Game Engine Programming

GMT Master Program
Utrecht University

Dr. Nicolas Pronost

Course code: INFOMGEP
Credits: 7.5 ECTS

Lecture #14

Scripting

Introduction

- Scripting is used to have code not hard-coded, modifiable in live without recompile
  - useful to model flexible AI
    - character personalities, behaviors
  - and adjusting gameplay (game logic content)
    - mission creation, dialogs, level design
- Scripts are satellites to the core engine
  - can be written in a different language (more accessible to non programmers)
  - run in safe environment (faults are not backfiring to the main application)
  - can be coded by different people (AI designers, content production people ...)
  - ➢ Good for team management and program security

Introduction

- Calling, interpreting and returning from script incurs a significant performance hit
- But the point is to add extensibility and flexibility to the engine, not speed
- Recent scripting languages have good performance and small memory footprint
  - Lua and Python

Introduction

- C++ code belongs to the core engine of the game
  – Everything that is CPU intensive should be implemented in the core engine
- Scripting code is best suited for gameplay
  – High-level logic and program flow mechanisms should be implemented in scripts

Introduction

- Three approaches
  - Creating your own scripting language in your game engine from scratch
  - Using embedded scripting language
  - Using socket interface scripting

Building a scripting language

- Three steps
  - Definition of the language syntax
    - what syntax for what features
  - Creation of the program loader
    - load the code into memory for execution
  - Execution of the program
    - can range from executing binary module to interpreting high-level language

Building a scripting language

- Parsing a simple language: a rule system example
  - composed of a list of rules
  - each rule consists of conditions and actions (similar to *if ... then ...*)
  - the run-time engine selects the first true rule and executes the associated action(s)

Building a scripting language

- 1st step is to define the syntax
  - tokens and expressions
  - example for a rule

```
defrule
    condition1
    condition2
    ...
=>
    action1
    action2
    ...
```

  - each rule can have multiple conditions (ANDed) and multiple actions (sequenced)

Building a scripting language

- Expressions are parsed by a grammar
  - For condition

```
condition -> float_function operator float | boolean_function
float_function -> DISTANCE | ANGLE | LIFE
operator -> GREATER | SMALLER | EQUAL
boolean_function -> LEFT | RIGHT
```

  - we can query Euclidean distances, angular distances to an enemy and life level
  - we can test if the enemy is on our left or right

Building a scripting language

- Expressions are parsed by a grammar
  - For action

```
action -> float_action float | unary_action
float_action -> ROTATE | MOVE
unary_action -> SHOOT
```

  - we can rotate and move
  - we can shoot at an enemy (*e.g.* the player)

Building a scripting language

- A rule system can now describe behaviors
  - Rule 1: avoid collision when too close

```
defrule
    DISTANCE SMALLER 5
    LEFT
=>
    ROTATE 0.01
    MOVE 0.1

defrule
    DISTANCE SMALLER 5
    RIGHT
=>
    ROTATE -0.01
    MOVE 0.1
```

Building a scripting language

- A rule system can now describe behaviors
  - Rule 2: shoot at the enemy

```
defrule
   ANGLE SMALLER 0.25
=>
   SHOOT
```

Building a scripting language

- A rule system can now describe behaviors
  - Rule 3: chase the enemy

```
defrule
   LEFT
=>
   ROTATE -0.01
   MOVE 0.1

defrule
   RIGHT
=>
   ROTATE 0.01
   MOVE 0.1
```

Building a scripting language

- The three rule systems are evaluated it that order
- When a rule is true, the corresponding actions are executed
- Enacts the entity to chase and shoot at player while avoiding collisions

Building a scripting language

- 2nd step is to create the program loader
  - Definition of a data structure to hold the rules (possible as syntax is simple and very regular)

```
typedef struct {
   int opcode;                 // ANGLE, MOVE, etc.
   int operation;              // SMALLER, etc.
   float param;
} fact;

typedef struct {
   std::list<fact> condition;
   std::list<fact> action;
} rule;
```

Building a scripting language

- The core program loads the script and sets up the list of rules
- Called a virtual machine

```
class vMachine {
        std::vector<rule> rules; // vector of rules
   public:
        void load(char * scriptFileName);
};
```

