdy, pokud se nachází více vpředu než je hodnota v depth bufferu; tím je zajištěno, že pixely nacházející se více vzadu nebudou přepisovat pixely nacházející se vpředu



Z-sort



Z-buffer method

Použití OpenGL ve knihovně Qt

- Chceme-li vykreslovat obsah widgetu pomocí OpenGL, musíme widget odvodit ze třídy `QOpenGLWidget` (ta je odvozena z `QWidget`)
- Do souboru musíme vložit příslušný hlavičkový soubor
  `#include <QOpenGLWidget>`
- Ve starých verzích Qt existovala podobná třída `QGLWidget`, která má různá omezení a od verze Qt6 už vůbec není k dispozici. V novém kódu tedy `QGLWidget` nepoužíváme!
- Ke zpřístupnění funkcí OpenGL využijeme vícenásobnou dědičnost. Widget odvodíme i ze třídy `QOpenGLFunctions` (můžeme použít režim **protected**, čímž funkce OpenGL skryjeme před uživateli widgetu).

```
#include <QOpenGLFunctions>
#include <QOpenGLWidget>

class GraphicWidget : public QOpenGLWidget, protected QOpenGLFunctions
{
  Q_OBJECT
  public:
    virtual void initializeGL();
  // Zde budou deklarace a definice dalsich clenu tridy
};
```

Inicializace OpenGL v Qt

- V konstruktoru widgetu neprovádíme žádná volání OpenGL funkcí
- Máme-li speciální požadavky na nastavení bufferů (např. barevnou hloubku či typ depth bufferu), použijeme v konstruktoru `setFormat()`
- Pro inicializaci knihovny OpenGL musíme ve widgetu překrýt virtuální metodu `initializeGL()`, na jejímž začátku zavoláme `initializeOpenGLFunctions()`
- Dále můžeme v `initializeGL()` provádět libovolná jednorázová nastavení OpenGL (zapínání volitelných funkcí, nastavení barvy pozadí, světel, materiálů, textur, atd.)

```
void GraphicWidget::initializeGL()
{
  // Nacteme knihovnu OpenGL a zpristupnime gl*() funkce
  initializeOpenGLFunctions();

  // Dale muze nasledovat jakakoli pocatecni priprava OpenGL sceny
}
```

Nastavení OpenGL

- Všechna počáteční nastavení OpenGL umístíme do `initializeGL()`
- Pro korektní renderování 3D objektů musí být zapnuto používání depth bufferu (standardně je jeho používání vypnuto)
- K zapnutí a vypnutí různých nastavení v OpenGL používáme funkce `glEnable()` a `glDisable()`, kterým předáme příslušný parametr
- Používání depth bufferu zapneme pomocí `glEnable(GL_DEPTH_TEST)`
- Barvu pozadí, používanou k mazání bufferů před každým kreslením, nastavíme funkcí funkcí `glClearColor()`

```
void GraphicWidget::initializeGL()
{
  initializeOpenGLFunctions();

  glEnable(GL_DEPTH_TEST);  // Zapneme používání depth bufferu
  // Specifikujeme barvu pro vyplnění color bufferu;
  // specifikují se hodnoty R, G, B, A v rozsahu 0.0 až 1.0
  glClearColor(0.0, 0.0, 0.0, 1.0);
}
```

Metoda paintGL()

- U widgetů odvozených z QOpenGLWidget vykreslujeme 3D objekty v metodě paintGL(), která virtuálně překrývá příslušnou metodu z QOpenGLWidget
- Metodu paintEvent() v tomto případě nepoužíváme

```
#include <QOpenGLFunctions>
#include <QOpenGLWidget>

class GraphicWidget : public QOpenGLWidget, protected QOpenGLFunctions
{
  Q_OBJECT
  public:
    virtual void initializeGL();
    virtual void paintGL();
};
```

```
void GraphicWidget::paintGL()
{
  // Zde budou prikazy pro vykreslovani pomoci OpenGL
}
```

