Tietokonegrafiikan perusteet (k2013), OT1, palautetta

Author: Paavo Nieminen (paavo.j.nieminen@jyu.fi)

Pisteytin tehtävät Korppiin seuraavasti:

- **0 pistettä**, jos ei ollut palautusta aikarajaan mennessä
- **0.2 pistettä**, jos oli tehty vain joko lineaarialgebra tai kuva
- **0.5 pistettä**, jos oli tehty sekä lineaarialgebra että kuva.

Mitään uusintapalautuksia en vaadi, koska tavallaan joka viikko palautetaan kehitetty versio, jossa toivon mukaan alkupään bugit korjautuvat matkan varrella. Tuohon alle koetin löytää mahdollisimman monesta palautetusta koodista jotakin pientä tai suurempaa kommentoitavaa. **Lukekaa läpi ja korjatkaa omasta asiat, jotka tuntuvat napsahtavan omalle kohdalle**. Jokaista kohtaa demonstroidaan vain yhdellä tai muutamalla koodiesimerkillä, mutta se ei tarkoita, etteikö omasta koodista löytyisi samoja ilmiöitä kuin tässä esitetyistä poiminnoista :). Myöhemmissä demoissa en sitten enää millään ehdi tekemään näin laajaa läpikäyntiä. Toivottavasti tämän lukeminen siis tässä vaiheessa herättää ajatuksia ja/tai keskustelua ja/tai kysymyksiä ja/tai väittelyä :). Pahempia tässä mainittuja puutteita en sitten mielelläni näkisi tulevissa vastauksissa.

Jotkut näistä ovat mielipidekysymyksiä, mutta jotkut ovat dramaattisia bugeja, jotka on syytä korjata. Tästä voi myös lueskelemalla saada hiukan vihjettä, millä tasolla kurssikaverit suurin piirtein ovat ohjelmoinnin oppimisessa. Näyttää siltä että varianssi on kovin suurta, mikä on normaalia. Koodatkaa vaan ahkerasti ja pysähtykää aina välillä miettimään ja oppimaan uutta siltä tasolta ponnistaen, jossa tällä hetkellä itse kukin on.

Sisältö

Yleistä

Väärin toimivat ohjelmat

Harva koodi oli ilman yhtään virhettä. Tarkistelkaa ja korjatkaa koodeja ensi kerran palautukseen (ja sen jälkeenkin). Yksikkötestit auttavat, ja jopa ilman testejäkin kannattaa opetella tunnistamaan lyhyistä koodeistaan, ovatko ne toimivia. Esimerkki:

```
/// <summary>
/// Palauttaa kahden vektorin välisen pistetulon
/// </summary>
static double pistetulo(double[] v1, double[] v2)
{
    double summa = 0;
    for (int i = 0; i < 4; i++)
    {
        summa = v1[i] + v2[i];
    }
    return summa;
}
```

Tällaista jos jää toteutukseen, niin ainakaan koodia ei ole testattu kovin täydellisesti... lisäksi sitä ei varmaan ole kovin tarkasti luettu sen jälkeen kun se on kirjoitettu. Pistetulon määritelmä ei tainnut olla “neljänsien komponenttien summa”, jonka tämä ohjelma palauttaa, laskeskeltuaan ensin turhan päiten kolmen ensimmäisen komponentin summat. Eihän tässä ole tietysti väärin kuin kaksi merkkiä:

```
summa  = v1[i] + v2[i];  // nykyinen
summa += v1[i] * v2[i];  // oikea
```

mutta ikävä kyllä ohjelmoinnissa jokainen merkki on armottoman merkitsevä. Suosittelen kehittämään ihan tätäkin taitoa, että pyrkii itse mielessään “kääntämään” ja “suorittamaan” lyhyen koodinpätkän merkki merkiltä ja toteamaan, onko se oikein. Testejä voi kirjoittaa virheiden havaitsemiseksi ja paikallistamiseksi, mutta varsinainen virheen korjaus edellyttää joka tapauksessa kykyä nähdä, missä kohtaa koodinpätkä tekee hassuja asioita.

Testailusta, ajankäytöstä ja koodin turvallisuudesta

Pointsit opiskelijalle, jonka koodissa oli mm. seuraavaa:

```
void Normalize()
{
  double len = Length();
  assert(len != 0);          // parempi kuin määrittelemätön tulos
  if (len != 0)
    (*this) *= (1.0/len);
}
```

Kun tiedetään selkeä invariantti, kuten että normalisoitava vektori ei saa olla nollavektori, niin tämän voi hyvillä mielin kirjata `assert`:illa. Kehitysvaiheessa ohjelma kaatuu assertiin, mutta bugien niittämisen jälkeen tarkistukset voidaan jättää pois käännettyä tuotanto-ohjelmaa hidastamasta. (Tai ilmeisesti ainakin Javassa assertit ovat defaulttina pois päältä ja ne pitää debug-vaiheessa laittaa erikseen päälle). Suorituskykyä vaativissa paikoissa, joissa poikkeukset yleensä tarkoittavat ohjelmointivirhettä, eivätkä hallitsemattomia ulkomaailman ilmiöitä (kuten tiedostojärjestelmä / netti / loppukäyttäjän syöte ...), `assert` on parempi menettely kuin poikkeuksen heittäminen.

Ylläolevassa muuten luotetaan siihen, että `Length()` ei ole rikki siten, että se antaisi negatiivisen pituuden. `assert(len > 0)` antaisi turvaa myös omia eikä vain luokan käyttäjän mokia vastaan.

Lisäksi tietysti tuo assertin jälkeinen `if` -tarkistus tuntuu ehkä tarpeettomalta, koska juuri `assert` on laitettu estämään nollalla jakamisesta tuleva ongelma. Jos tuonne laskuun mentäisiin nollavektorilla, niin nollat kerrottaisiin äärettömällä, mistä on tuloksena `NaN` tai mahdollisesti ohjelman kaatuminen, jos alusta määrittelee, että liukulukupoikkeuksiin pysähdytään. Jos assertia ei (tuotantokoodissa) olisi käytössä, mutta `if(len!=0)` olisi, niin mahdollisesti kaikesta testailusta huolimatta ilmenevä bugi jättäisi nollavektorit kaikessa hiljaisuudessa entiselleen, mikä voisi ehkä aiheuttaa yllätyksiä jossain muussa kohtaa koodia, missä oletetaan käytettävän yksikkövektoria... Assertin ansiosta tuo tietysti löytyy, jos syöte pystytään toistamaan “debug-laboratoriossa”. Kainosti voisin kuitenkin ehdottaa, että `assert` olisi tässä riittävä varmistus:

```
void Normalize()
{
  double len = Length();
  assert(len > 0);
  (*this) *= (1.0/len);  // luotetaan jo että len on OK.
}
```

Kun testejä tehdään, ne on hyvä sitten tehdä kunnolla... Seuraavassa koodissa oli mukana Comtestillä tehty yksikkötesti:

```
/**
 * Laskee n-ulotteisten vektorien summan.
 * @param v1 Ensimmäinen vektori.
 * @param v2 Toinen vektori.
```

```
     * @return Vektorien summa.
     */
    public float[] sum(float[] v1, float[] v2) {
        int n = v1.length;
        float[] vsum = new float[n];
        for(int i = 0; i < n; i++)
            vsum[i] = v1[i] + v2[i];
        return v2;
    }
```

Rikkihän tämä on... aika selvästikin rikki. Testi ei napannut tätä: Testissä tehdään summalasku ja määritellään oletettu tulos, mutta varsinaista tarkistamista testi ei tee. Siis testi oli rikki :). Testien tuloksiin voi luottaa vain, jos testitapaukset itsessään ovat kattavat ja OK. Hyvä tapa voi olla kirjoittaa testi ennen itse koodia, ja katsoa että se selvästi kertoo triviaalin, ei-vielä-toimivan koodin, olevan rikki. Sitten ainakin tietää, että kyseiselle ominaisuudelle tosiaan on olemassa testi, joka saa ainakin yhden virheen kiinni...

