Evan Arroyo and Joshua Sims

03 May 2017

Final Report

The aim of our project was to create a three-dimensional, gesture controlled adaptation of Pong. We selected this project because it involves subjects which interest us as well as many others in the computer science industry – potential employers included. Among these subjects are three-dimensional graphics, gestural control, data persistence, and, particularly, ubiquitous applicability, and ubiquitous accessibility. The most popular software applications are often both ubiquitously applicable and accessible. Google Search, Facebook, YouTube, Snapchat, Instagram, Ballz, and Spotify – these are examples of ubiquitously applicable and accessible software. Each is available on varying platforms, including mobile devices which are almost always within the user’s reach. Furthermore, these applications require little prerequisite knowledge. They are thus inherently accessible to most individuals and the bases for using them are common. They are predominantly used for entertainment, education, and/or socialization. The audience for these applications is thereby universal. Little knowledge and skill is required to play Pong and thus Pong is inherently accessible to most individuals. Furthermore, the basis for playing Pong is common amongst most individuals: it is the desire to be entertained. Our adaptation of Pong extends upon its classic counterpart with three-dimensional graphics, three-dimensional physics, additive gameplay effects, gestural control, and statistics. These extensions, however, have not hindered the ubiquitous applicability of the game nor its accessibility. We have gained not only proficiency in creating software which shares qualities common to tremendously popular software, we have also gained proficiency in the technologies which compose that software via the application of our chosen toolset.

Through our efforts, we have improved our ability to estimate the requirements of a project as well as the time to be dedicated to each requirement. The following MoSCoW analysis illustrates the estimations we made for this project.

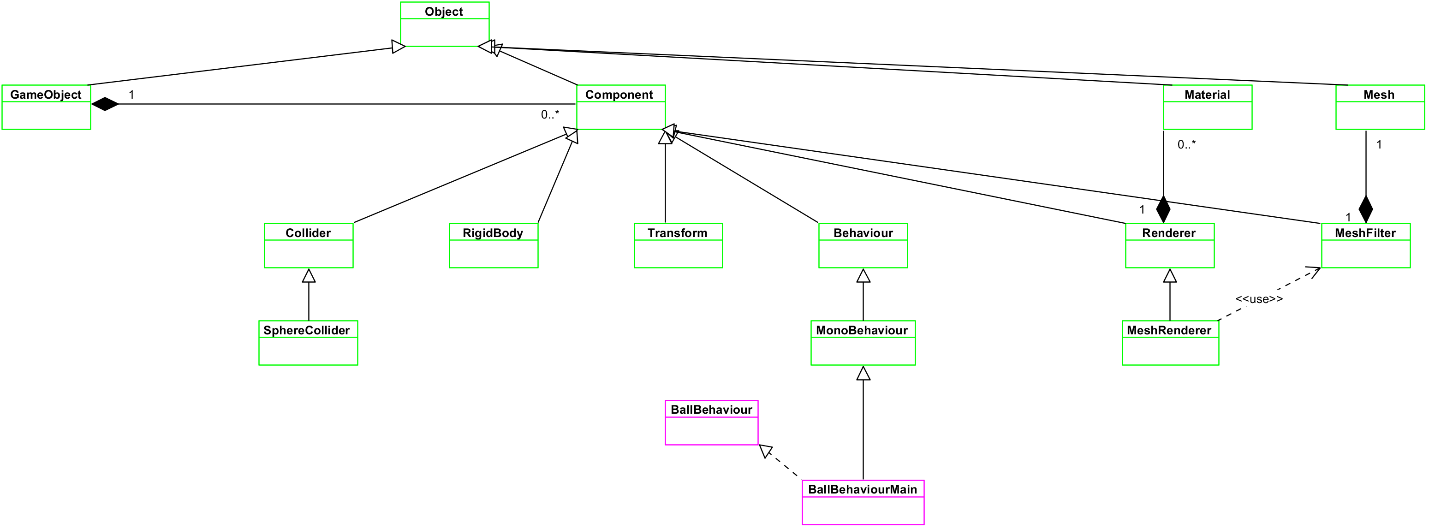
|  |  |  |  |
| --- | --- | --- | --- |
| Must have | Should have | Could have | Won’t have |
| * 3D arena * 2D movement of paddle * 3D movement of ball * Profiles (and profile creation) * Statistics menu (featuring accurate statistics) * Navigable UI * Settings menu * Gameplay derivative of classic Pong * Artificially intelligent opponent * Gestural control over UI and paddle via Kinect * Obstacles (random and static) for the ball to collide with * Elements which, when touched by the ball, modify the ball, paddle, or arena (including goal zones and barriers) | * Intuitively navigable UI (and clear visibility of UI elements) * Intuitive physics * Artificially intelligent opponent that challenges, but does not overwhelm, the player * Two player gameplay (neither player is artificially intelligent) * Profile deletion | * Sound effects * Background music * Choice of multiple camera angles * Official support on systems other than PC coupled with Kinect | * Networking * User insertion of music. sound effects, or graphics (the user cannot upload a sound to be played each time the ball collides with something nor can the user upload a picture and use it as the skin for their paddle) * Data encryption * Standardized method of importing/exporting settings or statistics * Persistent data technology other than the BinaryFormatter class in C# * Data visualization (other than descriptive text) |

