jamk.fi

Luaskriptin käyttö C++-aplikaatiossa

Jesse Hallinen

Opinnäytetyö

Tammikuu 2014

Ohjelmistotekniikan koulutusohjelma

Tekniikan ja liikenteen ala

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

KUVAILULEHTI

Tekijä(t) Hallinen, Jesse	Julkaisun laji Opinnäytetyö	Päivämäärä 17.01.2014
	Sivumäärä 42	Julkaisun kieli Suomi
		Verkkojulkaisulupa myönnetty (X)

Työn nimi
Luaskriptin käyttö C++-aplikaatiossa

Koulutusohjelma
Ohjelmistotekniikka

Työn ohjaaja(t)
Mieskolainen, Matti
Huotari, Jouni

Toimeksiantaja(t)
Star Arcade Oy

Tiivistelmä
Työn tavoitteena oli tehdä ristinollan kaltainen Five in a row -peli käyttäen Luaskriptiä toimeksiantona peliohjelmointiyritys Star Arcade Oy:lle sekä esitellä esimerkkien avulla, kuinka Lua voidaan käytännössä liittää C++-aplikaatioon. Luan liitännän toteutuksessa testattiin neljää erilaista Luan ja C++:n välistä liitäntäratkaisua, jotka olivat LuaBind, ToLua++, Lunar ja LuaBridge. Liitännät toteutettiin itse tehtyyn yksinkertaiseen pelimoottoriin. Lualla tehtiin myös yksinkertainen ohjelma, joka käytti edellä mainittua pelimoottoria. Liitäntäratkaisuja vertailtiin toisiinsa niiden ominaisuuksien, käytön helppouden ja liittämiseen vaaditun koodin määrän perusteella.

Pelin toteutus onnistui tavoitteiden mukaisesti aikataulussa. Peli tullaan luultavasti joskus julkaisemaan sellaisenaan tai hieman muunneltuna.

LuaBind on hyvä ratkaisu silloin, kun vaaditaan helppoa ja kehittynyttä C++:n ominaisuuksien käyttöä Luan kautta eikä välitetä ajotiedoston koon kasvamisesta.

LuaBridge on ominaisuuksiltaan LuaBindin ja Lunarin välistä, mutta kuitenkin paljon lähempänä Lunaria. LuaBridgen käyttö on miltei yhtä helppoa kuin LuaBindin eikä kuitenkaan ole riippuvainen mistään suurista kirjastoista. LuaBridge onkin oivallinen vaihtoehto, kun halutaan pitää ajotiedosto suhteellisen pienenä ja Luan liittämisprosessi yksinkertaisena.

Lunar on tutkituista ratkaisuista pienin ja nopein, mutta ominaisuuksiltaan vähäisin. Se vaatii luomaan C++-funktioista ja -luokista yhteensopivia Lunarin kanssa. Lunar soveltuukin erityisesti yksinkertaisiin projekteihin, joilta vaaditaan hyvää suorituskykyä ja pientä kokoa. Lunarin osittainen tarkoitus on toimia myös pohjana, jonka päälle jokainen voi rakentaa tarvittaessa oman ratkaisunsa.

ToLua++:n käyttö ei onnistunut, mutta ominaisuuksiensa perusteella se vaikuttaa mielenkiintoiselta vaihtoehdolta.

Avainsanat (asiasanat)
Lua, C++, LuaBind, LuaBridge, Lunar, ToLua++, peliohjelmointi

Muut tiedot

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

DESCRIPTION

Author(s)	Type of publication	Date
Hallinen, Jesse	Bachelor´s Thesis	17012014
	Pages 42	Language Finnish
		Permission for web publication (X)

Title
Using Lua script with C++ application

Degree Programme
Software Engineering

Tutor(s)
Mieskolainen, Matti
Huotari, Jouni

Assigned by
Star Arcade Oy

Abstract
The goal of this project was to create a Five in a Row game for Star Arcade Oy and study different ways of binding Lua script to a C++ application. Lua binding was tested with four different solutions: LuaBind, ToLua++, Lunar and LuaBridge. The target C++ application was a self-made simple game engine. A simple Lua program was also made to test this game engine. These four Lua solutions were compared to each other by features, how easy to use they are and the code amount needed for binding Lua to C++ program.

The game was easy to make because of Lua's simple syntax. It was created in time as expected and it will probably be published as it is or slightly modified.

LuaBind is a good solution when an easy to use and advanced way is needed. A downside of LuaBind is the increase in executable file size.

LuaBridge is between LuaBind and Lunar by its features; however, it is still more close to Lunar. LuaBridge lacks many features found in LuaBind. LuaBridge is a dependency free solution and it is almost as easy to use as LuaBind and a very good solution when the executable file needs to be small and Lua binding simple.

Lunar is the smallest and the fastest of these solutions; however, it also lacks many features. It requires C++ functions and classes to be made compatible with Lunar. Lunar fits well in simple projects requiring good performance and a small size. Lunar is also designed to be used as a base for users wanting to build their own Lua binding solutions.

Binding with ToLua++ did not work out; however, considering its features it seems like a good solution.

Keywords
Lua, C++, LuaBind, LuaBridge, Lunar, ToLua++, Game programming

Miscellaneous

Sisältö

Kuviot

1 Työn lähtökohdat

1.1 Työn tavoitteet

Tavoitteena oli tehdä ristinollan kaltainen Five in a Row -peli käyttäen Luaskriptiä toimeksiantona peliohjelmointiyritys Star Arcadelle sekä esitellä esimerkkien avulla, kuinka Lua voidaan käytännössä liittää C++-aplikaatioon. Kyseinen Five in a Row -peli oli työn konkreettinen tavoite toimeksiantajalle, ja Luan ja C++:n liittämisen tutkimisen tarkoituksena oli syventää tietämystä Lua-peleihin liittyen.

1.2 Toimeksiantaja

Star Arcade Oy on jyväskyläläinen peliohjelmointiyritys, joka on perustettu vuonna 2010, ja se työllistää noin 30 henkeä. Yritys keskittyy pääasiassa alati kasvaville mobiilipelimarkkinoille ja panostaa erityisesti sosiaalisiin alustariippumattomiin reaaliaikaisiin moninpeleihin. *(About Star Arcade 2013.)*

Star Arcade on luonut kahdeksan menestynyttä peliä ja sen peleillä on koko maailmassa yhteensä yli miljoona pelaajaa yli 180 maassa. Sen pelit ovat saatavilla useista sovelluskaupoista, ja niitä ovat mm. Jelly Wars, Diamonds Paradise, Diamonds Paradise 2, Battleships, King of Words, Tic Tac Toe, Spawned, Crazy Gardens, Memory Game ja Mancala. Star Arcaden tukemiin alustoihin kuuluvat mm. IOS, Android, Nokia, BlackBerry, Amazon Kindle, Samsung Bada, Windows Phone, Facebook ja PC. *(About Star Arcade 2013.)*

1.3 Five in a Row

Five in a Row on ristinollan kaltainen peli, jota pelataan perinteisesti valkoisilla ja mustilla Go-napeilla ja -laudalla, jossa on 19 kertaa 19 viivojen leikkauspistettä. Pelin aikana nappeja ei liikuteta enää paikalleen asettamisen jälkeen, joten peliä voidaan myös pelata kynällä ja paperilla käyttäen esimerkiksi ristejä ja nollia nappien symboleina. Tällöin sitä kutsutaan suomalaisittain ristinollaksi. Five in a Row'n tunnetuin nimitys lienee gomoku tai gobang, ainakin Aasian maissa. *(Gomoku 2013.)*

Pelin säännöistä on useita muunnoksia, mutta tässä opinnäytetyössä keskitytään vain käytettyyn hieman vapaatyyliseen peliin. Normaalissa gomokussa pitää saada tasan viisi nappulaa peräkkäin, mutta tämän työn pelissä voi voittaa myös yli viiden napin pituisella sarjalla. Pelilauta ei myöskään ole perinteinen, vaan siinä on 15 kertaa 15 leikkauspistettä. Pelinappulatkaan eivät ole perinteisen mustia ja valkoisia vaan sinisiä ja oransseja.

2 Teknologiat ja työkalut

2.1 C++

C++ on yksi suosituimmista ohjelmointikielistä. Sen kehittämisen aloitti Bjarne Stroustrup C-kielen pohjalta vuonna 1979 työskennellessään Bell Labsissa. C++ tukee olio-ohjelmointia, geneeristä ohjelmointia ja abstrakteja rakenteita. Sitä pidetäänkin keskitason ohjelmointikielenä, koska se sisältää sekä korkean tason että matalan tason ominaisuuksia. *(C++ 2013.)*

C++-ohjelmointikielen suunnittelussa on yritetty poistaa kaikki ylimääräinen ajonaikainen koodi, kuten esimerkiksi automaattinen roskienkeräys. Tällä tavoin on luotu ohjelmointikieli, joka soveltuu järjestelmään kuin järjestelmään. C++ onkin yksi suosituimmista ohjelmointikielistä graafisten aplikaatioiden kehityksessä ja suosituin kieli pelien kehityksessä. *(C++ 2013.)*

2.2 Työkalut

Työkaluna opinnäytetyössä käytettiin Windows 7 -käyttöjärjestelmällä varustettua tietokonetta. Ohjelmankehitysympäristönä oli Visual Studio 2010, johon asennettiin Lua language support -liitännäinen. Kaikki peliin liittyvä ohjelmointi tehtiin Lua-skriptikielellä, kun taas Luan liittämisen tutkimisessa käytettiin sekä C++- että Lua-ohjelmointikieltä.