Yksikkötestit olisi hyvä aina olla, mutta toki niiden kirjoittaminen vie jonkin verran aikaa (hyvien ja hyödyllisten testien kirjoittaminen vie tosi paljonkin aikaa) joka on pois itse toteutuksen kirjoittamiselta. Mutta toisaalta hyvin tehdyillä yksikkötesteillä saa bugit kiinni varhaisessa vaiheessa, jolloin myöhempiä ominaisuuksia tehdessä voi keskittyä olennaiseen eikä joudu miettimään aiempien koodien aiheuttamia yllätyksiä. Ajankäytön keskittäminen on taiteilua ristiriitaisten tavoitteiden kanssa.

Skalaarin tyyppi float/double/int?

No eipä ainakaan `int` ... Ohjelmoinnin peruskursseilla taidetaan tehdä aika paljon harjoittelua kokonaisluvuilla, kun vuodesta toiseen tulee grafiikkakurssin ekaan demoon jotakin seuraavanlaista:

```
public class Vektori {
    public int x, y, z, n;
    //...
```

Koetapa mallintaa tällä vektorilla vaikkapa pisteet yksikköympyrän reunalla... onhan siellä (1,0,0,1), (0,1,0,1), (-1,0,0,1) ja (0,-1,0,1), mutta ei muita kokonaislukuvektoreita. Tulee aika nykiviä animaatioita :). Hauska on myös metodi:

```
public void kerroVektoriReaaliluvulla(int a) {
```

sikäli että `int` ei mallinna reaalilukua millään tasolla :). Tähän on helppo haksahtaa, ja joka vuosi näitä alkuvaiheessa näyttäisi tulevan. Nämä koodit on syytä korjata jatkoa varten siten, että käytetään jotakin kontinuumia mallintavaa lukutyyppiä. Esim.:

```
public class Vektori {
    public double x, y, z, n;
    //... jne
```

Mutta pitääkö sen nyt olla sitten `float` vai `double`? Doublella laskut on tietysti tarkempia, eikä laskujen mahdollinen hitaus floatiin nähden meitä tämän kurssin puitteissa haittaa. Valmiit grafiikkakirjastot saattavat tarjota rajapinnassaan molempia tarkkuuksia. Grafiikkalaitteiston tarkkuuteen ohjelmoija ei voi juurikaan vaikuttaa (toki standardeissa on joitakin lupauksia minimitarkkuuksista ym.).

Toisesta koodista löytyi vähän vastaavaa:

```
public class Vektori {
  private double[] luvut;
  public Vektori(int i1, int i2, int i3, int i4){
    luvut = new double[]{i1, i2, i3, i4};
  }
  // ...
```

Sisäisesti kaikki hyvin, mutta rajapinta ei ole ihan mukana. Tässä ohjelmassa oli laaja yksikkötesti, mutta testitapauksissa käytettiin kokonaislukuja. Testien tarkoitus on omalla tavallaan tukea myös suunnittelua, joten niitä kirjoittaessa varmaan olisi hyvä miettiä myös tietotyyppiä, jota testataan. Eli reaalilukumatriisia kannattaa varmaan lähteä testailemaan reaaliluvuilla :).

Liukulukujen yhtäsuuruus?

Minusta on hiukan filosofinen kysymys, onko seuraava vektorien vertailu OK:

```
bool operator ==(const Vector& y) const
{
  if(abs(x[0] - y[0]) > EPSILON || abs(x[1] - y[1]) > EPSILON ||
     abs(x[2] - y[2]) > EPSILON || abs(x[3] - y[3]) > EPSILON)
       return false;

  return true;
}
```

Tämä samaistaa vektorit, jotka ovat “kovin lähellä” toisiaan. Se on järkevää, koska laskutoimituksissa voi tulla pyöristysvirheitä. Mutta minusta operaation nimi voisi mieluummin olla sitten “approxEqual(x,y)” tmv., ja operaattori == tarkoittaisi että asiat ovat bitilleen samat. Pyöristysvirheet kun propagoituvat ja tällöin voi (äkkiseltään ymmärtääkseni) olla, että jollekin epsilonille saataisiin aikaan

tilanne, josssa `x == y` pätee, mutta `1000*x == 1000*y` ei pädekään. Minusta tämä on filosofisesti hiukan arveluttavaa. Nimi `approxEqual(x,y)` lupaisi ehkä hiukan vähemmän siitä, mitä on `approxEqual(1000*x,1000*y)` tai sitten testaisi tarvittavissa paikoissa ihan pitkän kaavan mukaan että `dist(x,y)<epsilon` jolloin oltaisiin kartalla sekä matemaattisesti että numeerisesti ...

Julkinen vai yksityinen elementti?

Aiempaa esimerkkiä mukaillen, mutta reaalilukutyyppiä käyttäen, voisi määritellä:

```
public class Vektori4 {
    public double x, y, z, w;
    // ...
```

Onko tämä OK? Ohjelmoinnin peruskursseillahan paukutetaan päihimme sitä, että attribuutit pitää olla yksityisiä ja rajapinnan takana:

```
public class Vektori4 {
    private double x, y, z, w;
    // ...
    public void setX(ix) {x=ix;}
    public double getX() {return x;}
    // ...
```

Ei maailma ole ihan niin yksinkertainen kuin peruskurssien paukutuksesta voisi päätellä. Mielestäni saantimetodit tuovat 4-ulotteisen vektorin tapauksessa aika paljon koodia, jonka tarpeellisuus on vähän niin ja näin. Mielestäni siis julkiset attribuutit `x,y,z,w` ovat tässä aivan perustellut. Matriisin osalta olenkin sitten jo eri mieltä, koska voi hyvin olla että haluan jossain vaiheessa muuttaa sisäistä rakennetta (esim. rivivektoriesitys vs. sarakevektoriesitys vs. yksi taulukko, jota indeksoidaan lineaarisesti). Eli tästä makuasiasta sanoisin, että matriisin esitysmuoto mieluummin yksityiseksi ja vektorin esitys ehkä julkiseksi, jos käyttää erillisiä reaalilukuja komponenteille. Jos taas käyttää vektorin esitysmuotona taulukkoa, niin sitten olisi mielestäni syytä pitää se taulukko piilossa, ettei olioviite pääse vaeltelemaan hallitsemattomiin paikkoihin.

Vektorin ja matriisin tallennusrakenne?

Muinoisissa ohjelmointikielissä oli mahdollista (ja joskus pakollistakin) esittää vektorit ja matriisit perusmallin taulukoina, joita operoidaan funktioilla / proseduureilla / aliohjelmilla. Tämä on periaatteessa mahdollista nykyäänkin staattisia / luokkametodeja käyttäen:

```
public static double[] laskeVektorienErotus(double[] vektoriX,
  double[] vektoriY){ /* ... */ }
```

Toinen vastaava, eri kielellä:

```
/// <summary>
/// Kertoo kaksi matriisia. Jos matriisi ei ole oikean muotoinen,
/// tulee poikkeusta. Jos mat1 on m*n matriisi, mat2 on oltava
/// n*m matriisi.
/// </summary>
/// <param name="mat1»1. matriisi</param>
/// <param name="mat2»2. matriisi</param>
/// <returns></returns>
public double[,] MatriisiKerto(double[,] mat1, double[,] mat2){
//...
```

(Tästä lineaarialgebrallinen reunahuomautus: Jos mat1 on m*n -matriisi, ei mat2 tarvitse olla n*m -matriisi vaan n*p, eli riittää että rivejä on saman verran kuin mat1:ssä sarakkeita. Tulos on sitten m*p -matriisi. Ihan kiva, että yleisiä n*m -matriisin laskuja on mietitty. Tällä kurssilla tarvitaan pääasiassa 4x4 -matriiseja, joten voi olla että tulevaa kehittelyä helpottaa, jos koodissaan kiinnittää dimensiot.)