The features that we pursued and the timeline which ordered them appear in the following table.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Jan. 24th | Feb. 7th | Feb. 21st | Mar. 14th | Mar. 28th | Apr. 11th | Apr.25th |
| * 3D arena built * 3D movement of ball (and paddles) * 2D gestural control of paddle via Kinect | * Ball resets after goal * Game over after a number of goals specified by the user * Artificially intelligent opponent * GUI interaction via Kinect * Statistics menu * Settings menu | * Additive effects (such as randomly occurring obstacles, wind, and ball-modifying elements) | * Additive effects | * Additive effects | * Additive effects * Code freeze   *Time permitting*   * Two player capability * Player two paddle controlled via Kinect * Background music * Sound effects * Code refactorization | * Final report * Presentation |

We did not finish the development of every one of the features listed, but we did finish those which were of highest priority – specifically, the additive effects, three-dimensional graphics, three-dimensional physics, and gestural control.

First, we implemented the gestural control. The goal was to get the paddle to move in two dimensions rather than just one. We encountered difficulties with the Kinect API. Particularly, we had trouble navigating the API to find what we needed. The design of the API was not bad – it was just the layout and the navigability of it which hindered our comprehension of it.

Thereafter, we built the arena, introduced the ball, gave it movement and tested the physics provided by Unity. Thanks to Unity’s graphical scene editor, we did not have much trouble creating and positioning the 3D graphics for the scene. It is much simpler to drag elements around than to change source code, then render, then change source code, then render again, and so forth. Furthermore, Unity makes it easy to attach components and immediately witness the effects. This includes colliders and materials which helped determine if our gameplay physics were correct. The following diagram is a UML class diagram describing the Unity API in relation to our ball object.

Illustrates the structure of the Unity API

After confirming that the pong physics were correct, we programmed the ball to reposition itself at the middle of the arena after reaching a goal zone. This required minor changes from last semester’s work – our arena was 3D now so we had to take that into consideration.

We skipped some things which we had originally planned to do at this point. We did not enforce the ending of the game after a certain score, we did not reimplement the scoreboard, and we did not refactor the statistics to be compliant with the additional goal zone segments added. We did not enforce the ending of the game because we reconsidered that to be a secondary feature – it did not contribute to gameplay; it just stopped gameplay. We did not reimplement the scoreboard for the same reason – it does not contribute to gameplay. We abandoned statistics development for this reason as well, but also we decided to abandon the statistics because they were not maintainable as designed. We were saving statistics by serializing many fields of an object. The saves for each field had to be specified in the code and this became even more laborious once we added one more goal zone segment (originally there were three, now there were four). Most of the statistics are dependent on individual goal zone segments so the addition of the one goal zone segment required the inclusion of many more fields relative to that new goal zone segment. Then we have to load the statistics – more inefficient work. We decided to abandon the statistics not only because they did not contribute directly to the gameplay, but also because our current design was not maintainable and we would prefer to develop statistics according to a much more efficient and maintainable technique.