Building a scripting language

```
void vMachine::load(char *filename) {
    // assume the computation of number of rules in the file in numrules, and
    // the initialization of the vector of rules
    for (int i = 0; i < numrules; i++) {
        while (convert_to_opcode(readtoken(file)) != THEN) { // in defrule
                fact f;
                // read a condition
                char * stropcode = readtoken(file);
                f.opcode = convert_to_opcode(stropcode);
                switch (f.opcode) {
                        case ANGLE:
                                char * operation = readtoken(file);
                                f.operation = convert_to_opcode(operation);
                                // GREATER, etc.
                                f.param = atoi(readtoken(file));
                                rules[i].condition.push_back(f);
                                break;
                        // other cases (DISTANCE, LIFE, LEFT, RIGHT) ...
                }
        }
        // ...
```

Building a scripting language

```
        // ...
        // rule conditions ok, move on to actions
        while (!file.eof() && (convert_to_opcode(readtoken(file)) != DEFRULE))
        {
                fact f;
                // read an action
                char * stropcode = readtoken(file);
                f.opcode = convert_to_opcode(stropcode);
                switch (f.opcode) {
                        case ROTATE:
                                f.param = atoi(readtoken(file));
                                rules[i].action.push_back(f);
                                break;
                        // other cases (MOVE, SHOOT) ...
                }
        }
}
```

Building a scripting language

- We need to provide a routine to read token
  - strings or values separated by spaces and new lines, we can use operator >>
- *opcode* are defined as integer values (in an enum type for example)
- convert_to_opcode

```
int convert_to_opcode (char * opcode) {
    if (strcmp(opcode,"LEFT") == 0) return LEFT;
    // other cases ...
    if (strcmp(opcode,"=>") == 0) return THEN;
    return WRONG_OPCODE;
}
```

Building a scripting language

- 3rd step is to execute the script
- The virtual machine provides an execution function: run

```
class vMachine {
        std::vector<rule> rules; // vector of rules
   public:
        void load(char * scriptFileName);
        void run();
};
```

Building a scripting language

- Executing the program is just scanning the list of rules and applying the appropriate actions if conditions are fulfilled

```
void vMachine::run() {
    for (unsigned int r = 0; r < rules.size(); r++) {
        if (valid(rules[r])) { // evaluate conditions
            run_actions(rules[r]); // execute actions
            // break; to stop evaluating if rule found (optional)
        }
    }
}
```

Building a scripting language

- Two additional functions in the virtual machine
  - Evaluation of the conditions
  - Execution of the actions

```
class vMachine {
        std::vector<rule> rules; // vector of rules
  public:
        void load(char * scriptFileName);
        void run();
        bool valid (rule r);
        void run_actions (rule r);
};
```

Building a scripting language

- Evaluation of the conditions

```
bool vMachine::valid (rule r) {
    std::list<fact>::iterator pos = r.condition.begin();
    while (pos != r.condition.end()) { // for each condition
      switch (pos->opcode) { // what kind of condition
        case ANGLE:
         // compute angle (internal game code)
         if ((pos->operation == GREATER) && (angle <= pos->param))
           return false; // only return false as condition are ANDed
         if ((pos->operation == SMALLER) && (angle >= pos->param))
           return false;
         if ((pos->operation == EQUAL) && (angle != pos->param))
           return false;
        break;
        // other cases (DISTANCE, LIFE, LEFT, RIGHT) ...
      }
      pos.next();
    }
    return true;
}
```

Building a scripting language

- Execution of the actions

```
void vMachine::run_actions (rule r) {
    std::list<fact>::iterator pos = r.action.begin();
    while (pos != r.action.end()) {
         switch (pos->opcode) {
                  case ROTATE:
                           yaw += pos->param;
                           break;
                  case MOVE:
                           position.x += pos->param * cos(yaw);
                           position.y += pos->param * sin(yaw);
                           break;
                  // other cases (SHOOT) ...
         }
    }
}
```