2.3 Star Arcade SDK

Star Arcade SDK on pelien kehitykseen tarkoitettu paketti, ja se sisältää mm. Star Arcaden pelimoottorin. Pelimoottoreiden pääasiallinen tarkoitus on hoitaa peleissä tarvittavat perustoiminnot, kuten mm. fysiikan mallinnus, animaatiot, grafiikat, äänet sekä verkkoyhteydet. Star Arcaden pelimoottoriin on implementoitu Box2D-fysiikan mallinnus sekä useita OpenGL-, OpenGL ES- ja DirectX-toimintoja. Kyseinen pelimoottori on toteutettu siten, että se mahdollistaa alustariippumattoman pelien kehityksen. *(Kostilainen 2013, 15.)*

Star Arcade SDK sisältää myös sosiaalisen moninpelikerroksen, joka on rakennettu Star Arcaden pelimoottorin päälle. Sen avulla pelinkehittäjä pystyy lisäämään peliinsä

helposti tarvittavat sosiaaliset toiminnot, kuten keskustelut ja pelaamisen muiden pelaajien kanssa. Moninpelikerros hoitaa myös käyttäjän asetukset ja tilastot, ostostapahtumat, kirjautumisen ja rekisteröinnin. *(Kostilainen 2013, 14-15.)*

2.4 StarLua

StarLua on Star Arcaden pelimoottorille kehitetty Lua-rajapinta, joka mahdollistaa pelimoottorin ominaisuuksien hyödyntämisen Luan avulla. Tällä tavoin on siis mahdollista hyödyntää Luan yksinkertaista syntaksia, mutta kuitenkin valjastaa pelimoottorin tehokkuus käyttöön. StarLua on rakennettu C++-templaatteihin pohjautuvan Lunar-toteutuksen päälle, jotta suorituskyky pysyisi riittävän hyvänä mobiilipeleihin. Lunar on yksi tässä opinnäytetyössä testatuista ratkaisuista, ja sitä on tarkasteltu lähemmin luvussa 4.5.

2.5 MyEngine

MyEngine on itse tehty C++-ohjelma, jonka voisi luokitella erittäin yksinkertaiseksi pelimoottorin esiasteeksi. MyEngine pystyy piirtämään ja liikuttelemaan neliöitä ja ympyröitä, mutta siihen se sitten jääkin. MyEngine toimii tässä opinnäytetyössä esimerkkiohjelmana Luan liittämiselle.

MyEnginessä on neljä luokkaa, jotka ovat MyObjectManager, MyObject, MyRectangle ja MyCircle. MyObjectManager pitää listaa kaikista MyObject-objekteista ja kutsuu niiden päivitys- ja piirto-funktioita. MyObject on perusluokka kaikille objekteille ja se hoitaa objektin omat toiminnot, kuten liikkumisen, nopeuden muutoksen ja värin vaihdon. MyRectangle- ja MyCircle-luokat ovat perittyjä MyObject luokasta ja näin ol-

len saavat MyObject-luokan ominaisuudet käyttöönsä. MyRectangle ja MyCircle luokat sisältävät käytännössä ainoastaan erilaiset piirtofunktiot. MyRectangle piirtää nelikulmion ja MyCircle piirtää ympyrän.

Seuraavaksi on esitelty MyEngineä, ennen kuin siihen on liitetty Lua-koodia. Esiteltävinä ovat MyObject.h, MyObject.cpp sekä main.cpp-pääohjelma. Näitä kyseisiä tiedostoja vertaillaan myöhemmin tiedostoihin, joihin Luan liittäminen on toteutettu.

MyObject.h

```
#ifndef __MYOBJECT_H__
#define __MYOBJECT_H__

#include <stdlib.h>
#include <glut.h>

#define _USE_MATH_DEFINES
#include <math.h>

class MyObject {
public:
    bool alive;
    float x;
    float y;
    float width;
    float height;
    bool moving;
    float speed;
    float r;
    float g;
    float b;
    bool targetReached;

public:
    MyObject();
    virtual ~MyObject();

    void Instantiate();

    void Update();
    virtual void Draw();

    void SetPosition(float x, float y);
    void MoveTo(float targetX, float targetY);
    void Stop();
    void Destroy();
    void SetColor(float r, float g, float b);
    bool TargetReached();
    void SetSpeed(float speed);
```

```
protected:
    float endX;
    float endY;
    float targetX;
    float targetY;
    float movingAngle;
};

#endif __MYOBJECT_H__
```

MyObject.cpp

```
#include "MyObject.h"
#include "MyObjectManager.h"

MyObject::MyObject() {
    alive = true;
    x = 0;
    y = 0;
    width = 0.2;
    height = 0.2;
    endX = x + width;
    endY = y + height;
    targetX = 0;
    targetY = 0;
    moving = false;
    movingAngle = 0;
    speed = 0.001;
    r = 1;
    g = 1;
    b = 1;
    targetReached = false;
}

MyObject::~MyObject() {

}

// MyObjectista perityt luokat kutsuvat tätä funktiota konstruktorissaan
void MyObject::Instantiate() {
    // lisätään objekti listaan kaikista objekteista
    MyObjectManager::Instance()->objects.push_back(this);
}

void MyObject::Update() {
    if(targetReached) {
       targetReached = false;
    }

    if(moving) {
       float tempX = x;

       // liikkuminen
       x += speed * cos(movingAngle);
       y += speed * sin(movingAngle);
       endX += speed * cos(movingAngle);
       endY += speed * sin(movingAngle);

       // jos kohde on saavutettu
```

```
        if((tempX < targetX && targetX < x) || (x < targetX && targetX < tempX))
        {
            moving = false;
            targetReached = true;
        }
    }
}

void MyObject::Draw() {

}

void MyObject::SetPosition(float x, float y) {
    this->x = x;
    this->y = y;
    endX = x + width;
    endY = y + height;
}

void MyObject::MoveTo(float targetX, float targetY) {
    this->targetX = targetX;
    this->targetY = targetY;

    // lähtöpisteen ja kohdepisteen erotus
    float deltaX = targetX - x;
    float deltaY = targetY - y;

    // lasketaan liikkumiskulma
    movingAngle = (float)atan2(deltaY, deltaX);
    moving = true;
    targetReached = false;
}

void MyObject::Stop() {
    moving = false;
}

void MyObject::Destroy() {
    alive = false;
}

void MyObject::SetColor(float r, float g, float b) {
    this->r = r;
    this->g = g;
    this->b = b;
}

bool MyObject::TargetReached() {
    return targetReached;
}

void MyObject::SetSpeed(float speed) {
    this->speed = speed;
}
```

main.cpp

Kaikki gl- tai glut-alkuiset sanat liittyvät vain grafiikoiden piirtoon eivätkä ole oleellisia tämän opinnäytetyön tai Luan ja C++:n liittämisen kannalta.

```
#include <stdlib.h>
#include <glut.h>
#include "MyObjectManager.h"
#include "MyRectangle.h"
#include "MyCircle.h"

// muistivuotojen havaitseminen
#define _CRTDBG_MAP_ALLOC
#include <crtdbg.h>

// objectManager
MyObjectManager* objectManager = MyObjectManager::Instance();

void Update() {
    // päivitetään objektit
    objectManager->Update();

    // merkataan nykyinen ikkuna uudelleen piirrettäväksi
    glutPostRedisplay();
}

void Draw() {
    // puhdistetaan ruutu
    glClear(GL_COLOR_BUFFER_BIT);

    // piirretään objektit
    objectManager->Draw();

    // vaihdetaan puskurit
    glutSwapBuffers();
}

// Initialisointi
void Initialize() {
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
}

int main(int argc, char** argv) {
    // Initialisointi
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(600, 600);
    glutInitWindowPosition(200, 200);
    glutCreateWindow("MyEngine");
    Initialize();

    // asetetaan päivitys-funktio
    glutIdleFunc(Update);

    // asetetaan piirto-funktio
    glutDisplayFunc(Draw);
```

```
    // käynnistetään glutMainLoop
    glutMainLoop();

    // muistivuotojen havaitseminen
    _CrtDumpMemoryLeaks();

    return 0;
}
```

3 Lua

3.1 Lua yleisesti

Lua on tehokas, kevyt ja todella nopea skriptikieli. Siinä yhdistyvät yksinkertainen proseduraalinen syntaksi ja tehokkaat tietorakenteet, jotka perustuvat hakurakenteisiin ja laajennettavaan semantiikkaan. Lua on dynaamisesti tyypitetty, mikä käytännössä tarkoittaa sitä, että muuttujien tyyppi määrittyy ajon aikana. Lua on tulkattava ohjelmointikieli, eli Lua-koodia voidaan muokata ajon aikana ja saada muokkaukset vaikuttamaan ilman ohjelman uudelleen kääntämistä tai käynnistämistä. Luassa on myös automaattinen muistinhallinta inkrementaalisella roskienkeräyksellä, jonka vuoksi se soveltuu hyvin konfigurointiin, skriptaukseen ja nopeaan prototyyppien luomiseen. Lua myös toimii suoraan kaikilla alustoilla, joilla on standardi C-ohjelmointikielen kääntäjä. *(Lua About 2013.)*

Tässä opinnäytetyössä käytetään Lua 5.2 -versiota, ellei toisin mainita.

