Script System Architecture for Quick Game Production

Rodrigo de Godoy Domingues, Alexandre Cardoso
Universidade Federal de Uberlândia
FEELT, FACOM, GRVA
Uberlândia, Brasil
rodrigo@facom.ufu.br, alexandre@ufu.br

Bruno Barbieri Gnecco
Corollarium Tecnologia
São Paulo, Brasil
brunobg@corollarium.com

Abstract

In this paper we propose the architecture of a software layer for Game Engines, either for simulation or entertainment applications, with the goal to define a configurable module that allow quick production and maintenance of scripts through modular code developed using a script language. Through a fixed manager, responsible for the administration of events and the structure of interaction, the script code is responsible for the definition of checkpoints, for the control of some behavioral aspects of the application models, for the definition and management of the contextual logic of the system and for the definition and internal logic of the script, generating and processing events. With this we were able to quickly define and produce script code and a detailed evaluation method of the users performance.

Keywords: Game Engines, Script Systems, Configurable System, Serious Games, Simulation, Performance Evaluation

I. INTRODUCTION

Simulations are now standard in many applications. They are less costly, less dangerous and permit users to be tested in controlled scenarios. Computers have aided immensely in this task [1].

As [2] demonstrated, game engines such as Unity [3] are now sophisticated tools, very useful for developing simulators. They provide most of the basic needs: high level manipulation of graphic elements, sophisticated rendering pipelines with shadows and effects such as as particles, physics and more.

When designing sophisticated simulators, however, one still has to deal with the states and logic of the objects being simulated, checkpoints, messages to the user and other aspects which are better handled by separate system that does not depend on the renderization and the low level events, such as input from user devices and others.

We propose in this paper an architecture based on a software layer that separates completely the logic for simulation. It only receives high-end events and handles the internal state of the simulation. With this we have a fixed manager kernel for any projects and we can develop new simulators much more quickly, easily and with fewer errors.

Since development is simpler and based on a script language and a basic data structure, it is also faster, demands less of the developers and easier to debug [4]. AI is notoriously difficult, for example, though it is completely detached from the rendering pipeline [5].

As an added bonus, since we establish a common structure and a series of checkpoints, we can easily evaluate user sessions, checking with the same code for any application whether they skipped steps, made errors or spent too much time in certain tasks.

II. ARCHITECTURE

Our architecture is based on two parts:

- the core, which handles the lower level events from the application, such as input from keyboard and other devices, the GUI, management of objects and states, network, loading scenes etc.
- the script system, which is based on a JSON description [6] and can run Python [7] code. This can run code outside the main Unity3D loop or its supported languages, and can be changed without recompilation of the main application. This makes development much easier and faster. Also, patches and upgrades are quite small in size, since the core is not changed, only a few small scripts. All the application logic is defined by these scripts.

Since Unity3D uses .NET, we can use any CLI language. Unity3D already supports 3 languages, C#, Boo and UnityScript. Unfortunately these languages are not completely dynamic within Unity3D; they have to be coupled to a GameObject. This couples the behaviour to the actual game objects, which is exactly what we wanted to avoid. Our script system is similar to the HTML/JavaScript model, in which the objects (HTML) and the code (JS) are independent, and linked through events or object searches at runtime, and not at compile time.

Therefore we needed another solution. We chose Python, through the IronPython implementation, due to its power, maturity and set of libraries. We can call Unity directly from Python wherever necessary, avoiding wrappers or other mechanisms if we need to manipulate objects or perform any changes in the scene. We can even generate code procedurally at runtime and execute it.

The script system is composed of the following parts:

- Events: whenever a registered event happens in the scene, it is sent through the script system to capture a list containing which elements respond to it. Then, each element is processed regarding the event occurred;
- Checkpoints: Checkpoints are named events that are stored for evaluation purposes. They can be sent to a server for processing and parametrize the evaluation system to generate the reports and
- Rules: This mechanism is responsible for storing and controlling the state flow of the application, according to the activities or interactions performed by the user. This component is responsible to validate a state transition, if it happens, according to an activity and is also responsible to manipulate the state of the game, ensuring the final state of the transition is valid and correctly setup.

