Prototyping Decomposed Cloud Software: A Case Study on 3D Skeletal Game Engine

Minchen Li[1,2], Wei Cai[1], Ke Wang[3], Hong Ji[3] and Victor C.M. Leung[1]
[1]The University of British Columbia, Canada
[2] Zhejiang University, China
[3] Beijing University of Posts and Telecommunications, China
minchenl@cs.ubc.ca, weicai@ece.ubc.ca, {wangke, jihong}@bupt.edu.cn, vleung@ece.ubc.ca

***Abstract*—Recent studies have investigated a novel software paradigm that enables program decomposition and dynamic partitioning for cloud gaming system. Among the challenges for the proposed software architecture, to design and implement the decomposed software, which follows the platform's specification, is most critical in practice. In this work, we investigate a fundamental application in gaming scenario, the 3D skeletal game engine, which is often used to animate avatars. A graphics demo program is developed to demonstrate the feasibility of proposed system, while experiments have been conducted to show the efficiency of cognitive resource allocation. We also made contributions to seeking the appropriate design pattern in developing programs on our platform. In addition, we provide future visions on alternative implementations of some specific parts, which dedicates on improving the platform's compatibility.**

I. Introduction

Conventional cloud gaming systems that streams gaming video from cloud to players' terminals are suffering from the contradiction between high quality of service (QoS) and insufficient network bandwidth. To solve this problem, a decomposed cloud gaming platform [1] achieves flexible resource allocation by decomposing the game program into inter-dependent components that can be distributed to either cloud or terminal for execution. Accordingly, the paradigm of cognitive computing [2] has motivated us to build a context-aware platform that adapts cloud-terminal workload to the runtime environment to optimize the use of cloud, network and terminal resources while meeting the QoS objectives.

Intrinsically, the idea behind is to optimize the system by offloading graphic rendering modules from cloud to terminals, thus, to eliminate the video transmission overhead. In fact, a similar investigation has recently been proposed in [3], which ease the communication burden by decoupling the creation of rendering instructions from its execution and transmitting only small-sized rendering instructions over the Internet. In this work, we aim to develop a component-based 3D Skeletal Game Engine using JavaScript on our platform to evaluate the cognitive dynamic allocation performance. The program is a Skeletal Animation Editor. Its small scale memory needs will be good for our browser based program and our Node.js developed platform, and its high interaction would make our test and demonstration both interesting and convincing.

The remaining sections of the paper are organized as follows. We review related work in Section II and present the proposed program design in Section III. Then, in Section IV, we discuss the challenges of implementing such a multi-component program and provide the first-hand solutions for them. Implementation screenshots are illustrated in Section V. We further conduct experiments on cognitive cloud gaming platform to demonstrate the efficiency of cognitive engine in Section VI. Section VII concludes the paper and provide future visions on improving the design of the platform.

II. Related works

A. 3D Skeletal Animation

A skeletal animation consists of a skin mesh and an associated bone structure (skeleton). Moving a bone will move the associated vertices of the mesh, which exactly as what happens in reality.[4] The most popular technique to deform the skin according to the skeleton is the well-known Linear Blend Skinning method (LBS)[5][6]. It uses a 4×4 matrix to represent the transformation. Each bone in the skeleton has a skinning weight on each vertex of the mesh. LBS derives the transformation of a vertex by blending the transformation of the bones linearly according to the skinning weights. It can efficiently generate visually acceptable animation. Here we implement the basic Rigid Skinning method[6], which means each vertex will only be driven by one bone. And we use cuboids to construct the skin mesh.