3.2 Käyttökohteet

Lua on tällä hetkellä suosituin skriptikieli peleissä ja todella suosittu ohjelmissa ylipäätään. Sitä käytetään varsinkin ohjelmien kriittisissä osissa, jotka vaativat nopeaa

suoritusta, kuten pelien tekoälyssä. Luan käyttökohteista tunnetuimpina esimerkkeinä mainittakoon Adobe Photoshop Lightroom, Angry Birds, Apache HTTP Server, Apache Traffic Server, Firefox-verkkoselain, MediaWiki ja World of Warcraft. Lua onkin saanut laajalti tunnustusta varsinkin peliteollisuudelta, esimerkkinä mainittakoon 2011 Game Developer Magazine Front Line Award, jonka Lua-ohjelmointikieli voitti ohjelmointityökalu-kategoriassa. *(Merrell 2013.)*

Luan käyttökohteet voidaan yleensä jakaa kolmeen kategoriaan, joista ensimmäisenä on Luan käyttö sulautettuna toiseen aplikaatioon, toisena on Lua itsenäisenä ohjelmana ja kolmantena Luan käyttö yhdessä C-ohjelmointikielen kanssa. Star Arcaden pelimoottorille tehty StarLua-rajapinta on hyvä esimerkki Luan käytöstä sulautettuna C++-aplikaatioon ja siitä, kuinka C++-pelimoottoria voidaan käyttää yksinkertaisen Lua-rajapinnan kautta. *(Lerusalimschy 2003-2004.)*

Monet ihmiset käyttävät Luaa sulautettuna toiseen ohjelmaan, kuten CGILuaan, jota käytetään dynaamisten verkkosivujen luonnissa, tai LuaOrbiin, jota käytetään CORBA-objektien käsittelyssä. Nämä ohjelmat käyttävät Lua-C-ohjelmointirajapintaa rekisteröimään uusia funktioita, luomaan uusia tyyppejä ja muuttamaan jonkun operaation toimintaa. Tämä on kenties yleisin tapa käyttää Luaa, ja esimerkkeinä tästä ovatkin monet pelit, kuten World of Warcraft ja Angry Birds. Juurikin tähän käyttökohteeseen tämä opinnäytetyö lähinnä perustuu. *(Lerusalimschy 2003-2004.)*

Luaa voidaan myös käyttää itsenäisenä ohjelmointikielenä, vaikkakin tällöin sitä käytetään lähinnä tekstin käsittelyyn ja pieniin testiohjelmiin. Siinä tapauksessa käytetään pääasiassa Luan omia standardikirjastoja, jotka tarjoavat monia merkkijonojen käsittelyyn tarkoitettuja funktioita. *(Lerusalimschy 2003-2004.)*

Ohjelmat, jotka käyttävät Luaa kirjastona, ovat myös melko yleisiä. Silloin käytetään enemmän C-ohjelmointikieltä kuin Luaa, mutta hyvä Luan ymmärrys on tarpeen, jot-

ta voidaan tehdä tarpeeksi yksinkertaisia ja helppokäyttöisiä rajapintoja. *(Lerusalimschy 2003-2004.)*

3.3 Hyvät puolet

Ajonaikainen muokkaus

Luan kenties paras puoli on mahdollisuus Lua-koodin ajonaikaiseen muokkaamiseen. Se mahdollistaa nopean ohjelman kehityksen ilman turhia aikaa vieviä ohjelman käännösvaiheita. Käytännössä siis Lua-koodia voidaan muokata, vaikka ohjelma olisi käynnissä, ja sitten yksinkertaisesti vain suorittaa muokatut tiedostot uudestaan. Tämä on valtava etu verrattuna esimerkiksi C++-ohjelmaan, joka joudutaan pienenkin muokkauksen jälkeen kääntämään uudelleen.

Suorituskyky

Lua on nopea ja kevyt. Sen kehityksessä on panostettu nopeuteen, ja sen virallisilla kotisivuilla sanotaankin sen olevan useiden vertailuanalyysien perusteella nopein tulkattava skriptauskieli. Lua 5.2.2 -version lähdekoodi ja dokumentaatio vievät pakattuna vain noin 246 K muistia ja pakkaamattomana 960 K. *(Lua About 2013.)*

Yksinkertainen syntaksi

Hyvä puoli on myös Luan helppo ja yksinkertainen syntaksi, joka jo muutaman tunnin opettelulla alkaa luonnistua melko hyvin. Syntaksin helppous on suuri tekijä, varsinkin jos kehittäjäryhmään kuuluu henkilöitä, jotka eivät varsinaisesti ole ohjelmoijia. Esimerkiksi pelien kehityksessä tasosuunnittelijat ovat monesti enemmän graafisen-alan kuin ohjelmistoalan henkilöitä.

Muuttujat

Luassa kaikki muuttujat ovat tauluja, tämä onkin yksi suurimmista syistä Luan syntaksin helppouteen. Periaatteessa tämä tarkoittaa sitä, että ohjelmoijan ei tarvitse tietää muuta kuin miten taulut toimivat ja hän osaa jo käyttää Luan muuttujia. Luan tauluilla saa aikaan monimutkaisiakin rakenteita, kunhan pääsee jyvälle asioista.

Siirrettävyys

Lua on toteutettu puhtaasti ANSI C -ohjelmointikielellä ja kääntyy muokkaamattomana kaikille tunnetuille alustoille. Kaikki mitä Luan kääntämiseen tarvitaan on ANSI C -kääntäjä, kuten GNU Compiler Collection (GCC). *(Lua Frequently Asked Questions 2013.)*

3.4 Huonot puolet

Debuggaus

Debuggaus eli virheen etsintä ja korjaus on vaikeaa Luassa, jos Luaa käytetään toiseen ohjelmointikieleen liitettynä. Tämä johtuu siitä ettei ole olemassa kunnollisia debuggereita kyseisen kaltaisille tapauksille. Näin ollen Luassa voi olla vaikea havaita syntaksivirheitä.

Muuttujat

Sitä että kaikki muuttujat ovat tauluja voidaan pitää myös huonona puolena. Jotkut ihmiset näkevät taulut vaikeana hahmottaa. Esimerkiksi jos funktio ottaa parametrina muuttujan *myVar*, niin siitä ei näe suoraan minkälainen muuttuja on kyseessä. Lua ei myöskään varoita mitenkään parametrin väärästä rakenteesta, vaan väärä rakenne tulee esille vasta funktion sisällä kun parametria yritetään käyttää. Tietenkin olisi

mahdollista käyttää muuttujien nimissä unkarilaista notaatiota, joka kertoisi aina muuttujan tyypin. Esimerkiksi *iMyVar* olisi int-tyyppinen kokonaisluku.

4 Luan liittäminen

4.1 Yleistä

Monille ohjelmille ja etenkin pelimoottoreille halutaan yleensä luoda yksinkertainen rajapinta, joka mahdollistaa pelimoottorin käytön helposti. Niinpä yleensä esimerkiksi C++-ohjelmointikielellä luotuihin pelimoottoreihin luodaan liitäntä johonkin skriptikieleen, kuten esim. Lua-, C#- tai Python-kieleen. Lua on näistä vaihtoehdoista kevyin ja todistetusti toimiva ratkaisu.

Tässä opinnäytetyössä Luan liitäntä kohteena käytetään MyEngineä. Tietenkin huomioitavaa on, että MyEngine on hyvin pieni ja yksinkertainen ohjelma ja suuremmissa monimutkaisissa ohjelmissa Luan liittäminen ei välttämättä ole yhtä helppoa.

Luan voi liittää C++-aplikaatioon suoraan käyttämällä Luan omaa C API -rajapintaa, mutta tämä voi olla melko työlästä ja aikaa vievää riippuen tapauksesta. Onneksi Luan liitännälle on kuitenkin olemassa myös valmiita ratkaisuja. Nämä ratkaisut yrittävät erinäisin tavoin helpottaa ja nopeuttaa liittämisprosessia. Jotkin ratkaisut esimerkiksi generoivat etukäteen otsikkotiedostojen perusteella tarvittavan C++-luokkien liitäntäkoodin, kuten esim. ToLua++. Toiset ratkaisut eivät tarvitse erillistä esikäsittelykäännöstä vaan osaavat käännös- ja ajonaikaisesti käsitellä luokkia ja niiden metodeja oikein, kuten LuaBind ja LuaBridge. Yksinkertaisimmissa ratkaisuissa luokat ja metodit täytyy muokata yhteensopiviksi Luan kanssa, kuten Lunarissa. Nämä edel-

lä mainitut ratkaisut ovat joitain lupaavimmilta vaikuttavia Luan liitäntäratkaisuja ja niitä tarkastellaan ja vertaillaan lähemmin luvuissa 4.3-4.6.

4.2 Lua-esimerkkiohjelma

Lua-liitännän toimivuutta testaamaan luotiin myös yksinkertainen Lua-esimerkkiohjelma käyttäen MyEngineä. Esimerkkiohjelmassa on tarkoitus liikuttaa ympyrää sekä generoida nelikulmioista tausta Lua-koodin avulla. Kaikkien MyEnginellä tehtyjen Lua-ohjelmien käynnistyspisteenä toimii entry.lua-niminen Lua-tiedosto. Kyseinen tiedosto näyttää suurin piirtein seuraavalta riippuen Luan liitäntäratkaisusta:

entry.lua

```
-- Suoritetaan Scene.lua-tiedosto
dofile("lua_src/Scene.lua")

-- initialisoidaan scene
local scene = Scene()

local ball = {}
ball['X'] = 0.4
ball['Y'] = 0.6
ball['Radius'] = 0.1
ball['Circle'] = nil

local targetReachedCount = 0
math.randomseed(os.time())
--
-- Käynnistysfunktio Lua-ohjelmalle
--
function MyEngineLua_InitializeGame()
    -- Luodaan uusi Cicle-objekti
    ball['Circle'] = Circle(ball['X'], ball['Y'], ball['Radius'])
    -- Aletaan liikuttamaan ympyrä-objektia sattumanvaraisesti
    ball['Circle']:MoveTo(math.random(), math.random())

    -- Tehdään scene
    scene:Create(0, 0, 1, 0.1)
end

--
-- Päivitys funktio, jota kutsutaan useita kymmeniä kertoja sekunnissa
--
function MyEngineLua_Update()
```

```
    -- jos ympyrä saavutti kohteensa
    if ball['Circle']:TargetReached() then
        targetReachedCount = targetReachedCount + 1
        -- Aletaan liikuttamaan ympyrää sattumanvaraisesti
        ball['Circle']:MoveTo(math.random(), math.random())
        -- Asetetaan sattumanvarainen nopeus
        ball['Circle']:SetSpeed(math.random(0.004, 0.02))
        -- Asetetaan sattumanvarainen väri
        ball['Circle']:SetColor(math.random(), math.random(), math.random())

        -- Jos ympyrä on saavuttanut kohteen 10 kertaa niin nollataan ympyrän
        -- arvot
        if targetReachedCount > 10 then
            ball['Circle']:Reset()
        end
    end
end

----- Lisätään uusi funktio Circle-luokalle --------------------

--
-- Resetoidaan ympyrän sijainti, väri ja pysäytetään ympyrän liike
--
function Circle:Reset()
    self:SetPosition(0.5, 0.5)
    self:SetColor(1, 1, 1)
    self:Stop()
end
```