Ehkä ihan kevyehkösti suosittelisin nimeämään ja kapseloimaan tyyppejä täsmällisemmin, koska se on nykyään mahdollista. Eli vektorit on mallia `Vektori` tai erityisesti 3d/4d -vektorit voisivat olla `Vektori3`, `Vektori4`, koska silloin ei voi mennä sekaisin erilaisiin erikoistarkoituksiin luodut tietorakenteet. Poikkeusten heittoakaan ei niin tarvitsisi miettiä, kun kääntäjästä ei ole voinut mennä läpi epäyhteensopivia laskutoimituksia... Eli jotakin seuraavanlaista saattaisin kevyesti ehdottaa:

```
class Vektori4 {
  double[] koord = new double[4];
  // ...
}
```

tai, mahdollisesti:

```
class Vektori4 {
  double x;
  double y;
  double z;
  double w;
  //...
}
```

Mutta koska asiat voidaan tehdä monin eri tavoin, en mitenkään pakota tiettyyn malliin, kunhan ratkaisut toimivat vaaditun speksin mukaisesti. Arvaukseni kuitenkin on, että yleiset mielivaltaisten dimensioiden ratkaisut vaativat enemmän koodia ja enemmän tarkkuutta virheellisten argumenttien käsittelyssä.

Matemaattisesti vektoria voi ajatella yksisarakkeisena matriisina (tai yksirivisenä; yleensä lähtökohtaisesti sarakkeena, jolloin se saadaan riviksi transponoimalla). Ohjelmakoodia ajatellen tulee kuitenkin mieleen, että matriisioperaatioissa saattaa pidemmän päälle tulla ylimääräistä pohtimista indeksien kanssa, jos vektori ei ole erillinen luokka, vaan erityistapaus matriisista.

Seuraava vektoriluokka on joka tapauksessa ohjelmakoodin mielessä redundantti:

```
public class Vektori {
  private double[][] vektori = new double[1][4];
  //...
}
```

Toteutukseen tulee tässä tavallaan tarpeeton muistiviittaus (kun kieli on esim. Java). Näitä "suorituskykyasioita" nyt oikeastaan pyydettiin olemaan miettimättä... ja rajapintahan piilottaa sisäisen toteutuksen, joten sitä voi tarpeen tullen muuttaa myöhemmin:

```
/**
  * Palauttaa annetun koordinaatin arvon
  *
  * @param koordinaatti haluttu koordinaatti (0,1,2,3)
  * @return Koordinaatin arvo
  */
 public double get(int koordinaatti) {
     return vektori[0][koordinaatti];
 }
```

Mutta ehkä tämä asia ei liity ainoastaan "suorituskykyyn" vaan myös siihen "käsitteiden selkeyteen", jota peräänkuulutettiin! Mieluummin siis olisi ihan vaan nelitaulukko kuin kaksidimensioinen rakenne.

Operaatiot samassa vai eri luokassa kuin data?

Filosofinen kysymys on, kuuluvatko vektoreita käsittelevät operaatiot samaan luokkaan tietorakenteen kanssa. Minun mielestäni kuuluvat - sehän on olio-ohjelmoinnin idea, että rakenteet ja operaatiot kulkevat käsi kädessä, ollen siis luokan ominaisuuksia. Muuten voi tulla pidemmän päälle hiukan verbaalia koodia:

```
class Operaatiot {
```

```
  // ...
  public static Vektori vektoriSumma(Vektori vektori1,
                                     Vektori vektori2) {
      Vektori summa = new Vektori(0,0,0,0);
      summa.set(0,vektori1.get(0)+vektori2.get(0));
      summa.set(1,vektori1.get(1)+vektori2.get(1));
      summa.set(2,vektori1.get(2)+vektori2.get(2));
      summa.set(3,vektori1.get(3)+vektori2.get(3));

      return summa;
  }
  // ...
}
```

Ehkä mieluummin siis Vektori-luokka sisältäisi perusoperaationsa:

```
class Vektori {
  // ...
  public static Vektori summa(Vektori a, Vektori b){
    Vektori s = new Vektori();
    for (int i=0;i<LEN;++i){
      s.val[i] = a.val[i] + b.val[i];
    }
    return s;
  }
```

Käyttäjän tarvitsisi importata vain yksi luokka, ja nimet olisivat lyhyempiä, esim. “Vektori.summa(a,b)” vs. “Operaatiot.vektoriSumma(a,b)”. Samoin koodi olisi simppelimpää, kun voi käyttää yksityisiä attribuutteja ilman saantimetodeja.

Tässä toinen esimerkki, jossa operaatiot ovat eri luokassa kuin data:

```
public class Operations {
  //sum of two Vectors
  public Vector sumV(Vector u, Vector v){
      double[] a = u.toArray();
      double[] b = v.toArray();
      return new Vector(a[0]+b[0], a[1]+b[1], a[2]+b[2], a[3]+b[3]);
  }
  // ...
```

Eri luokassa sitten itse rakenne:

```
public class Vector {
      private double x1;
      private double x2;
      private double x3;
      private double x4;
      // ...

      //vector to array
      public double[] toArray(){
              double[] x = {x1, x2, x3, x4};
              return x;
      }
      // ...
```

Tässäkin pääsisi sekä suorittava tietokone että itse ohjelmoija vähemmällä vaivalla, jos operaatiot olisivat saman luokan sisällä:

```
public class Vector {
      // ...
      public static Vector sum(Vector a, Vector b){
        return new Vector(a.x1+b.x1, a.x2+b.x2, a.x3+b.x3, a.x4+b.x4);
      }

      public static double dot(Vector a, Vector b){
        return a.x1*b.x1 + a.x2*b.x2 + a.x3*b.x3 + a.x4*b.x4;
      }
      // ...
```

Hankaluudeksi tulee ainakin Javan tapauksessa se, että Matriisi ja Vektori ovat olennaisesti eri luokkia (jos ne siis tehdään sillä tavalla, eikä ajatella Vektoria erityistapauksena Matriisista...), jolloin esim. matriisi-vektori -kertolasku on pakko tehdä jomman kumman luokan ulkopuolella ja siellä on pakko käytellä jonkinnäköistä saantimetodia sitten. Kovin eleganttia ratkaisua en keksinyt mallivastaukseenikaan, vaan jonkinlaista väkivaltaa siellä taidetaan tehdä. Jälkiviisautena ehkä voisin lähteä miettimään vektoria astetta “tyhmempänä” luokkana, jossa tosiaan olisi komponentit julkisina liukulukuina.

Oletusarvot?

