© Kiskapu Kft. Minden jog fenntartva

```
      [ 0 <1, 1, 1, 1> ]
      [ 1 <1, 1, 1, 0.7> ]
    }
    samples 10
  }
...
...
```

Az eddigi módosítások ellenére még mindig akad egy kis gond az árnyékokkal. Tegyük egy kicsit elmosódottabbá az éles árnyékokat! Többféleképpen is megoldhatjuk: alkalmazhatunk véletlenszerű zajt a részecskerendszerben a `jitter` kulcsszó megadásával. A másik lehetőségünk a felül-mintavételezés, ekkor a nagyobb fényességváltozásokat finomítjuk (az `aa_treshold` és az `aa_level` kulcsszó segítségével). A harmadik mód, amikor a teljes képre magasabb mintavételezési értéket alkalmazunk. Mivel ez utóbbi a leglassabb eljárás, először a többi módszert próbáljuk ki. A részecskerendszer meghatározásán belül általánosságban a véletlen zajt és a helyi felül-mintavételezést használjuk a következő módon:

```
...
...
    }
    samples 10
    aa_level 3
    aa_threshold 0.2
    jitter 0.1
    }
  }
...
...
```

Most már szinte tökéletes a poron áthaladó fénysugár megjelenítése, de nagyon ritkán találkozunk olyan hellyel, ahol így áll a levegő, és a por ennyire egyenletesen oszlik el a térben. Kavarjunk egy kis szellőt a `turbulence` utasítás alkalmazásával, és máris elégedettek lehetünk az eredménnyel:

```
...
...   box_mapping
    linear
    turbulence 1
    color_map {
...
...
```

Vegyük észre, hogy nem használtuk az állandó sűrűséget meghatározó `constant` sűrűségfüggvényt, hanem helyette a `linear` kulcsszó meghatározta egyenletes eloszlást adtuk meg. Ennek oka, hogy állandó sűrűségű térben nem lenne értelme a véletlenszerű változtatásoknak, a részecskesűrűség nem változna. Megjegyzendő, hogy a `turbulence` érték típusa vektor; a felhasználásával látványos hatásokat érhetünk el, ha az egyik irányban nagyobb értéket adunk meg, mint a másik két koordinátatengely mentén. Így készíthetünk például vízesést vagy más áramló rendszert.

4. kép Fény a porban…

Természetesen a por számára nemcsak fehér és szürke színeket adhatunk meg, hanem az alábbi részlet alapján akár a fehértől indulva – a képzelőerőnk szabta határokig – bármilyen színt. Fontos olyan színértékeket megadni, amelyek bizonyos színeket kiszűrnek, ezt a `rgbf` kulcsszóval tehetjük meg. Mivel azonban a részecskerendszerben a színeknek átlátszóknak is kell lenniük, a színek meghatározása során inkább a `rgbft` szót használjuk.

```
...
    color_map {
      [ 0 color rgbft <1, 0,
       ➥0, 0.5, 1.0> ]
      [ 1 color rgbft <1, 0,
       ➥0, 0.5, 0.7> ]
    }
...
...
```

Mielőtt a végére érnénk a részecskerendszerekkel való ismerkedésnek, fel kell hívnom a figyelmet néhány dologra: a részecskerendszert minden tárgyhoz a következő formában adjuk meg:

```
OBJETKUM {
  texture {
    pigment {...}
    normal {...}
    finish {...}
    halo {...}
  }
  hollow
}
```