Základy OpenGL

- Knihovna OpenGL používá asi 250 různých funkcí, které slouží k nastavení vlastností renderování a k vykreslování objektů
- Všechny základní funkce OpenGL začínají písmeny gl, za nimiž následuje název funkce; názvy maker začínají GL_
- Většina funkcí existuje v několika verzích, aby bylo možné používat typy **double**, **float**, **int** podle potřeby (knihovna nevyužívá přetížení funkcí, protože její návrh je v jazyce C, nikoliv C++)
- Tyto funkce jsou odlišeny posledním písmenem v názvu funkce (d pro **double**, f pro **float**, i pro **int** a s pro **short int**)
- Pro kreslení obdélníku jsou např. v OpenGL dostupné následující funkce (namísto standardních názvů typů používá knihovna OpenGL vlastní označení, tj. GLdouble, GLfloat, GLint atd.):

```
void glRectd(GLdouble x1, GLdouble y1, GLdouble x2, GLdouble y2);
void glRectf(GLfloat x1, GLfloat y1, GLfloat x2, GLfloat y2);
void glRecti(GLint x1, GLint y1, GLint x2, GLint y2);
void glRects(GLshort x1, GLshort y1, GLshort x2, GLshort y2);
```

Příprava OpenGL scény

- Než začneme v metodě `paintGL()` kreslit, je třeba vymazat případný obsah, který zbyl v bufferech po předchozím kreslení
- Knihovna Qt často buffery maže sama, ale je vhodné na to nespoléhat
- K mazání obsahu bufferů slouží funkce `glClear()`, jejíž parametr specifikuje, které buffery se mají vymazat; pro vyplnění color-bufferu použije tato funkce barvu, kterou jsme dříve specifikovali funkcí `glClearColor()`

```
void GraphicWidget::paintGL()
{
  // Napred smazeme color i depth buffer
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  // Tady budou dalsi prikazy provadejici jakekoli kresleni
}
```

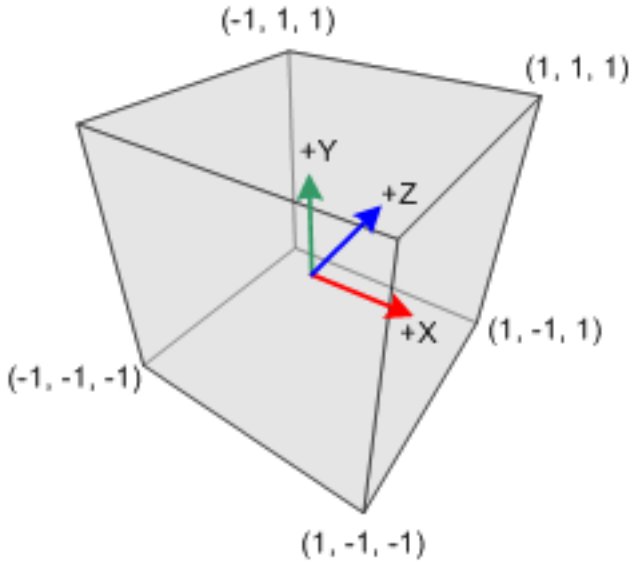
Transformační matice v OpenGL

- V OpenGL můžeme nastavit transformace vykreslovaných objektů (rotace, posunutí, zvětšení/zmenšení)
- Transformace se provádí pomocí transformační matice, se kterou zpravidla manipulujeme pomocí příslušných funkcí
- V OpenGL existují dvě matice: matice *projection* slouží pro nastavení pozice pozorovatele; matice *modelview* nastavuje transformace vykreslovaných objektů
- Funkce `glMatrixMode()` slouží pro výběr matice, se kterou budeme následně pracovat
- Funkce `glLoadIdentity()` nastaví do aktuální matice hodnoty jednotkové matice (tj. žádná transformace)
- Pro příslušné transformace se používají funkce `glTranslate()`, `glRotate()`, `glScale()` (existují ve verzích s příponou `f` nebo `d`)