B. Program Decomposition

There are basically two methods to decompose a game program, Fine-Grained Decomposition and Coarse-Grained Decomposition. Fine-grained decomposition segments the whole game program into a huge quantity of tiny components. Therefore, it leads to more opportunities in seeking optimal component allocation solution. But it's limited in state migration problem. A coarse-grained decomposition partitions the game program into a number of functional-independent and stateless components. The native states of the components are always invisible to each other, which eliminates the state transfer problem in fine-grained decomposition. However, the coarse-grained decomposition results in fewer components, which might affect the effectiveness of cognitive resource allocation. In addition, it is relatively difficult for game developers to write such program, since the components are strong coupling to each other. [7]

III. Program Design

A. Function Description

The editor has two modes - Display Mode and Edit Mode. In display mode, user can watch the animations and change the view with mouse drag or finger touch. In Edit Mode, user can edit the animation by adjusting the posture of the character model in the middle key frame. Other non-key frames will be automatically generated via frame interpolation.

B. Architecture Design

The consistency, stability, availability and portability of our program are very important because it is a web app. So we designed it based on the MVC(Model-View-Controller) architectural pattern, which also made the decomposition easy and clear (Fig. 1).

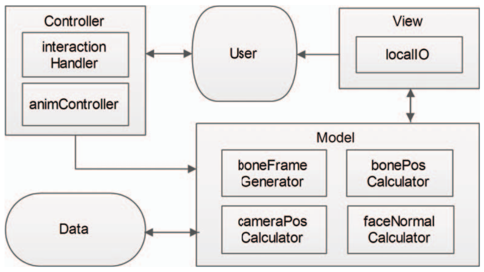


Fig. 1: MVC with Component View

The central component of MVC, the model, captures the application's behavior in terms of its problem domain and directly manages the application's data, logic and rules, independent of the user interface.[8] A view can be any output representation of information, such as the GUI Panel and 3D Scene here. The controller accepts input and converts it to commands for the model or view. In addition to dividing the application into three kinds of components, the MVC design defines the interactions between them: [9]

1) A controller sends commands to the model to update the model's state. It also sends commands to its associated view to change the view's presentation of the model.
2) A model notifies its associated views and controllers when there has been a change in its state. This notification allows the views to produce updated output, and the controllers to change the available set of commands.
3) A view requests information from the model that it uses to generate an output animation frame to the user.

C. Flow Control

There are 3 kinds of control flows in our Control Flow Diagram (CFD). The yellow arrows represent the initialization control flow, which is followed only in the initialization state. The blue arrow pairs represent the conditional call and back control flow, they are only followed when a user input appears. Once the handling process finishes, the control will immediately turn back to where it was. The green arrows sketched the main loop, which is followed continuously. It will

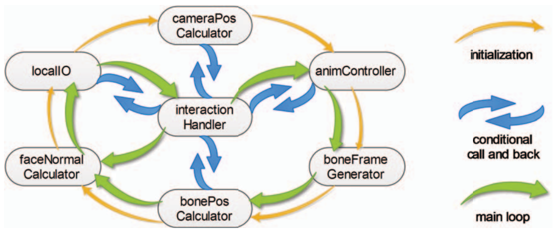


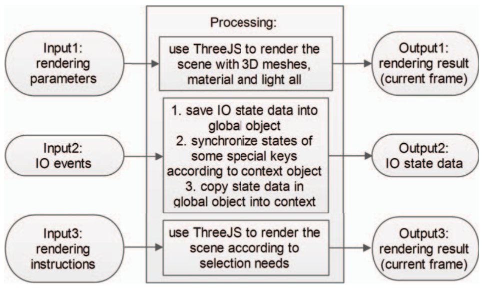
Fig. 2: Control Flow Diagram

pass through 3 components in Edit Mode and 6 components in Display Mode respectively. The animController and localIO component are fixed, so we only got 2 and 4 free components in each mode.

D. Component Design

By using Three.js[1] as the basic render engine and implementing the skeletal animation algorithms ourselves, we divided our program into 7 components as shown in Fig. 1.

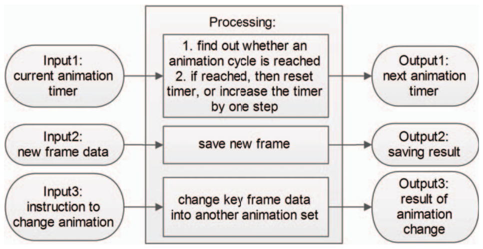
The localIO (Fig. 3) component is fixed on the client for Three.js and some browser related variables. It mainly does the rendering works and help synchronizing the I/O states.



