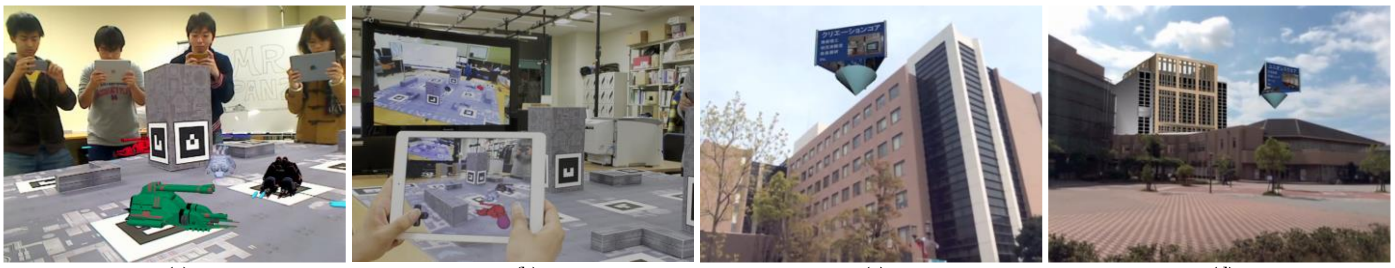
A Distributed Framework for Creating Mobile Mixed Reality Systems

Tsubasa Ogino*, Yuki Matsuda*, Le Van Nghia*, Daisuke Kawabata*,
Kento Yamazaki*, Asako Kimura*, and Fumihisa Shibata*
*Graduate School of Information Science and Engineering, Ritsumeikan University


(a) (b) (c) (d)
Fig. 1. Scenes of applications developed with our proposed framework. (a) and (b) are scenes of MR Panzer. (c) and (d) are scenes of BKC Guide.

***Abstract*—This paper describes development of a distributed framework for creating mobile mixed reality (MR) systems. The goal of the framework is providing the same MR space for a variety of mobile devices which connect via wireless network. This paper discusses a design policy of our framework, system architecture for supporting diverse mobile devices, including an easy-to-implement script language for developing applications. We implemented our framework based on the proposed design and developed trial applications using our script language. As a result, we confirmed that applications are easily developed using our framework.**

Keywords—mixed reality, augmented reality, mobile devices, distributed framework

I. Introduction

Recently, mobile devices such as smartphones have rapidly spread, and mixed reality (MR) systems on mobile devices have been drawing attention. However, if we keep in mind the spread of this type of system, it is inefficient to develop each application individually from scratch. Therefore, an easy way to develop such MR applications is imperative to the further progress of mobile MR systems.

Bauer et al. proposed a framework for wearable computers, called DWARF (Distributed Wearable Augmented Reality Framework) [1]. The main aim of DWARF is to enhance reusability of systems based on the idea of modularizing AR functions. Piekarski et al. proposed software architecture for developing AR applications, called Tinmith-evo5 [2]. It has a high level of rendering component, a tracker to estimate the position and orientation. However, it is not designed for current mobile devices, such as smartphones and tablets. MacIntyre et al. proposed Argon AR Web Browser [3]. This is an architecture that is focused on the cooperation in WWW services and AR. Schmalstieg et al. proposed a framework for handheld devices, called Studierstube ES [4]. This does not consider the low-performance mobile devices such as cellular phones. Metaio releases Junaio [5], which is a mobile augmented reality browser, and AREL [6], which is a JavaScript binding, for developing mobile AR applications. However, they adopted a stand-alone architecture, so they do not support to share the same MR space with multiple devices. Recently, the number of software for game developments has increased. Some of them support AR functions. For example, Unity released by Unity Technology [7] has APIs for creating AR applications. However, it is required to implement the code for networking if the developers want to create the AR application which synchronizes virtual contents among mobile devices.

In addition, outstanding technological advances can be expected due to the rapid development of mobile devices including performances and styles. It is therefore preferable to implement such a framework that could absorb variety of future innovations.

From this viewpoint, we designed and implemented a distributed framework for mobile MR systems, based on three demands.

(1) The contents of motion in MR space can be synchronized and shared with multiple mobile devices
(2) Differences in types of mobile devices or perfomances can be absorbed
(3) Application developers can create mobile MR systems easily

II. Distributed Framework

A. Design Policy

In our study, we aim to achieve a framework that is capable of changing the style of information presented depending on the performance of mobile device. We assume that multiple users holding different mobile device working together in the same space.