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
  // Rotace o uhel 60 stupnu kolem vektoru (0.0, 1.0, 0.0), tj. osy y
glRotatef(60, 0.0, 1.0, 0.0);
  // Rotace o uhel 20 stupnu kolem vektoru (1.0, 0.0, 0.0), tj. osy x
glRotatef(20, 1.0, 0.0, 0.0);
  // Zde se budou renderovat objekty, transformace se aplikuji
  // v obracenem poradi, tj. nejdrive rotace kolem osy x pak y
```

Souřadnicový systém v OpenGL

- Souřadnice v OpenGL neodpovídají pozicím pixelů (jak tomu zpravidla bývá při 2D renderování)
- Souřadnice v OpenGL se zpravidla specifikují jako neceločíselné hodnoty, jejich výchozí nastavení je v rozsahu -1.0 až 1.0, pozice 0.0, 0.0, 0.0 odpovídá středu okna (a středu scény ve směru osy z)
- Výchozí nastavení souřadnic v OpenGL je následující:
  - x: **-1.0** (levý okraj okna), **0.0** (střed), **1.0** (pravý okraj okna)
  - y: **1.0** (horní okraj okna), **0.0** (střed), **-1.0** (spodní okraj okna)
  - z: **1.0** (zadní stěna), **0.0** (střed), -**1.0** (přední stěna)
- Pomocí funkcí OpenGL lze nastavit jiný rozsah souřadnic



Kreslení v OpenGL

- Kreslení v OpenGL spočívá ve vykreslování čar, trojúhelníků, čtyřúhelníků a polygonů, všechny 3D objekty musí být poskládány z těchto útvarů
- Tyto útvary vykreslujeme tak, že nejdříve zavoláme funkci `glBegin()`, do které předáme argument specifikující typ vykreslovaných objektů (čáry, trojúhelníky, čtyřúhelníky, polygony)
- Souřadnice vrcholů těchto útvarů specifikujeme funkcí `glVertex()`
- Kreslení dokončíme zavoláním `glEnd()`
- Funkce `glVertex()` existuje v několika verzích `glVertex`*`Nx`*`()`, kde *N* specifikuje počet předávaných souřadnic (2, 3, 4) a *x* specifikuje typ předávaných proměnných (i, s, f, d, vysvětlení viz dříve)