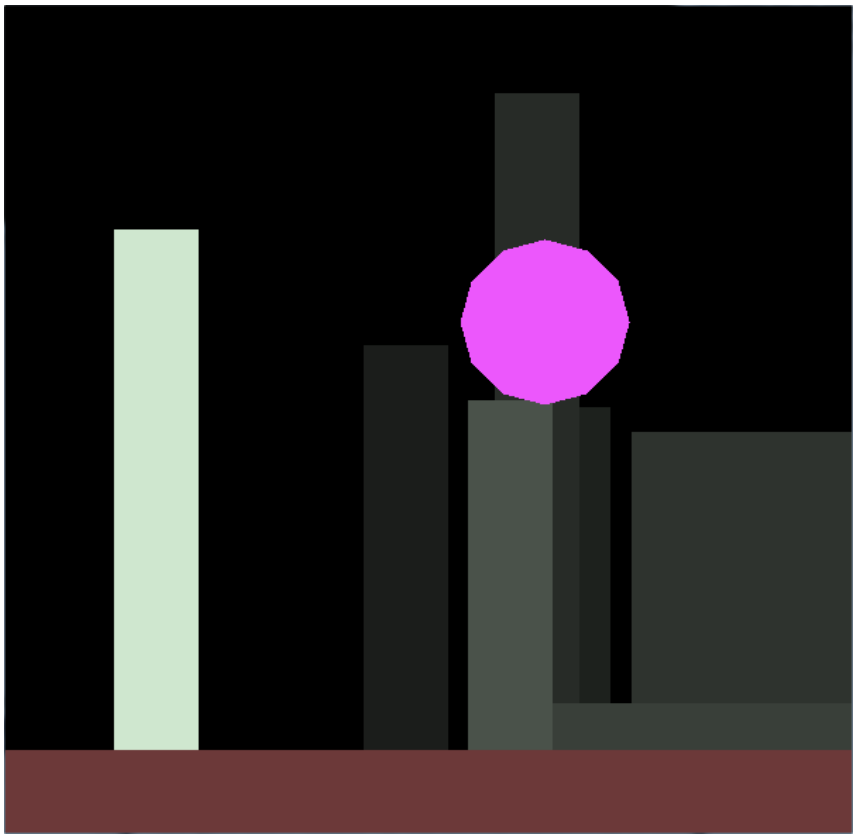
Edellä esitellyssä entry.lua-tiedostossa oleva MyEngineLua_InitializeGame-funktio toimii käynnistyspisteenä MyEnginen Lua-ohjelmalle ja MyEngineLua_Update-funktiota puolestaan kutsutaan MyEnginen Update-funktiosta, jota suoritetaan useita kymmeniä kertoja sekunnissa. MyEngineLua_InitializeGamessa vain luodaan uusi Circle-objekti, aletaan liikuttaa sitä sattumanvaraisesti johonkin suuntaan ja luodaan uusi tausta Scene-luokan ilmentymän avulla. Circle:Reset-funktio lisää Circle-luokalle uuden Reset-funktion, jota voidaan nyt käyttää Circle-luokan ilmentymien kautta.

Scene-luokka on Luassa tehty luokka. Tämän luokan tarkoitus on lähinnä näyttää, miten Luassa voidaan luoda luokkaa vastaava rakenne, vaikka käytössä on pelkkiä taulumuuttujia. Scene-luokalla ei ole kuin yksi funktio: Create, joka luo ruskean nelikulmion lattiaksi ja harmaita nelikulmioita taustalle. Scene.lua-tiedosto näyttää seuraavanlaiselta:

Scene.lua

```
Scene = {}
Scene = function()
    -- objekti
    local self = {}

    -- floor-taulu, sisältää lattiaan liittyvää tietoa
    local floor = {}
    floor['X'] = 0
    floor['Y'] = 0
    floor['Width'] = 0
    floor['Height'] = 0
    -- lattian väri
    floor['Red'] = 0.4
    floor['Green'] = 0.2
    floor['Blue'] = 0.2
    floor['Rect'] = nil

    -- tausta
    local backgroundItems = {}     -- sisältää kaikki taustaobjektit
    local itemCount = 10    – taustaobjektien määrä

    --
    -- Luodaan lattia ja tausta
    -- @param x = lattian alku x-koordinaatti
    -- @param y = lattian alku y-koordinaatti
    -- @param w = lattian leveys
    -- @param h = lattian korkeus
    --
    self.Create = function(s, x, y, w, h)

        -- luodaan lattia
        floor['X'] = x
        floor['Y'] = y
        floor['Width'] = w
        floor['Height'] = h
        floor['Rect'] = Rectangle(x, y, w, h)
        floor['Rect']:SetColor(floor['Red'], floor['Green'], floor['Blue'])

        math.randomseed(os.time()) -- os.time on hyvä satunnainen siemenluku

        – luodaan taustaobjektit
        for i = 1, itemCount do
            -- tämä auttaa saamaan satunnaisempia numeroita
            math.random() math.random() math.random()

            -- luodaan nelikulmio satunnaiseen kohtaan
            backgroundItems[i] = Rectangle(math.random(), 0, math.random(0.1,
                                                0.4), math.random())
            -- asetetaan väri nelikulmiolle
            backgroundItems[i]:SetColor(0.8 / i, 0.9 / i, 0.8 / i)
        end
    end

    return self
end
```

Käynnistettynä esimerkkiohjelma näyttää suunnilleen kuvion 1 mukaiselta:



Kuvio 1. Lua-esimerkkiohjelma

4.3 LuaBind

LuaBind on hyvin kattava ja erittäin helppokäyttöinen paketti Luan ja C++:n liittämiseen, mutta se vaatii myös liittämään suuren osan valtavasta Boost-paketista. Tämä taas johtaa siihen, että LuaBind ei sovellu kunnolla mobiilisovelluksiin, jotka vaativat pientä kokoa. Yksi suurimmista syistä, miksi ihmiset käyttävät Lua-skriptikieltä ohjelmissaan on, että he haluavat pitää ohjelmansa mahdollisimman pieninä ja nopeina. LuaBind kasvattaakin exe-tiedoston eli ajotiedoston kokoa niin paljon, että monet suosiolla siirtyvät etsimään kevyempää vaihtoehtoa.

LuaBind tukee seuraavia ominaisuuksia :

- Ylikuormitettuja metodeja ja vapaita funktioita
- C++-luokkien ja niiden muuttujien käyttöä Luassa
- operaattoreita
- enum-muuttujia
- Lua-luokkien ja -funktioiden käyttöä C++:ssa
- C++-luokkien virtuaalisten funktioiden ylikirjoitusta
- rekisteröityjen tyyppien epäsuoraa konvertointia
- parhaiten vastaavan funktion allekirjoituksen tarkistusta
- palautusarvo- ja parametrikäytäntöjä. *(LuaBind 2005.)*

LuaBind on toteutettu hyödyntäen templaatti-ohjelmointia, jonka vuoksi projekti ei tarvitse ylimääräistä esikäsittelykäännöstä. Tämän vuoksi käyttäjän ei yleensä tarvitse tietää tarkkaa allekirjoitusta rekisteröitävälle funktiolle, koska kirjasto generoi koodin perustuen käännösaikaiseen funktiotyyppiin, joka sisältää itsessään allekirjoituksen. Huonoin puoli tässä lähestymistavassa on käännösajan kasvu tiedostolle, joka suorittaa rekisteröinnin. Tämän takia onkin suositeltavaa rekisteröidä kaikki samassa cpp-tiedostossa. *(LuaBind 2005.)*

Luaskriptin liittämisestä LuaBindin 0.9.1 -version avulla MyEngineen suoriuduttiin todella nopeasti ja vaivattomasti. MyEnginen tiedostoihin ei tarvinnut lisätä kovinkaan paljon uutta koodia. Itse asiassa ainoastaan main.cpp-tiedostoon tarvittiin joitain lisäyksiä. Seuraavaksi tarkastellaan main.cpp-tiedostoa tarvittavien lisäysten jälkeen. LuaBindiin liittyvät lisäykset ovat merkitty kirkkaan punaisella ja niiden yläpuolella on myös selitetty kunkin lisäyksen tarkoitus.