At this point, all the elements of basic Pong were complete, except for the opponent paddle. We then implemented the artificial intelligence. This too required little change from last semester’s work. Our AI from last semester tracked the ball’s x position and mimicked that position at a certain speed such that it could be constantly able to deflect the ball. Refactoring that functionality included mimicking the ball’s y position as well as its x position. Additionally, we programmed the AI to return to the center position after striking the ball rather than following the ball even when the ball is not heading in the direction of the AI. This made the AI appear smarter and it did make it more capable since returning to the center put the paddle in a position equidistant from the areas that the ball may go to.

After programming the opponent paddle to follow the ball, we focused on developing the additive effects. At the inception of this phase, we imagined triggering additive effects via ball collision with a special object in the arena. However, soon after beginning the development of the additive effects, we realized that it would be best to remove that extra variable of difficulty and that time would be a simpler trigger. So, we decided that we would begin by programming the additive effects to occur at random times and end after random time intervals rather than be dependent on collision with a special object.

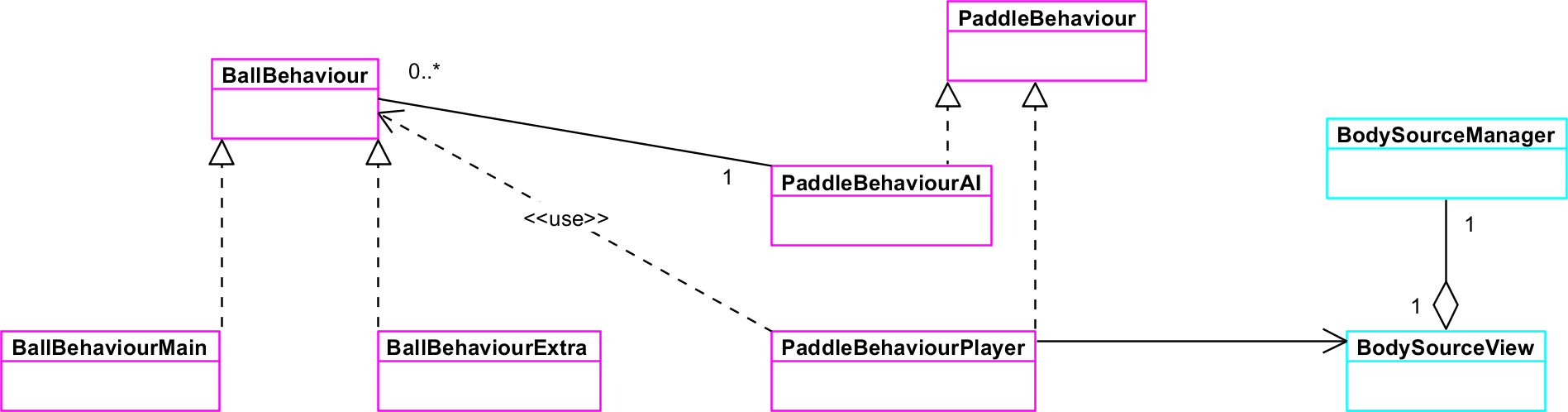
The first additive effect which we developed was the disabling of the goal zone segments. We wrote a script which toggled a boolean at random times. This boolean would control the ability of a goal zone segment to consume a ball. Then we wrote into the ball behaviour script that the ball should check that boolean upon collision with a goal zone segment. If the boolean was true, the ball would reset at the center of the arena. If it was false, the ball would just rebound off the goal zone segment. We also programmed the goal zone segments to turn the same color as the arena walls whenever they became disabled.

After disabling goal zone segments, we proceeded to award extra points for scoring on a particular goal zone segment. We denoted such goal segments by making them blink a bright color and we enabled that functionality by writing a blink script which causes objects to blink if the appropriate boolean is toggled. At this point we also incorporated the crosshairs which award extra points for hitting the border between goal zone segments. These extra points were not something we had to program – Unity was already performing this for us. Originally, it was a bug – we did not want multiple goal zone segments to each trigger the accrual of points if the ball struck the border separating the goal zone segments. We spent a couple of days trying to work around this issue, but after much effort we decided to just make it a feature.