B. System Architecture

We design the system architecture of our framework to meet three needs described above (see Fig. 2).

This system is composed of MR Server, Streaming Server, Mediator, Client, and Thin Client. We adopt the client-server architecture to meet (1). Client and Mediator send contents' information to MR Server, and MR Server receives them. MR

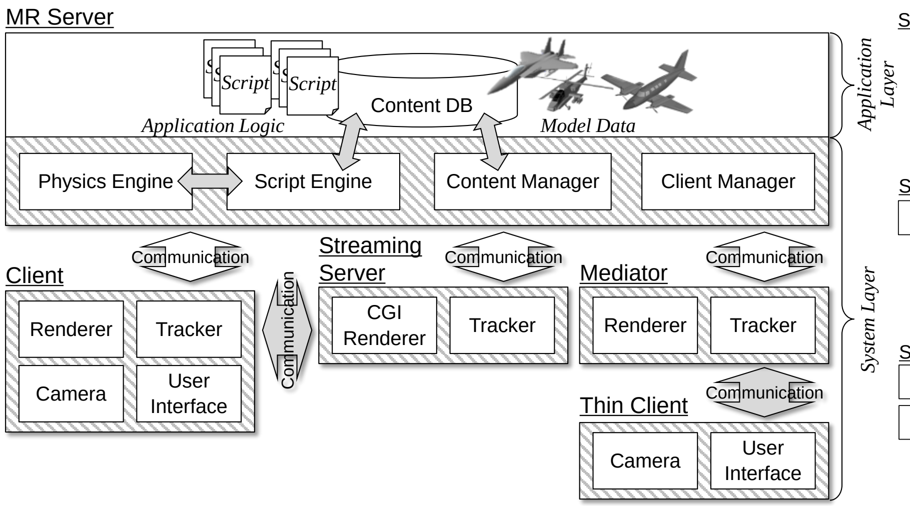


Fig. 2. System Architecture of our proposal framework

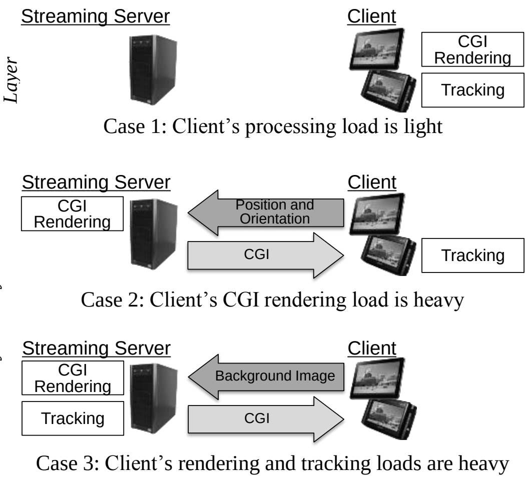


Fig. 3. Cooperation of Streaming Server and Client

Table 1. Properties of motion data

Type and name	Meaning
`int state`	play (0), pause (1), stop (2)
`int loop`	loop playback counts
`float speed`	playback speed

Table 2. Properties of a virtual object

Type and name	Meaning
`int id`	ID
`int classId`	class ID
`Position3D pos`	3D position
`Orientation3D ori`	3D orientation
`Scale3D scale`	3D scale
`int presen`	visible (0), translucent (1), hidden (2), invisible (3)

Table 3. Properties of a client

Type and name	Meaning
`int id`	ID
`int classId`	class ID
`Position3D pos`	3D position
`Orientation3D ori`	3D orientation

Server manages all contents and updates this information using a script engine. The script engine is a module to analyze and run scripts. In order to meet (2), we employed Streaming Server and Mediator. In our framework, mobile devices are divided into Thin Client and Client depending on their performances or intended uses. Thin Client is a low-performance mobile device such as cellular phones. Therefore, Mediator consistently does both rendering and tracking instead of Thin Client. Client is a high-performance mobile device such as smartphones and tablets. As shown in Fig. 3, Streaming Server renders computer generated image (CGI) and tracks according to Client's states. Client's rendering and tracking loads dynamically change using Streaming Server. In case 1, Client tracks and renders MR Image by itself. In case 2, Client sends its position and orientation to Streaming Server. Streaming Server renders CGI using received information. In case 3, Client sends a background image to Streaming Server. Streaming Server tracks using received information and renders CGI. In these two cases, Streaming Server sends compressed CGI to Client, and Client renders a MR image with it. Client's rendering and tracking loads are estimated based on each processing time. To meet (3), functions are separated into Application Layer and System Layer to enhance the reusability of System Layer. Furthermore, we designed our own script language to facilitate the content manager.