Processing:
Input1: rendering parameters
use ThreeJS to render the scene with 3D meshes, material and light all
Output1: rendering result (current frame)
Input2: IO events
1. save IO state data into global object
2. synchronize states of some special keys according to context object
3. copy state data in global object into context
Output2: IO state data
Input3: rendering instructions
use ThreeJS to render the scene according to selection needs
Output3: rendering result (current frame)

Fig. 3: The IPO diagram of localIO

The animController(Fig. 4) component is fixed on the server because the FPS recording process inside must be executed on the server. It controls the animation timer and the content of the animation.



Processing:
Input1: current animation timer
1. find out whether an animation cycle is reached
2. if reached, then reset timer, or increase the timer by one step
Output1: next animation timer
Input2: new frame data
save new frame
Output2: saving result
Input3: instruction to change animation
change key frame data into another animation set
Output3: result of animation change

Fig. 4: The IPO diagram of animController

The rest of the components are all free. They all take their input data to do one specific computation respectively, and then output the results. Since their IPO(Input-Processing-Output) diagram are quite simple, we will not show them here.

[1]http://threejs.org/

IV. Challenges and Solutions

A. *Use of Global Variable*

Our platform currently provides two mechanisms for global variable among multiple components. However, neither of them seem to be perfect.

One is to save global variables into a JavaScript object named ”global” whose lifetime spans the whole runtime of the executing program. While this can only make variables global on either client or server, not both. So two components running on different sides could not share any variables, and components will lose their variables due to client-server migration. Another is to pack global variables into a JavaScript Object Notation (JSON) object named ”context” to be transmitted to the next executing function during each function jump. But JavaScript variables with member functions could not be represented in JSON form. Then most of the components using 3rd party objects must be fixed on either side to apply the former approach, making no contribution to our partitioning optimization. What's more, during each function jump, the program must transmit the object ”context” containing all the global variables including those do not need to be synchronized.

Our solution is to use only one 3rd party library (Three.js) for basic rendering tasks, saving all the related objects into ”global”, and implementing other algorithms ourselves to reach as many free components as possible. This scheme just satisfied the local rendering technique [10]. To make our platform more compatible with 3rd party libraries, we can make the ”global” object consistent between client and server by adding a synchronizer. While this may influence the cost evaluation of optimization, thus ruining the current theory and the cognition of our platform. Besides, to compress and decompress ”context” object for every transmission could minimize the time cost of function jumps.

B. *Synchronization of I/O State*

In our program, one single I/O event may be handled by multiple components, so the events received by localIO should be saved into its ”context” object to be transmmited to the sequent components. However, due to the asynchronization between the I/O event and the program's main control flow, there might be a WAR(Write after Read) data hazard [11]. If the I/O event appears during the execution of other components, the I/O state information received by localIO would be overwritten by the ”context” object of the former component after the function jump, thus ignoring the I/O request without handling.

To avoid WAR data hazard, we save the I/O event's information into ”global” when it appears. Then after the control transferred to localIO with the new ”context” object, we copy the I/O state information from ”global” object into this ”context” object, then these I/O state information could be transmitted forward to the following components to trigger the processing functions, which also synchronizes the I/O handling process and the execution of the program. Note that we only update the I/O state for those which just triggered. We don't want to overwrite other I/O states, which might have been reset in the processing components to avoid reprocessing. Since I/O operation is a necessity in browser based games, it would be better if our platform could provide I/O handling APIs for the programmers. So that they can be free from I/O state synchronization and can focus on their games.