Main.cpp

```
#include <stdlib.h>
#include <glut.h>
#include "MyObjectManager.h"
#include "MyRectangle.h"
#include "MyCircle.h"

// liitetään mukaan Luan kirjastot
extern "C" {
    #include "lua.h"
    #include "lualib.h"
    #include "lauxlib.h"
}

// liitetään mukaan LuaBind
#include <luabind/luabind.hpp>

// muistivuotojen havaitseminen
#define _CRTDBG_MAP_ALLOC
#include <crtdbg.h>

// Osoitin dynaamiseen lua_State-rakenteeseen, joka toimii tavallaan tulkkina
// Luan ja C++:n välillä
lua_State *L;

MyObjectManager* objectManager = MyObjectManager::Instance();


void Update() {
    // päivitetään objektit
    objectManager->Update();

    // Kutsutaan LuaBindin call_function-funktiolla lua_Statesta löytyvää
    // MyEngineLua_Update-funktiota. MyEngineLua_Update löytyy entry.lua-
    // tiedostosta, jolla testataan Luan liitännän onnistumista
    luabind::call_function<void>(L, "MyEngineLua_Update");

    // merkataan nykyinen ikkuna uudelleen piirrettäväksi
    glutPostRedisplay();
}

void Draw() {
    glClear(GL_COLOR_BUFFER_BIT);

    // piirretään objektit
    objectManager->Draw();

    glutSwapBuffers();
}

// Rekisteröidään MyObject, MyRectangle ja MyCircle luokat ja niiden funktiot
// LuaBindiin, joka käytännössä tarkoittaa luokkien rekisteröimistä lua_Stateen
void RegisterClasses() {
    luabind::module(L) [
       // MyObject
       luabind::class_<MyObject>("Object")
          .def(luabind::constructor<>())
          .def("SetPosition", &MyObject::SetPosition)
          .def("MoveTo", &MyObject::MoveTo)
          .def("Stop", &MyObject::Stop)
```

```
            .def("Destroy", &MyObject::Destroy)
            .def("SetColor", &MyObject::SetColor)
            .def("TargetReached", &MyObject::TargetReached)
            .def("SetSpeed", &MyObject::SetSpeed),

        // MyRectangle
        luabind::class_<MyRectangle>("Rectangle")
            .def(luabind::constructor<float, float, float, float>())
            .def("SetPosition", &MyObject::SetPosition)
            .def("MoveTo", &MyObject::MoveTo)
            .def("Stop", &MyObject::Stop)
            .def("Destroy", &MyObject::Destroy)
            .def("SetColor", &MyObject::SetColor)
            .def("TargetReached", &MyObject::TargetReached)
            .def("SetSpeed", &MyObject::SetSpeed),

        // MyCircle
        luabind::class_<MyCircle>("Circle")
            .def(luabind::constructor<float, float, float>())
            .def("SetPosition", &MyObject::SetPosition)
            .def("MoveTo", &MyObject::MoveTo)
            .def("Stop", &MyObject::Stop)
            .def("Destroy", &MyObject::Destroy)
            .def("SetColor", &MyObject::SetColor)
            .def("TargetReached", &MyObject::TargetReached)
            .def("SetSpeed", &MyObject::SetSpeed)
    ];
}

// Initialisointi
void Initialize() {
    // initialisoidaan lua_State. Nyt lua_State on vielä tyhjä eikä sisällä
    // mitään funktioita tai luokkia
    L = luaL_newstate();

    // ladataan Luan peruskirjastot lua_Stateen, jotta päästään käsiksi niiden
    // funktioihin
    luaL_openlibs(L);

    // yhdistetään LuaBind tähän lua_Stateen
    luabind::open(L);

    // Rekisteröidään luokat Luaan
    RegisterClasses();

    glClearColor(0.0, 0.0, 0.0, 0.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);

    // suoritetaan entry.lua tiedosto ja samalla sen funktiot tallentuvat
    // lua_Stateen, jotta niitä voidaan kutsua myöhemmin
    luaL_dofile(L, "lua_src/entry.lua");

    // Kutsutaan entry.lua:ssa olevaa MyEngineLua_InitializeGame-funktiota,
    // joka toimii pienen Lua esimerkkiohjelman käynnistyspisteenä
    luabind::call_function<void>(L, "MyEngineLua_InitializeGame");
}

int main(int argc, char** argv) {
    // Initialisointi
    glutInit(&argc, argv);
```

```
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(600, 600);
    glutInitWindowPosition(200, 200);
    glutCreateWindow("MyEngine");
    Initialize();

    // asetetaan päivitys-funktio
    glutIdleFunc(Update);
    // asetetaan piirto-funktio
    glutDisplayFunc(Draw);
    glutMainLoop();

    // Siivotaan roskat eli kutsutaan Luan roskienkeräystä
    lua_gc(L, LUA_GCCOLLECT, 0);
    // Suljetaan Lua
    lua_close(L);

    // Muistivuotojen havaitseminen
    _CrtDumpMemoryLeaks();
    return 0;
}
```

Kokeilun perusteella LuaBind soveltuu ohjelmiin, joissa ei keskitytä täysin suorituskykyyn ja vaaditaan kehittynyttä rajapintaa Luan ja C++:n välille. LuaBind on luultavasti valmein Luan liitäntäratkaisu.

4.4 ToLua++

ToLua++ 1.0 -versio perustuu C++-otsikkotiedostoihin. Otsikkotiedostojen muuttujien, funktioiden ja luokkien perusteella ToLua++ generoi tarvittavan Luan liitäntäkodin, jotta Lua pääsee käsiksi C++-koodin ominaisuuksiin. *(Celes & Manzur 2003.)*

ToLua++ on yksi kevyimmistä ratkaisuista, minkä takia monet ihmiset kokeilevat sitä. ToLua++:n huonoja puolia ovat dokumentaation puute ja bugit. Huonona puolena voidaan pitää myös ToLua++:n perustumista Lua 5.0 -versioon, eikä uusimpaan 5.2-versioon. Tämän takia Lua 5.2 -versiota ei voi käyttää suoraan ToLua++ 1.0 -version kanssa.

Tässä opinnäytetyössä kokeiltiin ToLua++:n käyttöä Lua 5.2- sekä Lua 5.0 -versioden kanssa. Lua 5.2:n kanssa havaittiin liian monta yhteensopimattomuutta ja päätettiin siirtyä Lua 5.0:aan. Lua 5.0:n kanssa onnistuttiin generoimaan tarvittavat liitäntäkooditiedostot, mutta ToLua++ generoi näistä tiedostoista virheellisiä. Virheiden korjaaminen olisi vaatinut joko ToLua++:n tai generoitujen tiedostojen suurta muokkaamista, joten lopulta päätettiin hylätä ToLua++ ja siirtyä seuraavaan vaihtoehtoon.

4.5 Lunar

Lunar on paranneltu versio Lunasta ja koostuu ainoastaan yhdestä lunar.h-tiedostosta, joka sisältää vain noin 215 riviä koodia. Alkuperäisen staattisen Luna-templaattiluokan loi Lenny Palozzi käyttäen hyväkseen Luan C API:a, C++-templaatteja ja Luan liitännäismekanismejä tarkoituksenaan luoda pieni ja helppokäyttöinen, mutta samalla tehokas keino luokkien ja funktioiden rekisteröimiseen. *(Palozzi 2003.)*

Lunar ei ole niin kehittynyt liitäntäratkaisu kuin esimerkiksi LuaBind. Sellaisenaan se onkin tarkoitettu lähinnä pelkkien luokkien ja funktioiden rekisteröimiseen. Lunarin oikea tarkoitus onkin toimia pohjana, jonka päälle käyttäjä voi rakentaa juuri omaan projektiinsa sopivan ratkaisun.

Lunarin yksinkertaisuuden takia Luan liittäminen MyEngineen vaatiikin uutta koodia MyObject-, MyRectangle- ja MyCircle-luokkiin. Koska tarkoituksena on pystyä käyttämään kyseisiä C++-luokkia myös C++-koodissa, päätettiin niihin tehdä erikseen Lua-funktiot ja jättää alkuperäiset funktiot koskemattomiksi. Seuraavaksi onkin esitetty tarvittavat lisäykset MyObject.h- ja MyObject.cpp-tiedostoihin. Vihreällä kirjoitetuissa kommenteissa selitetään kaikkien lisäysten tarkoitus.

MyObject.h

MyObject.h:n alkuun piti lisätä tarvittavien kirjastojen liitännät:

```
// Luan kirjastot
extern "C" {
    #include "lua.h"
    #include "lauxlib.h"
    #include "lualib.h"
}

// Liitetään mukaan lunar
#include "lunar.h"
```

MyObject-luokan esittelyn sisälle lisättiin seuraavat:

```
// lunarin vaatimukset
public:
    // konstruktori, joka ottaa parametrina osoittimen lua_Stateen
    MyObject(lua_State *L);
    // className-jäsenmuuttuja, joka pitää sisällään luokan nimen,
    // jolla luokkaa käytetään Luaskriptissä
    static const char className[];
    // methods-jäsenmuuttuja, joka pitää sisällään kaikki metodit,
    // joita voi käyttää Luaskriptissä
    static Lunar<MyObject>::RegType methods[];

// Metodit, joita on tarkoitus käyttää Luaskriptissä.
// Kaikkien metodien tulee ottaa parametrina osoitin lua_Stateen
// ja palauttaa kokonaisluku int
public:
    int SetPosition(lua_State *L);
    int MoveTo(lua_State *L);
    int Stop(lua_State *L);
    int Destroy(lua_State *L);
    int SetColor(lua_State *L);
    int TargetReached(lua_State *L);
    int SetSpeed(lua_State *L);
```

MyObject.cpp

MyObject.cpp:ssä puolestaan määriteltiin edellä tehdyt muutokset. Tiedoston alkuun lisättiin:

```
// Luokan nimen määrittely
const char MyObject::className[] = "Object";
// Metodien rekisteröinti
Lunar<MyObject>::RegType MyObject::methods[] =
{
    {"SetPosition", &MyObject::SetPosition},
    {"MoveTo", &MyObject::MoveTo},
    {"Stop", &MyObject::Stop},
    {"Destroy", &MyObject::Destroy},
    {"SetColor", &MyObject::SetColor},
    {"TargetReached", &MyObject::TargetReached},
```

```
    {"SetSpeed", &MyObject::SetSpeed},
    {0,0}
};

// lunarin vaatima konstruktori
MyObject::MyObject(lua_State *L) {

}
```