Nem használhatók a `pigment`, `color_map`, `pigment_map`, `texture_map` és `material_map` utasításokon belül. Amennyiben többrétegű mintázatot szeretnénk használni, a részecskerendszert mindig a legalsó rétegben kell meghatároznunk, mely rétegnek természetesen átlátszónak kell lennie. Szintén ne feledkezzünk meg róla, hogy egymást átfedő tárolóobjektumok esetében a PoV-Ray az eredményt nem képes megfelelő módon kiszámítani. Ilyenkor minden tárolóobjektumot a többitől függetlenül számol ki, és az eredmény összeadódik. Az ebből származó hibák elkerülhetők, ha megfelelően nagy méretű tárolóobjektumot adunk meg.
További hiányosság, hogy a többféle színtérképpel (`color_map`) létrehozott `attenuating` típusú részecskerendszer, amelyet a felhők készítésénél tárgyaltunk, jelenleg nem használható. Amint azt a leírás elején említettem, a kamera látóterében lévő objektumoknak üregesnek kell lenniük, ezt a `hollow` kulcsszó teszi lehetővé.
Fontos megjegyeznünk, hogy a `scale` kulcsszót a megfelelő helyen kell alkalmaznunk. Amennyiben a hordozó tárgyat át szeretnénk méretezni, például akkor, amikor a részecskéket rendezetlenné (`turbulence` kulcsszó) tesszük, az átméretezést még a rendszer meghatározása előtt el kell végezni; míg ha a részecskerendszer méretét szeretnénk megváltoztatni, a `scale` utasítást a rendszer meghatározásán belül kell használnunk.
Végül az ismétlés kedvéért jegyezzük meg, hogy a rendezetlenség nincs hatással az állandó sűrűségű rendszerre, tehát a `turbulence` és a `constant` kulcsszó együttes alkalmazása nincs hatással a részecskék eloszlásváltozására.
Végül nézzük meg a korábban elkezdett tárgyat, amely most poros térben lebeg, és szabadon alkalmazzuk rá új ismereteinket!

Fábián Zoltán (dzooli@freemail.hu, dzooli@yahoo.com) 23 éves, jelenleg programozóként dolgozik. Szabadidejében szívesen kirándul, túrázik. Emellett szeret rajzolni, érdekli a 3D grafika és a Linuxszal kapcsolatban minden olyan program és programnyelv, amit még nem ismer vagy nem próbált ki.

© Kiskapu Kft. Minden jog fenntartva

Bevezetés a Tkinter használatába (2. rész)

Egy számológép elkészítése során is fedezhetünk fel új dolgokat: tudtad, hogyha véletlenül meglököd az egeredet, máris folyamatok egész sorát indíthatod el?

Tkinter-tanfolyamunk második részéhez érkeztünk. Az előző részben már láthattuk, hogyan néz ki egy egyszerű Tkinter-alkalmazás, ezúttal pedig egy kicsit bonyolultabb feladattal, egy számológépes példával folytatjuk. A „Szia Világ!”-os példa sokat elmond, amikor egy programnyelv vagy rendszer alapjaival ismerkedünk, a Tkinter viszont túlmutat ezen; és mivel terjedőben van a szokás, hogy a grafikus alkalmazások fejlesztésére szánt rendszereket egy-egy számológépes példán keresztül ismertetik meg a felhasználókkal, tegyünk így mi is. Eközben fény derül olyan titkokra, minthogy miként tudunk egy szövegbeviteli mezőt programsorból módosítani, de az is kiderül, mi minden történik olyankor, amikor látszólag nem történik semmi, csak az egerünkkel bóklászunk a képernyőn. Vágjunk bele!

Ismerkedés a számológéppel

Számológépet már mindenki látott, ezért a feladattal nagyjából tisztában vagyunk. Vegyük sorra, mi minden szükséges ahhoz, hogy az elképzeléseinket valóra váltva egy egyszerű számológép jelenjen meg a képernyőn, amely összead és kivon, továbbá a másik két alapművelettel is tisztában van. Ha a számítógép „fejével” gondolkodunk, mindjárt elakadunk, hiszen a „szegény” gép nem tudja, hogyan néz ki egy számológép, tehát pontról pontra mindent el kell neki magyaráznunk. Meg kell mondanunk például, hogy a gombok ne „csak úgy” megjelenjenek a képernyőn, hanem meg is lehessen őket nyomni; és azt is meg kell értetnünk, hogy olyankor mi történjen, ha meg is nyomjuk ezeket a gombokat. Tudnia kell, hol legyenek az egyes gombok, a többiről nem is beszélve. Nem olyan bonyolult ám ez, csak elsőre szokatlan, hiszen ezúttal olyan partnerrel akadtunk össze, aki nem biztos, hogy mindent azonnal ugyanúgy gondol, ahogyan mi elvárnánk.

