Varga Dániel

Játék készítést támogató függvénykönyvtár

KONZULENS

Bányász Gábor tanársegéd

Automatizálási és Alkalmazott Informatikai Tanszék

2011

Tartalomjegyzék

Bevezetés....3
Célkitűzés....3
    Jól használható függvénykönyvtár....3
    Könnyen továbbfejleszthető....3
    Betekintés a használt komponensek működésébe....3
GObject....4
    Bevezetés....4
    Elnevezési konvenciók....4
    Osztályok és interfészek....5
    Memória-kezelés....5
    Eszközök....6
    Teljesítmény....9
Vala....9
Játék készítést támogató függvénykönyvtár - GtkGame....10
    A fejlesztés szakaszai....10
    Tervezési megfontolások....10
    A függvénykönyvtár megvalósítása....13
    Erőforrás menedzsment....13
    Megjelenítés....14
Fordítás és telepítés....16
    Függőségek....16
    Fordítás....17
    Telepítés....18
Példa alkalmazások....19
Továbbfejlesztési lehetőségek....22
Konklúzió....24
Irodalomjegyzék....24

Bevezetés

Játszani mindenki szeret, azonban saját játékprogramot készíteni keveseknek van lehetősége. Ennek oka, hogy bár nagyon sok grafikus könyvtár létezik, ezek használata kezdők számára nehézkes. Gyakran túl általánosak és szerteágazó funkciókkal rendelkeznek, mert nem csak játékok fejlesztésére tervezték őket. Ezért célom nem egy újabb grafikus függvénykönyvtár tervezése, hanem egy keretrendszer létrehozása, amely összefogja a játéklogikában közös elemeket és egységes interfészen teszi elérhetővé ezeket a funkciókat.

Az elkészítendő rendszerben jól elkülöníthetőnek kell lennie a megjelenítésnek és a játéklogikának. A könnyű használhatóság mellett a sebességet is fontos szempontnak tartom. Nem hagyhatjuk figyelmen kívül a manapság olyan népszerű objektum-orientált szemléletet, amely amellett, hogy közel áll az emberi gondolkodáshoz nagyon jól illeszkedik a játék fejlesztéshez.

Célkitűzés

Jól használható függvénykönyvtár

Célom egy olyan szintű játékkészítést támogató függvénykönyvtár elkészítése amely szignifikánsan könnyebbé teszi néhány klasszikus játék program fejlesztését. A sebesség mellett fontosnak tartom, az emberi közeli gondolkodást és a logikus felépítést. A TDK dolgozat keretében példa alkalmazásokon keresztül mutatom be a függvénykönyvtár használatát.

Könnyen továbbfejleszthető

A továbbfejlesztéshez elengedhetetlen, hogy az elkészült függvénykönyvtár legyen jól dokumentált, ezért arra törekszem, hogy a forráskód legalább harminc százaléka megjegyzés legyen, valamint a publikus programozási felület minden függvénye és metódusa dokumentált legyen.

Fontosnak tartom a előre tervezést, azonban egy ilyen programnál mindenképpen számítani kell arra, hogy fejlesztés alatt újabb igények merülnek fel, illetve a korábbiak változnak. A tervezés során követni fogom a már jól bevált tervezési mintákat annak érdekében, hogy a program működése más fejlesztők számára kiszámítható legyen. A megfelelő tervezési minták megválasztásával, részben biztosítható, hogy a későbbi követelmény változtatásokat könnyebben lehessen alkalmazni a már elkészült programon.

Betekintés a használt komponensek működésébe

A fejlesztés során arra törekszem, hogy ne írjak olyan komponenseket amelyek elérhetőek szabad szoftverként és megfelelnek a célnak. A komponensek kiválasztásakor figyelembe veszem a komponensek elérhetőségét a választott platformokon, illetve igyekszem jól kipróbált, széleskörben alkalmazott komponenseket választani. Amennyiben funkcionalitásban több komponens is megfelelőnek bizonyul azt választom amelyik kisebb, gyorsabb vagy amelynek kevesebb függősége van. A TDK dolgozat keretében röviden bemutatom a felhasznált függvénykönyvtárakat és segédprogramokat.

GObject

Bevezetés

A GObject [1] egy keretrendszer amely megkönnyíti az objektum-orientált programozást C nyelven. A C nyelvet nem kifejezetten erre tervezték, ennek ellenére a gyakorlatban elég jól működik. Az osztályok tulajdonképpen C struktúrák, a virtuális függvényeket függvény mutatóként valósítja meg. A GObject részben biztosítja a reflexiót, az osztályokat regisztrálni kell a keretrendszerben, ugyancsak így adhatunk az osztályokhoz property-ket és szignálokat.

Sajnos nagyon sok körítés, tucatnyit makró deklaráció szükséges minden egyes osztály létrehozásához. Ezt a feladatot érdemes automatizálni, erre készült a Gob2 segédprogram. Az osztályokat egy saját speciális nyelven, Java-hoz hasonló szintaktikával, írjuk le. A Gob2 csak egy buta kódgenerátor, a metódusok törzse teljes egészében C nyelvű, ezt nem értelmezi. Az eszköz használatával semmivel sem lesz lassabb programunk, hiszen kódot még mindig C nyelven írjuk, csupán kényelmesebben dolgozhatunk. A program áttekinthetővé válik, a körítést amely minden osztálynál nagyrészt ugyan az, nem kell megírnunk újra meg újra.

A gob2 formátum szintaktikai hibáira szerencsére szinte mindig nagyon jól érthető hibaüzenetekkel tud szolgálni. A generált forráskódban #line makrókat helyez el a C fordítónak, így a hibaüzenetek legtöbbször a gob fájl megfelelő sorára mutatnak és nem generált C kód és header fájlra, így könnyebb a hibát megtalálni. Bizonyos hibáknál amikor nem kapunk jól érthető hibaüzenetet és nem jövünk rá mi a hiba oka, érdemes a generált fájlokat is megvizsgálni.

A GObject Generator nem az egyetlen kódgenerátor amely a GObject könyvtárra épül. A Vala egy objektum-orientált programozási nyelv, amely a kódot C nyelvre fordítja le, az osztályokat pedig GObject osztályokra képezi le. Vala nagyszerűen együttműködik a nagyrészt C nyelven írt Gnome asztali környezet komponenseivel, sőt Vala nyelven készített függvénykönyvtárak azonnal használhatóak C programokból.

Elnevezési konvenciók

A C nyelvben nincsenek névterek, hogy elkerüljük a szimbólumok ütközését a publikus felület minden elemének meg kell felelnie a GObject elnevezési konvenciójának.

Típusokat és struktúrák azonosítói CamelCase stílusúak. Az első szó mindig a modul azonosítója, a további részben névtereket alakíthatunk ki. A define makrókat az elterjedt C konvenció alapján csupa nagy betűvel kell írni. A metódusok neveit csupa kis betűvel írjuk. A define makrók és metódusok nevének eleje meg kell hogy egyezzen az osztály nevével amihez tartozik, azonban CamelCase alaknak megfelelően a szavakat aláhúzásjellel kell elválasztani. A konstruktorok általában „new” kifejezésre végződnek. Többalakú konstruktor (constructor overload) a konvenció szerint a „new” után elhelyezett megkülönböztetéssel hozható létre. A kényelmesebb használat érdekében gyakran az osztály tulajdonságaihoz létrehoznak get_tulajdonsag és set_tulajdonsag alakú metódusokat, de ez nem kötelező.

Példa:

```
MylibraryBarclass              // osztály struktúra neve
MYLIBRARY_BARCLASS_GET_TYPE    // Define makró a típus azonosítónak
mylibrary_barclass_new         // Alapértelmezett konstruktor
```

```
mylibrary_barclass_new_full      // Constructor overload
mylibrary_barclass_get_width     // „width” property lekérdezése
mylibrary_barclass_do_something  // „Valami” metódus
```

Az aláhúzásjeleket mindenképpen tegyük ki a megfelelő helyen, ugyanis a GObjectIntrospection eszközt összezavarja, ha nem tartjuk be nagyon pontosan az elnevezési konvenciókat. A rosszul generált programozó felületből megtalálni, hogy az eredeti kódban hol rontottuk el az elnevezéseket nem minden esetben triviális.

Osztályok és interfészek

A Java és C# nyelvhez hasonlóan minden osztálynak legfeljebb egy szülő osztálya lehet és tetszőleges szánú interfészt megvalósíthat. Minden osztály őse a GObject amely alapvető funkciókat biztosít, köztük a referencia számlálást, amely könnyebbé teszi a memória-kezelést.

Memória-kezelés

A GObject referencia számlálással segíti a memória-kezelést. Amikor egy objektum referenciáinak száma nullára csökken automatikusan felszabadítja a keretrendszer. A referenciák számát növelni és csökkenti a következő metódusokkal lehet:

```
g_object_ref (object); // Növeli a referenciák számát.
g_object_unref (object); // Csökkenti a referenciák számát.
```

Az objektum a g_object_unref metódus hívás során szabadul fel amikor a referenciák száma eléri a nullát.