Tiedoston loppuun lisättiin:

```
// Metodien määrittely
// Kaikki metodit vain lukevat lua_Statesta tarvittavat parametrit ja
// kutsuvat sitten näillä parametreilla alkuperäisiä C++-metodeja, jotka vasta
// suorittavat määrätyt asiat.
//
// Ainoa poikkeus on TargetReached-metodi, joka palauttaa arvon.
// Tämä tapahtuu työntämällä oikean TargetReached-metodin palautusarvo
// lua_Staten pinoon.

int MyObject::SetPosition(lua_State *L) {
    SetPosition(lua_tonumber(L, 1), lua_tonumber(L, 2));
    return 0;
}

int MyObject::MoveTo(lua_State *L) {
    MoveTo(lua_tonumber(L, 1), lua_tonumber(L, 2));
    return 0;
}

int MyObject::Stop(lua_State *L) {
    Stop();
    return 0;
}

int MyObject::Destroy(lua_State *L) {
    Destroy();
    return 0;
}

int MyObject::SetColor(lua_State *L) {
    SetColor(lua_tonumber(L, 1), lua_tonumber(L, 2), lua_tonumber(L, 3));
    return 0;
}

int MyObject::TargetReached(lua_State *L) {
    lua_pushnumber(L, (int)TargetReached());
    return 1;
}

int MyObject::SetSpeed(lua_State *L) {
    SetSpeed(lua_tonumber(L, 1));
    return 0;
}
```

Samanlaiset muutokset kuin edellä mainitut täytyi myös tehdä MyRectangle- ja MyCircle-luokkiin. Tosin niissä ei täytynyt tietenkään määritellä funtioita uudestaan, kos-

ka ne ovat perittyjä MyObject-luokasta. Funktiot täytyi vain rekisteröidä myös näille luokille. Esimerkkinä MyRectangle-luokan MyObject-luokasta perittyjen metodien rekisteröinti:

```
// Luokan nimen määrittely
const char MyRectangle::className[] = "Rectangle";
// Metodien rekisteröinti
Lunar<MyRectangle>::RegType MyRectangle::methods[] =
{
    // Peritty MyObject luokasta
    {"SetPosition", &MyObject::SetPosition},
    {"MoveTo", &MyObject::MoveTo},
    {"Stop", &MyObject::Stop},
    {"Destroy", &MyObject::Destroy},
    {"SetColor", &MyObject::SetColor},
    {"TargetReached", &MyObject::TargetReached},
    {"SetSpeed", &MyObject::SetSpeed},
    {0,0}
};
```

Seuraavaksi on tarkastelussa main.cpp:hen tehdyt lisäykset. Kaikki lisäykset on merkitty taas kirkkaan punaisella asian havainnollistamiseksi ja lisäysten yläpuolella selitetään lisäyksen tarkoitus.

main.cpp

```
#include <stdlib.h>         // this must be before glut.h
#include <glut.h>
#include "MyObjectManager.h"
#include "MyRectangle.h"
#include "MyCircle.h"

// Liitetään mukaan Luan kirjastot
extern "C" {
    #include "lua.h"
    #include "lauxlib.h"
    #include "lualib.h"
}

// muistivuotojen havaitseminen
#define _CRTDBG_MAP_ALLOC
#include <crtdbg.h>

// Osoitin dynaamiseen lua_State-rakenteeseen, joka toimii tavallaan tulkkina
// Luan ja C++:n välillä
lua_State *L;

MyObjectManager* objectManager = MyObjectManager::Instance();

void Update() {
    // päivitetään objektit
    objectManager->Update();

    // Työnnetään MyEngineLua_Update-funktion arvo pinoon
    lua_getglobal(L, "MyEngineLua_Update");
```

```
    // Kutsutaan edellä mainittua funktiota
    lua_call(L, 0, 0);

    // merkataan nykyinen ikkuna uudelleen piirrettäväksi
    glutPostRedisplay();
}

void Draw() {
    // puhdistetaan ruutu
    glClear(GL_COLOR_BUFFER_BIT);

    // piirretään objektit
    objectManager->Draw();

    // vaihdetaan puskurit
    glutSwapBuffers();
}

// Rekisteröidään luokat lua_Stateen
void RegisterClasses() {
    // Rekisteröidään MyObject
    Lunar<MyObject>::Register(L);
    // Rekisteröidään MyRectangle
    Lunar<MyRectangle>::Register(L);
    // Rekisteröidään MyCircle
    Lunar<MyCircle>::Register(L);
}

// Initialize everything
void Initialize() {
    // initialisoidaan lua_State. Nyt lua_State on vielä tyhjä eikä sisällä
    // mitään funktioita tai luokkia
    L = luaL_newstate();

    // ladataan Luan peruskirjastot lua_Stateen, jotta päästään käsiksi niiden
    // funktioihin
    luaL_openlibs(L);

    // Rekisteröidään luokat Luaan
    RegisterClasses();

    glClearColor(0.0, 0.0, 0.0, 0.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);

    // suoritetaan entry.lua tiedosto ja samalla sen funktiot tallentuvat
    // lua_Stateen, jotta niitä voidaan kutsua myöhemmin
    luaL_dofile(L, "lua_src/entry.lua");

    // Työnnetään entry.lua:ssa olevan MyEngineLua_InitializeGame-funktion arvo
    // pinoon. Kyseinen funktio toimii Lua esimerkkiohjelman käynnistyspisteenä
    lua_getglobal(L, "MyEngineLua_InitializeGame");
    // Kutsutaan edellä mainittua funktiota
    lua_call(L, 0, 0);
}

int main(int argc, char** argv) {
    // Initialisointi
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(600, 600);
```

```
    glutInitWindowPosition(200, 200);
    glutCreateWindow("MyEngine");
    Initialize();

    // Asetetaana päivitys-funktio
    glutIdleFunc(Update);
    // Asetetaan piirto-funktio
    glutDisplayFunc(Draw);
    glutMainLoop();

    // Siivotaan roskat eli kutsutaan Luan roskienkeräystä
    lua_gc(L, LUA_GCCOLLECT, 0);
    // Suljetaan Lua
    lua_close(L);

    // Muistivuotojen havaitseminen
    _CrtDumpMemoryLeaks();
    return 0;
}
```

Edellä olevasta koodista nähdään, että LuaBindin ja Lunarin main.cpp-tiedostot ovat miltei samanlaiset. Se suuri ero Lunarilla ja LuaBindilla kuitenkin on, että LuaBindissa ei tarvitse tehdä mitään muutoksia itse MyObject-, MyRectangle- ja MyCircle-luokkiin, kun taas Lunarissa muutoksia oli melko paljon. LuaBind myös tunnistaa funktioiden palautustyypit, esim. MyObject-luokan TargetReached-funktio palauttaa boolean-muuttujan, joten entry.lua:ssa voitiin vain kirjoittaa:

```
if ball['Circle']:TargetReached() then
```

Kun taas Lunarin kanssa TargetReached-funktion täytyi palauttaa int-muuttuja, joten kun boolean-arvo muutettiin int-arvoksi, niin täytyi kirjoittaa:

```
if ball['Circle']:TargetReached() == 1 then
```

Muuten entry.lua-tiedostot olivatkin identtiset.

Lunar on suorituskykyisin, mutta kehittymättömin valmis ratkaisu Luan liittämiseen ja soveltuukin hyvin hyvää suorituskykyä vaativiin yksinkertaisiin ohjelmiin. Lunar ei välttämättä ole paras ratkaisu jo valmiille C++-ohjelmille, koska Lunar vaatii joko vanhojen funktioiden ja luokkien muokkaamista tai uusien Lunarin kanssa yhteensopivien funktioiden tekoa.

4.6 LuaBridge

LuaBridge on pieni ja kevyt ratkaisu Luan ja C++:n yhteen liittämiseen eikä sillä ole riippuvuuksia muihin kirjastoihin. Sillä voidaan rekisteröidä luokkia, funktioita ja muuttujia Luan ja C++:n välillä. LuaBridge onkin hyvä vaihtoehto suurimpaan osaan tapauksista. *(Falco 2012.)*

LuaBridge on suunniteltu yksinkertaiseksi ja suorituskykyiseksi, joten luonnollisesti siitä puuttuu joitain ominaisuuksia. Puuttuvia ominaisuuksia ovat:

- Enum-muuttujat
- yli kahdeksan parametria funktiossa tai metodissa, tosin määrää on mahdollista kasvattaa
- ylikuormitetut funktiot, metodit ja konstruktorit
- globaalit muuttujat
- Standard Template Library (STL) container -tyyppien ja Lua-taulujen välinen automaattinen konversio
- Lua-luokkien perintä C++-luokista
- nil-arvon antaminen C++-funktiolle, joka vaatii osoittimen tai referenssin
- standardit container-tyypit, kuten esim. std::shared_ptr. *(Falco 2012.)*

LuaBridgen yhdistämisessä MyEngineen onnistuttiin helposti ja nopeasti. MyEngineen ei myöskään tarvinnut tehdä paljoakaan lisäyksiä. MyEnginen luokkiin ei tarvinnut tehdä mitään muutoksia ja main.cpp-tiedoston pääohjelmaankin vain vähän. LuaBridgen liittämisen jälkeen main.cpp oli miltei samanlainen kuin Lunar-ratkaisussa. Ainoina eroina olivat LuaBridgen main-tiedoston alkuun lisätty:

```
// Liitetään mukaan LuaBridge
#include "luabridge/luabridge.h"
```

sekä RegisterClasses-funktio, joka korvattiin seuraavalla:

```
// Rekisteröidään luokat lua_Stateen
void RegisterClasses() {
    luabridge::getGlobalNamespace(L)
        .beginClass<MyObject>("Object")
            .addConstructor <void (*) (void)>()
            .addFunction("SetPosition", &MyObject::SetPosition)
            .addFunction("MoveTo", &MyObject::MoveTo)
            .addFunction("Stop", &MyObject::Stop)
            .addFunction("Destroy", &MyObject::Destroy)
            .addFunction("SetColor", &MyObject::SetColor)
            .addFunction("TargetReached", &MyObject::TargetReached)
            .addFunction("SetSpeed", &MyObject::SetSpeed)
        .endClass()
        .deriveClass<MyCircle, MyObject>("Circle")
            .addConstructor <void(MyCircle::*)(float, float, float)>()
        .endClass()
        .deriveClass<MyRectangle, MyObject>("Rectangle")
           .addConstructor <void(MyRectangle::*)(float, float, float, float)>()
        .endClass();
}
```