Building a scripting language

- From here, we can add many rules to improve the AI behavior
- Game logic can be coded that way by defining over 100 facts

```
typedef struct { int opcode; int operation; float param;} fact;
```

  - example: Age of Empire
- The rule execution is still binary compiled code, so quite fast
  - Overhead from the loop (*run*) and switches (*valid* and *run_actions*)
  - The more time consuming the actions, the more negligible the overhead

Building a scripting language

- Parsing structured languages
  - Possible to parse code for more complex languages, but has its limits
    - difficult to handle hundreds of structures, function calls and symbols
  - Lexical scanners are then used to detect whether a token is valid or not
    - such as the Lex analyzer

Building a scripting language

- Context-free grammars allow to declare languages by using substitution rules
  - if statement in C

```
if_statement :
      IF '(' expression ')' statement
    | IF '(' expression ')' statement ELSE statement
;
```

  - and numeric expression

```
expression :
      NUMBER
    | expression opcode expression
    | '(' expression ')'
    ;
opcode :
     '+' | '-' | '*' | '/' ;
```

Building a scripting language

- When the grammar is defined, the input script is
  - parsed, converted to tokens and checked for syntactic correctness
  - Yet Another Compiler Compiler (Yacc)
    - generates the code to parse a grammar
- We finally decide what to do during parsing the script
  - interpret the script
  - generate binary code
  - convert to different format, ...

Embedded languages

- If you do not need specific functionalities, regular C-like scripting languages are available, called embedded languages
  - called from the host application (C++ game)
  - provide internal programming and API for host
- Two approaches
  - Designed to be embedded (Python, Lua)
  - Regular languages embedded using special tools (Java Native Interface JNI to execute Java from C/C++ applications)

Python

- Dynamic programming language
- High-level OO interpreted language
- Used in games for game logic and server control
  - Battle Field 2, Civilization IV, Freedom Force, Disney's Toontown, Frets On Fire, ...

Python

- Control flow (if, for, break, continue ...)
- Data structures (list, sequence, ...)
- Function
- I/O
- Error and exception
- Class
- Template
- Multi-threading
- Module organization
- *and more*

Embedding Python

- Download and install Python
  - http://www.python.org/
- In the VS project properties
  - include path: PYTHON_HOME/include
  - library path: PYTHON_HOME/libs
- Header file to include

```
#include "Python.h"
```

Embedding Python

- Initialize the Python interpreter

```
Py_Initialize();
```

- Run the script

```
// Giving directly the sequence of commands in parameter
PyRun_SimpleString("/*script commands/*");

// Referencing a script file
FILE * file = fopen(script_name,"r");
PyRun_SimpleFile(file,script_name);
```

- Finalize the Python interpreter

```
Py_Finalize();
```

Embedding Python

- Example

```
#include "Python.h"

int main () {

    Py_Initialize();

    PyRun_SimpleString("from time import time,ctime\n
                        print('Today is ',ctime(time()))\n");

    FILE * file = fopen("script.py","r");
    PyRun_SimpleFile(file,"script.py");

    Py_Finalize();
    return 0;
}
```

script.py

```
from time import time,ctime
print('Today is ',ctime(time()))
```

Embedding Python

- Exchanging data requires more code
  - Convert data values from C++ to Python
  - Perform a function call to a Python interface routine using the converted values
  - Convert the return data from Python to C++

Embedding Python

- Importing script (python module)

```
PyObject * imported = PyImport_Import(scriptModuleName);
```

- Defining the function to call

```
PyObject * function = PyObject_GetAttrString(imported,functionName);
```

- Defining the function parameters

```
PyObject * parameters = PyTuple_New(nbParameters); // here 2
PyObject * p1 = PyLong_FromLong(value1);
PyObject * p2 = PyFloat_FromString("10.9");
PyTuple_SetItem(parameters, 0, p1);
PyTuple_SetItem(parameters, 1, p2);
```

- Running the function

```
PyObject * result = PyObject_CallObject(function,parameters);
```

Embedding Python

- Error checking after each call
  - as you cannot assume anything from an external user (script programmer)

```
if (imported != NULL) ...                   // to check the file loading
if (function != NULL &&                     // to check if function exists
    PyCallable_Check(function))             // to check if function callable
if (p1 != NULL && p2 != NULL) ...           // to check if type conversion ok
if (result != NULL) ...                     // to check if call succeed
```