C. *Content Control Mechanism*

In order to develop MR application easily, it is important to simplify the placement of contents in virtual space. This means that the mechanism to easily control contents' position and orientation in virtual space is required. We call it content control mechanism, and we designed it with an emphasis on three points.

(1) Application developers can control complicated motions of contents.
(2) Application developers can alter motions of contents such as users' intercations.
(3) Application developers can easily develop a program code to control contents.

In Fig. 2, contents control mechanism includes a physics engine, a script engine, a content manager, and a client manager. The contents controlled with our proposed framework consist of clients that users hold and virtual objects such as CG. Clients are controlled with the client manager. Virtual objects are controlled with the content manager. Virtual objects consist of more than one component. The components include 3D CG, a sound, a skeletal animation. The skeletal animation consists of 3D CG model and the motion made by dedicated software. The skeletal animation has properties to control itself with scripts (see Table 1). We have considered "play," "pause," and "stop" as the minimum requirement to control the skeletal animation. In additional, there are "loop playback counts" and "playback speed" as properties.

Virtual objects have properties (see Table 2), which are updated by a script language to control virtual objects. Furthermore, application developer can deploy transparent virtual objects to realize hidden surface removal and collision detection. Clients also have properties (see Table 3), which are used in the scripts.

D. *Connection and Interpolation*

Contents control mechanism is executed on MR Server. Clients show the MR space to users, so MR Server needs to send MR information to clients according to the interval for updating the screen of the client. However, it is difficult to connect between MR Server and multiple clients constantly because of data size and communication bandwidth. In our study, MR space is updated with the script engine on MR Server constantly about 30 times per second, and Clients receives information of MR space from MR Server at a constant interval *ΔT*. While waiting for next connection, Clients display CG objects by interpolating between last and current information received from MR Server.

We employ Bezier curve for interpolation (see Fig 4). In order to smoothly reconstruct motion of CG object on the client, by the current and previous positional information P_t and $P_{t\text{-}3dt}$ (Note that ΔT=3dt) sent from the MR server, the system generates control points $P_{t\text{-}dt}$ and $P_{t\text{-}2dt}$ from the velocity vector V_t and $V_{t\text{-}3dt}$. In this case, a point at where the velocity vector changes rapidly, such as a ball bounce, is defined as cusp, and two curvilinear intervals including the cusp are interpolated.

E. *Time Synchronization Mechanism*

We consider communication delay between MR Server and Clients when application developers use sounds and animation contents. The communication delay of each Client is different, so each client receives contents' information from MR Server at different time. Due to this time difference, each Client starts playing sound and animation at different time. Therefore, we designed a mechanism to synchronize the starting time of interpolating, playing sounds, and animations. MR Server and Clients synchronize application time when each Client starts. Time synchronization is carried out by measuring round-trip communication time. In order to eliminate the deviation between the Clients caused by the communication delay, we set a time of delay between the Client and the MR Server as *delay* [ms], which is uniformed in all the Clients. As shown in Fig. 5, Clients render contents back to delay with interpolation. In order to unify time to play sound and animation with all the Clients, MR Server additionally sends the time when the sound and animation started to play. Clients begin to play when it reaches the starting time received from the MR Server. If this received time is beyond the starting time, the Clients calculate the difference between the current time and play from the middle. With this mechanism, all Clients can play simultaneously regardless to the communication delay.