Kun tehdään uusi vektori tai matriisi, niin mikä sen alkuarvo pitäisi olla? Javassa ja C#:ssa tulee kiusaus jättää muistinhallinnan nollaksi lupaamat komponentit, ja luottaa niihin. Näin taidan tehdä esimerkkikoodissanikin... C:ssä ja C++:ssa näin ei voi tehdä, koska tunnettua alkuarvoa ei ole luvattu. Nostetaan vaihtoehtona esille opiskelijan ehdotus:

```
/// <summary>
/// Constructor to create an empty 4D Vector.
/// </summary>
public Vector4D()
{
    _x1 = double.NaN;
    _x2 = double.NaN;
    _x3 = double.NaN;
    _x4 = double.NaN;
}
```

Voihan sitä näinkin ajatella. Alkuarvon halutaan tässä olevan määrittelemätön, ja liukulukustandardi antaa keinon myös olla ihan eksplisiittisesti määrittelemätön. NaN eli Not-a-number, kuten vaikkapa määrittelemättömän laskun 0/0 tulos. Niin tai näin, voisi olla kohteliasta todeta dokumenttikommentissa, millainen vektori pullahtaa ulos. Esim.:

```
/// <summary>
/// Constructor to create a 4D Vector with NaNs as
/// its components.
/// </summary>
```

Silloin käyttäjälle ei jää epäselvää, mitä hän saa, kun kutsuu parametritonta konstruktoria. Tietää sitten sijoittaa kaikkiin komponentteihin lukuja, jos haluaa vektorillaan laskea. Tämä on yksi suunnittelukysymys, että halutaanko parametrittomia konstruktoreja sallia ollenkaan, koetetaanko niistä palauttaa joku yleisesti käyttökelpoinen olio, vai pakotetaanko käyttäjä eksplikoimaan koodissaan aina, millaisen vektorin hän milloinkin haluaa. Paras ratkaisu on jossain määrin makuasia, mutta tietysti sillä on implikaationsa sitten luokan käyttöön. NaN-vektorin kaikki komponentit on asetettava luonnin jälkeen. Nollavektori on sellaisenaan käyttökelpoinen. Kumpi on parempi... vaikeampi sanoa. Kielissä, jotka joka tapauksessa tekevät automaattisesti sen työn, että asettavat muuttujat nolliksi, voi olla perusteltua olla tekemättä enää sen päälle lisätyötä...

Täysin oma kuvapuskuri vai platformin valmis?

Toistaiseksi on mielestäni OK, jos kuvapuskurina on käytetty esim. Javan, C#:n tai SDL:n valmista komponenttia. Se tukee suoraan kokeilemista kuvaruudulle tai tiedostoon piirtelyn kautta, mikä oli tavoitekin. Saa nähdä, pärjääkö ratkaisulla loppuun asti - luultavasti pärjää. Omassa toteutuksessa on tietysti se hyvä puoli, että väriarvon tallennustapa ei ole kiinnitetty esim. kokonaisluvuksi 0..255 per kanava, vaan voi käyttää vaikkapa liukulukuja.

Nimiavaruudet - hyvä

Aikoinaan ei ollut nimiavaruuksia. Aikoinaan aliohjelman nimen maksimimitta saattoi olla vaikkapa vain kahdeksan merkkiä. Aikoinaan oli tästä johtuen hiukan haastavaa yhdistellä ohjelmistokomponentteja - mikä `summa` esimerkiksi on mihinkin tyyppiin liittyvä `summa`. Nykyään meillä on nimiavaruudet, joita on ihan kiva käyttää:

```
namespace Ubergraphics {
  // ...
```

Kysymys siitä, onko `Ubergraphics` aivan paras mahdollinen nimi nimiavaruudelle, jääköön pohdittavaksi, mutta nimiavaruuksien käyttö sinänsä on hyvä juttu kielissä, jotka niitä tukevat. Esim. Javan "pakettien" nimet muodostavat nimiavaruudet, joilla operaatiot voidaan eriyttää - `java.lang.Object` on eri kuin `fi.jyu.tgp.grafiikka.Object` vaikka en tietenkään järkevänä päivänä menisi missään nimessä antamaan omalle luokalleni nimeksi `Object`. Huhhuh...

Taikavakiot - ei hyvä

Tässä koodissa on huomioitu kommenttiin eräs seikka:

```
public class Vektori {
  // ehkä vähän turhaan tämä dimensio tässä,
  // voisi luoda muodostajissa vain 4 pituisia taulukkoja
  private static final int dim = 4;
  double[] vektori = null;

  public Vektori() {
      vektori = new double[dim];
      vektori[0] = 0;
      vektori[1] = 0;
      vektori[2] = 0;
      vektori[3] = 0;
  }
  // ...
```

Havainto on hyvä sinänsä.. neljän mittaiset vektorit riittävät (varmaankin, ehkä..) melko pitkälle perusgrafiikkajutuissa. Lukuarvon lokalisointi on hyvä, koska myöhemmin olisi helppo vaihtaa dimensioksi jotakin muuta kuin 4, tai sen voisi vapauttaa käyttäjän määrittelemäksi. Nyt ei kuitenkaan ole viety asiaa ihan loppuun asti, koska koodissa on yhä taikavakioita!! Ilman muuta pitäisi olla:

```
public Vektori() {
    vektori = new double[dim];
    for (int i=0;i<dim;i++){
      vektori[i] = 0;
    }
}
```

Silloin vektorin pituuden muuttaminen myöhemmin olisi oikeastikin mahdollista, koska koodin toiminta ei enää riipu vektorin pituudesta. (Reunahuomiona: Java lupaa, että uusi vektori on täynnä nollia, joten tuossa nimenomaisessa kohdassa silmukka on sinänsä tarpeeton.) Muualla kyseisessä koodissa näköjään käytettiinkin `dim`-vakiota ihan asiallisesti.

Huonon syötteen nielaisu - paha, paha, paha!

Voihan tällaisia tarkistuksia tehdä:

```
private void setAlkio(int i, int j, double a) {
        if (i<0 || i>KOKO-1 || j<0 || j>KOKO-1) return;
        this.alkiot[i][j] = a;
}
```

Mutta kun kyseessä on oma `private` -apufunktio, niin ehkä tässä kohtaa voisin luottaa sen verran, ettei tarkistusta tehtäisi joka kerta kun asetan alkion arvon. Joka tapauksessa olisi syytä heittää poikkeus mieluummin kuin olla tekemättä mitään. Kyseessä on kuitenkin metodin käyttäjän tekemä ohjelmointivirhe, josta hän varmaan mielellään olisi tietoinen! Huonon syötteen hiljaiset nielaisut ovat aina vaarallisia paikkoja!

Tässä tapauksessa olisi ehkä mukavampaa saada vaikka ihan Javan ArrayIndexOutOfBounds -poikkeus, joka tuolta sijoituksesta luonnostaan napsahtaisi huonoilla indekseillä. Sen sijaan nyt on luvassa mahdollisesti hyvinkin outoja toiminnallisia virheitä, joista ei koskaan kuulu mitään...

Homma menee erityisen vaaralliseksi seuraavassa pätkässä:

```
/**
 *
 * @param i
 * @param j
 * @return matriisin alkion rivillä i, sarakkeessa j tai nollan, jos
 * tällaisia indeksejä ei ole olemassa.
 */
```

```
public double alkio(int i, int j) {
        if (i<0 || i>KOKO-1 || j<0 || j>KOKO-1) return 0;
        return this.alkiot[i][j];
}
```

Näin ei sitten menetellä enää 2000-luvulla, ei koskaan! Julkisen metodin käyttäjä haluaa tietää, jos hänen koodissaan on virhe. Hän ei todellakaan halua, että koodi ikään kuin toimii ja antaa vielä oikeanlaisen paluuarvon, jota voi käyttää myöhemmissä laskutoimituksissa. Muinoin saatettiin joutua palauttamaan "erikoinen" arvo huonolla syötteellä, mutta tämä oli aikaa ennen poikkeuksia. Silloinkin olisi ollut karmivaa, jos virhetilanteessa olisi saatu niinkin tavanomainen arvo kuin liukuluku `0.0`. Pikakorjaus edelliseen:

```
public double alkio(int i, int j) {
        return this.alkiot[i][j];
}
```

Nyt on hyvä: `java.lang.ArrayIndexOutOfBoundsException` pelastaa minut kaatamalla ohjelmani, jos teen ohjelmointivirheen käyttäessäni matriisiluokkaasi.