  - you can at anytime print the last entry from the Python error log

```
PyErr_Print(); // or access it by: PyObject* PyErr_Occured()
```

Embedding Python

- Reference counts
  - In C++, allocation/de-allocation are managed by new/delete operators
  - Python uses reference counting to avoid memory leaks
    - each object contains a counter which is incremented when a new reference to the object is created and is decremented when a reference to it is deleted
    - when counter reaches zero the object is de-allocated

```
Py_INCREF(x);    // to increase counter of x
Py_DECREF(x);    // to decrease counter of x
```

Embedding Python

- Example (without error checking)
  - calling a max scripted function

```
def scriptMax(a,b)                                          myMax.py
    c = 0
    if a > b : c = a
    else : c = b
    print(“Max value between ”,a,“ and ”,b,“ is ”,c)
    return c
```

Embedding Python

- Example (without error checking)

```
long int value1 = 2;
long int value2 = 10;

Py_Initialize();

PyObject * imported = PyImport_Import(“myMax”);
PyObject * function = PyObject_GetAttrString(imported,“scriptMax”);

PyObject * parameters = PyTuple_New(2);
PyObject * p1 = PyLong_FromLong(value1);
PyObject * p2 = PyLong_FromLong(value2);
PyTuple_SetItem(parameters, 0, p1);
PyTuple_SetItem(parameters, 1, p2);

// ...
```

Embedding Python

- Example (without error checking)

```
// ...

PyObject * callResult = PyObject_CallObject(function,parameters);
Py_DECREF(p1);
Py_DECREF(p2);
Py_DECREF(parameters);

long int maxValue = PyLong_AsLong(callResult);
Py_DECREF(callResult);

Py_DECREF(function);
Py_DECREF(imported);

Py_Finalize();
```

Embedding Python

- As long as the function signature does not change (module and function names, type and number of parameters and return value)
  - you can change the code of scriptMax
  - without compiling again
  - useful when someone else is coding external functionalities (game logic, AI ...)
  - only the interface with the C++ application is needed

Embedding Python

- Extending embedded Python
  - The Python interpreter might need access to functions in the main C++ program
  - The main program can provide an API for the Python interpreter
  - By creation of modules in main program

```
PyObject* PyModule_Create(PyModuleDef * module)
```

Embedding Python

- Example
  - we want to call the C++ function *getNumPlayers* from the script

```
import GameEngine                                              script.py
print("Number of players: ", GameEngine.getNumPlayers())
```

  - we need to import the module before the Py_Initialize()

```
// ...                                                          main.cpp
PyImport_AppendInittab("GameEngine", &PyInit_GameEngine);

Py_Initialize();
// call to script.py
```

Embedding Python

- Example

```
static PyObject * getNumPlayers (PyObject * self, PyObject * args) {
    if (!PyArg_ParseTuple(args, ":getNumPlayer")) return NULL;
    return PyLong_FromLong(GameEngine::getInstance()->getNumPlayers());
}

static PyMethodDef GameEngineMethods [] = {
    {"getNumPlayers", getNumPlayers, METH_VARARGS, "print #player"},
    {NULL, NULL, 0, NULL}
};

static PyModuleDef GameEngineModule = {
    PyModuleDef_HEAD_INIT, "GameEngine", NULL, -1, GameEngineMethods,
    NULL, NULL, NULL, NULL
};

static PyObject * PyInit_GameEngine (void) {
   return PyModule_Create(&GameEngineModule);
}
```

Embedding Python

- Extending embedded Python
  - When user types need to be exposed from the main program, use the boost-Python library

```
#include “className.h”
#include “boost/python.hpp”
#include “boost/ref.hpp”
#include “boost/utility.hpp”

BOOST_PYTHON_MODULE (moduleName) {
 class_<className, bases<baseClassName>, std::auto_ptr<className>>(“className”)
   .def(“memberFunction1”, &className::memberFunction1)
   .def(“memberFunction2”, &className::memberFunction2)
 ;
 implicitly_convertible<std::auto_ptr<className>,
                        std::auto_ptr<baseClassName>>();
}
```