```
  // Kreslime dve cary
glBegin(GL_LINES);
    // Prvni cara zacina v bode (-0.5, 0.5, 0.0)
  glVertex3f(-0.5, 0.5, 0.0);
    // a konci v bode glVertex3f(0.5, -0.5, 0.0)
  glVertex3f(0.5, -0.5, 0.0);
    // Druha cara zacina v bode (0.5, 0.5, 0.0)
  glVertex3f(0.5, 0.5, 0.0);
    // a konci v bode (-0.5, -0.5, 0.0)
  glVertex3f(-0.5, -0.5, 0.0);
glEnd();
```

Kreslení v OpenGL

- Mezi voláním `glBegin()` a `glEnd()` se může nacházet libovolný počet volání `glVertex()` - tyto vrcholy jsou pak postupně zpracovávány a jsou vykreslovány příslušné objekty
- Funkce `glBegin()` přijímá následující hodnoty:

  GL_LINES - kreslení jednotlivých čar, které na sebe nenavazují; vždy se dva za sebou jdoucí vrcholy interpretují jako počáteční a koncový bod čáry

  GL_LINE_STRIP – kreslí čáry které jsou vzájemně propojené

  GL_LINE_LOOP – podobně jako předchozí ale první a poslední vrchol jsou propojeny čárou

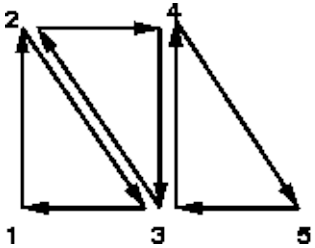
  GL_TRIANGLES – kreslí trojúhelníkové plochy, tři vrcholy jdoucí za sebou jsou vždy interpretovány jako souřadnice jednoho trojúhelníku

  GL_TRIANGLE_STRIP a GL_TRIANGLE_FAN – kreslí na sebe navazující trojúhelníky

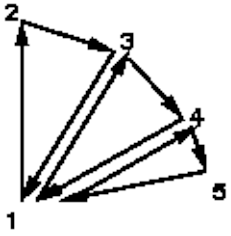
  GL_QUADS a GL_QUAD_STRIP – kreslení obdélníků

  GL_POLYGON – kreslení konvexních polygonů

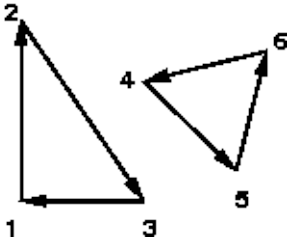
  GL_POINTS – nakreslí bod v pozici každého vrcholu



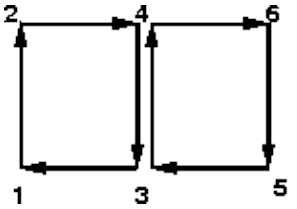
Triangle strip



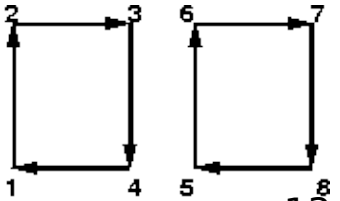
Triangle fan



Independent triangles



Quad strip

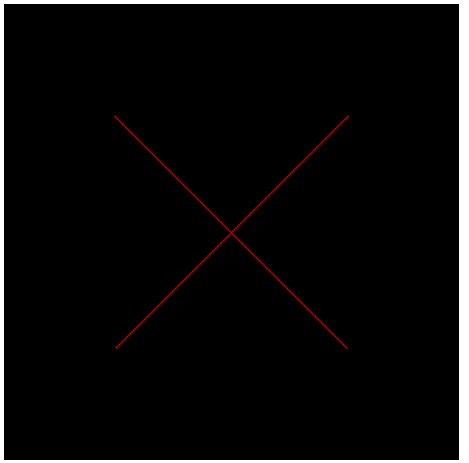


Independent quads

Nastavení barvy v OpenGL

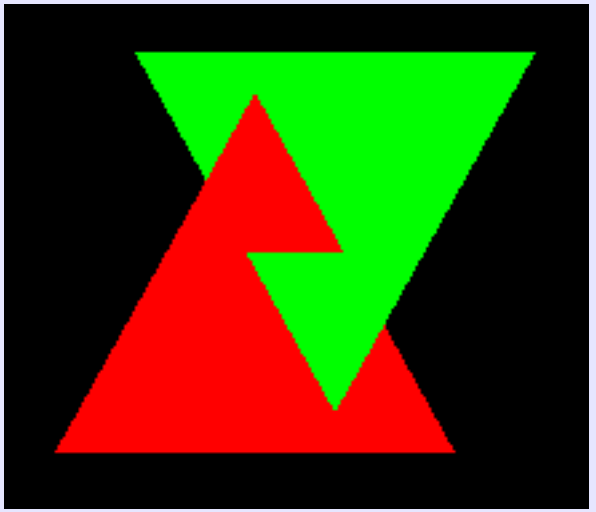
- Barvu vykreslovaných objektů specifikujeme voláním funkce `glColor()`
- Specifikují se jednotlivé složky barvy: **R** (*red*), **G** (*green*), **B** (*blue*) a případně průhlednost **A** (alfa kanál; 0 = průhledné, 1 = neprůhledné)
- Funkce `glColor()` existuje v několika verzích `glColor`*Nx*`()`, kde *N* specifikuje počet předávaných hodnot barev (3, 4) a *x* specifikuje typ předávaných proměnných (i, s, f, d a další, vysvětlení viz. dříve)
- Funkci můžeme zavolat před začátkem kreslení (tj. před zavoláním `glBegin()`) ale také kdykoliv mezi voláními `glBegin()` a `glEnd()` - hodnota barvy se vždy uplatní od okamžiku volání `glColor()`