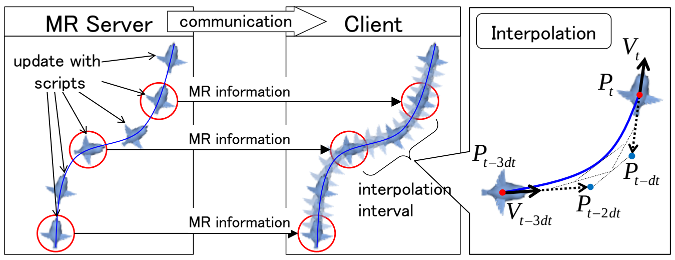


Fig. 4. The outline of interpolation

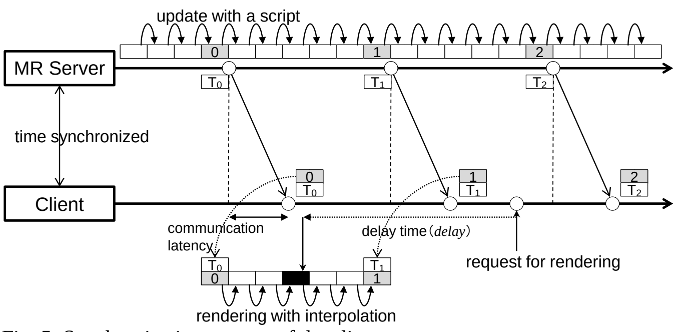


Fig. 5. Synchronization process of the client

F. *Script Language*

Our script language is designed like object-oriented language, such as Java and C++. Even when application developers use this language for the first time, they will be able to develop easily only by understanding the concept of it. Moreover, it is specialized in controlling MR contents, and it is possible to easily use this function. We show a part of variable types, operators, and statements in Table 4 to 6.

Our script language has three basic classes, named MRWorld, MRObject, and Client. In order to create applications, the developers add codes into sub classes that inherit these three basic classes. They can also declare variables in classes' fields and methods.

MRWorld class: MRWorld class stores the entire information of MR spaces where there are clients and objects. MRWorld has methods, such as adding objects into MR space and deleting objects from there. Moreover, when the client connection states are changed, or the event such as a collision between objects occurs, call back methods are called. We show some methods in Table 7.

MRObject class: MRObject class stores the information of objects. Field variables that store properties such as the position and orientation are prepared beforehand and used for

Table 4. Variable Types

Type name	Meaning	Size
`int`	integer type	4 byte
`float`	floating-point type	4 byte
`boolean`	boolean type	1 byte
`string`	character string	flexible length

Table 5. Operators

Operator	Symbols
assignment	`=, +=, -=, *=, /=, %=`
arithmetic	`+, -, *, /, %`
comparison	`>, <, >=, <=, ==, !=`
logical	`&&, ||`
other	`new, isInstanceOf`

Table 6. Statements

Statement	Symbols
selection	`if, if~else, switch`
iteration	`for, while`
jump	`break, continue, return`

Table 7. A part of methods of MRWorld class

Class methods	
Method definition	Meaning
`void addObject(MRObject obj)`	adding an object
`void removeObject(MR Object obj)`	removing an object
instance methods for callback	
Method definition	Meaning
`void connectedClientCallback (int clientClassId)`	called at time to connect a new client
`void onCollisionCallback (MRObject obj1, MRObject obj2)`	called at time to collide objects

controlling the motion of the objects. A callback method is called every time MR space is updated, so some processes such as moving objects are described in this method. We show some methods in Table 8.

Client class: Client class stores the information of clients. When interaction is detected, a callback method is called. Therefore, processes related to interactions are described within this method. We show some methods in Table 9.

In addition to these basic classes, application developers can use Position3D class to store a 3D position of the camera, Orientation3D class to store a 3D orientation of the camera, and Math class for numerical processes such as trigonometric function.

III. Evaluation of Our Framework

A. *Use of Streaming Server*

Equipment We implemented MR Server, Client, and Streaming Server and evaluated their performance. Table 10 shows the equipment for the evaluation.

Evaluation Method We calculate the processing time [ms/frame] and the delay time [ms/frame] in case 1 and case 2 (see Fig. 3). The processing time is the fastest time in all processes and the delay time is the total time of each process between capturing background image and displaying MR image. The number of CG objects placed in MR space is varied 2 to 20 in all even numbers. The CG object has a skeletal motion and the number of the polygon of this object is 47187. We calculate 100 times, and show the averages graphically. We execute ARToolKit [8] on the Client to estimate the position and orientation.

Result and Consideration In Fig. 6, we show the comparison of processing time in case 1 and case 2 graphically. In Fig 7, we show comparison of the delay time in case 1 and case 2 graphically. In case 1, if the number of CG objects increases, both processing time and delay time also increase. In case 2, the processing time is about 30 [ms/frame] constantly. The delay time in case 2 is increasing, but the increased amount in case 1 is less than one in case 2. In case 2, Streaming Server renders CGI, so we consider that the performance for rendering CGI improves with increasing CG objects. On the other hand, in case 1, we consider that mobile devices cannot fill demands of the performance to render CGI with increasing CG Objects.

B. *Develop Trial Applications*

We developed trial applications, called MR Panzer and BKC Guide (see Fig. 1), using our implemented framework.