LuaBridgessä ei ole mahdollista lisätä metodeja C++-luokille Lua-tiedostojen kautta, joten entry.lua:ssa olevaa Circle:Reset-metodia jouduttiin muuttamaan, jotta ohjelma toimisi odotetulla tavalla. Uusi Reset-funktio ottaa parametrina Circle-objektin ja resetoi sitten kyseisen objektin sijainnin, värin ja pysäyttää liikkeen. Reset-funktio näyttää tältä:

```
function Reset(circle)
    circle:SetPosition(0.5, 0.5)
    circle:SetColor(1, 1, 1)
    circle:Stop()
end
```

Sitä kutsutaan näin: `Reset(ball['Circle']).`

LuaBridge on pieni, nopea ja helppokäyttöinen ja soveltuu hyvin suurimpaan osaan ohjelmista. Toisin kuin Lunarissa, LuaBridgellä pystyy myös rekisteröimään helposti muuttujia Luan ja C++:n välille. LuaBridgellä ei tarvitse muokata valmiita funktioita ja luokkia vaan se osaa käsitellä niiden palautusarvojen ja parametrien muuttujien tyypit automaattisesti. LuaBridge sijoittuisikin ominaisuuksiensa perusteella johonkin Lunarin ja LuaBindin välille.

5 Five in a Row -pelin toteutus

5.1 Aikataulu

Pelin tekeminen aloitettiin 25.9.2013 ja lopetettiin 20.11.2013. Ottaen huomioon että työpäiviä oli viikossa vain neljä, maanantaista torstaihin, niin saadaan kokonaisajaksi noin 280 tuntia. Peliä tehdessä tuli huomattua kuinka helpoilta ja yksinkertaisilta vaikuttavat asiat vievät usein eniten aikaa.

Suurimmat yksittäiset aikaa vievät tekijät olivat tekoäly, nappuloiden animaatiot ja pelin grafiikoiden reaaliaikainen mukautuminen ruudun koon muutokseen. Näistäkin ominaisuuksista perustoiminnot saatiin valmiiksi melko nopeasti, mutta koska esimerkiksi tekoäly ei saa olla liian tyhmä ja animaatiot liian tylsistyttäviä, niin aikaa käytettiin huomattavia määriä niiden hiomiseen paremmiksi.

5.2 Suunnittelu

Pelin toteutus käynnistettiin palaverilla, jossa mietittiin pelejä, jotka tehtäisiin Star-Lua-alustalle. Palaverin tuloksena päätettiin, että ensimmäinen peli tulisi olemaan Five in a Row. Palavereja pidettiin joka keskiviikkona ja niissä seurattiin projektin etenemistä sekä suunniteltiin uusia ominaisuuksia. Näissä palavereissa oli mukana myös graafikoita, jotka suunnittelivat peleille ulkoasut. (Ks. kuvio 2.)

Kuvio 2. Five in a Row'n pelitila

Suunnitteluun ei tarvinnut käyttää kovin paljoa aikaa, koska ristinolla on pelinä kaikille tuttu ja melko yksinkertainen. Siinä on myös selkeät osat, joiden perusteella se oli helppo jakaa luokkiin. Suunnittelussa täytyi tosin ottaa huomioon suurempi konteks-

ti, nimittäin tämä peli luultavasti tulisi osaksi jotain suurempaa kokonaisuutta, joten täytyi miettiä myös miten peli siinä toimisi.

5.3 Rakenne

Luassa ei varsinaisesti ole luokkarakennetta ollenkaan vaan kaikki koostuu tauluista. Näiden taulujen avulla on kuitenkin mahdollista luoda rakenne, joka muistuttaa paljolti luokkaa. Five in a Rowissa tehtiin juurikin tällä tavalla. Pelissä on kaiken kaikkiaan kahdeksan luokkaa sekä itse pääohjelma, joka käyttää näitä luokkia ja hoitaa suurimman osan pelin logiikasta. Kaikki luokat ovat globaaleja, joten niiden nimiin on liitetty FiveInARow-etuliite, esim. FiveInARowGameView, jotta Luan nimiavaruudessa ei varmasti olisi kahta samannimistä muuttujaa. Selkeyden vuoksi tässä opinnäytetyössä luokkien nimissä ei kuitenkaan käytetä etuliitteitä.

FiveInARow-pääohjelma vastaa tapahtumien käsittelystä ja niiden lähettämisestä eteenpäin oikeille luokille. Se vastaa myös esimerkiksi pelin logiikasta ja pelaajien siirtojen lähettämisestä ja vastaanottamisesta.

Commons-luokka sisältää paikallisia muuttujia, joiden ei haluta olevan globaaleja ja joita käytetään useissa muissakin luokissa.

GameView-luokka vastaa pelitilasta ja siihen liittyvistä asioista, kuten tietojen välittämisestä eteenpäin GameBoard- ja TurnIndicator-luokille.

GameBoard-luokka vastaa pelilaudasta ja sen toiminnoista, kuten esimerkiksi tarkastuksesta onko joku voittanut pelin ja erinäisistä pelinappuloiden animaatioista. GameBoardissa myös luodaan pelilauta, joka koostuu GameCell-olioista.

GameCell-luokka vastaa pelilaudalla olevista soluista, jotka tarkoittavat tässä tapauksessa kohtia joihin pelinappulat on mahdollista sijoittaa. Jokaisella solulla on omistaja, joka määrittää kenelle solu kuuluu. Soluilla on myös kuva -muuttuja, joka on pelaajasta riippuen joko sininen tai oranssi pelinappula.

TurnIndicator-luokka vastaa kaikista vuoron vaihtumiseen ja ilmaisemiseen liittyvistä toimista, kuten jäljellä olevan ajan näyttämisestä tai ilmoituksesta kun pelaaja yrittää tehdä siirron toisen pelaajan vuorolla.

ResultView-luokka vastaa tulos näkymästä, joka sisältää huipputulos-listoja sekä mahdollisuuden ehdottaa revanssia tai sitten etsiä täysin uutta vastustajaa.

MysteryView-luokka vastaa pelin päätyttyä esiin hyppäävästä alueesta, jossa on eräänlainen arvoituslaatikko, josta pelaaja saa osittain sattumanvaraisen määrän kokemuspisteitä tai muita palkintoja. MysteryViewissä myös ilmoitetaan pelin voittaja.

Bot-luokka vastaa tekoälystä, kuten sen siirroista ja logiikasta.

5.4 Tekoäly

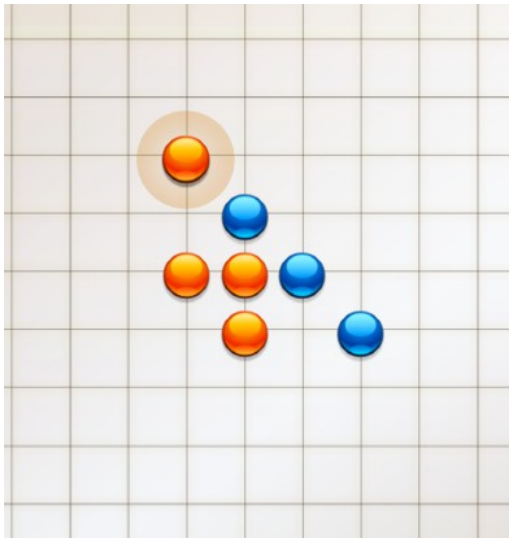
Five in a Rowissa on mahdollista pelata myös tekoälyä vastaan, vaikkakin peli on suunniteltu niin, etteivät pelaajat välttämättä huomaisi pelaavansa tekoälyä vastaan. Tekoälyn tarkoituksena ei tietenkään ole huijata pelaajaa vaan tarjota mahdollisuus pelata vaikka muita ihmispelaajia ei sillä hetkellä löytyisikään. Kyseinen tilanne on mahdollinen varsinkin juuri julkaistussa pelissä, jossa pelaajamäärät saattavat olla

vielä pieniä. Käytännössä peli tekoälyä vastaan on toteutettu siten, että kun peli on etsimässä vastustajaa pelaajalle ja silloin kun muita vapaita pelaajia ei löydy, niin peli asettaa tekoälyn vastustajaksi.

Tekoälyn vaikeusaste luokiteltaisiin luultavasti vähän keskitason yläpuolelle. Tekoäly onkin monimutkaisin asia koko pelissä ja sen tutkimiseen käytettiin jonkin verran aikaa. Yhdessä vaiheessa jouduttiin myös kirjoittamaan tekoälyn koko rakenne uusiksi, kun huomattiin ettei se palvellut kunnolla tarkoitustaan.