```
  // Nastavime barvu na cervenou
glColor3f(1.0, 0.0, 0.0);
  // Kreslime dve cary
glBegin(GL_LINES);
  glVertex3f(-0.5, 0.5, 0.0);
  glVertex3f(0.5, -0.5, 0.0);
  glVertex3f(0.5, 0.5, 0.0);
  glVertex3f(-0.5, -0.5, 0.0);
glEnd();
```



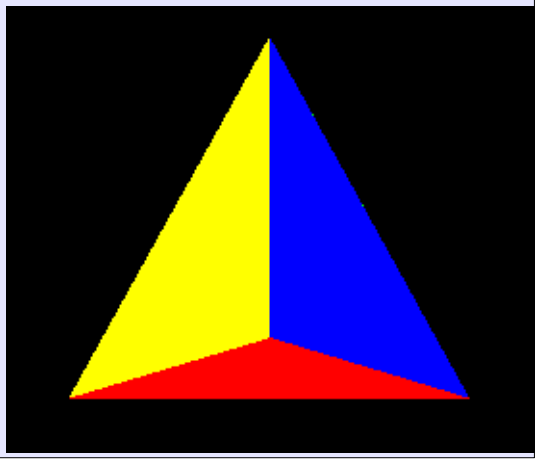
Renderování v OpenGL – příklad 1

```
// Ukazka kresleni dvou protinajicich se trojuhelniku
void GraphicWidget::paintGL()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();

  // Nakresli se dva trujuhelniky vzajemne posunute a protnute skrz
  glBegin(GL_TRIANGLES);
      // Prvni trojuhelnik bude cerveny
    glColor3f(1.0, 0.0, 0.0);
    glVertex3f(-0.6, -0.5, 0.5);
    glVertex3f(0.4, -0.5, 0.5);
    glVertex3f(-0.1, 0.4, -0.5);
      // Druhy trojuhelnik bude zeleny
    glColor3f(0.0, 1.0, 0.0);
    glVertex3f(-0.4, 0.5, 0.5);
    glVertex3f(0.6, 0.5, 0.5);
    glVertex3f(0.1, -0.4, -0.5);
  glEnd();
}
```



Renderování v OpenGL – příklad 2

```
// Ukazka kresleni pravidelneho ctyrstenu
void GraphicWidget::paintGL()
{
  // Zde bude podobna priprava sceny jako v predchozim prikladu
  // Objekty budou pootocene o 10 stupnu kolem osy x
  glRotatef(10.0, 1.0, 0.0, 0.0);
  // Objekty budou pootocene o 60 stupnu kolem osy y
  glRotatef(60.0, 0.0, 1.0, 0.0);
  glBegin(GL_TRIANGLES);  // Kreslime ctyrsten
    glColor3f(1.0, 0.0, 0.0); // Jedna stena bude cervena
    glVertex3f(0.0, -0.4, 0.8);
    glVertex3f(0.7, -0.4, -0.4);
    glVertex3f(-0.7, -0.4, -0.4);
    glColor3f(0.0, 1.0, 0.0); // Druha stena bude zelena
    glVertex3f(0.0, -0.4, 0.8);
    glVertex3f(-0.7, -0.4, -0.4);
    glVertex3f(0.0, 0.8, 0.0);
    glColor3f(0.0, 0.0, 1.0); // Treti stena bude modra
    glVertex3f(0.0, -0.4, 0.8);
    glVertex3f(0.0, 0.8, 0.0);
    glVertex3f(0.7, -0.4, -0.4);
    glColor3f(1.0, 1.0, 0.0);  // Ctvrta stena
    glVertex3f(0.7, -0.4, -0.4); // bude zluta
    glVertex3f(0.0, 0.8, 0.0);
    glVertex3f(-0.7, -0.4, -0.4);
  glEnd();
}
```



Interaktivní manipulace pomocí myši v Qt

- Programy pro vizualizaci 3D objektů často poskytují možnost otáčet s objektem pomocí myši
- Pro zpracování pohybu myši slouží v knihovně Qt funkce `mouseMoveEvent()` která je zavolána pokud změníme polohu myši a zároveň je stisknuto tlačítko myši
- Na pohyb myši zpravila program reaguje tak, že nastaví hodnoty pro geometrickou transformaci objektů a poté okno překreslí
- Pro překreslení okna používajícího OpenGL grafiku voláme metodu `update()`, stejně jako u běžných widgetů

Manipulace pomocí myši – příklad – část 1