Előfordulhat, hogy nincs olyan szoros kapcsolat két objektum között, nem szeretnénk megakadályozni az objektum felszabadulását, ezért nem helyezünk el referenciát. Nem akarjuk életben tartani az objektumot, de mivel használjuk, szeretnénk tudni róla, hogy megsemmisült. Az osztályra mutató pointerből ez nem lehet eldönteni egyértelműen, ezért létezik úgynevezett „gyenge referencia” (weak reference).

A gyenge referencia tulajdonképpen egy callback függvény, amelyet az objektum megsemmisülése után hív meg a keretrendszer. Fontos kiemelni, hogy gyenge referencia esetén a callback függvény olyan pointert kap, amelyet az objektum korábban használt, de már felszabadított memóriaterületére mutat, azonban ezt sem írni sem olvasni nem szabad.

A gyenge referencia hasonlóan a referenciához elhelyezhető és meg is szüntethető:

```
// Gyenge referencia callback bekötése.
g_object_weak_ref (object, notify, data);
// Gyenge referencia megszüntetése.
g_object_weak_unref (object, notify, data);
```

Minden GObject létrehozásakor rendelkezik egy referenciával, tulajdonképpen aki létrehozta az fenntart rá egy referenciát. A G:Initially:Unowned a G:Object leszármazottja, közöttük csupán annyi a különbség, hogy az előbbi létrehozásakor nem rendelkezik referenciával, úgy is fogalmazhatunk, hogy kezdetben nincs tulajdonosa.

Eszközök

GObject Generator (GOB2)

GObject osztályok készítéséhez sok körítést kell írni, ezt végzi el a programozó helyett a GOB2. A GObject Generator egy előfordító amely megkönnyíti GObject osztályok létrehozását. Az osztályok leírásának szintaktikája hasonlít a Java nyelvhez, azonban a metódusok kódja teljes egészében C nyelvű. A GOB2 segédprogram szintaktikáját úgy tervezték, hogy szinte bármit le lehet írni vele ami szükséges a GObject osztályhoz így a generált forráskódokat nem kell utólag szerkeszteni. Az osztály generálása mellett nyújt némi segítséget a típusok ellenőrzésében is.

Osztály

Osztályt más objektum-orientált nyelvekhez hasonlóan hozhatunk létre. A sok nyelvben ha nem adunk meg ősosztályt akkor is rendelkezni fog az objektumunk valamilyen alapvető funkciókkal, tehát mindenképpen van ősosztálya. A Java nyelvben például az Object minden osztálynak az őse.

A GOB2 azonban megköveteli, hogy az osztály rendelkezzen egy ősosztállyal, ezt nem hagyhatjuk el. Ezt azért fontos kihangsúlyozni, mert a GObject lehetőséget ad olyan osztályok létrehozására amelyeknek nincs szülő osztályuk, ezeket a Vala nyelv „compact” osztálynak hívja, azonban ezt a GOB2 nem támogatja.

Ha új osztályt kívánunk létrehozni, a G:Object osztályból vagy a G:Initially:Unowned osztályból kell származtatnunk, az osztályunk már eleve rendelkezni fog minimális képességékkel, mint referencia számlálás és propertyk.

```
class Nevter:Osztaly from G:Object {
	...
}
```

Metódus

Metódust a Java nyelv szintatkájához hasonlóan kell definiálni. Minden metódus előtt specifikálni kell a láthatóságát amely lehet: private, protected vagy public. A privát metódusokat külön „-priv.h” végződésű fejléc fájlban deklarálja a GOB2 így azok nem lesznek elérhetőek más osztályból. A metódus nevéhez a GObject Generator automatikusan fűzi az elejére az elnevezési konvencióknak megfelelően az osztály nevét.

A metódusok első argumentuma a általában a self, ha elhagyjuk statikus metódust hozunk létre. A self típusát nem kell megadni a deklarációban. Az osztályok referenciáit ajánlott GOB2 szintaktikának megfelelően átvenni: a CamelCase alakban azokon a helyeken ahol aláhúzást alkalmazunk a metódus neveknél kettősponttal kell elválasztani. Ez azért szükséges, mert a GOB2 nem feltételezi, hogy minden objektum elnevezése helyesen betartja a konvenciókat. Ennek oka, hogy sok korábban írt GObject osztály referenciáját is át kell tudni venni amelyek nem tartják be teljesen a konvenciókat.

Ajánlott előírni az argumentumok ellenőrzését a „check” kulcsszóval. Osztályok referenciájánál a típus ellenőrzést írjuk elő (type), illetve ha nem engedjük meg, hogy NULL értékű legyen a mutató akkor ennek ellenőrzését (null). A relációjelekkel vizsgálhatjuk, hogy szám argumentumok megfelelnek-e az elvárásainknak.

A metódusok kódját C nyelven kell írni, azt nem értelmezi a GObject Generator. A metódusokban használhatunk néhány makrót amely megkönnyíti a munkát. A Self makró az aktuális osztály típusa, míg a SELF az aktuális osztály típusára castolásnak felel meg. Egy leszármazott osztályból a PARENT_HANDLER makróval hívhatjuk meg kényelmesen a szülő felüldefiniált (overridden)

metódusát.

Példa

```
public void metodus_nev (self,                  // Self típusát nem kell megadni
     Gtk:Button *button (check type null),  // Objektum ellenőrzése
     int i (check > 5))                     // Szám ellenőrzése
{
     …                                      // Metódus törzse nyelven
}
```

Virtuális metódus felüldefiniálása leszármazott osztályban

Metódus felüldefiniálása (override) kicsit körülményes, hiszen a GOB2 csak egy kódgenerátor, nem értelmezi a fájlokat, emiatt meg kell adnunk feleslegesnek tűnő adatokat is. Szigorúan követnünk kell a felüldefiniálandó függvény szignatúráját hiszen ezt sem ellenőrzi a fordító.

Példa:

```
override (Névtér:Szülőosztály) metódus_neve (Névtér:Szülőosztály *self, …) {
     …
}
```

Statikus metódus

Statikus az a metódus amely nem objektum példányhoz kötődik, de mégis köze van az osztályhoz. A metódus nem kötődik objektum példányhoz ezáltal válik statikussá, nem veszi át az objektum példányra mutató pointert, egyszerűen csak hagyjuk el a „self" argumentumot a paraméterlista elejéről.

```
class Nevter:Osztaly from G:Object {

     public void teszt(int i) {
          ...
     }

}
```

Statikus metódus használható szignálok kezelésére, ekkor ugyanis nem kapjuk meg automatikusan a példányra mutató pointert. Amennyiben mégis egy példány metódusához szeretnénk kötni a szignált adatként adjuk át a példányra mutató pointert. Ekkor is statikus metódust kell alkalmaznunk, mert az mutatót nem a szokásos módon első paraméterként kapjuk, az adat argumentum az utolsó, ebben kapjuk meg a példány mutatóját.

Property

A legtöbb objektumnak vannak tulajdonságai, éppen ezért néhány objektum-orientált programozási nyelvben van ennek megfelelő nyelv elem. A GObject is rendelkezik ezzel a képességgel. A property azonban más feladatot is ellát: egy objektum létrehozásakor ez az egyetlen mód, hogy adatot adjunk át az objektum konstruktorának.

Szignál

Szignálok használatakor érdemes odafigyelni arra, hogy soha ne semmisítsünk meg az objektum példányt olyan függvényben amelyet az adott objektum szignálja hívott meg. Ebben az esetben ugyanis az objektum megsemmisülése ellenére minden a szignálra feliratkozott függvény le fog futni, azonban a később hívott függvények már egy megsemmisült objektumra mutató pointert kapnak, amiből sehogy sem lehet eldönteni, hogy érvénytelen vagy sem.

Hiányosságok

A GObject Generator nem támogatja saját interfész létrehozását, így azt nekünk kell kézzel elkészíteni. A többszörös öröklés nem gyakori, így erre a funkcióra nincs gyakran szükség.

A GOB2 nem támogatja könnyített osztályok, úgynevezett Boxed Type létrehozását. A könnyített osztály nem rendelkezik referencia számlálással és a GObject egyéb képességeivel. Átadható érték szerint is, szemben a GObject osztályokkal amelyeket minden esetben referenciaként kell átadni. Definiálhatunk konstruktort és destruktort valami másoló konstruktort is, így osztály propertyben használva biztonságosan másolódik.

A GObject szignálok sokkal többre képesek, mint amennyit használni tudunk, ha GOB2-re bízzuk a szignál létrehozását. A GObject Generator nem támogatja megfelelően a szignál visszatérési értékének összegyűjtését. Cserébe a szignál létrehozása nagyon egyszerű.

