10/06/2013

FACULTATEA DE AUTOMATICA SI CALCULATOARE

ELEMENTE DE GRAFICA PE CALCULATOR

Laborator 1

1. C/C++

In cadrul laboratoarelor de EGC va fi folosit C/C++ cu STL(doar liste/vectori). In general nu se vor folosi mecanisme de tipul mostenire, polimorfism, metaprogramare cu template-uri, etc. La acest link gasiti un tutorial de C/C++ ce acopera tot ce veti avea nevoie in cadrul laboratoarelor de EGC: http://www.cplusplus.com/doc/tutorial/.

2. IDE

Microsoft Visual Studio este IDE-ul (Integrated Development Enivornment) in care vom lucra la laborator. Aceasta nu inseamna ca sunteti fortati sa il folositi pe statiile voastre. Codul laboratoarelor poate fi usor mutat in alte IDE-uri si/sau in alte sisteme de operare. Pentru cei ce nu sunt familiari cu vreun IDE este bine de punctat ca exista 2 mari notiuni: solutiile si proiectele. La nivelul cel mai simplu de exprimare proiectul este „programul vostru" cu toate setarile de rigoare(dependinte, link, etc), iar solutiile sunt „manageri" de proiecte(proiecte dependinta, batch builds, etc).

Cunostintele minime de utilizare de IDE-uri de care veti avea nevoie la laborator sunt:

- Debug/run -> incepe executia programului

- Compile -> compileaza programul si creeaza executabilul

Cunostinte extra ce va vor fi utile:

- Toggle breakpoint -> in timpul executiei programul se va opri si va astepta, oferind oportunitatea de a observa variabile/clase/etc in starea lor curenta

- Step Into/Step Over -> executie instructiune cu instructiune

3. Primii pasi in grafica

Primul lucru pe care il puteti intreba la o ora de grafica este „cum apar obiectele pe ecran?". Explicatia intuitiva ar fi urmatoarea:

- se pleaca de la niste date reprezentate prin puncte sau alte metode (ex: o carte cu coperti verzi si un logo pe prima pagina).

- Aceste date sunt intai procesate de programul pe care il scriem a.i. sa aiba o semnificatie (ex: cartea este deschisa, cu coperta in sus, luminata de sus cu lumina rosie si de jos cu o lumina albastra)

- Apoi datele sunt trimise la placa grafica ce printr-o serie de transformari (pe care le vom explora incepand din acest laborator) ajung intr-un format usor de procesat pentru programele de pe placa video.

- Dupa procesare, driverul video afiseaza rezultatul pe ecran.

Grafica se preda cel mai usor plecand de la conceptele simple, clare din punct de vedere matematic si apoi avansand la cele mai complexe (atat ingineresti cat si algoritmice si matematice). De aceea in cadrul laboratorului vom folosi intai un framework simplu pentru primele concepte si apoi o biblioteca pentru a facilita interactiunea dintre cod si driverul video.

4. Obiectele

Primul pas pentru a desena ceva este acela de a defini obiectele pe care ne dorim sa le desenam. Framework-ul oferit va da posibilitatea de a desena urmatoarele obiecte 2D:

- Linii (clasa ***Line2D***):
  - O linie este definita prin doua puncte (clasa ***Point2D***):
  - Mod de creare a unei linii:

    *Line2D *line = new Line2D(Point2D(x1,y1),Point2D(x2,y2));*

    - creeaza o linie de culoare neagra care uneste punctele *(x1,y1)* si *(x2,y2)*

    *Line2D *line = new Line2D(Point2D(x1,y1),Point2D(x2,y2),Color(r,g,b));*

    - similar cu primul constructor, dar stabileste si culoarea liniei
- Dreptunghiuri (clasa ***Rectangle2D***):
  - Un drepunghi este definit printr-un punct, lungimea si latimea
  - Mod de creare a unui dreptunghi:

    *Rectangle2D *rectangle = new Rectangle2D(Point2D(x,y),width,height)*

    - creeaza un dreptunghi cu coltul stanga jos *(x,y)*, de latime *width* si inaltime *height*, neumplut, de culoare neagra

    *Rectangle2D *rectangle = new Rectangle2D (Point2D(x,y), width, height, Color(r,g,b), fill)*

    - similar cu primul constructor, dar stabileste si culoarea dreptunghiului prin atributele *(r,g,b)*, si modul de desenare al dreptunghiului – umplut *(fill = true)* sau neumplut *(fill = false)*
- Cercuri (clasa ***Circle2D***):
  - Un cerc este definit prin centru si raza
  - Mod de creare a unui cerc:

    *Circle2D *circle = new Circle2D(Point2D(x,y),radius)*

    - creeaza un cerc cu centrul in *(x,y)*, de raza *radius*, neumplut, de culoare neagra

    *Circle2D *circle = new Circle2D(Point2D(x,y),radius,Color(r,g,b),fill)*

    - similar cu constructorul anterior, dar stabileste si culoarea cercului prin atributele *(r,g,b)*, si modul de desenare – umplut *(fill = true)* sau neumplut *(fill = false)*.
- Poligoane oarecare (clasa ***Polygon2D***)
  - Un poligon este definit printr-o lista de puncte, unite in ordinea in care sunt date
  - Mod de creare a unui poligon:

    Polygon2D polygon = new Polygon2D();*

- creeaza un obiect poligon, care va fi desenat in culoarea neagra, neumplut

- ulterior, se adauga punctele *(x1,y1), (x2,y2),...(xn,yn)* la poligon:

```
polygon->addPoint(Point2D(x1,y1)
polygon->addPoint(Point2D(x2,y2)
....
polygon->addPoint(Point2D(xn,yn)
```

```
Polygon2D* polygon = new Polygon2D(Color(r,g,b),fill)
```

- similar cu primul constructor, dar stabileste si culoarea, si modul de desenare a poligonului (umplut/neumplut).

Toate aceste clase (***Line2D, Rectangle2D, Circle2D, Polygon2D***) sunt derivate din clasa generala ***Object2D***.

5. Transformarile

Pentru a modela si reprezenta un obiect trebuie sa putem sa il transformam din starea sa initiala in starea dorita. Pentru aceasta, intr-un spatiu 2D exista mai multe transformari:

Translatie:

$x' = x + tx;$

$y' = y + ty;$

Rotatie (in jurul originii):

$x' = x*cos(u) - y*sin(u)$

$y' = x*sin(u) + y*cos(u)$

Rotatia relativa la un punct oarecare se rezolva in cel mai simplu mod prin:

- translatarea atat a punctului asupra carui se aplica rotatia cat si a punctului in jurul caruia se face rotatia a.i. cel din urma sa fie originea sistemului de coordonate.

- rotatia normala (in jurul originii),

- translatarea rezultatului a.i. punctul in jurul caruia s-a facut rotatia sa ajunga in pozitia sa initiala

Scalare:

$x' = x* sx;$

$y' = y* sy;$

Daca $sx=sy$ atunci avem scalare uniforma, altfel avem scalare neuniforma. Scalarea relativa la un punct oarecare se rezolva similar cu rotatia relativa la un punct oarecare.

In framework-ul de la laborator vom folosi matricile transformarilor in coordonate omogene.

Translatia:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Scalarea:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Rotatia fata de origine:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(u) & -\sin(u) & 0 \\ \sin(u) & \cos(u) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

De ce sunt necesare matricile? Pentru a reprezenta printr-o singura matrice de transformari o secventa de transformari elementare, in locul aplicarii unei secvente de transformari elementare pe un anume obiect.

Deci, daca dorim sa aplicam o rotatie, o scalare si o translatie pe un obiect, nu facem rotatia obiectului, scalarea obiectului urmata de translatia lui, ci calculam o matrice care reprezinta transformarea compusa (de rotatie, scalare si translatie), dupa care aplicam aceasta transformare compusa pe obiectul care se doreste a fi transformat.

In framework-ul de laborator folosim scrierea in vectori coloana (spre deosebire de scrierea vectori linie, $[x'\ y'\ 1]$). Astfel, daca dorim sa aplicam o rotatie (cu matricea de rotatie *R*), urmata de o scalare (*S*), urmata de o translatie (*T*) pe un punct *(x,y)*, punctul transformat *(x',y')* se va calcula astfel:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}}_{T} \cdot \underbrace{\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{S} \cdot \underbrace{\begin{bmatrix} \cos(u) & -\sin(u) & 0 \\ \sin(u) & \cos(u) & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{R} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Deci, matricea de transformari compuse *M* este *M = T * S * R.*

Transformarile sunt implementate in clasa ***Transform2D***, ce contine:

- atributul *TransformMatrix*, care pastreaza matricea curenta de transformari
- metodele:
  - *loadIdentityMatrix()* – aduce matricea curenta de transformari la matricea identitate (adica la nici o transformare)
  - *multiplyMatrix(matrix)* – inmulteste matricea *matrix* cu matricea curenta de transformari

- *translateMatrix(tx,ty)* – creeaza matricea de translatie cu pasii de translatie *tx* si *ty* si inmulteste aceasta matrice de translatie cu matricea curenta de transformari
- *scaleMatrix(sx,sy)* – creeaza matricea de scalare cu factorii de scalare *sx* si *sy* si inmulteste aceasta matrice de scalare cu matricea curenta de transformari
- *rotateMatrix(u)* – creeaza matricea de rotatie in jurul originii, cu unghiul *u*, si inmulteste aceasta matrice de rotatie cu matricea curenta de transformari
- *applyTransform(object)* – aplica matricea curenta de transformari pe obiectul 2D *object*.