```
class GraphicWidget : public QOpenGLWidget, protected QOpenGLFunctions
{
  Q_OBJECT
  protected:
    virtual void initializeGL();
    virtual void paintGL();
    // Metoda pro zpracovani stisknuti tlacitka mysi
    virtual void mousePressEvent(QMouseEvent *event);
    // Metoda pro zpracovani pohybu mysi
    virtual void mouseMoveEvent(QMouseEvent *event);
  private:
    // Do nasledujicich promennych se pri kazdem pohybu mysi
    // ulozi hodnoty uhlu rotace kolem os x a y
    double rotationX = 10, rotationY = 60;
    // Do nasledujicich promennych se ukladaji souradnice mysi
    // po predchazejicim pohybu
    int lastPosX = 0, lastPosY = 0;
};
```

Manipulace pomocí myši – příklad – část 2

```
void GraphicWidget::paintGL()
{
  // Zde je uvedena pouze cast kodu, ktera zajisti rotaci objektu
  glRotatef(rotationX, 1.0, 0.0, 0.0);
  glRotatef(rotationY, 0.0, 1.0, 0.0);
}

void GraphicWidget::mousePressEvent(QMouseEvent *event)
{
  cout << "Stisknuto tlacitko mysi!" << endl;
  // Pri stisknuti tlacitka mysi si ulozime aktualni pozici mysi
  lastPosX = event->x();
  lastPosY = event->y();
}

void GraphicWidget::mouseMoveEvent(QMouseEvent *event)
{
  const double rotScale = 0.5; // Nastaveni citlivosti rotace na pohyb mysi
  // Uhel pro rotaci spocitame jako rozdil mezi aktualni pozici
  // mysi a predchozi pozici, vynasobeny vhodnym skalovacim faktorem.
  rotationX += rotScale * (lastPosY - event->y());
  rotationY += rotScale * (lastPosX - event->x());
  update();  // Pozadavek na prekresleni okna
  // Ulozime aktualni pozici mysi
  lastPosX = event->x();
  lastPosY = event->y();
}
```

Použití časovače v Qt

- V programech někdy potřebujeme aby se v pravidelných intervalech provedla určitá akce
- Pro tento účet používáme tzv. časovač jehož funkci plní v knihovně Qt objekty třídy `QTimer`
- Objekt třídy `QTimer` zpravidla definujeme jako člen třídy
- Časovač spustíme pomocí metody `start()` třídy `QTimer`, jejímž parametrem je časový interval v milisekundách; časovač poté bude opakovaně generovat signál `timeout()` v uvedených intervalech
- Časovač lze zastavit voláním metody `stop()`
- Pomocí standardních mechanizmů zpracování signálů zajistíme volání vhodné metody slotu při generování signálu `timeout()`
- Časovače se hojně využívají při animaci objektů

Použití časovače v Qt – příklad – část 1