Szignálok használata esetén gyakran olyan attribútumokat is kapunk a szignált kezelő függvényben amelyre nincsen szükségünk, erre természetesen figyelmeztet a fordító. A sok téves figyelmeztetéstől nem tudunk megszabadulni, mert az osztály metódusainak paramétereit a GOB2 értelmezi és hibát jelez, ha jelzést helyezünk el a GCC-nek a warning kihagyásáról. Annyit tehetünk, hogy az osztályon kívül valósítjuk meg a szignálhoz bekötött függvényt így megkerülve a GOB2 értelmezőjét.

Minden fájlnak pontosan egy osztályt kell tartalmaznia, tehát nem tehetjük meg azt, hogy a Gob2 segítségével egy külön fájlban saját felsorolt típust (enum), flag-et vagy hibát definiálunk egy közös fájlban, de osztályt nem. Ebben az esetben nekünk kell a GOB2 segítsége nélkül definiálni a közös típusokat.

Gtk-Doc

A Gtk-Doc megjegyzés formátuma kicsit eltér a Doxygen és Javadoc-ban megszokottaktól. Minden blokkban fel kell tüntetni mi a neve az elemnek amelyet dokumentálunk. Ebből kifolyólag a dokumentáció és az elemnek nem kell egymás mellett lennie, sőt akár másik fájlban is lehet.

Nyilván célszerű a metódusok leírását a metódus mellé tenni, hiszen ezzel segítjük azt aki a forráskódot olvassa, azonban időnként mégis szükséges ez a funkció: a GOB2 generálta fájlokban nem mindig tudjuk a metódus mellé írni a leírását. Gtk-Doc ajánlásnak megfelelően érdemes a kód fájlba tenni a dokumentációt. A header fájlban található dokumentációval az a probléma, hogy a dokumentáció megváltoztatása miatt megváltozik a header fájl, tehát a környezet újrafordítja az összes fájlt amiben használtuk.

GObjectIntrospection

A GObjectIntrospection egy eszköz amely képes automatikusan előállítani az illesztést számos objektum-orientált programozási nyelvhez. A támogatott nyelvek között találhatjuk a Python, Vala, Ruby, JavaScript nyelvek mellett még számos más programozási nyelvet, azonban támogatásuk jelenleg kísérleti.

A metódusok leírását ajánlott kiegészítenünk többlet információval, ez megkönnyíti a GObject

objektumaink használatát más programozási nyelvekből. Az osztály egyik metódusa pointert vesz át paraméterként, nem egyértelmű, hogy az kimeneti vagy bemeneti paraméter, egyáltalán kötelező-e átadni valamit a NULL érték is megengedett-e. Ez tehetjük egyértelművé speciális jelzésekkel.

Teljesítmény

A GObject objektum-orientált programozást tesz lehetővé C nyelven, azonban futás közben sok konverziót és ellenőrzést végez. Ez a rengeteg függvényhívás miatt minden bizonnyal lassabb mint a C++. Némi gyorsulást érhetünk el az ellenőrzések kikapcsolásával, ehhez a programot G_DISABLE_ASSERT és G_DISABLE_CHECK define makrókkal kell fordítani.

A GObject osztály példányok létrehozása igen költséges, sebesség kritikus részeknél jobb ha Boxed Typeot alkalmazunk vagy másképp oldjuk meg a problémát.

Vala

A Vala [9] egy modern objektumorientált programozás nyelv, a szintaktikáját a C# mintájára alakították ki. A Vala fordító a forrásfájlokból C kód és fejléc fájlokat készít, így pontosan olyan mintha az alkalmazás vagy függvénykönyvtár C nyelvű lenne. A Gob2 segédprogramhoz hasonlóan az osztályokat GObject osztályokra képezi le, azonban a Vala egy teljesen új nyelv nem egy buta előfordító.

A Vala nyelven írt programban használhatunk más C nyelven írt függvénykönyvtárakat, ehhez az API-t leíró úgynevezett vapi fájl szükséges, amely leírja a Vala nyelvi elemek leképezését C kódra. Vala nyelven írt programból C fordító készíti tárgykódot, ezért többnyire közvetítő réteg nélkül, közvetlenül használható más C nyelven írt függvénykönyvtár, a vapi fájl csak a leképezést írja le.

Vala nyelvből használhatjuk könnyedén a C nyelvű függvénykönyvtárakat, ez persze visszafelé is működik: mivel minden Vala kódból C kód és header fájlok készülnek, ezért egy Vala nyelven írt függvénykönyvtár ugyan úgy használható C programokban mint bármelyik másik GObject alapú C könyvtár.

A Vala nyelv szintaktikája nagyon hasonló a C# nyelvvel. Mindkét nyelv tartalmaz delegate-et, osztály property-t ezek szintaktikája is nagyon hasonlít. A C# event-nek pedig a GObject/Valában a signal a megfelelője. A C#-al ellentétben a Vala programok nem virtuális gépen futnak, ezáltal jobb teljesítmény érhető el.

Játék készítést támogató függvénykönyvtár - GtkGame

A fejlesztés szakaszai

A függvénykönyvtár elődjének fejlesztését 2009 októberében kezdtem, a célom a játék fejlesztés könnyítése volt C programozási nyelven. Kezdetben nem tudtam pontosan mit és hogyan lehetne megoldani, ezért kevés tervezéssel láttam neki a fejlesztésnek. A C nyelv ellenére objektum-orientált szemlélettel alakítottam ki a programozási felületet. A könyvtárral leegyszerűsítette az ablakos alkalmazások készítését, valamint kezdetleges segítséget nyújtott a játék logikában is. A megjelenítés akkor még hardver gyorsítás nélkül a Gtk+ DrawingArea felületi elemére közvetlen rajzolással történt. A sebesség növelése érdekében csak a változásokat rajzolta újra, ez jelentősen bonyolította a függvénykönyvtárat, azonban komolyabb játékokhoz így is használhatatlanul lassú volt. A menet közben felmerülő újabb igényeknek megfelelően nagyon sok módosításon esett át. A megfelelő tervezés és egységes konvenciók hiányában hamar karbantarthatatlanná vált.

Végül úgy döntöttem, hogy a megszerzett tapasztalatokra építve újraírom a függvénykönyvtárat. Az új változatban már a Gtk+ kapcsán megismert GObject keretrendszert alkalmaztam. A keretrendszer jó alapot adott az objektum-orientált fejlesztéshez, a korlátozások pedig biztosították a kód könnyebb karbantartását. A megjelenítést lecseréltem hardver gyorsítást alkalmazó megoldásra, így a megjelenítés jelentősen felgyorsult.

Tervezési megfontolások

A korábbi hibákból tanulva a fejlesztést aprólékos tervezés előzte meg. Összegyűjtöttem a megvalósítandó feladatokat és számba vettem az elérhető függvénykönyvtárakat amelyek segíthetnek a feladatok megoldásában. A kiválasztott komponensek függőségei lesznek a függvénykönyvtárnak, tehát anélkül nem lehet lefordítani, telepíteni és használni a GtkGame alapú játék programokat. Nem megfelelő komponensek választásával a későbbi GtkGame felhasználók munkáját teszem nehezebbé.

Csak olyan külső komponensek jöhetnek szóba amelyek maguk is szabad szoftverek. Nagyon fontos, hogy az összes nem opcionális komponens elérhető legyen a platformok széles skáláján, hiszen ez biztosítja a GtkGame portolhatóságát más platformokra. A függvénykönyvtár egyes részei, például a megjelenítés nagyon hardver és környezet függő, a könyvtár ezen részeit jól el kell különíteni annak érdekében, hogy könnyedén le lehessen cserélni más környezetnek megfelelő megoldásra. Lehetséges, hogy egyes környezetekben más függvénykönyvtárakat kell alkalmazni ugyan arra a feladatra. Fontos szempontnak tekintem olyan komponensek választását amelyek nagyobb felhasználói bázissal rendelkeznek, hiszen ezek stabilabbak és több platformon támogatottak.

Programozási nyelv, környezet kiválasztása

A játékkészítést támogató függvénykönyvtár elődje C nyelven készült, nem volt szigorúan objektum-orientált, azonban az elejétől fogva objektum orientált szemlélettel terveztem meg és implementáltam. Az objektum-orientált programozás nagyon közel áll az emberi gondolkodáshoz, ugyanakkor ez a gondolkodás jól illeszkedik a játékok fejlesztéséhez is, egy játék világa általában kényelmesen leírható objektumokkal.

A fenti szempontok alapján célszerű a fejlesztéshez objektum-orientált nyelvet választani. A korábban bemutatott GObject függvénykönyvtár lehetővé teszi az objektum-orientált programozást C nyelven. A Gtk+ és Gnome környezet nyelve a C, ebben a környezetben logikus ezt a nyelvet választani. A Govject alapú függvénykönyvtárhoz GObjectIntrospection eszközzel hozzákapcsolható több script nyelv, ami különösen hatékonnyá teszi a játék prototípus fejlesztését, illetve megkönnyíti a kezdő programozók dolgát. Ezzel olyan előnyt biztosít amelyet más nyelvet választva nem tudunk megvalósítani, illetve nem olyan hatékonyan.