Revenind la exemplul anterior, daca dorim sa aplicam o rotatie, urmata de o scalare, urmata de o translatie, pe obiectul *object*, vom apela urmatoarele functii:

loadIdentityMatrix() //aduce matricea curenta de transformari la matricea identitate (ca sa //nu se aplice si alte transformari care poate au fost anterior aplicate pe //alte obiecte, sau chiar pe obiectul *object*)

//matricea curenta $M = I$ (matricea identitate)

rotateMatrix(u) //acum matricea curenta de transformari este $M = R * I = R$

scaleMatrix(sx,sy) //acum matricea curenta de transformari este $M = S * R$

translateMatrix(tx,ty) // acum matricea curenta de transformari este $M = T * S * R$

applyTransform(object) // se aplica matricea curenta de transformari pe obiectul *object*

Clasa ***Obiect2D*** are, pe langa vectorul de puncte ale obiectului, si un vector de puncte transformate ale obiectului. Atunci cand se deseneaza obiectul, se deseneaza de fapt tinand cont de punctele transformate, nu de cele initiale. De ce nu se doreste modificarea punctelor initiale ale obiectului? De exemplu, daca dorim sa avem un obiect, si sa cream mai multe instante ale sale, este convenabil sa avem intotdeauna acces la punctele componente initiale. La fel, daca dupa o serie de transformari aplicate unui obiect, dorim sa-l aducem in starea initiala, este mult mai usor daca avem salvate punctele sale initiale.

6. Clasa DrawingWindow

Pentru a putea face primii pasi in grafica fara a ne lovi de cum accesam driverul video sau de cum sincronizam firul de executie sau alte probleme vom folosi framework-ul de la laborator. Singurele fisiere ca va sunt necesare atat pentru laborator cat si pentru viitoarea tema sunt *main.cpp* si *Transform2d.cpp*. In rest nu este necesar sa lucrati cu sau sa modificati codul suport, insa sunteti incurajati sa explorati!

- Frameworkul este initializat prin constructor astfel:

```
//creare fereastra
DrawingWindow dw(argc, argv, 600, 600, 200, 100, "Laborator EGC");
//se apeleaza functia init() - in care s-au adaugat obiecte
dw.init();
//se intra in bucla principala de desenare - care face posibila desenarea, animatia si procesarea evenimentelor
dw.run();
```

Functiile din *main.cpp*:

init()

- e o functie de initializare, care e apelata o singura data, inainte de a incepe procesul de desenare
- permite adaugarea de obiecte
- un obiect, dupa ce este creat, trebuie adaugat in lista de obiecte ce vor fi desenate pe ecran (exemplu: *addObject2D(line)* – adauga obiectul *line* - exemplificat in sectiunea 4)
- transformarile pot fi aplicate inainte sau dupa adaugarea obiectului la lista de obiecte de desenat pe ecran

onIdle()

- aceasta functie permite animatia – se declanseaza la fiecare *x* milisecunde
- daca dorim de exemplu sa facem o animatie de translatie pentru un anumit obiect, putem ca la fiecare declansare a functiei *onIdle()* sa-l translatam cu un pas foarte mic. Daca pasul este foarte mic, atunci se creeaza senzatia de animatie continua.
- pentru animatie, trebuie sa tinem cont de faptul ca la apelarea functiei *applyTransform()* din ***Transform2D***, matricea curenta de transformari se inmulteste cu punctele initiale ale obiectului, iar rezultatul este salvat in punctele transformate ale obiectului, care vor fi folosite si la desenarea acestuia. Deci, daca dorim sa cream o animatie de translatie (pe axa Ox), cu un pas foarte mic, ar trebui sa crestem de fapt succesiv pasul de translatie cu o valoare foarte mica:

```
tx += du;  // (dx e foarte mic)
loadIdentityMatrix();//ca sa elimin alte transformari care poate au fost aplicate
                     //pe alte obiecte, sau poate chiar pe obiectul curent, in
                     //frame-urile anterioare
translateMatrix(tx,0);
applyTransform(object);
```

onKey()

- functie care defineste ce se intampla cand se apasa pe o tasta

onMouse()

- functie care defineste ce se intampla cand se da click pe mouse

Responsabil laborator: Anca Morar