MR Panzer is a shooting game in which four users can play at the same time. Each player uses buttons arranged on a touch panel to control own CG tank and attacks other tanks with CG missiles. The system can detect collisions between CG objects and real obstacles by placing unseen virtual objects. The system can play BGM and sound effects. Furthermore, MR Panzer has minimum game elements. For example, if tanks are attacked 5 times by other tank's missiles, each tank returns to the initial position. In MR Panzer, client's position and orientation are estimated using ARToolKit. In order to estimate them unintentionally, we prepared an actual field and real

Table 8. A part of methods of MRObject class

Method definition	Meaning
Instance methods	
`Position3D getPosition()`	getting a positon
`void setPosition (float x, float y, float z)`	setting a position
`void setAsBox (float h, float w, float l)`	setting a cuboid shape for collision
Instance methods for callback	
`void updateCallback(float dt)`	called at time to update MR space. `dt` is an elapsed time from last update

Table 9. A part of methods of Client class

Method definition	Meaning
Instance methods for callback	
`void updateCallback(float dt)`	called at time to update MR space. `dt` is an elapsed time from last update
`void onInteraction (int interaction_type)`	called at time to occur interaction

Table 10. Equipment for the evaluation

	Client	MR Server
CPU	Apple A7	Intel Corei7-2600
Memory	1 GB	8 GB
Network	IEEE802.11n (5GHz band) 300Mbps	wired LAN (1.0 Gbps)

	Streaming Server
CPU	Intel Corei7-3960X
Memory	16 GB
Video Card	NVIDIA GeForce GTX 560
Network	wired LAN (1.0 Gbps)

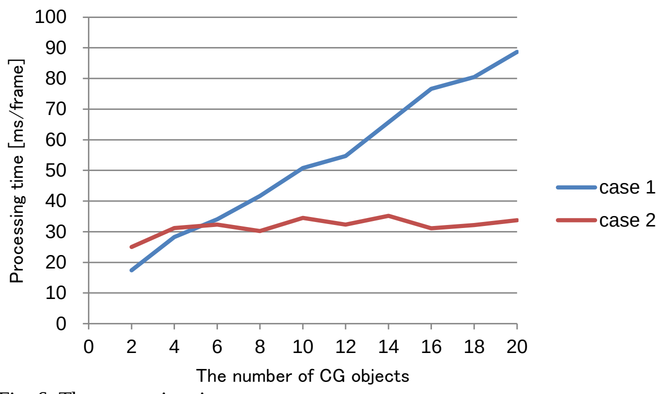


Fig. 6. The processing time

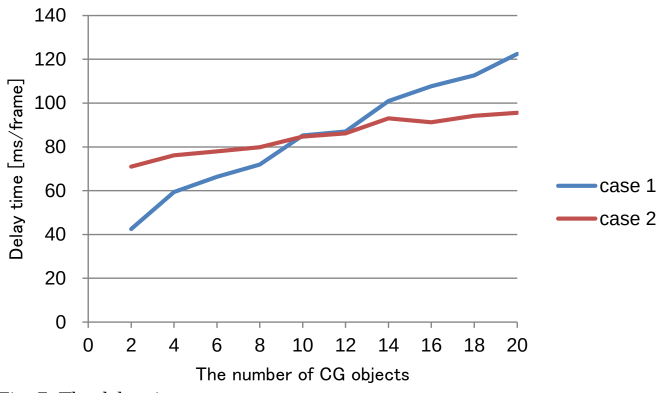


Fig. 7. The delay time

objects. In Fig. 1, four users are playing MR Panzer. As shown in Fig. 1 (b), the entire field is displayed in the large monitor placed in the back. We could see the synchronization between the large monitor and the mobile device held by the user in front of the image.

BKC Guide is an application for visitors to our campus named Biwako Kusatsu Campus. The CG nameboard is superimposed on the top of the building to indicate the name and facilities of the building. As shown in Fig. 1 (d), the building that is planned to be constructed in the future is superimposed as CGI on the back of the existing building. We place unseen virtual objects on the real buildings to occlude them. In BKC Guide, client's position and orientation are estimated with physical sensors on the mobile device because users can use them in every place with a minimum precision. The GPS receiver is used for estimating the position, and each user can estimate the orientation with the gyro sensor and the direction sensor.