A C programozási nyelv választása ellentmondásnak tűnhet azonban látnunk kell, hogy megfelelő köröltekintéssel ez a környezet megfelelő objektum-orientált alkalmazások fejlesztéséhez ugyanakkor biztosítja, hogy a sebesség kritikus részek optimális teljesítmény nyújtsanak.

Architektúra

Az osztályok és interfészek kezelését a GObject függvénykönyvtár C programozási nyelven biztosítja, erre építve hoztam létre a más játék-készítést támogató programokban már jól bevált objektum-modellt. Mindennek az alapja a játéktér amelyben az objektumok mozoghatnak.

A grafikus alkalmazások, így a játék programok is esemény vezéreltek. A játék menetét események befolyásolják, ennek megfelelően támogatott ez esemény-vezérelt program fejlesztés (event driven). A GObject környezetben az osztályok szignálokkal rendelkeznek, amely leginkább a C# nyelv event fogalmára hasonlít. Minden objektum a rá jellemző szignálokkal rendelkezik. A szignálokra feliratkoztathatunk függvényeket, amelyek sorra lefutnak a szignál bekövetkezésekor.

Nem minden játék program tétlen a felhasználói beavatkozások között, felmerül az igény beavatkozásra egy idő letelte után vagy adott időként. Elfogadhatatlan az, hogy a program folyamatosan foglalja a processzort egy ciklusban, a játék menetébe így végezze egyéb beavatkozásait. A tevékeny várakozás feleslegesen foglalja a processzort így más folyamatok nem férnek hozzá. Hordozható eszközöknél további gondot okoz a nagyobb áramfelvétel, ugyanis a legtöbb processzor kevesebb energiát fogyaszt ha nem terhelik, így energia felhasználási szempontból sem jó ez a megoldás. A processzor tehermentesítésére az ilyen tevékenységekhez időzítőt kell felhúzni, amelyek ismétlődve adott idő után futnak le.

Objektumok szerializálása

Perzisztencia vagyis az objektumok állapotának mentése és visszatöltése sok esetben hasznos. Hálózati játékoknál az adatcsomagokban valamilyen szöveges vagy bináris adatot vihetünk át, a küldő oldalon szerializálni kell az objektumokat, majd a fogadó oldalon visszaállítani. Az szerializált objektumokat elmenthetjük a merevlemezre, hogy később betöltsük és visszaállítsuk őket, így mentést készíthetünk a játék aktuális állapotáról. Bizonyos esetekben hasznos lehet az, hogy az objektum szerializált formájából meg tudjuk állapítani két objektum azonosságát.

Megvalósításához valamilyen módon szöveges vagy bináris adattá kell alakítani az objektum állapotát, majd később vissza kell tudni állítani. Objektumok állapota alatt általában az objektum struktúra adatmezőit értjük. Bizonyos mezők ideglenes vagy érdektelen adatokat tartalmazhatnak, gyakran nem akarjuk minden egyes mező értékét menteni. Néhány nyelvnél nyelvi elem a property vagyis tulajdonság, ennek az értékét is menteni kell.

A bináris formátum előnye, hogy jóval kisebb méretű, a merevlemezre mentve kevesebb helyet foglal, gyorsabban át lehet küldeni a hálózaton. A bináris formátummal szemben a szöveges formátum előnye, hogy ember számára könnyedén olvasható, ez jelentősen megkönnyíti a hibakeresést. A szöveges formátumban tárolt objektumok hatékonya tömöríthetőek, így a sebesség rovására csökkenthetjük a méretét. A formátumokra több szabvány létezik, egyes formátumokat számos nyelv támogat. A szerializáció biztosítja az átjárhatóságot nyelvek és programok között, az objektumok szerializált formában átvihetők egy másik programozási nyelven írt programba.

A C nyelv és a GLib/GObject függvénykönyvtár sem nyújt beépített támogatást szerializálására és perzisztenciára. A C nyelv az egyik legelterjedtebb nyelv, nagyon régóta alkalmazzák sikerrel, több jól kipróbált megoldást is találhatunk az adatok szerializálásra. Ennek kiválasztásában a fő szempontom az volt, hogy támogassa a GObject objektumokat. Ez nagyban megkönnyíti az integrálását a függvénykönyvtárba, és a későbbiekben a játékok fejlesztői is egyszerűbben biztosíthatják saját osztályaik szerializálását és visszaállítását. A jobb együttműködés érdekében fontos szempontnak tartottam azt, hogy a formátum szabványos legyen és minél több programozási nyelvhez elérhető legyen támogatás.

A feladatra két megfelelő könyvtárat találtam, a json-glib JSON (JavaScript Object Notation) formátumba szerializálja az objektumokat, míg a libyaml-glib a YAML formátumba. Mindkettő igen elterjedt szöveges formátum, számos programozási nyelvet támogatnak.

A json-glib előnye, hogy érett projekt, hosszú ideje stabilan használható, több alkalmazásban sikerrel használják. Több platformon, köztük a Maemo/MeeGo [7] rendszeren is elérhető. A JSON formátum nagyon egyszerű, ezáltal könnyű használni a függvénykönyvtárat.

A YAML formátum sokkal inkább alkalmas objektumok tárolására szemben a JSON formátummal amely tulajdonképpen típustalan rekordokat tárol. A libyaml-glib projekt jelenleg kísérleti állapotú, az programozási felülete nem stabil, több súlyos hibával és hiányossággal is küzd.

Végül elsősorban stabilitása miatt a json-glib függvénykönyvtár mellett döntöttem.

Programot összeállító segédprogram

Linux környezetben a legelterjedtebb eszköz a programok összeállítására a GNU Make. A program lefordításának és összeállításának menetét egy speciális nyelven úgynevezett „Makefile”-ok írják le. A Makefile nyelve deklaratív, szabályokat fogalmazhatunk meg, hogy az egyes célokat (target), hogyan érhetjük el. A make parancs feladata meghatározni, hogy mely lépéseket kell elvégezni a megadott cél elérése érdekében. Ha nem adtunk át célt a make parancsnak, akkor a Makefileban leírt legelső célt próbálja elérni, ezt általában „all”-nak szokták nevezni.

A GNU Make Előnye, hogy egy igen megbízható eszköz, régóta fejlesztik és nagyon sok projekt használja. A nagy felhasználótábornak hála sok példa található az interneten bizonyos problémák megoldására. Hátránya, hogy bizonyos dolgokat nehéz leírni make fájlokkal. A platformfüggetlen fordítás megvalósítása is igen nehézkes.

Ezeket a problémákat küszöböli ki több másik segédprogram, amelyek saját leíró nyelvvel rendelkeznek, illetve rugalmasabb sablonszerű kiegészítéseket tesznek lehetővé a „Makefile”-ban. A megoldások nagyban eltérnek, azonban a végeredmény közös: kapunk egy „Makefile”-t amit valamilyen módon egy másik eszköz generált, figyelembe véve az adott környezetet. Ezután a szokásos módon fordíthatjuk le a programot a make segédprogrammal. Ilyen megoldás például az Automake/Autotools illetve a cmake.

A hagyományos eszközök használata nehézkes, ezért a függvénykönyvtár fordításához és összeállításához SCons segédprogramot választottam. A SCons Erejét az adja, hogy a program összeállítását Python nyelven írhatjuk le. Szemléletében hasonló a Make környezettel azonban nem találtak ki egy új nyelvet, így sokkal egyszerűbben leírhatjuk azt ami hagyományos eszközökkel igen bonyolult lenne.

Szálbiztosság

A többszálú programozást megkönnyíti a szálbiztos programozási felület (API), azonban ennek biztosítása igen költséges, a szemaforok és aszinkron értesítések jelentősen lelassíthatják a programot. Az egyszerűbb játékok nem igényelnek önálló szálakat csak a játék logikában a nagyobb számításigényű feladatokat érdemes külön szálba tenni. Nem reális olyan alkalmazásokkal számolni

amelyek több különböző szálon folyamatosan különféle módokon befolyásolja a játékot. A nagyobb számítási feladatok általában csak a számítás végén, egy jól definiálható módon avatkoznak bele a játék folyásába. Ennek a szinkronizációját viszonylag egyszerű megoldani, felesleges azzal terhelni az összes publikus metódust, hogy garantáljuk a szálbiztosságot. Például sakkprogramnál amikor elkészül a mesterséges intelligencia a következő lépés kiszámításával léptetnie kell a bábut.

A szálbiztos programozási felület nem könnyíti meg a kezdő programozók dolgát, tekintve, hogy csak a bonyolultabb játékok igényelnek szálakat, és a szálak alkalmazása már eleve körültekintést igényel. A sebességet szem előtt tartva úgy döntöttem, hogy a publikus programozási felület nem lesz szálbiztos. A játék fejlesztő feladata biztosítani, hogy az alkalmazott szálak ne okozzanak hibát. A függvénykönyvtár belső megvalósításának egyes részei alkalmazhatnak szálakat a tevékenységek párhuzamosítására, ezt azonban a publikus programozási felület elfedi.