Reunahuomautus: Grafiikkasovellusten koodaus ei onneksi ole rakettitiedettä... raketeissa, sädehoitolaitteissa ym. kriittisissä sovelluksissa, joissa väärä toiminta vaarantaa ihmishenkiä, ei välttämättä ole sallittua heittää poikkeuksia eikä (toivottavasti) riitä edes pelkkä erityistapauksien testaaminen, vaan koodi täytyy todeta kertakaikkiaan oikeaksi kaikissa mahdollisissa tilanteissa. Nyt ei onneksi ole ihan näitä paineita, mutta opetellaan tekemään semi-kohteliasta koodia, eli vaikka ihan vaan sellaista joka kaatuu aina tarpeen tullen.

Maagiset paluuarvot virheilmoituksina - paha

Edelliseen liittyen, koska meillä on nykyisissä kielissä poikkeukset, niin seuraavanlaiselle kommentille ei tulisi olla mitään tarvetta:

```
* Huom! Pienin luku, jota voi sijoittaa on  -1000000000 + 1, koska
 get-metodi palauttaa tämän virheilmoituksenaan.
 TODO: keksi kivempi virheilmaisin
```

Voin kertoa, että "kivempi virheilmaisin" on poikkeuksen heittäminen. Muinoisissa kielissä, jotka eivät tue poikkeuksia (esim. C), voisin suositella ohjelman kaatamista siihen paikkaan:

```
if (!isOK) die_with_message("Index out of bounds in vec4SetElem()");
```

Tällaisissa voisi tulla kyseeseen myös `assert`:in käyttö, jossa on se hyvä puoli, että tuotantokoodista voi jättää tarkistukset pois, mutta kehitysvaiheessa on käytössä standardi menettelytapa virheiden löytämiseen.

Tässä on toinen hiukan vastaava kommentti:

```
/**
 * Sijoittaa matriisin alkiot olion omaan taulukkoon (luvut)
 * @param sijoitettavat sijoitettavat luvut sisältävä taulukko
 * @return 0 jos sijoitus onnistui, -1 jos sijoitus ei onnistunut,
 * eli sijoitettavat-taulukko on väärän kokoinen
 */
public int set(double[][] sijoitettavat) {
```

Jos haluaa tarkistella, voisi mieluummin olla:

```
public void set(double[][] sijoitettavat)
          throws DimensionMismatchException {
```

Tällä kurssilla tarkoituksena on opetella pääasiassa grafiikkaa, joten massiivisia virheentarkistuksia ei sinänsä vaadita. Hyvä puoli on, että näitä on ylipäätään tavalla tai toisella mietitty. Näin kuuluu tietenkin aina ohjelmoidessa tehdäkin! Mutta maagiset paluuarvot eivät ole oikea ratkaisu, ja silloinkaan niitä ei saisi oikeastaan pitää taikavakioina vaan nimettyinä (esim. C:ssä)

```
#define INVALID_VALUE -1000000000
/*...*/
if (value == INVALID_VALUE) processSomehow();
```

Tämä on vaaroille altista hasardihommaa, josta poikkeuksia tukevissa kielissä ei pitäisi enää joutua näkemään painajaisia. Kehitys sillä tavalla kehittyy kivasti...

C++

Tässä huomioita liittyen C++:aan. Ekassa demossa tuli kaksi palautusta kyseisellä kielellä. Toisen niistä sain nopeasti rikki, mutta toisesta en välitöntä heikkoutta löytänyt. Suosittelen katsastelemaan uutta standardia, jossa on muinoisia C++:n ongelmia kovasti korjailtu. Mm. `int huhhuh = NULL;` on historiaa, kuten koko C-kielen `NULL` -hirvitys muutenkin.

Muistivuoto - paha, paha, paha!

Muistivuoto on maailman helpoin tehdä C++:ssa, jossa ei ole automaattista roskienkeruuta. Seuraava esimerkki on adaptoitu opiskelijan koodista (joka on syytä korjata välittömästi :)):

```
/* Sisäiseksi esitysmuodoksi on päätetty muistiosoitin: */
class Matrix4
{
private:
  double **matrix;
  //...
}

/* Konstruktori varaa tilaa muistista: */
Matrix4::Matrix4(int w, int h)
{
      width = w;
      height = h;
      matrix = new double*[w];
      //...
        matrix[x] = new double[h];
      //...
}

/* Destruktori ei tee mitään: */
Matrix4::~Matrix4()
{
  // tässä oli sommiteltu deletet, mutta ne oli
  // jostain syystä kommentoitu pois.. virhe :).
  // mutta paras korjaus voisi olla poistaa kokonaan
  // tarve niille deleteille. Read on...
}

/* Lopputuloksena jokainen luotu matriisi varaa muistia, jota ei
   vapauteta koskaan. Riittävän monen matriisilaskun jälkeen ohjelma
   (ja luultavasti muutkin ohjelmat samassa tietokoneessa) kaatuvat
   siihen, ettei uutta muistia enää saa varattua. Siihen asti kaikki
   on ollut jo pitkään sairaan hidasta, kun keskusmuistin lisäksi on
   jouduttu käyttämään kovalevyllä olevaa heittovaihtomuistia eli
   swappia.  */
```

Muistivuodot ovat myrkkyä, eikä niitä yksinkertaisesti saa tehdä. Ikävä kyllä niitä on varsin helppo tehdä, joten näitä bugeja luultavasti jää vaikka kuinka varoisi. Sitäkin tärkeämpää on siis alusta lähtien kulkea turvallisinta mahdollista reittiä. Muistivuotojen välttämiseksi voisi olla nämä perussäännöt:

1. Jokaista `new`:tä täytyy vastata `delete`. (kuten C:ssä jokaista malloc()ia pitää vastata free()).
2. Itse asiassa jokainen `new` voi olla lähtökohtaisesti heikko ratkaisu käsillä olevaan tilanteeseen.

C++:n kehittäjä Bjarne Stroustrup muistuttaa joissakin opetusmateriaaleissaan: “this is not Java”... tai sarkastisemmin: “I’m a Java programmer ... and I need my garbage collector!” C++:ssa kohtuullisen kokoiset tietorakenteet lienee fiksuinta tehdä ilman new-operaattoria ja ylipäätään eksplisiittisiä pointtereita.

Esim.:

```
template<class ScalarType>
class Matrix4{
  ScalarType elem[4][4];
  //...
};
```

Tällöin destruktoria ei tarvitse edes kirjoittaa. “Hintana” on että laskuissa syntyy suorituspinoon väliaikaisia matriiseja, joita jonkin verran kopioidaan paikasta toiseen. Hinta on kuitenkin mielestäni pieni, koska muistivuotoa ei voi syntyä. Ylipäätään “hintakysymys” ei ole ihan suoraviivainen - väliaikainen tila pinosta voi olla paljon halvempaa kuin dynaamisen muistitilan anominen kekopuolelta. Myöskään ei tarvitse itse kirjoitaa kopiointioperaatiota eikä destruktoria, jos ei käytä pointtereita. Siitä lisää seuraavassa.

Lisää pointteriongelmia

Jos matriisin elementit ovat eksplisiittisen osoittimen päässä, C++ auliisti generoi automaattisesti mm. sijoitusoperaattorin, joka kopioi osoittimet, mutta ei datoja. Tällöin seuraava on mahdollista:

```
Matrix4 m1; // Uusi matriisi. osoitin elementteihin. (auts..)
Matrix4 m2; // Uusi matriisi. osoitin elementteihin. (auts..)

m2 = m1;    // m2:n aiemmat elementit vuotaneet (muistia ei
            // vapautettu missään vaiheessa) (auts!!)

            // m2:n elementit samoja kuin m1:n - siis
            // konkreettisesti samoja muistipaikkoja!!

m2.setElem(1,2,42.0); // muuttuu sekä m1 että m2 !!
```

Edellä mainittuja ongelmia ei tule, jos elementit ovat luokan osa, eivätkä siis new:llä luotuja pointtereita. Tämä on C++:mainen tapa tehdä kohtuullisen kokoisten olioiden luokkia.