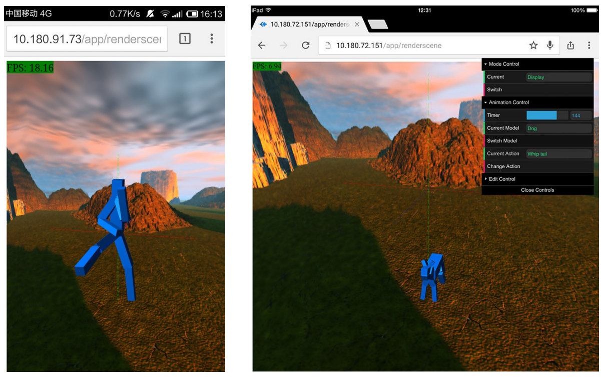
C. *Implementation of Main Loop*

In 3D animations, there will be a main execution loop to continuously get user input and process it to generate the new image frame back to user. In our program, this main execution loop is implemented by periodically transfering control among multiple components, as shown in our CFD in Fig. 2. But due to the runtime environment of browser, the all client side allocation will crash the browser if there's no delay in the loop.

Our solution is to fix animController on the server. Then we would never encounter the situation when all components are on the same side because localIO have already been fixed on the client. We didn't use the setTimeOut() function as the usual cases because it will influence our QoE measurement. The browser won't get blocked within this approach because there are always some delays during the client-server transitions, providing the browser with enough time to finish its built-in tasks.

V. Implementation Screenshots

We here focus on cross-platform. A laptop client with Chrome running on Windows 8.1 can reach more than 50 FPS displaying the default dog animation. Fig. 5 shows mobile client situation. The cellphone has quadruple 1GHz core and 1GB memory, reaching 10+ FPS in dog animation and 20+ FPS in human animation. The iPad has some trouble with its OS version, it only reaches less than 10 FPS.



(a) Xiaomi (b) iPad

Fig. 5: Chrome in Mobile Devices

VI. Experiments and Results

Our platform seeks the best QoE by dynamically allocating program components according to real-time device and network configurations. To evaluate its performance, we

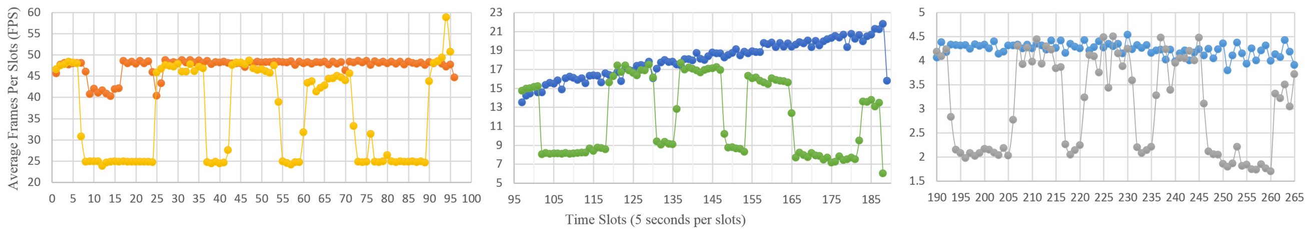


Fig. 6: Experiment Results

conducted the experiment under Display Mode to compare the FPS between 2 specific situations during 24 minutes. In the first situation, we let our platform automatically allocate the components, and we divide the time into 3 8-min stages, setting their bandwidth to 1000KB/S, 500KB/S and 100KB/S respectively. In the second situation, we also divide the time averagely into 3 parts with the bandwidth decreases from high to low just as the first situation, while we manually ensure every allocation occur 30 seconds in each part.

Since we have 4 free components in the main loop under Display Mode, there exists $2^4 = 16$ allocations. We need to switch to these allocations manually as fast as possible to avoid recording dirty FPS of other allocations. Hence, we use Gray Code[12] to arrange our switching sequence, so that we can switch to every next configuration by just one click.

We used 2 computers in the same LAN to conduct the experiment with NetBalancer[2] running on the server to limit the bandwidth of Node.js[3] process, so as to control the network bandwidth. The computers are both with Intel(R) Core(TM) i7-4770 CPU and 8.00 GB memory (RAM), 64-bit Windows 7. The browser on the client is Chrome.