A függvénykönyvtár megvalósítása

Játéktér

Minden grafikus játék alapja a játéktér, amely nyilvántartja a benne található objektumokat, követi a mozgásukat. A játékban egyszerre több játéktér is létezhet, azonban egy kamera egyszerre csak egy játékteret követhet. A legtöbb játék véges játéktérben játszódik ezért játéktérnek megadható a kiterjedése.

Erőforrás menedzsment

A GObject könyvtár referencia számlálást biztosít minden GtkGame objektumnak, ennek megfelelően jelezni kell a referencia eldobását, ugyanis az objektum csak akkor fog ténylegesen megsemmisülni, ha a referenciái száma nullára csökkent.

Nagyon speciális esetben előfordulhat, hogy az objektumok körben egymásra hivatkoznak, ez azért probléma, mert akkor mindegyikük referenciája legalább egy, így egyik sem tud felszabadulni. Ennek elkerülésére alkalmazhatunk úgynevezett gyenge referenciát, ez azt jelenti, hogy nem tartunk fenn referenciát az objektumon, de az általunk megadott callback függvény meghívódik az objektum megsemmisülése után. Paraméterként kapunk egy mutatót amely a valaha volt objektum helyére mutat a memóriában, azonban az objektum már nem létezik, ne próbáljunk metódusokat hívni rajta, vagy elérni az objektum struktúra elemeit.

Gyakran előfordul, hogy a játéktérben több entitásnak is azonos a képe, illetve hogy a játéktér mozaikszerűen áll össze néhány képből, tehát az erőforrásokat több helyen is használjuk. A fájlból betöltött képeket szinte sohasem változtatják meg a játék programok a betöltés után. Több oka van, hogy erőforrás kezelés ezen különösnek tűnő módját választottam. Először is megfigyeléseim szerint as játékokban alkalmazott legtöbb erőforrást több helyen is használják azonban sohasem változtatják meg. Tipikusan kevés memóriát foglaló erőforrások megváltoztatása helyett keletkező új példányok viszonylag kis költséggel létrehozhatóak, sőt ha már korábban használtuk lehet, hogy már létezik is, ebben az esetben az erőforrás-kezelő nem hoz létre új objektumot.

Az erőforrások hatékony használata érdekében a GtkGame támogat egy fejlettebb erőforrás kezelést. A GtkGameResource interfészt implementáló objektumok JSON formátumba szerializálhatóak, valamit az erőforrás kezelő biztosítja azt, hogy az immutable erőforrások, ne forduljanak elő duplán a memóriában. Az úgynevezett immutable erőforrásokkal szemben a követelmény, hogy létrehozásuk után ne lehessen módosítani őket. Az ilyen objektumok tipikusan nem rendelkeznek módosító műveletekkel, az eredeti példány megváltoztatása helyett a metódusok egy új példányt adnak vissza.

Néhány alkalmazásban mégis előfordulhat olyan erőforrás amelyet igen költséges másolni, mégis

szeretnénk gyakran módosítani. Megengedett olyan erőforrások létrehozása amelyek nem immutable tulajdonságúak, azonban ezeket az objektumokat az erőforrás kezelő nem deduplikálja, nekünk kell gondoskodni arról, hogy ne legyen létezzen több azonos példány belőlük.

Az erőforrás kezelő egy szingleton objektum, a program futása során legfeljebb egy példány jön belőle létre. Feladata nyilvántartani az erőforrásokat, az erőforrások szerializált alakját összehasonlítva az azonos példányokat deduplikálja. Ha megpróbálunk létrehozni egy olyan erőforrás példányt amely már létezik a konstruktor a már létező példány referenciáját adja vissza, a fejlesztőnek nem kell tudni, sőt nem is nem tudja megállapítani, hogy létrejött-e példány vagy egy korábbit kapott vissza.

Megjelenítés

Minden mai játék programnak szüksége van grafikus felületre. A felület két részből áll: a kezelőfelület, és a játéktér. Jelenleg az egyetlen támogatott kezelőfelület a Gtk+ ablak amelyet alapértelmezetten gombok és menüpontok nélkül jön létre. A megjelenítés és a játék logika interakciója jól definiált interfészen keresztül zajlik, tehát a függvénykönyvtár megjelenítési rétege egyszerűen bővíthető újabb megjelenítőkkel.

Kezelőfelület

A játéktér mellett gyakran szükséges egy felhasználóbarát kezelőfelület, a GtkGame függvénykönyvtár lehetővé teszi Gtk+ ablak létrehozását. A Gtk+ vezérlőkkel tetszőlegesen bonyolult kezelőfelület alakítható ki egyszerűen a Glade Editor grafikus felület szerkesztővel. A felületet GtkBuilder XML formátumban kell elkészíteni, a korábbi, mára elavult libglade nem támogatott.

Ablak létrehozható Clutter [2] grafikus függvénykönyvtárral, ebben az esetben az ablak teljes tartalma a játéktér, nincs lehetőség kezelőfelület kialakítására, ez a feladat teljes mértékben a fejlesztőre hárul. Hátránya továbbá, hogy jelenleg nincs lehetőség az ablak több nézetre osztására. A Clutter/COGL felület ezért elsősorban érintőképernyős táblagépeken vagy telefonokon ajánlott, ahol a kezelőfelület egyébként is minimális a kis képernyő miatt.

A fenti problémák kiküszöbölhetőek Clutter-Gtk vagy GtkGLExt alkalmazásával. Mindkét megoldás GtkDrawingArea felületi elem képességeit bővíti. A program felületén több GtkDrawingArea vezérlővel több nézet is kialakítható, így létrehozhatunk osztott képernyős megjelenítést, vagy kis térképet amely segíti a játékost a tájékozódásban.

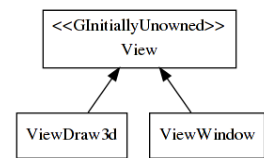
A GtkGame hardver gyorsítással jeleníti meg a játékot. A beépített grafikus elemek azonos módon jelennek meg az összes támogatott megjelenítőn, a GtkGame elfedi a rajzolási megoldások különbségeit. A tényleges rajzolás GtkGLExt esetén a szabványos OpenGL függvényekkel történik, míg a Clutter-Gtk könyvtárat alkalmazva a COGL elfedi az OpenGL és OpenGL CE verzióinak különbségeit. Saját grafikus elem készítésekor ajánlott minkét implementációt elkészíteni a kód hordozhatósága érdekében.

Nokia Maemo [7] rendszeren a GtkGLExt támogatás hiánya, illetve a Gtk+ nagyon régi verziója miatt csak a Clutter/COGL és a Clutter-Gtk felület támogatott.

Játéktér leképezése a megjelenítőre

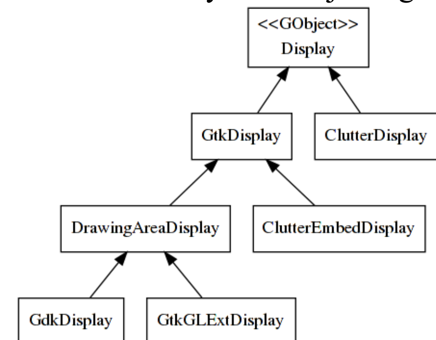
A megjelenítés a könnyebb bővíthetőség miatt két részre oszlik, az egyik rész a világ hardver független leképezése, míg a másik rész a tényleges rajzolással foglalkozik.

View



A View a megjelenítés hardver független része. Betekintést ad a játéktér egy adott területére, nyilvántartja a látható objektumok mozgását. A nézet ablak mozgatható, bejárhatjuk vele a teljes játékteret. A kamera kezelése és a látható objektumok meghatározása is jelentősen különbözik a két- és háromdimenziós megjelenítésben. Kétdimenziós megjelenítésnél a kamera [3] egy téglalap alakú ablak amely kisebb vagy azonos méretű a játéktérrel, míg háromdimenziós megjelenítésnél egy kamera amely képes tetszőleges irányba fordulni. A különböző részeket a View két leszármazott osztálya valósítja meg.

Display



A Display csak rajzolás operációs rendszer és hardver függő részleteivel foglalkozik. A tényleges rajzolással foglalkozó osztályok ősosztálya a Display, származtatással minden megjelenítőre külön implementáció készíthető.

Bővítési lehetőségek

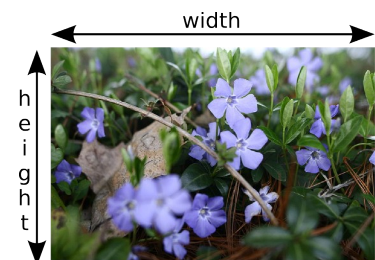
A GtkGame könnyedén bővíthető újabb megjelenítővel, általában elegendő a Display osztály származtatásával létrehozni az új megjelenítőt. Amennyiben használni szeretnénk a beépített grafikus elemeket az új megjelenítővel azokat is származtatni kell és implementálni a rajzolást az új megjelenítőre.