As shown in Table 11 and Table 12, MR Panzer's code is about 300 lines, and BKC Guide's code is about 150 lines. We show a part of the code for placing CG nameboards in BKC Guide because this is the typical code for initializing virtual objects (see in Fig. 8). Although both Player class in MR Panzer and User class in BKC Guide inherit Client class, the lengths of these codes are different. The main reason for this is that Player class includes the code for interactions to control the Player's tank. In Fig. 9, we show the codes for these interactions in MR Panzer. According to this code, each player can control own tank using interactions.

The developer of these applications is a student who has knowledge about our framework and has an experience to develop our script code. It takes 15 hours to develop MR Panzer and an hour to develop BKC Guide. Scripts are mainly composed of simple selection statements and assignment statements. According to these results, we confirmed that our framework could provide an easy way to develop this kind of MR applications.

IV. Conclusion

In this paper, we proposed a distributed framework that can create MR systems providing the same MR space for a variety of mobile devices. In particular, we described the system architecture of our proposed framework, the mechanism to control contents, and the original script language for developing mobile MR applications. As a result, by using Streaming Server, we can distribute the loads. Although each trial application's script codes are about 300 lines, it gained a lot of popularity in the demonstration for high school and college students. As a result, we could indicate MR applications are easily developed with our framework. The current framework employs ARToolKit and physical sensors as a module to estimate client's positions and orientations. We can easily employ other methods by adding them into a module, so we are planning to combine other tracking methods using 3D feature points and to build other trial applications.

References

[1] M. Bauer, B. Bruegge, G. Klinker, A. MacWilliams, T. Reicher, S. Riß, C. Sandor, and M. Wagner, "Design of a component-based augmented reality framework," Proc. Int'l Symp. on Augmented Reality 2001 (ISAR 2001), pp. 45 - 54 (2001.10)

[2] W. Piekarski and B. H. Thomas, "Tinmith-evo5 – An architecture for supporting mobile augmented reality environments," Proc. Int'l Symp. on Augmented Reality 2001 (ISAR 2001), pp. 177 - 178 (2001.10)

[3] B. MacIntyre, A. Hill, H. Rouzati, M. Gandy, and B. Davidson, "The Argon AR web browser and standards-based AR application environment," Proc. Int'l Symp. on Mixed and Augmented Reality (ISMAR 2011), pp. 65 - 74 (2011.10)

[4] D. Schmalstieg and D. Wagner, "Experiences with handheld augmented reality," Proc. 6th Int'l Symp. on Mixed and Augmented Reality (ISMAR 2007), pp.3 - 15 (2007.11)

[5] Junaio: https://dev.metaio.com/junaio/ (last access 31 July 2014.)

[6] AREL: https://dev.metaio.com/arel/overview/ (last access 31 July 2014.)

[7] Unity: http://unity3d.com (last access 28 May 2014.)

[8] H. Kato, M. Billinghurst, I. Poupyrev, K. Imamoto, and K. Tachibana, Virtual object manipulation on a table-top AR environment, Proc. Int'l Symp. on Augmented Reality (ISAR2000), pp.111 - 119 (2000.10)

Table 11. Lengths of code for MR Panzer

Class name	Super class	Length
TankWorld	MRWorld	71
Player	Client	50
Observer	Client	12
Tank	MRObject	73
Rocket	MRObject	41
Explosion	MRObject	21
Building	MRObject	10
Wall	MRObject	10

Table 12. Lengths of code for BKC Guide

Class name	Super class	Length
BKCWorld	MRWorld	61
User	Client	12
Nameboard	MRObject	57
NewBuilding	MRObject	10
UnseenBuilding	MRObject	10

```
class BKCWorld : MRWorld {
    BKCWorld() {
        Nameboard gym;
        gym = new Nameboard(0);
        gym.setPos(90.0, 200.0, -500.0);
        gym.serOri(0.0, 45.0, 0.0);
        gym.setScale(2.0, 2.0, 2.0);
        MRWorld.addObject(bkc_gym);
        …
    }
    …
}
```

Fig. 8. The code of placing the CG Object for BKC Guide

```
class Player : Client {
    Tank myTank;
    …
    void onInteraction(int interaction_type) {
        switch(interaction_type) {
            case 1: // key up
                this.myTank.moveUp(0.5);
                break;
            case 2: // key down
                this.myTank.moveDown(0.5);
                break;
            case 3: // key left
                this.myTank.turnLeft(4.5);
                break;
            case 4: // key right
                this.myTank.turnRight(4.5);
                break;
            case 5: // key s
                this.myTank.shootRocket();
                break;
        }
    }
}
```

Fig. 9. The code of interactions for MR Panzer