Pointterit ovat elimellinen osa C++:aa, ja niiden käyttö pitää osata ja ymmärtää, mutta ohjelmien teossa pitäisi ehkä ensimmäisenä hakea ratkaisua, johon ei liity osoitinta. Eksplisiittinen osoitin ja new:n käyttö konstruktorissa implikoi saman tien, että myös vastaava destruktori sekä kopiointi- ja sijoitusoperaatiot on kirjoitettava huolellisesti.

Okei, tässä demossa nyt tehtiin myös kuvapuskuri, joka sisältää esim. laajakuvanäytöllisen pikseleitä eli esim. 1920*1080*4 = 8294400 tavua dataa. Sellainen köntsä varmaan kannattaa varata new:llä, ja sitten vaan kirjoitella huolelliset destruktorit ja kopioinnit itse, ettei tule vuotoja. (Kopioinnin voi myös kieltää eksplisiittisesti laittamalla vastaavan konstruktoriin privaatiksi; uusi C++11 tarjoaa tähän myös kauniimman näköisen ratkaisun).

“Osoittimen kaltaisia” turvallisia apuluokkia on lisätty uuteen C++11 -standardiin. Näihin `std::unique_ptr`:iin ja kavereihin kannattaa tutustua siinä vaiheessa kun tekee mieli kirjoittaa tähti `*` - jospa se tähti ei kuitenkaan olisi vieläkään perusteltu ratkaisu, vaan ehkä pointteriluokilla pärjäisi...

Melkein turvallinen pointteri?

Meillä on koodi:

```
class Matrix {
private:
  double a[16];
  // ...
public:
  /**
   * Returns the matrix elements in a linear array as row after row
   */
   const double* Get() const {
     return a;
   }
   // ...
}
```

Kaikki hyvin, kunnes käyttäjä tekee:

```
const double *elems = mat1.Get();
mat1 *= mat2;
process(elems);
```

... sikäli kuin käyttäjä huolimattomuuksissaan kuvittelee, että elems olisi välttämättä samassa tilassa kuin siinä vaiheessa, kun siihen ikäänkuin sijoitettiin. Käyttäjä ei itse voi muuttaa elementtejä (onneksi), mutta hän ei saa olettaa niitä muuttumattomiksi. Asia ei välttämättä ole täysin selvää rajapinnasta, joten ehkä `Get()` saisi olla privaatti. Lisäksi julkinen `Get()` estää muuttamasta sisäistä tallennusmuotoa myöhemmin. Hiuksien halkomista tietysti, mutta jotain ikäänkuin horjuvaa oli tasapuolisuuden nimessä pakko löytää tästä toisestakin palautetusta C++ -koodista.

Tottakai seuraava olisi erinomaisen kielletty toimenpide matriisiluokkasi käyttäjän koodissa:

```
Matrix m1;
const double *hmm;
hmm = m1.Get();
std::cout << m1.getElem(0,0) << std::endl;
free((void*)hmm); // en saisi, mutta voin...
std::cout << m1.getElem(0,0) << std::endl;  // kaatumus!
```

Jonkinmoinen probleemi on, että C++ sinänsä sallii tämän erinomaisen kielletyn toimenpiteen. (Hmm... tätä pitää reflektoida.. miksi tuo ylipäätään on teknisesti mahdollista. Näyttää ainakin omassa kääntäjässäni kyllä olevan). Ihanaa vapautta ja vauhdin hurmaa! ...Eihän me tietenkään mentäisi mitään tuollaista tekemään omassa grafiikkasoftassa, koska osataan käyttää omaa kirjastoamme sillä tavoin kuin olemme tarkoittaneet. Mutta asia on hyvä tiedostaa, ja pääsääntönä voisi olla syytä pitää ne pointterit ihan siellä omalla puolella (ts. rajapinnan takana) niin pitkälle kuin mahdollista. Mieluummin tehtäisiin tarvittaessa vaikka metodi, jolla käyttäjä saa arvoista kopiot omaan taulukkoonsa.

Sekalaisia, vähemmän olennaisia C++ -huomioita

Nuo #include “jotakin.h” voi olla hyvä laittaa vain niihin tiedostoihin, joissa niitä kutakin tarvitaan. Siis vain implementaatioon (esim. pikselikuva.cpp), jos otsikkotiedosto (esim. pikselikuva.hpp) ei sitä varsinaisesti tarvitse. C++:lla kirjoitettaessa otsikkotiedostojen nimi saisi päättyä esim. `.hpp` kun tuo `.h` tuo ehkä mieleen C-koodin.

Luokille voi olla hyvä kirjoittaa `operator<<`, jolla ne voi sitten heittää suoraan vaikka konsoliulostuloon:

```
Matrix4 m; cout << m;
```

Formaatin voi määritellä niin, että olioita voi sitten striimailla sekä sisään että ulos:

```
Matrix4 m; my_file_input_stream >> m;
```

Molemmat operaattorit on silloin tietysti kirjoitettava eksplisiittisesti... Ei tämä tietenkään välttämätöntä ole, mutta saattaa helpottaa joissain tilanteissa testailua/debuggailua tai tiedostojen käsittelyä.

Epäilyttävää:

```
const int NULL = 0;
const double PI = 3.14159265359;
```

Onneksemme C++11:een on tullut tyypitetty `nullptr`. Tuota `NULL` -tekstiä ei soisi näkevänsä tulevaisuuden ohjelmissa. Monissa C++ -ympäristöissä sattuu löytymään `M_PI` otsikkotiedostosta `cmath` mutta standardissa sitä ei taideta mainita, joten portaabelin koodin tekijä joutuu määrittelemään tuon itse tavalla tai toisella (toivottavasti oikein). Jos pelkää, ettei muista piin likiarvoa tai saattaa kirjoittaa sen vahingossa väärin, niin voipihan sen laskettaa matikkakirjaston kautta vaikkapa näin:

```
const double PI = atan(1)*4;
```

Sitten on mahdollisesti väärän lukuarvon syy ainakin vieritetty omalta kontolta kirjastolle ja laitteistolle...

C#

Itselläni on aika vähän kokemusta C#:sta, mutta ensikosketusten perusteella se tuntuisi olevan aika Java-mainen, tosin ilmeisesti se myös lainailee joitakin C++:n hyviä ominaisuuksia, kuten operaattorien kuormittamista. Joka tapauksessa tämäkin kieli sallisi liittää poikkeukseen jotakin lisätietoa poikkeuksen syystä:

```
if (input.Length != 4)
{
  throw new Exception();
}
```

Tällainen vaikuttaa aika geneeriseltä poikkeukselta. Sinänsä hyvä, että se heitetään :) mutta olisi hyvä olla mukana tieto, mitä tilannetta tämä poikkeus kuvaa.