A későbbiekben szeretném a megjelenítési réteget bővíteni karakteres módú megjelenítéssel. Elvileg a játékteret könnyedén megjeleníthetjük majd karakteres módban. A karakteres megjelenítés nagyban eltér a grafikus megjelenítéstől, ezért nyilván nem lehet automatikusan bármelyik grafikus játék program jól használható karakteres képernyőn, ehhez a játék fejlesztő közreműködése is szükséges. Terveim szerint a képek automatikusan karakteressé alakulnak majd az aalib függvénykönyvtár segítségével, azonban a megfelelő megjelenítéshez képeket érdemes karakteresen is megrajzolni egy ilyen játékhoz. A karakteres felület kialakítása semmivel sem lesz bonyolultabb mint két- és háromdimenziós megjelenítés egyidejű támogatása, nyilván a háromdimenziós modellek automatikus kétdimenziós leképezése sem fog megfelelően illeszkedni a játék többi eleméhez.

Grafikus primitívek

Számos alapvető grafikus elem rendelkezésre áll a játékok fejlesztéséhez. Az összes beépített grafikus primitív támogatja a beépített megjelenítőket, azonban előfordulhat, hogy a későbbiekben megjelennek olyan elemek amelyek csak bizonyos megjelenítőket támogatnak. Minden grafikus elemnek van szélessége és magassága, azonban a mérete nem feltétlenül változtatható meg közvetlenül.

Image



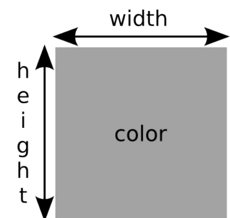
Játékokban leggyakrabban alkalmazott elem a kép, a GtkGame környezetben az Image erőforrás a kép megfelelője. A kép a konstruktornak átadott fájlnévvel hozható létre. Az erőforrás kezelő gondoskodik arról, hogy az azonos képeket csak egyszer töltsük be, ezért azonos fájlnévvel tetszőleges példányban létrehozható az Image erőforrás.

Úgynevezett „Team Color” megvalósításához bizonyos színű képpontok színezhetőek a képen a megadott színnel. A képpontok kiválasztására két mód áll rendelkezésre: a GRAY és a WESNOTH.

Az előbbiben csak a pontosan szürke árnyalatú (piros=zöld=fekete) képpontokat színezi. A WESNOTH módban a Battle For Wesnoth játékban alkalmazott színezésnek [8] megfelelően színezi ki a képet.

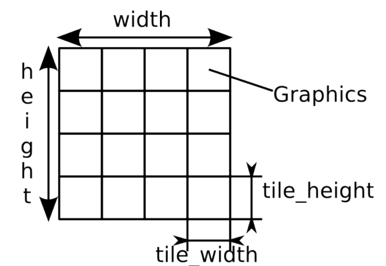
A képek úgynevezett immutable erőforrások, nem változtathatóak meg a létrehozásuk után, a fájlnevet és a színezést létrehozáskor kell megadni.

Filled



Filled grafikus elem egy színnel kitöltött téglalap. A legtöbb grafikus elemmel ellentétben a mérete lehet végtelen. Egyaránt alkalmas egyszínű háttérnek valamint, kombinálva Tile vagy Layermap elemmel bonyolultabb struktúra kialakítására. A kitöltésre használt szín alfa komponensével félig átlátszó téglalap is képezhető. A tényleges megvalósításban a rajzolás GL_QUADS primitívvel történik, míg Clutter/COGL [2] esetén cogl_rectangle metódussal.

Tile

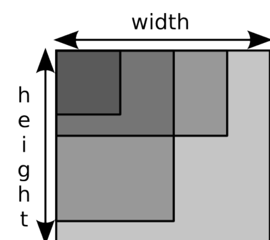


A Tile más grafikus elemekből felépülő grafikus elem. A grafikus elemek egy homogén táblázatban helyezkednek el, vagyis a táblázat minden cellája azonos méretű. A Tile tényleges méretét a „csempék” mérete és a száma határozza meg.

Egymásba ágyazott Tile elemekkel kialakíthatóak különféle struktúrák. A Tile mutable grafikus elem, ez azt jelenti, hogy változtatása gyors, azonban az erőforrás kezelő nem akadályozza meg az azonos példányok létrehozását.

Tile létrehozható egy másik Tile elem alapján, ekkor tulajdonképpen lemásolódik a tartalma. A Tile elem, a többi grafikus elemhez hasonlóan, támogatja az adat mentését JSON formátumba.

Layermap



A Layermap más grafikus elemekből felépülő grafikus elem. A tartalmazott praktikusan azonos méretű elemeket egymáson helyezi el, az utoljára hozzáadott eltakarja a korábban hozzáadott elemeket. A Layermap mutable grafikus elem, tehát az erőforrás kezelő nem akadályozza meg az azonos Layermap objektumok létrehozását.

Jól használható réteges háttérnek, alkalmas más grafikus elemekkel kombinálva összetett struktúrák kialakítására.

Line

A Line grafikus elem egyszerű törött vonal, tetszőleges számú pont adható hozzá. A color tulajdonsággal állítható a rajzolási szín.

Fordítás és telepítés

Függőségek

A GtkGame nagy mértékben épít más függvénykönyvtárak funkcióira, azonban a ezeket rugalmasan kezeli, a legtöbb függőség opcionális. Néhány könyvtárak több verziója is támogatott, korábbi rendszereken így Maemo mostanra igen elavult környezetében is lefordítható. A C forrásfájlok nagy

részét külső eszközök állítják elő, ezek használata azonban korábbi rendszereken nehézkes, például nem érhető el Maemo csomagként a Gob2 kódgeneráló, a Vala pedig igen elavult. A függőségeik pedig tovább bonyolítják a telepítést. A kiadott GtkGame forráscsomag tartalmazza az összes generált C kód és header fájlt, a kódgeneráló segédprogramok csak a GtkGame fejlesztéséhez szükségesek.

A GtkGame több megjelenítőt is támogat, ajánlott a környezethez legjobban megfelelőt választani. A Clutter/COGL függvénykönyvtár jól támogatott a Nokia Maemo/MeeGo platformján míg a GtkGLExt inkább az asztali számítógépen előnyös.

A fordítást és összeállítást a SCons segédprogram irányítja, amely működéséhez Python értelmező szükséges. A legrégebbi Python 2.3 verzió amely még elegendő a fordításhoz, ezt a verziót tartalmazza a Nokia Maemo [7] környezete alapértelmezetten.

Több Linux disztribúció szétbontja a csomagokat több részre, a GtkGame programok fordításához mindenképpen szükséges az adott függőségek fejlesztői csomagja, ugyanis ez tartalmazza a C header fájlokat.

Csak fordításidejű függőségek:

- Python 2.3
- SCons-2.0.1
- Gob-2.17 (opcionális)
- Vala-0.14 (opcionális)

Futásidejű függőségek:

- GLib-2.12
- Json-GLib-0.10.4
- Gtk+-2.10 (opcionális, Gtk+-2.22 a GtkBuilder támogatáshoz)
- Clutter-0.8 (opcionális)
- Clutter-Gtk-0.8 (opcionális)
- GtkGLExt-1.2.0 (opcionális)

Fordítás

A függvénykönyvtár fordítását és összeállítását SCons segédprogrammal oldottam meg. A SCons egy szoftver összeállító eszköz, a klasszikus make parancs egy alternatívája. Erejét az adja, hogy a program összeállítását Python nyelven írhatjuk le, így sokkal egyszerűbben leírhatjuk azt ami hagyományos eszközökkel igen bonyolult lenne.

A fordítást a forráskód gyökér könyvtárban kiadott scons paranccsal indíthatjuk el. Az elérhető függőségek számbavétele automatikus, a folyamatba kapcsolókkal avatkozhatunk be.

--disable-gtkglext

GtkGLExt támogatás letiltása, nem kerülnek lefordításra azok a részek amelyek használják ezt a könyvtárat.

--disable-gtkclutter

GtkClutter támogatás letiltása, nem kerülnek lefordításra azok a részek amelyek használják ezt a könyvtárat.

--disable-clutter

Clutter támogatás letiltása, nem kerülnek lefordításra azok a részek amelyek használják ezt a könyvtárat.

--disable-vala-tools

Vala fordító és vapi generálás letiltása, ebben az esetben a forráscsomagban található generált fájlokat fogja használni a GtkGame.

--disable-gob2

Gob2 előfordító letiltása, ebben az esetben a forráscsomagban található generált forráskódokat fogja használni a GtkGame.

--install-prefix=PREFIX

Telepítési gyökérkönyvtár megadása. Alapértelmezetten /usr Linux rendszereken.