These three components allow us to know exactly what happened and where (checkpoints), respond to game events ("if you click the green button, the machine is started") and keep track of the complete flow, following a script, which is a basic requirement for evaluation of how the user performed in the simulation.

III. RESULTS

We present on the sequence some screenshots, of an application and a console system using the script system to manipulate the scene and the game states, a sample code representing the system state configuration, in python and a sample code to the rules of state configuration and events.

The test system is a room containing 3 switches and a lever to be manipulated and start a cog. Once the three switches are on a determined position, the lever is released and the cog can be started.

In the fig. 1 the camera is positioned near the middle switch, which was identified as “chave2” and a checkpoint added so it can generate events. On click the checkpoint triggers the event with the same name and the process begins.

It is also possible to view the “in-game” console, which can be used to manually trigger events, such as the one the checkpoints triggers, meaning we can test the features of the game at any time.

The console is printing the state of the system previously to the switch switching.

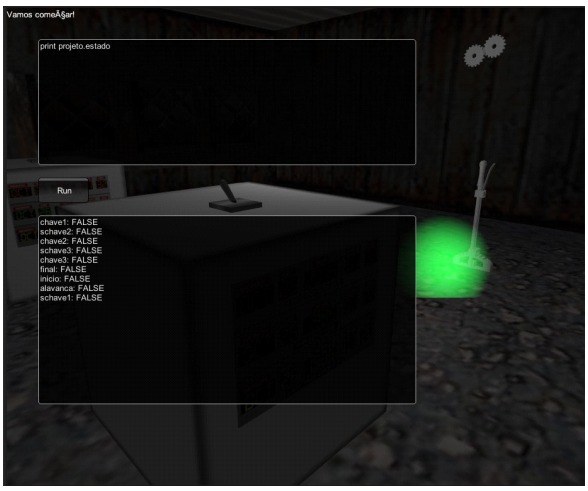

Fig. 1. Python console and the initial state. In green is a checkpoint activated by proximity.

In the fig. 2 the situation is similar to the one presented by fig. 1, however the switch was switched and then the state of the system printed again. Observe the value for chave2.

Fig. 2. Console and printing a state after an event on the lever

Fig. 3 is an excerpt of the state and event configuration. It uses the JSON notation so it is easy to parse and process. Note that the first configurations indicates the name of the system, the name of the script that represents or implements the state and the modules that python shall require.

Following the initial setup comes the states of the system. Each state has its rules for transitions and evaluation. A state can be simple, leading to another state, or scripted, which means it executes a code that will consult the overall state of the system, through the indicated script, and decide to which state transit.

```
{
    "sistema":"ExemploSala",
    "VersaoDeRegras":"1.0",
    "scripts":"salaRuleScripts.py",
    "modulos":["GameState","RulesEngine","json","ScriptEngine","UnityEngine"],
    "regras": {

            "inicio":{
                "notipo":"Simples",
                "tutorial":"Vamos comecar!",
                "requerido":true,
                "final":false,
                "sucesso":true,
                "left":0,
                "top":0,
                "transicoes":{
                    "login":{
                        "estado":"GameStart",
                        "valor":0
                        },
                    "error":{
                        "estado":"inicio",
                        "valor":0
                        }

                }
            },
            "GameStart":{
                "notipo":"Script",
                "tutorial":"",
                "requerido":true,
                "final":false,
                "sucesso":true,
                "left":0,
                "top":0,
                "transicoes":{
                    "script":"iniciaJogo()"
                }
            },
```

Fig. 3. The initial setup in the rules file. The configuration and the states definition.