```
#include <QTimer>

class GraphicWidget : public QOpenGLWidget, protected QOpenGLFunctions
{
  // Zde budou definice a deklarace ostatnich clenu tridy
  public slots: // Verejne sloty volane jinymi widgety
    // Pro spusteni a zastaveni casovace pouzijeme dve tlacitka
    // pro nez definujeme nasledujici dve metody slotu
    void startRotation();
    void stopRotation();
  protected slots: // Sloty pro interni pouziti
    // Metoda timerTick() bude volana po kazdem 'tiknuti' casovace
    void timerTick();
  private:
    QTimer* timer;
};
```

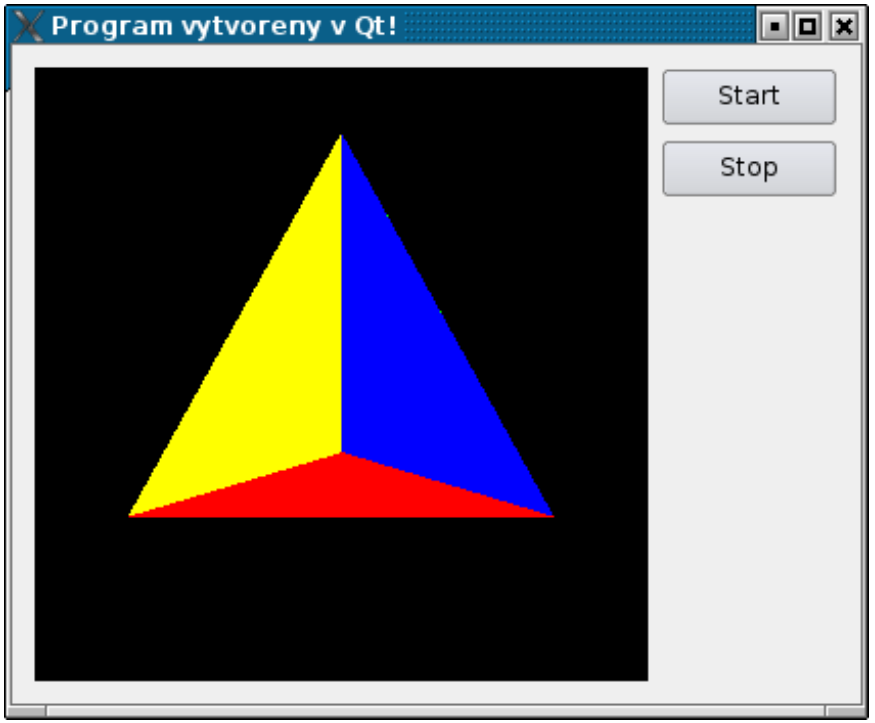
```
GraphicWidget::GraphicWidget() : timer(new QTimer(this))
{
  // V konstruktoru vytvorime propojeni mezi signalem timeout()
  // z casovace a metodou timerClick() tridy GraphicWidget
  connect(timer, &QTimer::timeout, this, &GraphicWidget::timerTick);
}
```

Použití časovače v Qt – příklad – část 2

```
void GraphicWidget::startRotation()
{
  cout << "Bylo stisknuto tlacitko Start" << endl;
  // Spustime casovac, bude generovat signal kazdych 100 ms
  timer->start(100);
}

void GraphicWidget::stopRotation()
{
  cout << "Bylo stisknuto tlacitko Stop" << endl;
  // Zastavime casovac
  timer->stop();
}

void GraphicWidget::timerTick()
{
  cout << "Tiknuti casovace" << endl;
  // Nastavime rotaci kolem osy y o 10 stupnu
  rotationY += 10;
  // Zasleme pozadavek na prekresleni OpenGL widgetu
  update();
}
```

Cvičení

1. Vytvořte program (vycházející z programu z předchozího cvičení), který bude v okně widgetu `GraphicWidget` zobrazovat objekt pravidelného čtyřstěnu s různě zbarvenými stěnami. Objektem bude možné rotovat pomocí myši. Dále bude možné pomocí dvou tlačítek *Start* a *Stop* spustit a zastavit animaci jeho rotace kolem osy y. Widget `GraphicWidget` odvoďte od tříd `QOpenGLWidget` a `QOpenGLFunctions`. **3 body**



Bonus: Nastavení vlastností materiálu

- Modelované vlastnosti materiálu použitého k tvorbě 3D objektů můžeme upravovat pomocí funkce `glMaterialfv()`
- Jako první argument předáváme většinou `GL_FRONT` (nastavení lícové strany materiálu)
- Druhý argument vybírá, jakou vlastnost chceme nastavovat, třetím argumentem je pole `GLfloat` s hodnotou či hodnotami vlastnosti
- Pomocí `glEnable(GL_COLOR_MATERIAL)` můžeme zapnout přebarvování materiálu v závislosti na voláních `glColor()`
- Vše provádíme v metodě `initializeGL()`

```
// Nastavíme základní barvu odlesků (RGBA, zde neprůhledná bílá)
GLfloat specular[] = { 1.0, 1.0, 1.0, 1.0 };
glMaterialfv(GL_FRONT, GL_SPECULAR, specular);

// Parametr určující ostrost/rozptyl odlesku (rozsah 1.0-128.0)
GLfloat shininess[] = { 50.0 };
glMaterialfv(GL_FRONT, GL_SHININESS, shininess);