Tässä esimerkki operaattorin kuormittamisesta, joka on siis mahdollista ainakin C#:ssa ja C++:ssa. Javassa tätä aina kaipailee:

```
public static Vec4 operator +(Vec4 vec1, Vec4 vec2)
{
```

```
    double[] data = new double[4];
    for (int i = 0; i < 4; i++)
    {
        data[i] = vec1[i] + vec2[i];
    }
    return new Vec4(data);
}
```

Koodi, josta tämä on, suorittaa hiukan turhia kopiointeja ja olioiden luonteja.. Tässä tapauksessa voisi noin periaatteessa olla ihan vaan:

```
public static Vec4 operator +(Vec4 vec1, Vec4 vec2)
{
    return new Vec4(vec1[0]+vec2[0], vec1[1]+vec2[1],
                    vec1[2]+vec2[2], vec1[3]+vec2[3]);
}
```

Voip olla (?) että C# alustaa arvot automaattisesti nollaksi, kuten Java, joten seuraavassa sijoitus suattaapi olla tarpeeton:

```
public Mat4() {
    _data = new double[4, 4];
    for (int i = 0; i < 4; i++) {
        double[] column = new double[4];
        for (int j = 0; j < 4; j++) {
            _data[i, j] = 0.0; }}}
```

Kyseinen koodi taitaa olla jonkin verran vaiheessa.. tuota `column` -taulua ei taideta käyttää paljon mihinkään. Uskoisin äkkiseltään että yo. koodin lopputulema on sama kuin tämän:

```
public Mat4() {
    _data = new double[4, 4];
}
```

Jos ymmärrän C#-kieltä oikein, niin seuraavassa koodissa try-catch on aika tarpeeton:

```
try {
    // ... Matriisitulon laskeminen ...
}
catch (Exception)
```

```
{
    throw;
}
```

Jos tässä on ideana poimia poikkeus ja heittää se sama heti eteenpäin, niin eikös tämä ole oletustoiminto joka tapauksessa myös C#:ssa... jättämällä try-catchit pois, jäisi vähemmän ylimääräistä tauhkaa lähdekoodiin, jos ette meinaa käsitellä poikkeusta paikan päällä kuitenkaan...

C-kieli

C-kieli on aina yhtä muodikas, aina yhtä vaarallinen ja kiehtovan raudanläheinen. Esimerkkikoodia C-kielisestä vastauksesta:

```
// uusi vektori
struct vector newVector(double x, double y, double z, double w)
{
  struct vector temp = {x,y,z,w};
}
```

Hups. Testasitko mitenkään? Vektori luodaan, mutta sitä ei palauteta. Fiksu kääntäjä ei generoisi tällaisesta mitään koodia, koska tämähän ei voi vaikuttaa ulkomaailmaan millään tavalla. Toisaalta lupaat palauttaa vektorin, mutta ilmeisesti sitten on unohtunut kirjoittaa `return temp`. Pieni **vinkki, joka on syytä ottaa käyttöön niin C:llä kuin muillakin kääntäjillä tehdessä**: Pyydä kääntäjältä mahdollisimman paljon varoituksia!!! Esim:

```
[nieminen@kettu demo1]$ gcc -Wall matrix.c
matrix.c: In function 'newVector':
matrix.c:12:17: warning: unused variable 'temp' [-Wunused-variable]
matrix.c:13:1: warning: control reaches end of non-void function [-Wretu
```

Ei kait siinä.. C:llä on ihan hauska koodailla. Kääntäjän varoituksia on syytä kuunnella (ja pyytää) ihan samoin kuin muillakin kielillä kirjoittaessa...

Alla olevassa kommentissa varmaan on totta, että yksinkertaisemmaksi ja lyhemmäksi tuo toteutus varmaan silmukalla tulisi:

```
// Matriixin transponaalimatriisi
// resultti annetaan parametrina
// TODO tee loopilla
```

```
void matrixTranspose(double matrix[4][4], double result[4][4])
{
  result[0][0] = matrix[0][0]; // diagonaali
  result[0][1] = matrix[1][0];
  result[0][2] = matrix[2][0];
  result[0][3] = matrix[3][0];

  result[1][0] = matrix[0][1];
  result[1][1] = matrix[1][1]; // diagonaali

  // ...
```

Kääntäjä osaa kyllä optimoida vakiokokoisen silmukan, joten ei pittäis olla kummempaa syytä kirjoittaa indeksejä auki. Kommentissa saisi ehkä lisäksi lukea, että resultti ei saa olla sama kuin syötematriisi. Kukaanhan ei estä käyttäjää tekemästä:

```
matrixTranspose(matA,matA);
```

Varmasti rikki, koska siirrellään saman matriisin elementtejä paikasta toiseen. Kommentti saisi kertoa, että tuollainen käyttö on kielletty. Eikä ehkä liikaa maksaisi `assert(matrix != result);`. Jos haluaa ultimaattisen tehokkaan transpoosin, voisi transponoida yhtä matriisia 'in-place'. Jos haluaa ultimaattisen turvallisen transpoosin, voisi ottaa sisään yhden matriisin, ja palauttaa (arvona, eli elementit automaattisesti kopioituen) uuden matriisin.

Sen sijaan samasta vastauksesta löytyi kertolasku:

```
// vektorien kertolasku AB
// paluuarvo annetaan parametrina
// TODO toteuta loopilla
void matrixMultMatrix(double a[4][4], double b[4][4], double result[4][4
{
  // Toteutus, joka sallisi että result==a tai result==b
  // ...
}
```

Tuonkin ominaisuuden voisi kertoa kommentissa, jotta kumulatiivisen matriisitulon uskaltaisi laskea "paikoillaan":

```
matrixMultMatrix(matA,matB,matA); // kumuloidaan matA:ta.
```

Toki jotakin täytyy jossain vaiheessa kopioida, kun lasketaan matriisituloa, mutta sikäli kuin kopiointi ja aputilan varaaminen tapahtuu aliohjelman sisällä, eikä siis ole käyttäjän vastuulla, voisi tämän kohteliaasti ilmoittaa rajapinnan kommentissa.

Tarkkana siellä sitten, vaikka tehdään debug-tulosteita vain kaiken varalta:

```
// vekotorien esitys konsoliin (kaiken varalta)

void PresentVector3d(float src[3]){

        printf("%f ",   src[0]);
        printf("%f ",   src[2]);
        printf("%f \n", src[3]);
}
```

Toinen komponentti `src[1]` ei tulostu ollenkaan, ja viimeinen tulostettava luku `src[3]` on mitä sattuu eli "neljäs" elementti kolmen elementin mittaisesta taulukosta. Siihen ei ole mitään kontrollia... Kääntäjä ei valita eikä varoita... C-kieli, aina niin muodikas, aina niin ihanan vaarallinen... Näillä sanoilla onnea matkaan C-koodaajille. Pitäkäähän varanne siellä..

D-kieli

Itselläni ei ole juurikaan kokemusta D:stä, mutta opiskelijan koodi näytti minusta oikein selkeältä ja robustilta. Minun mielestäni oikein kivalta näytti myös asserteilla tehty yksikkötesti:

```
unittest {
    import std.stdio;
// ... määritellään sopivat A,At,B,SUM_AB,AB ym. ...
    assert(transpoosi(A) == At);
    assert(summa(A, B) == SUM_AB);
    assert(summa(A, B) == summa(B, A));
    assert(tulo(A, I) == A);
    assert(tulo(I, A) == A);
    assert(tulo(A, B) == AB);
    assert(tulo(B, A) != AB);
// ...
    writeln("Kaikki ok.");
}
```

Go

Huomioidaan, että yksi vastaus tuli Go-kielellä, joka onpi Googlen uusi ja päällisin puolin varsin viehättävä kieli. Vastaus tosin oli kääräisty valmiiden kirjastojen päälle, vaikka tässä demossa oli tarkoituksena

tehdä vastaavat itse ... Propsit uuden ja erilaisen kielen käyttelystä, mutta demopisteitä ei kokonaan valmiin kirjaston käytöstä oikein voi jakaa.