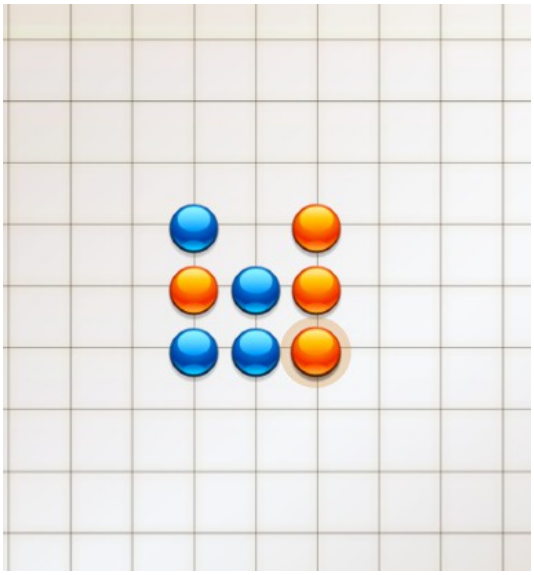
Tekoälyn logiikan taustalla on melko yksinkertainen painoarvoihin perustuva järjestelmä. Järjestelmässä käydään läpi jokainen vapaa napin sijoituspaikka ja annetaan sille painoarvo. Painoarvo annetaan joko suoraan laskemalla vapaan paikan ympärillä olevat napit tai sitten muutaman erityistapauksen perusteella. Erityistapaukset ovat tilanteita, joissa tekoälyn on pakko laittaa nappi tiettyyn paikkaan estääkseen ihmispelaajaa voittamasta tai voittaakseen itse. Näistä erityistapauksista mainittakoon pari tärkeintä.

Ensimmäisenä tapauksena on tilanne, jossa vastustajalla on molemmista päistä avoin kolmen napin jono. (Ks. kuvio 3.) Tässä tapauksessa tekoälyn on pakko asettaa nappinsa jompaankumpaan päähän jonoa.



Kuvio 3. Tekoäly estää sinistä vastustajaansa saamasta molemmista päistä avointa neljän napin jonoa

Toisessa erityistapauksessa vastustaja on saamassa seuraavalla siirrollaan vähintään kaksi kolmen napin jonoa, jotka ovat molemmista päistä avoimia. (Ks. kuvio 4.) Tämä tarkoittaisi sitä, että pelaaja voittaisi kahden siirron päästä, jos tekoäly ei estäisi sitä. Jos tekoäly ei havaitse vastustajan olevan voittamassa peliä, niin se sijoittaa nappinsa siten, että se itse saisi mahdollisimman hyvät voittomahdollisuudet.



Kuvio 4. Tekoäly estää sinistä vastustajaansa saamasta kahta kolmen napin jonoa

5.5 Ongelmat

Peliä kehittäessä oli joitain ongelmia, mutta ei mitään ylitsepääsemättömiä. Näistä merkittävimmät olivat yleensä ongelmia, jotka johtuivat huonosta ohjelman rakenteesta ja näin ollen myös rakenteen puutteellisesta suunnittelusta. Kyseisissä tapauksissa jouduttiin järjestelemään koodia uudelleen, joskus jopa suuriakin määriä.

Pieniä ongelmia aiheutui myös vaadittavien ominaisuuksien puuttumisesta. Tämä oli aivan ymmärrettävää, koska StarLua-rajapinta on vielä kehitysvaiheessa ja siihen lisäilläänkin ominaisuuksia sitä mukaa kun niitä tarvitaan.

6 Pohdinta

6.1 Luan ja C++:n liitäntä

Monimutkaisemmille matalan tason ohjelmointikielillä kirjoitetuille ohjelmille kannattaa tehdä korkeamman tason skriptauskieli. Se helpottaa ja nopeuttaa kehitystä huomattavasti. Hyvänä esimerkkinä toimii C++-pelimoottori. C++ ei ole mikään helpoin ohjelmointikieli, joten jos tarkoituksena on että pelimoottoria käyttävät muutkin kuin sen kehittäjä, niin kannattaakin nähdä vähän vaivaa ja liittää moottoriin jokin skriptauskieli, kuten Lua.

Parhaan Luaskriptin ja C++:n yhdistämistavan löytäminen projektille ei ole mikään helppo asia. Siinä täytyy ottaa huomioon monia asioita, kuten kuinka nopea ohjelman täytyy olla tai kuinka paljon muistia ohjelma saa viedä. Etsiessään oikeaa Luan liitäntäratkaisua projektille kannattaakin olla kärsivällinen. On hyvin mahdollista että ensimmäiseksi valittu ratkaisu ei sovikaan projektille. Seuraavissa kappaleissa onkin yritetty kiteyttää tämän opinnäytetyön johtopäätökset koskien Luan ja C++:n liittämistä toisiinsa.

LuaBind on hyvä ratkaisu silloin kun vaaditaan helppoa ja kehittynyttä C++:n ominaisuuksien käyttöä Luan kautta eikä välitetä ajotiedoston koon kasvamisesta.

LuaBridge on ominaisuuksiltaan LuaBindin ja Lunarin välistä, mutta se on kuitenkin paljon lähempänä Lunaria. LuaBridgestä puuttuukin useita LuaBindissa olevia ominaisuuksia. LuaBridgen käyttö on miltei yhtä helppoa kuin LuaBindin eikä kuitenkaan ole riippuvainen mistään suurista kirjastoista. LuaBridge onkin oivallinen vaihtoehto kun halutaan pitää ajotiedosto suhteellisen pienenä ja Luan liittämisprosessi yksinkertaisena.

Lunar on tutkituista ratkaisuista pienin ja nopein, mutta ominaisuuksiltaan vähäisin. Se vaatii luomaan C++-funktioista ja -luokista yhteensopivia Lunarin kanssa. Tämä on tosin suhteellisen yksinkertainen operaatio, kun siitä pääsee selville. Lunar soveltuukin erityisesti melko yksinkertaisiin projekteihin, joilta vaaditaan hyvää suorituskykyä ja pientä kokoa. Tällaisia projekteja ovat esimerkiksi mobiilisovellukset ja -pelit. Lunar soveltuu myös hyvin projekteihin, joihin halutaan rakentaa oma juuri sopiva Luan liitäntäratkaisu. Itse asiassa Lunarin osittainen tarkoitus onkin toimia pohjana, jonka päälle jokainen voi rakentaa tarvittaessa oman ratkaisunsa.

ToLua++ vaikuttaa lupaavalta, vaikka tässä opinnäytetyössä sen käyttöönotto ei onnistunutkaan. Kun ToLua++:n bugit saadaan korjatuksi niin se saattaa hyvinkin olla yksi parhaista ratkaisuista. Ainakin käyttäjien kokemusten ja ToLua++:n dokumentaation perusteella se vaikuttaa tehokkaalta ja monipuoliselta ratkaisulta.

6.2 Five in a Row

Five in a Row -pelissä onnistuttiin hyvin ja se toimiikin mainiona esimerkkinä Luan käytöstä C++-aplikaatiossa. Pelin kehityksessä alkuun pääseminen tapahtui nopeasti, jopa yllättävän nopeasti. Tästä voikin kiittää Star Arcaden pelimoottorille luotua StarLua-rajapintaa. Kyseinen StarLua-rajapinta toimiikin erittäin hyvänä esimerkkinä siitä, kuinka Luan avulla voidaan helpottaa ja nopeuttaa uusien pelinkehittäjien mukaan pääsyä.

Pelin toteutus ei kuitenkaan ole täydellinen. Jos pelin saisikin toteuttaa uudelleen niin erityisesti pelin rakenteesta tulisi parempi ja selkeämpi. Lua-ohjelmointikieli oli melko uusi tuttavuus pelin aloitushetkellä, joten sekin on varmasti vaikuttanut lopputulokseen. Toisaalta Luaa voidaan pitää myös syynä miksi peli valmistui niin nopeasti.

6.3 Tulevaisuus

Luan käyttö toiseen ohjelmaan sulautettuna skriptauskielenä tulee varmasti lisääntymään ja valmiiden liitäntäratkaisujen määrä kasvamaan. Luan suosion kasvaessa tässä opinnäytetyössä opitut asiat ovatkin entistä arvokkaampia ja tulevatkin luultavasti tarpeeseen tulevaisuudessa.

Five in a Row -peliin tullaan luultavasti tekemään pieniä muutoksia ja se julkaistaan jossain vaiheessa tulevaisuudessa.

Lähteet

About Star Arcade. 2013. Star Arcaden markkinointimateriaali. Viitattu 12.11.2013. http://www.star-arcade.com/corporate/index.php

C++. 2013. Wikipedia – vapaa tietosanakirja. Viitattu 11.11.2013. http://en.wikipedia.org/wiki/C++

Celes, W. & Manzur, A. 2003. toLua++ - Reference Manual. Viitattu 5.12.2013. http://www.codenix.com/~tolua/tolua++.html

Falco, V. 2012. LuaBridge 2.0 Reference Manual. Viitattu 9.12.2013. http://vinniefalco.com/LuaBridge/Manual.html#s1

Gomoku. 2013. Wikipedia – vapaa tietosanakirja. Viitattu 13.11.2013. http://en.wikipedia.org/wiki/Five_in_a_Row_(game)

Kostilainen, S. 2013. Wordcraft – Alustariippumaton sosiaalinen monipeli. Opinnäytetyö. Jyväskylän ammattikorkeakoulu, tekniikan ja liikenteen ala, ohjelmistotekniikan koulutusohjelma. Viitattu 12.11.2013. https://publications.theseus.fi/bitstream/handle/10024/54370/Kostilainen_Sami.pdf

Lerusalimschy, R. 2003-2004. Programming in Lua. Viitattu 12.11.2013. http://www.lua.org/pil/p1.1.html

Lua About. 2013. Lua-ohjelmointikielen viralliset kotisivut. Viitattu 11.11.2013. http://www.lua.org/about.html

LuaBind. 2005. LuaBindin dokumentaatio. Viitattu 4.12.2013. http://www.rasterbar.com/products/luabind/docs.html

Lua Frequently Asked Questions. 2013. Lua-ohjelmointikielen viralliset kotisivut. Viitattu 5.12.2013. http://www.lua.org/faq.html

Merrell, P. 2013. Where Lua Is Used. Artikkeli Luan virallisilla kotisivuilla. Viitattu 12-.11.2013. https://sites.google.com/site/marbux/home/where-lua-is-used#8S4UcLlroV5fq8i3WSheIA

Palozzi, L. 2003. Technical Note 5. Viitattu 6.12.2013. http://www.lua.org/notes/ltn005.html