// Zapneme barvení pomocí glColor()
glEnable(GL_COLOR_MATERIAL);
```

Bonus: Osvětlení scény

- Světelné zdroje můžeme vytvářet pomocí funkce `glLightfv()`
- Podobně jako u `glMaterialfv()` můžeme nastavovat různé vlastnosti, vybírané druhým argumentem
- `GL_POSITION` určuje polohu světla pomocí čtyř reálných hodnot: x/y/z souřadnice a příznak určující, zda je světlo v nekonečnu v daném směru (0) nebo jde o bodový zdroj v dané poloze (1)
- Pomocí `glEnable(GL_LIGHTING)` zapínáme podporu pro světla, potom pomocí `glEnable(GL_LIGHT0)` rozsvítíme první světlo
- Nastavování opět provádíme v metodě `initializeGL()`

```
// Definujeme polohu prvního světla: uprostřed roviny XY,
// u pozorovatele (z = -1), zdroj v nekonečnu (0),
// tzn. slunce za zády pozorovatele
GLfloat lightPosition[] = { 0.0, 0.0, -1.0, 0.0 };
glLightfv(GL_LIGHT0, GL_POSITION, lightPosition);

// Aktivujeme zpracování světel při renderování
glEnable(GL_LIGHTING);

// Rozsvítíme světlo LIGHT0
glEnable(GL_LIGHT0);
```

Bonus: Normálové vektory

- Výpočet osvětlení vyžaduje znát pro každou plochu její normálový vektor (vektor jednotkové délky směřující ven z modelovaného tělesa)
- Normálové vektory se nastavují pomocí funkce `glNormal3f()`
- Nastavený normálový vektor se aplikuje na všechny následující vrcholy (`glVertex`), dokud není změněn
- Způsob stanovení normálového vektoru závisí na modelovaném tělese (např. z vektorového součinu hran trojúhelníku, derivace křivky)

```
// Kreslíme čtyrstěn
glBegin(GL_TRIANGLES);
// Červená stěna
glColor3f(1.0, 0.0, 0.0);
// Vezmeme protější vrchol (0, 0.8, 0), tedy ten,
// který není použit jako glVertex pro tuto stěnu
// Obrátíme jen všechna znaménka, normalizovat nemusíme
glNormal3f(0.0, -0.8, 0.0);
glVertex3f(0.0, -0.4, 0.8);
glVertex3f(0.7, -0.4, -0.4);
glVertex3f(-0.7, -0.4, -0.4);
// analogicky další stěna, atd.
```

Bonus: Skládání složitějších objektů

- Složitější grafické objekty často potřebujeme skládat ze základních tvarů, k jejichž umisťování do sestavy využíváme transformační funkce glTranslate(), glRotate(), ...
- Protože tyto funkce modifikují globální transformační matici GL_MODELVIEW, aplikovaly by se na všechny další operace
- Stav transformační matice můžeme uložit pomocí glPushMatrix() a později se k němu vrátit pomocí glPopMatrix()

```
void GraphicWidget::drawSphere(float x, float y, float z, double r)
{
  // Napřed uložíme aktuální transformační matici
  // (např. globální natočení)
  glPushMatrix();

  glTranslatef(x, y, z);
  // Metoda vykresluje kouli v počátku soustavy souřadnic,
  // nastavená translace ji ale umístí do požadovaného místa
  sphereAtOrigin(r, 10, 10);

  // Obnovíme uloženou transformační matici
  glPopMatrix();
  // Další kreslení nebude ovlivněno předchozím glTranslate()
}
```

Bonus: Cvičení

1. *Povinnou úlohu naleznete na straně 23.*
2. Program z první úlohy upravte tak, že se v každém vrcholu čtyřstěnu vykreslí koule libovolné barvy (poloměr např. 0.1). Pro kreslení koule můžete použít metodu `GraphicWidget::sphereAtOrigin()`, jejíž implementaci si do svého `graphicwidget.cpp` překopírujte ze souboru `/home/tootea/C3220/data/sphereAtOrigin.cpp`. Pro všechny objekty použijte uživatelsky definovaný materiál a scénu nasviťte. Budete muset doplnit normálové vektory pro stěny čtyřstěnu, stačí aproximace odvozená z protějšího vrcholu (viz str. 26). **nepovinná, 3 body**