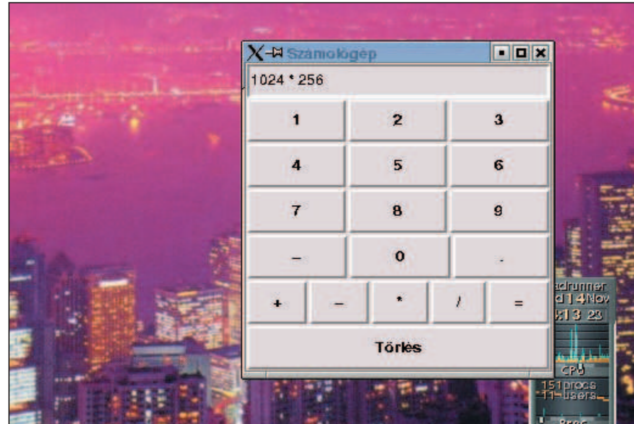
Első lépésben tehát az úgynevezett felhasználói felületet szükséges felépítenünk: ki kell találnunk, hogy milyen elemeket szeretnénk kitenni a képernyőre, és hol legyen a helyük. Esetünkben gombokra és kijelzőre lesz szükség. A következő lépésben pedig azt kell kifundálnunk, hogy az egyes gombokra kattintva mi történjék. Végül az sem árt, ha arra is felkészülünk, hogy mi legyen olyankor, ha a monitor előtt ülő felhasználó olyan dolgokat cselekszik, amelyre programunk nincs felkészülve: egyszerűen lépjünk ki valamilyen hibával a programból, ne is vegyük észre a hibát vagy tudassuk a felhasználóval, hogy rosszul csinált valamit?

Amennyiben mindezt kigondoltuk, a programtervezés nagy részén már túl is vagyunk, innentől már csak ujjgyakorlat az egész.

Az első nekifutás

Mint fentebb már említettem, számológépünkhöz két dologra lesz szükségünk: egy kijelzőre és sok-sok apró pici gombra, továbbá egy ablakra, ahol mindezt elhelyezhetjük. Az előző részben láthattuk, hogy egy gomb létrehozása csupán egy pillanatig tart: létrehozunk egy gombpéldányt és valamelyik felületkezelővel kitesszük a képernyőre. Szemléltetve:

```
w = Button(root, text= sz veg ,
           command=eljÆrÆs)
w.pack(side=LEFT, expand=YES, fill=BOTH)
```



Munkánk gyümölcse

Lambda kifejezések

Egy `lambda` kifejezés egy szokványos `def` függvénytől annyiban különbözik, hogy olyan helyeken is előfordulhat, ahol a `def` nem. Tartalma csak valamilyen egyszerű kifejezés lehet, bonyolultabb szerkezetek, mint `if` vagy `for`, nem. A `lambda` kifejezéseket elsősorban visszahívó függvényeknél használják. Visszatérési értéke mindig egy függvényobjektum, amelyet egy változóhoz rendelve hívhatunk meg. Az alábbi példa lefutása után a *b(18)* visszatérési értéke 36:

```
b = lambda a: a * 2
print b(18)
```

Minden `lambda` kifejezés saját névtérrel bír, amely nem azonos a hívó programrész névterével. Ezt a gondot egy trükkel szokták megoldani: a `lambda` értéklistájához azokat a változókat teszik hozzá, amelyekre a programrész névteréből szükségünk van. Számológépprogramunkban erre is találunk példát.

Az első sorban dől el, hogy mi lesz a gombunkra írva, és mi fog történni, ha valaki véletlenül rákattint. A w változó innentől kezdve az általunk megálmodott gombra mutat, ez azonban egyelőre csak a memóriában létezik. Ahhoz, hogy valódi, kattintható és látható gomb váljék belőle, ki kell tennünk a képernyőre. Ezt a nagyon fontos és cseppet sem elhanyagolható feladatot a Packer nevű felületkezelőre bízzuk. Ennek a `pack()` eljárásnak a dolga a gomb képernyőre történő helyezése, egészen pontosan az alkalmazásunk ablakába, lehetőség szerint a bal oldalra igazítva.