Fig. 4 indicates the ending of the states configuration and the two final configurations, the events and the response of elements regarding an event.

```
            "finalizar":{
                "notipo":"Script",
                "tutorial":"",
                "requerido":true,
                "final":false,
                "sucesso":true,
                "left":0,
                "top":0,
                "transicoes":{
                    "script":"finaliza()"
                }
            }
    },
    "elements":{
        "chave":{
            "chave1":"projeto.estado.switchChave1()"
        },
        "chave2":{
            "chave2":"projeto.estado.switchChave2()"
        },
        "chave3":{
            "chave3":"projeto.estado.switchChave3()"
        },
        "Alavanca":{
            "switch":"projeto.estado.switchAlavanca()"
        }
    },
    "events":{

        "chave1":["chave"],
        "chave2":["chave2"],
        "chave3":["chave3"]

    }
}
```

Fig. 4. Final part of the rule file. The ending of the states configuration and the event and event on element configuration

Fig 5. presents the script that represents the overall state of the system. It is a python script with functions to manipulate the values itself as well as to manipulate Unity objects, triggering animations etc.

```
import UnityEngine
import GameState
import GrabMe

class Estado:
    '''
    Classe que armazena estados internos do roteiro
    '''

    #Dicionario para armazenar os estados
    genericState={
        'chave1':False,
        'chave2':False,
        'chave3':False,
        'alavanca':False,
        'inicio':False,
        'final':False,
        'schave1':0,
        'schave2':0,
        'schave3':0
    }

    def __init__(self):
        print "#########################Iniciando o estado do projeto##############################"
    ##Estado Liga Chave
    def ligaChave(self,c):
        if(c==1):
            self.genericState['chave1']=True;
            self.genericState["schave1"]="1"
        if(c==2):
            self.genericState['chave2']=True;
            self.genericState["schave2"]="1"
        if(c==3):
            self.genericState['chave3']=True;
            self.genericState["schave3"]="1"
        return self.genericState["schave1"]+self.genericState["schave2"]+self.genericState["schave3"]
    ##Estado Desliga Chave
    def desligaChave(self,c):
        if(c==1):
            self.genericState['chave1']=True;
            self.genericState["schave1"]="0"
        if(c==2):
            self.genericState['chave2']=True;
            self.genericState["schave2"]="0"
        if(c==3):
            self.genericState['chave3']=True;
            self.genericState["schave3"]="0"
        return self.genericState["schave1"]+self.genericState["schave2"]+self.genericState["schave3"]

    def switchChave1(self):
        if(self.genericState["chave1"]):
            UnityEngine.GameObject.Find("chave").animation.Play("desliga")
        else:
            UnityEngine.GameObject.Find("chave").animation.Play("liga")
```

Fig. 5. The systems overall state and scene manipulation functions

IV. CONCLUSION

We have shown our architecture for a simulator that decouples game rendering and devices to the actual game logic, event treatment and user evaluation. Such a system makes development of game logic quite easy, particularly for serious games and simulators.

We can change rules without recompiling the code -- in fact, we can reload the scripts in real time, during execution, so it is quite easy and fast to test changes and fix bugs. Debugging is also very easy: access to Unity allows use of its features (such as the console or showing data on the screen).

These are yet preliminary results and we will extend this architecture further.

V. BIBLIOGRAPHY

[1] GNECCO, B. B, DOMINGUES, R. G., BRASIL, G. J. C., DIAS, D. R. C., TREVELIN, Luis Carlos Ferramentas para desenvolvimento de jogos para ambientes livres. Tendências e Técnicas em Realidade Virtual e Aumentada. Cuiabá, p.104 - 120, 2013.

[2] Domingues, R. G., Battaiola, A. L.: Projeto de um framework para auxilio no desenvolvimento de aplicações com gráficos 3D e animação, São Carlos, 2003

[3] Unity3d Game Engine - www.unity3d.com, último acesso em 08/07/2013

[4] Varanese, A. & Lamothe, A.: Game Scripting Mastery, Premier Press, Dez. 2003

[5] Bourg, D. M., Seeman, G.: AI for Game Developers, O`Reilly Media Inc, Jul. 2004

[6] www.json.org, último acesso em 26/07/2013

[7] Linguagem Python - www.python.org, último acesso em 08/07/2013