--vala-save-temps

A Vala fordító tartsa meg az ideiglenesen léterehozott C fájlokat.

--build-examples

Példa programok fordításának letiltása, illetve engedélyezése.

--build-testcases

Tesztesetek fordításának tiltása, illetve engedélyezése.

--help

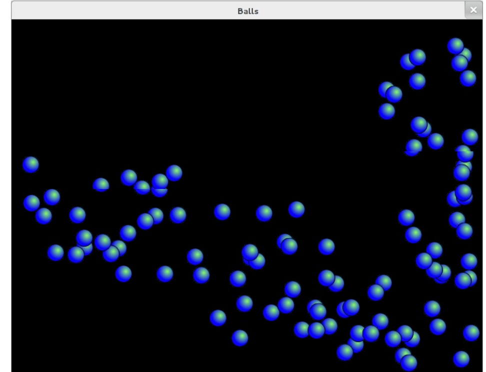
Az opciók teljes listájához listája.

Telepítés

A függvénykönyvtár telepítését szintén az SCons eszköz végzi. A telepítő elhelyezi a header fájlokat és a dinamikus könyvtárakat a nekik megfelelő helyen, így a fejlesztői környezet könnyedén megtalálja a szükséges függőségeket. Játékok fejlesztéséhez ajánlott a GtkGame könyvtárat telepíteni. A telepítést a „scons install” parancs hajtja végre. A telepítési könyvtárat az „--install-prefix” opció segítségével jelölhetjük ki, ez Linux környezetben alapértelmezetten a /usr könyvtár.

Példa alkalmazások

Az examples könyvtárban a különböző programozási nyelveken írt példaprogramok amelyek a függvénykönyvtár használatát mutatják be. Az egyes támogatott programozási nyelvek példa programjai külön alkönyvtárakban találhatóak. A legtöbb példa nem játszható játék, csak a könyvtár képességeit és használatát mutatja be a lehető legegyszerűbben. Néhány összetettebb példa is itt kapott helyet.

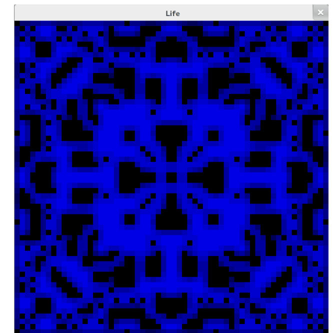


Balls

Használt képességek:

- Egyszerű ablak
- Image
- Játéktér elhagyás esemény

Az ablak széléről visszapattanó labdák az egyik legegyszerűbb példa program. A játéktér pontosan az ablakkal megegyező méretű, követi az ablak átméretezését is. Kezdetben labdák sebessége és iránya véletlenszerű. A labda a játéktér szélét elérve játéktér elhagyás eseményt okoz, az esemény hatására a megfelelő koordináta tengely menti sebesség ellentettjére változik, így tulajdonképpen visszapattan a játéktér széléről.



Life

Használt képességek:

- Egyszerű ablak
- Tile
- Filled
- Időzítő
- Egér

A Life példa program egy egyszerű életjáték, hasonlít a klasszikus life játékra azonban a szabályai egyszerűbbek. A játéktér négyzetes cellákra oszlik, minden cellában legfeljebb egy élőlény élhet. A következő generáció élőlényei azokban a cellákban maradhatnak életben vagy születhetnek ahol a szomszédos élőlények száma legfeljebb hét, de legalább három. A programban a kék mezők jelölik az élőlényeket a fekete mezőkben pedig nincs élet. Az egyszerű szabályok ellenére igen változatos mintákat kapunk.

Az ablakban egyetlen nagyméretű Tile grafikus elem található, cellái különböző színű Filled elemeket tartalmaznak. A generáció váltást az időzítő indítja el másodpercenként, ekkor egy másik Tile elemre kerül az új generáció térképe. Végül a két Tile helyet cserél.

Minesweeper

Használt képességek:

- Egyszerű ablak
- Egér
- Image
- Tile
- LayerMap

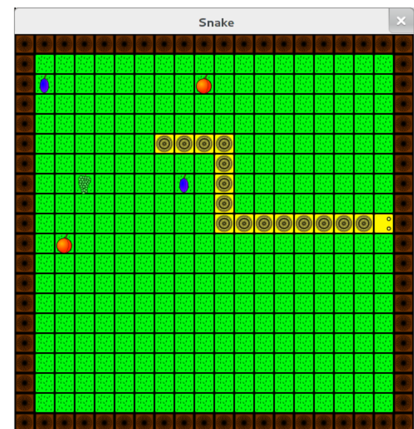
Egyszerű hagyományos aknakereső, a játék célja, hogy felfedjük az összes mezőt amelyen nem található akna.

A játéktéren nincsenek objektumok a megjelenítéshez két réteg Tile grafikus elemet alkalmaz. Az alsó réteg az aknákat és számokat tartalmazza, a felső réteg pedig kezdetben teljesen elfedi a játékteret, a felhasználó egyesével fedheti fel az alattuk található területet.

Az aknákat a játék az első kattintás után helyezi el így nem fordulhat elő, hogy azonnal az első lépésben aknára lépünk.

Jobb gombbal zászlót helyezhetünk el a cellán. Hogy a játék érintőképernyőn is játszható legyen, ha hosszabb ideig tartjuk lenyomva az egérgombot akkor a cella felfedése helyett zászlót helyez el a cellán.

Snake

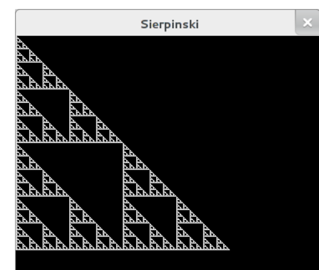


Használt képességek:

- Egyszerű ablak
- Egér
- Billentyűzet
- Image
- Tile
- LayerMap

A Snake példa program egy hagyományos kígyós játék. A játékosnak feladata a kigyóval összezgyűjteni a megjelenő gyümölcsöket. Minden újabb falattal egyre hosszabb lesz a kígyó és egyre nehezebb úgy irányítani, hogy ne ütközzön saját magába, ami a játékos vesztét okozza. A program támogatja az egeres irányítást is, hogy érintőképernyőn is kényelmesen játszható legyen.

Sierpinski



Használt képességek:

- Egyszerű ablak
- Tile

Tile grafikus elemek egymásba ágyazásával kirajzol egy Sierpinski háromszöget. Ez a példaprogram bemutatja a Tile grafikus elemek ügyes használatát, a megfelelő kialakítással spórolhatunk a grafikus elemekkel.

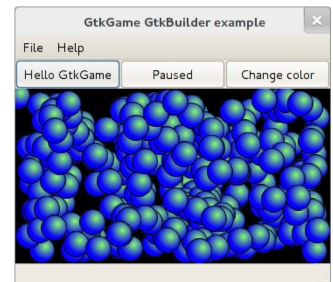
Snowfall

Használt képességek:

- Egyszerű ablak
- Image
- Háttérkép
- Játéktér elhagyása esemény
- Időzítés

Egyszerű hóesést utánzó alkalmazás. A hópelyhek véletlenszerűen mozognak, az irányukat és sebességüket fél másodpercenként változtatja egy időzítő. A játéktér mérete megegyezik a képernyő méretével. A játékteret a képernyő szélén elhagyó hópelyhek átkerülnek a játéktér másik oldalára, míg a játéktér alját elérő hópelyhek ismét a képernyő tetejére kerülnek.

GtkUI

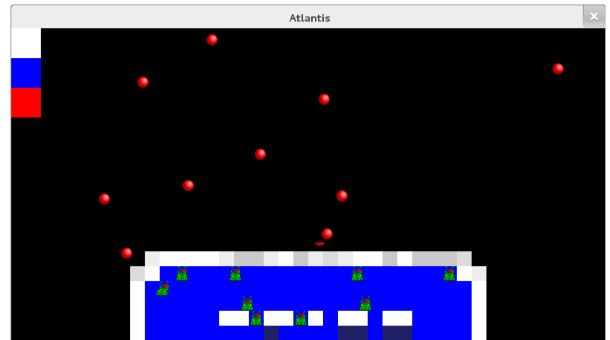


Használt képességek:

- GtkBuilder ablak, menüvel, gombokkal
- Image
- Játéktér elhagyása

Nagyon egyszerű GtkBuilder példaprogram. A játéktérben labdák pattognak a Ball példához hasonlóan. A játék megállítható a Running/Pause gombbal. A háttér színe véletlenszerűen megváltozik a Change color gombbal. A help menüből elérhető a névjegy.