© Kiskapu Kft. Minden jog fenntartva

Számológép

```
1. from Tkinter import *
2.
3. class Calculator(Frame):
4.   def __init__(self):
5.   Frame.__init__(self)
6.   self.pack(expand=YES, fill=BOTH)
7.   self.master.title('SzÆmol gØp')
8.   self.error = 0
9.   self.ans = 0
10.
11.  display = StringVar()
12.  disp = Entry(self, relief=SUNKEN,
           textvariable=display)
13.  disp.pack(side=TOP, expand=YES,
     fill=BOTH)
14.
15.  for key in ("123", "456", "789", "-0."):
16.    fkey = frame(self, TOP)
17.    for char in key:
18.      button(fkey, char, lambda w=display,
         c=char, s=self: s.setscreen(w, c,
         1))
19.
20.  fops = frame(self, TOP)
21.  for char in "+-*/=":
22.    if char == '=':
23.      btn = button(fops, char)
24.      btn.bind('<ButtonRelease-1>',
25.        lambda e, s=self, w=display:
           s.calc(w))
26.    else:
27.      btn = button(fops, char,
28.        lambda w=display,
           c=' %s '%char, s=self:
           s.setscreen(w, c, 0))
29.
30.  clearF = frame(self, BOTTOM)
31.  button(clearF, 'T rlØs', lambda
     w=display, s=self: s.clear(w))
32.
33.  def setscreen(self, w, c, clear):
34.    if self.error == 0:
35.      if self.ans == 1:
36.        self.ans = 0
37.        if clear:
38.          w.set('')
39.      w.set(w.get() + c)
40.    else:
41.      self.clear(w)
42.      w.set(c)
43.
44.  def clear(self, w):
45.    w.set('')
46.    self.error = 0
47.
48.  def calc(self, display):
49.    try:
50.      if self.error == 0:
51.        display.set(`eval(display.get())`)
52.        self.ans = 1
53.      else:
54.        display.set("kattints a torlesre")
55.    except:
56.      display.set("HIBA")
57.      self.error = 1;
58.
59. def frame(root, side):
60.  w = Frame(root)
61.  w.pack(side=side, expand=YES, fill=BOTH)
62.  return w
63.
64. def button(root, text, command=None):
65.  w = Button(root, text=text,
       command=command)
66.  w.pack(side=LEFT, expand=YES, fill=BOTH)
67.  return w
68.
69. if __name__ == '__main__':
70.  Calculator().mainloop()
```

Egy felületkezelőnek ennél azonban jóval több dologra kell ügyelnie. Kezdetben az ablak mérete pontosan akkora kell legyen, hogy minden elem, amelyet belepakoltunk, elférjen benne és látszódjék, hacsak másképpen nem rendelkeztünk. Amennyiben az ablakunk méretét növeljük, gombjainknak több hely áll rendelkezésére, mint amennyire szükségünk van, és ilyenkor ugyancsak a felületkezelő feladata, hogy megmondja, miképpen változzon meg az ablakban található gomb vagy bármilyen más elem mérete és elhelyezkedése, hogy a felhasználók igényeinek a legjobban megfeleljen. Ebben az esetben feltételezzük, hogyha a felhasználó növeli az ablak méretét, bizonyára nagyobb gombokra van szüksége, így gombunk tulajdonságai közé vesszük, hogy a rendelkezésre álló helyet mindkét irányban töltse ki (`fill=BOTH`), és nőjön együtt az ablakkal (`expand=YES`). A `side` értékkel mondhatjuk meg, hogy a Packer az ablak melyik oldalára igazítsa a gombunkat.

A Packeren kívül két másik felületkezelő is létezik: a Grid és a Placer. A Grid az ablakot rácsokra osztja, és nekünk csak azt kell megadnunk, hogy az egyes elemek mely rácspontokba kerüljenek. Ez a felületkezelő a legjobban talán a HTML-ből ismert TABLE-höz fogható. Használata a Packerhez hasonlóan könnyű és egyszerű, és igazából csak szokás kérdése, hogy melyiket kedveljük jobban – viszont kétségkívül igaz, hogy olyan feladatok is akadnak, amelyek az egyikkel vagy a másikkal könnyebben kivitelezhetők.
A felületkezelők feketebáránya a legegyszerűbb, mégis a legtöbb odafigyelést igénylő ***Placer***, amelynek segítségével elemeinket pontosan igazítva helyezhetjük el az ablakban. Fontos, hogy egy kereten (vagy ablakon) belül csak egyetlen felületkezelő használható! Ez józan ésszel is belátható, hiszen mindannyian az elemek elhelyezéséért felelősek. Tegyük fel, hogy egy ablakot már rácsokra osztottunk, ilyenkor már nem pakolgathatunk benne ide-oda bármit!