Haskell

Haskell, tai siis ylipäätään puhdas funktio-ohjelmointi, on jonkin verran eri maailma kuin nämä "perinteiset" kielet, joilla yleensä on tullut koodattua. Itsehän opiskelen tässä samalla Funktio-ohjelmointi 2 -kurssia, kuten ilmeisesti myös ne grafiikkakurssin opiskelijat, jotka ovat ekassa demossa Haskelliin tarttuneet. Omana kurssiportfolionani yritän saada aikaan tämän grafiikkakurssin mallivastaukset (koko sarja) Haskellilla. Huhhuh... Toistaiseksi meidän parhaat neuvot tulevat funktiokurssin opettajalta eli Ville Tirroselta sekä Jarkolta, jolla on pidempi kokemus funktiomaailmasta. Tähän voisin jakaa jotakin, mitä tähän mennessä on tullut vastaan...

Otetaan tähän opiskelijan koodista muutama näpsäkkä funktionaalinen lineaarialgebraoperaatio:

```
type Matrix a = [[a]]
type Vector a = [a]
-- ...

-- Vektorien yhteenlaskuoperaatio
(.+) :: Num a => Vector a -> Vector a -> Vector a
(.+) u v = zipWith (+) u v

-- Vektorin kertominen skalaarilla
(.*) :: Num a => a -> Vector a -> Vector a
(.*) a v = map (*a) v

-- Vektoreiden sisätulo
dot :: Num a => Vector a -> Vector a -> a
dot u v = sum $ zipWith (*) u v

-- Matriisin ja vektorin tulo
(.**) :: Num a => Matrix a -> Vector a -> Vector a
(.**) m v = map (dot v) m

-- Kahden matriisin tulo
(<|>) :: Num a => Matrix a -> Matrix a -> Matrix a
(<|>) m l = map ((.**) (transpose l)) m
```

Aika paljon vähemmänhän tuohon tarvitsi kirjoittaa kuin vaikkapa Javalla :). Mutta jonkin verran enemmän tarvitsin kynää ja paperia, että uskoin että implementaatio toimii kuten sen pitää. Hauskaa puuhaa tämä sinänsä on... katsotaan, mihin tässä päädytään.

Oma toteutuksemme ei ollut kaikistellen ihan näin läpensä “funktionaalinen” ... lisäksi lähdimme hiukan varoen liikkeelle ja teimme kiinteän mittaisen vektorin. Tässä muutama operaatio meiltä:

```
-- tietotyyppi
data Vektori = V4 {-#UNPACK#-}!Float
                  {-#UNPACK#-}!Float
                  {-#UNPACK#-}!Float
                  {-#UNPACK#-}!Float

--lisäys
(.+) :: Vektori -> Vektori -> Vektori
(V4 x y z w) .+ (V4 a b c d) = V4 (x + a) (y + b) (z + c) (w + d)

--ristitulo
(><) :: Vektori -> Vektori -> Vektori
(V4 x y z 0) >< (V4 a b c 0) =
        V4 (y * c - b * z) ((-1) * x * c + a * z) (x * b - a * y) 0
```

Öhöm.. voi olla että ristitulo-operaattorin nimeksi voi jossain vaiheessa tulla jotakin muuta kuin `><` vaikka se aavistuksen verran hauska näin ensilähtöön onkin... XD ... ><.

Tuo `{-#UNPACK#-}` -pragma saatiin Tirroselta ihan ensimmäisenä vinkkinä, jolla toivottavasti vältyttäisiin tulevilta infernaalisilta suorituskykyongelmilta. Se estää arvojen pitämisen “laatikossa”, joiden käsittely kait konepellin alla on raskaampaa kuin yksittäisten arvojen. Samaa lopputarkoitusta palvelee huutomerkki `!` jolla kielletään elementtien laiskuus. Tietyllä tapaa olisi ollut ihan kiva ensin törmätä noihin infernaalisiin suorituskykyongelmiin, ja sen jälkeen “löytää” niihin tämä huutomerkkiratkaisu ja unpack-pragma, mutta saimme nämä nyt tässä vaiheessa vähän niinkuin annettuna.

Aiemmin näytetty opiskelijan vastaus, jossa vektorit ovat yleisiä listoja ja matriisit listoja listoista, on kovin kaunis, mutta voi olla että tulevaisuudessa siinäkin saattaa olla odottamassa infernaalisia suorituskykyongelmia. Tällä kurssilla toki olisi syytä olla miettimättä suorituskykyä ja keskittyä grafiikka-algoritmeihin... Mutta Haskellin kanssa meillä taitaa olla enemmän mahdollisuuksia todellisiin ’no-can-do’ -pommeihin, johtuen juuri kaikesta siitä yksinkertaisten perusrakenteiden kauneudesta. Näitähän on funkkarikurssilla nähty: mm. merkkijonojen katenointi `(++)`:lla räjähtää heti, jos liitetään perään eikä eteen.

Jännää on. Katsotaan, miten meidän itse kunkin tässä käy, kun yritetään haskelloida grafiikkaa... Positiivista on, että erillisenä ongelmana esim. seuraavan demon puurakenne on Haskellilla kovin helppo mieltää ja toteuttaa. Hyvä, että meillä on tässä kolme erillistä tekijää ja yhteinen opettaja (eli Ville) jolta voi kysyä.

Java

Java oli nähtävästi aika suosittu kieli tässä palautuksessa. Tuolla kohdassa Yleistä olleet esimerkit tulivat pääasiassa Javalla koodatuista vastauksista (Ja C#:lla, joka on hyvin samankaltainen).

Python

Itselläni on Pythonista aika kursorinen kokemuspohja. Tuomas osannee auttaa tämän kanssa enemmän. Pythonilla palautuksen tehnyt opiskelija on tehnyt mielestäni ihan hyvännäköistä koodia, jota silmäiltyäni luotan vastaajan python-taitoihin enemmän kuin omiini. Yleiseen tietouteen nostaisin pythonista sen alkeisohjelmointipedagogisesti mielenkiintoisen seikan, että siinä olioinstanssin viite viedään eksplisiittisesti kuhunkin metodiin, esim:

```
class Matrix:
  """Luokka matriisien käsittelyä varten."""
  #...
  def __mul__(self, other):
    #... (syötteiden tarkistukset ja poikkeusten heitot)
    product = Matrix(n=self.n, m=other.m)
    other = other.transpose()
    for x in range(self.n):
      for y in range(other.n):
        product[x][y] = sum([i*j for i, j in zip(self[x], other[y])])
    return product
```

Muissakin oliokielissä tapahtuu vastaava ilmiö implisiittisesti, ja “itseen” viitataan sitten nimellä `self` tai `this`. Tämä on hyvä pitää mielessä. Pythonissa se on tosiaan eksplisiittistä eli metodien ensimmäinen parametri on viite instanssiin, jolle operoidaan.

Pythonissa on paljon hyviä ominaisuuksia, joita opiskelija näyttää mukavasti käytelleen. Kieli on voimakas, mutta lyhyen kokemukseni mukaan se ei anna ohjelmoijalle paljonkaan varaa laiskuuteen tietyissä asioissa, koska dynaamisella tyypityksellä on helppo ampua jalkaan itseään ja kaveria. Kääntäjä ei ilmoita tyyppivirheistä, koska käsitettä ei ihan sellaisenaan ole olemassa. Silloin kun viimeksi kajosin Pythoniin, niin myöskään yksityisen attribuutin käsitettä ei ollut, eikä oikeastaan kiinteästi määriteltyjen attribuuttienkaan käsitettä... Mutta hätää ei ole, jos nämä asiat tiedostaa. Python luottaa siihen, että ohjelmoija tietää, mitä hän tekee. Siinä mielessä kieli on kyllä ihan C:n sukulainen.

Onnea vaan tuleviin demoihin pythonin parissa... ensimmäinen vastaus vaikutti aika robustilta tyyppitarkistuksineen ja yksikkötesteineen (jotka näköjään voi kirjoittaa “comtest-tyylisesti” metodien kommentteihin). ///