Atlantis

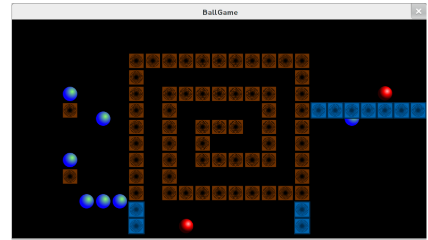


Használt képességek:

- Egyszerű ablak
- Tile
- Image
- Egér
- Időzítő
- A* útvonalkereső

Az Atlantis egy kreatív stratégiai játék. A játékos célja, hogy megvédje bázisát a bombázástól. A falak és utak építését és karbantartását automata robotok végzik. A felhasználó tervet ad a bázis építésére, a terv módosításával ösztönzi a robotokat hatékonyabb védelem kialakítására, közvetlenül nem avatkozhat bele a robotok munkájába. Az unatkozó robotok folyamatosan újabb munkát keresnek, a célterületre útvonalkeresővel találnak utat. A robotok nem veszik figyelembe a bombázást, ezért ügyelni kell arra, hogy ne merészkedjenek védtelen területre.

Ballgame



Használt képességek:

- Egyszerű ablak
- Tile
- Image

A Ballgame csak a Nokia N900 telefonon játszható, mert a játék csak a telefon gyorsulásérzékelőjével irányítható. A játékban labdák gurulnak a telefon dőlésszögének megfelelően, a játékos feladata a labdák célba juttatása. A labdák színe különböző, érdekesebbé teszi a játékot, hogy a labda vele azonos színű falon áthaladhat, míg a többi visszapattan. A valósághű mozgást az Ode fizikai szimulációs függvénykönyvtárral biztosítja.

Lords

Használt képességek:

- GtkBuilder ablak
- Tile
- Image
- Kép színezése
- Egér

A Lords nem játszható példa játék, csak bemutatja egy egyszerű körökre osztott stratégiai játék készítését. Az egységeket kattintással jelölhetjük ki, a kijelölt egységet üres területre kattitással mozgathatjuk. Bemutatja a GLib IniFileRead használatát, az egységek adatait ini szerű fájlból tölti be.

Továbbfejlesztési lehetőségek

A GtkGame jelenleg csak csak a legszükségesebb funkciókat támogatja, sok továbbfejlesztési lehetőség van. A TDK dolgozattal nem ér véget a fejlesztés, a későbbiekben folytatom a GtkGame funkcióinak bővítését. A következő pontokra fogok koncentrálni a jövőben:

Hálózati játékok támogatása

Ma az Internet korában nagyon népszerűek a hálózaton játszható játék programok. Nagyban megkönnyít a fejlesztők dolgát, ha az objektumok változásait képes a könyvtár maga szinkronban tartani. Jelenleg a hálózati támogatás nem használható, kísérleti állapotú, az üzeneteket JSON formátumban küldi át.

Csak immutable grafikus objektumok

A GtkGame könyvtár számos funkciójának működését is egyszerűsíti az, ha a grafikus objektumokat nem lehet megváltoztatni. Néhány elemnél igen lassú lehet a módosítás helyett újabb példányt csinálni, azonban megfelelő adatszerkezettel ez javítható. Az esetek többségében amikor módosítunk grafikus elemeket nem megfelelően találtuk ki a játék program működését.

GObjectIntrospection támogatás

Játékok fejlesztése C nyelven igen nehézkes, főleg a kezdő programozóknak. Jelenleg csak a Vala az egyetlen magas szintű nyelv amely támogatott. A GtkGame vala illesztését a vala-gen-introspect eszköz végzi, amely időközben elavult, átveszi a helyét a GObjectIntrospection. A GObjectIntrospection lehetővé teszi több script nyelv támogatását, így a játék programok készülhetnek JavaScript, Python, Ruby nyelven. A GObjectIntrospection jelenleg mintegy tíz nyelvet támogat, azonban ez a technológia még kísérleti, ma (2011) a Vala és a Python illesztés stabil.

Fejlett útvonalkeresés

Nagyon sok alkalmazásban nem elégséges két pont között a legrövidebb út megtalálása. A jelenleg támogatott A* útvonalkereső nagyszámú egység irányítására teljesen alkalmatlan. Nem veszi figyelembe más egységek mozgását emiatt azok gyakran egymásba ütköznek, természetellenesen mozognak. Az A* másik hátránya, hogy nagyon lassú, megfelelő világ reprezentációval jelentősen növelhető az útvonalkeresés sebessége.

Pontos kijelölés támogatása a beépített megjelenítőkkel

A grafikus elemek összetettsége és az átlátszóság miatt nagyon nehéz megmondani, hogy a képernyő egy adott pontját mely objektumok érintik. Több játék programban fontos, hogy ezt pontoan meg tudjuk állapítani, például ez alapján jelöljük ki az egységeket. A kijelölés támogatása részben készen van, azonban állapota kísérleti. A legnagyobb problémát a Clutter/COGL környezet okozza, mert az OpenGL CE nem támogatja a selection render OpenGL módot.

Teljes 3D támogatás

A mai gyors video kártyákkal nagyon népszerűek a háromdimenziós játékprogramok. A fejlesztés elején nem akartam támogatni ezt a megjelenítést, ugyanis sokkal bonyolultabb mint a kétdimenziós világ ezáltal kezdő programozóknak nagyobb nehézséget okoz. Egy játék program nem attól lesz jó, hogy nagyon drága számítógép kell a futtatásához. A jövőben szeretném biztosítani a teljes háromdimenziós támogatást, mert számos játék programot nem lehet két dimenzióban elkészíteni. Jelenleg a könyvtár nagy része, beleértve a megjelenítést, készen áll a háromdimenziós alkalmazásokra, azonban még az alapvető grafikus elemek hiányoznak, illetve néhány osztály gondot okozhat amelyeket két dimenziót feltételezve írtam meg korábban.

Fizikai szimulációs könyvtár integrációja

Manapság nagyon népszerűek a nagyrészt pontos fizikai szimulációt alkalmazó játékok, ennek oka, hogy igen kiszámíthatatlanná, tehát érdekesség teszi a játékot. Több szabad forráskódú függvénykönyvtár is rendelkezésre áll, így könnyű ilyen játékokat készíteni. A GtkGame jelenleg nincs felkészítve a fizikai szimulációt alkalmazó játékokra. A problémát elsősorban az okozza, hogy GObject környezetben nem áll rendelkezésre megbízható fizikai szimulációs függvénykönyvtár, így a Vala és más GObjectIntrospection támogatott script nyelveken nem lenne elérhető ez a képesség.

Android port

Az Android az egyik legelterjedtebb mobil platform, a legtöbb telefonokon és táblagépen Android fut. A kernel Linux, azonban a környezet ami erre épül kevésbé támogatja C alkalmazások fejlesztését. A GtkGame futtatásához legelőször a GLib függvénykönyvtárat kell működésre bírni. Jelenleg elérhető egy kísérleti GLib Android platformra, így nem tartom kizártnak, hogy valamikor

a jövőben a GtkGame alkalmazások változtatás nélkül Androidon is futtathatóak legyenek.

Konklúzió

Munkám során a számítógépes játékokban felmerülő problémákkal foglalkoztam. Célom volt megtervezni és elkészíteni egy játék készítést támogató függvénykönyvtárat. A függvénykönyvtár támogatja az alapvetően szükséges funkciókat, azonban a későbbiekben, még sok hiányzó funkciót kell hozzáadni.

A GObject használta során viszonylag ritkán találkoztam rejtélyes hibákkal. A hibaüzeneteket midig hamar megértettem, szemben a C++ gyakran igen bonyolult üzeneteivel. GObject osztályok írása igen körülményes a sok körítés miatt, ezért mindenképpen érdemes valamilyen kódgenerátort használni. Megfelelő körültekintéssel az osztályok egyszerűen refaktorálhatóak. Megfelelő szintű C nyelvi ismerettel a tanulási görbe nem meredekebb mint bármelyik objektum-orientált programozás nyelvnél. A fejlesztés végére teljes kód mindennel együtt elérte a 15000 sort,azonban egy ekkora méretű program is jól átlátható, karbantartható.

Talán a legnagyobb nehézség, hogy mindennek hosszú neve van az elnevezési konvekció alapján. Zavaró, hogy a speciális formátumú GObject Generator forrás fájlokban nem képes egyik program sem hatékony kódkiegészítéssel segíteni a munkát. Összességében a C nyelven a GObject nagyszerűen használható mint objektum-orientált programozási nyelv, támogat minden funkciót amire szükség lehet. A GLib könyvtár biztosítja az alkalmazás portolhatóságát számos platformra.

Irodalomjegyzék

[1] http://developer.gnome.org/gobject/2.30/

[2] http://docs.clutter-project.org/docs/cogl/1.7/

[3] Dr. Szirmai-Kalos László: Számítógépes grafika, CoomputerBooks, 2001

[4] Bányász Gábor, Levendovszky Tihamér: Linux programozás, Szak kiadó, 2003

[5] http://www.json.org/

[6] http://www.yaml.org/

[7] http://maemo.org/

[8] http://wiki.wesnoth.org/Team_Color_Shifting

[9] http://live.gnome.org/Vala