Fig. 6 shows our results. Each point in the chart represents the average FPS in a 5-sec time slice. The curves with orange, deep blue, and light blue color show the data in the first situation where we let our platform automatically allocate components. The curves with yellow, green, and gray color show where we switch the allocation manually. We can easily and clearly find out that, except the inception stage and the 94th-96th time slice FPS, the cognitively automatic allocation always reaches the optimized FPS. Besides, we derive very similar patterns from the three iteration schemes: the 1st, 5th, 6th, 8th, 9th, 11th, 12th, 16th configurations all reach the optimized FPS, and the rest all reach the worst case FPS. And the worst case FPS value is approximately half of the optimized FPS value. Based on the analysis of our CFD(Fig. 2), we figured out that all worst cases have 4 client-server transitions in one main loop, and all optimized cases have only 2 transitions. This means the time cost of data transmission dominated our program.

VII. Conclusions and Future Works

From our experiment and analyses in the previous chapter, we conclude that in order to better serve the 3D cloud games nowadays, the computation of CG primitives, such as mesh vertex coordinate, needs to be computed locally to prevent frequent massive data communication. Only the components with computations that are not directly related with the CG primitives, i.e. a physics engine, could be deployed on the server. For our future works, we should try to improve the compatibility of our platform, so that we can deploy more 3D games with various 3rd party libraries conveniently to further research this emerging trend.

[2]https://netbalancer.com/
[3]http://nodejs.org/

VIII. Acknowledgements

This work was supported by a Mitacs Globalink Internship, a University of British Columbia Four Year Doctoral Fellowship, the Canadian Natural Sciences and Engineering Research Council under grant STPGP 447524-13, and the Fundamental Research Funds for the Central Universities of China.

References

[1] W. Cai, C. Zhou, V. Leung, and M. Chen, "A cognitive platform for mobile cloud gaming," in *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, vol. 1, pp. 72–79, Dec 2013.
[2] D. S. Modha, R. Ananthanarayanan, S. K. Esser, A. Ndirango, A. J. Sherbondy, and R. Singh, "Cognitive computing," *Commun. ACM*, vol. 54, pp. 62–71, Aug. 2011.
[3] S. Gorlatch, D. Meilaender, F. Glinka, W. Zhang, and X. Liao, "Bringing mobile online games to clouds," in *Proceeding of 33rd IEEE International Conference on Computer Communications*, INFOCOM2014, 2014.
[4] Newacct, Sp4cerat, Pieman, Codehead, and Rcrmn, "OpenGL Tutorials: Basic Bones System." Website, Mar. 2014. http://content.gpwiki.org/OpenGL:Tutorials:Basic_Bones_System.
[5] N. Magnenat-thalmann, R. Laperrire, and D. Thalmann, "Joint-dependent local deformations for hand animation and object grasping," in *Proceedings on Graphics interface*, pp. 26–33, 1988.
[6] A. Jacobson, Z. Deng, L. Kavan, and J. Lewis, "Skinning: Real-time shape deformation," in *ACM SIGGRAPH 2014 Courses*, 2014.
[7] W. Cai and V. Leung, "Decomposed cloud games: Design principles and challenges," in *Multimedia and Expo Workshops (ICMEW), 2014 IEEE International Conference on*, pp. 1–4, IEEE, 2014.
[8] S. Burbeck, *Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC)*. ParcPlace System, Inc., 1992.
[9] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, "Pattern-oriented software architecture: a system of patterns," *Computacin y Sistemas*, 1996.
[10] W. Cai, M. Chen, and V. Leung, "Toward Gaming as a Service," in *IEEE Internet Computing Magazine*, pp. 12–18, IEEE Computer Society, May 2014.
[11] E. Nurvitadhi, J. Hoe, T. Kam, and S. Lu, "Automatic pipelining from transactional datapath specifications," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*, pp. 1001 – 1004, 2010.
[12] R. Doran, "The gray code.," *Journal of Universal Computer Science*, vol. 13, no. 11, pp. 1573–1597, 2007.