Embedded C++ in script

- The complete reverse approach is often possible
  - main application is the script interpreter and not the C++ program
  - add the game engine functionalities to the script language
  - script interpreter provides a wrapper of another language / library
  - Example: Python-Ogre www.pythonogre.com
    - to run Ogre applications from Python interpreter
    - Ogre application (game components) can be compiled as Python dynamic library

Java scripting

- To use java functionalities with libraries and built-in routines as embedded language
- The choice between embedded oriented (Python, Lua) and language oriented (Java) scripting comes down to user preferences
- We need an additional tool to connect a java module to an application: the Java Native Interface (JNI)

Java Native Interface

- Specific set of calls within the Java programming language
- Bidirectional mechanism
  - A Java program can call C/C++ routines
  - A C/C++ program can access methods written in Java using the Invocation API

Java Native Interface

- Example
  - Calling a simple “Game Over!” Java program

```
public class JavaScript {                                    JavaScript.java
    public static void main(String[] args) {
          System.out.println("Game Over!");
    }
}
```

  - assuming we have the JavaScript.class file by calling javac JavaScript.java

Java Native Interface

- Example

```
#include <jni.h> // JNI calls

#define USER_CLASSPATH "." // where JavaScript.class is

int main() {
   JNIEnv * env;          // JNI environment
   JavaVM * jvm;          // Java virtual machine
   JDK1_1InitArgs vm_args;
   char classpath[1024];
   vm_args.version = 0x00010001;
   JNI_GetDefaultJavaVMInitArgs(&vm_args);

   // append where our .class files are to the classpath
   sprintf(classpath, "%s;%s", vm_args.classpath, USER_CLASSPATH);
   vm_args.classpath = classpath; // update the classpath

   // ...
```

Java Native Interface

- Example

```
    // ...

    // create the java VM
    jint res = JNI_CreateJavaVM(&jvm, &env, &vm_args);
    if (res < 0) { exit(1); } // can't create the VM

    jclass cls = env->FindClass("JavaScript");
    if (cls == 0) { exit(1); } // can't find the class we are calling

    jmethodID mid = env->GetStaticMethodID(cls,"main","([Ljava/lang/String;)V");
    if (mid == 0) { exit(1); } // can't find JavaScript.main

    jvalue * args; // function parameters
    env->CallStaticVoidMethod(cls, mid, args);
    jvm->DestroyJavaVM();
    return 0;
}
```

Java Native Interface

- jni.h is in the JDK include folder
- Function parameters can be specified

```
jstring jstr = env->NewStringUTF("parameter");
jvalue * args = env->NewObjectArray(1,env->FindClass("java/lang/String"),jstr);
env->CallStaticVoidMethod(cls, mid, args);
```

- Call function name has to be adapted to called function specification

```
CallCharMethod(...);
CallNonVirtualBooleanMethod(...);
CallStaticFloatMethod(...);
// ...
```

- Values can be returned from Java code

```
jfloat jf = env->CallFloatMethod(cls, mid, args);
```

Socket-based scripting

- Main program is server while script is client
- Calling functionalities from sockets
  - separate running environment (safer)
  - platform independent architecture
  - but not suited for time-critical tasks (socket access quite slow)
- Script module can be compiled
  - faster but lost of flexibility (same language)
- Script and application do not need to be physically on the same machine

Socket-based scripting

- Coding principle of the script module

```
// open socket to main game program
while (!end) {
   // read opcode from socket
   switch (opcode) {
        case QUIT:
                end = true;
                break;
        case MOVEPLAYER:
                // read optional parameters of opcode
                // ... move the player ...
                // take actions and send back data to main program
                break;
        // other opcode specific operations
   }
}
// close socket
```

Socket-based scripting

- Socket-based scripting is better designed to receive parameters, to perform local calculations, and to return a result
- Accessing and returning objects, structures and algorithms is again difficult
  - One solution consists in making them visible and callable from the script

End of lecture #14

Next lecture

Game engine standards