Then we implemented the random obstacles. This required us to learn about prefab instantiation at run time because the random obstacles would be instantiations of a prefab. Thanks to the physics engine, this required us to only insert and remove the random obstacles. Unity takes care of the incident when a random obstacle spawns on a ball and it handles how the ball should rebound off of a random obstacle.

We then implemented the paddle shrink which is similar to the other effects in that it occurs at a random time and lasts for a random amount of time. We wrote a script to determine when the effect should happen and then we modified the paddle scripts such that the paddles could be shrunken and would remain so for the random amount of time.

The additive effect which required the most attention was the multiple balls effect. To incorporate multiple balls, we had to refactor the artificial intelligence to consider the different scenarios which it would encounter when pursuing multiple balls. To get the artificial intelligence to not just mimic the x and y coordinates of a single ball, we assigned it a queue of balls in the artificial intelligence paddle script. The artificially intelligent paddle constantly peeks at the top of the queue to determine which ball it should pursue and then, after making its decision, it pursues that ball just as it did the main ball. Though, it is not just this simple. The introduction of multiple balls brought about edge cases that had to be dealt with. We had to consider what the AI should do if a ball hit a disabled goal zone segment. Ordinarily, a ball is dequeued from the queue of tracked balls if it either scores or strikes the AI paddle. However, a ball can hit a disabled goal zone segment and thus remain in the threat range of the AI paddle. Since the ball definitely moves either past the artificially intelligent paddle or rebounds off of it back toward the AI’s goal zone, we determined that the AI paddle should just consider it as dequeued. We then had to consider the edge case of when the paddle hits a ball which it was not actually pursuing. To handle this, we programmed the AI to later disregard the ball (when it eventually rises to the top of the queue) which it accidentally hit and to continue pursuing the ball it was initially pursuing. To do this, we added another boolean to the balls for the paddle to consider. This boolean indicates to the paddle whether or not it should pursue the ball, regardless of if the ball was in the queue. Fortunately, we did not have to deal with the other troubles of spawning in multiple balls such as “at which position should the balls spawn?” and “at what angle should they proceed at?” We simply spawned the extra balls at the position of the main ball, gave them a velocity, and Unity’s physics engine handled the collision that ensued between the main ball and the extra balls. Thus, the extra balls “explode” off the main ball at varying angles. The last thing we did for the multiple balls effect was, attach previously created blinking color script to the main ball and programmed the main ball to blink before extra balls were spawned.

Following is a UML class diagram depicting the integration of the Kinect API and the gestural control of the player paddle in relation to the behavior of the artificially intelligent paddle and the balls.

Illustrates the implementation of the gestural control and the relationships between the paddles and balls

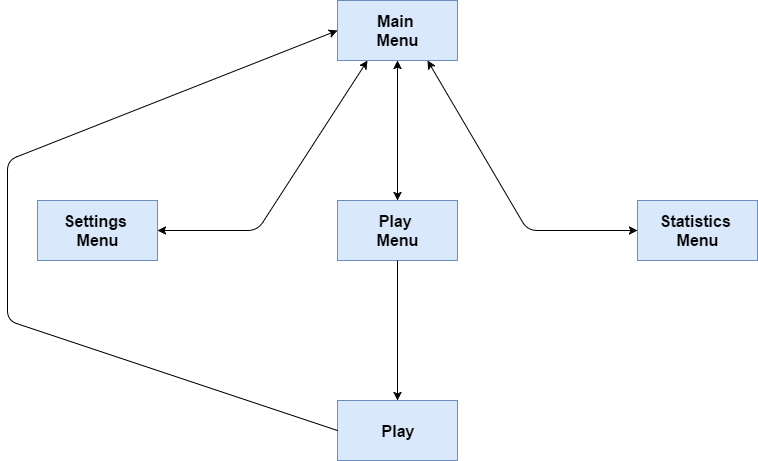
Once the additive effects were completed we realized that there was an issue: the effects were being introduced all at once and this made the gameplay overwhelming especially for a novice. So, we introduced a levelling system: after the player reaches a certain score, a gameplay effect is introduced. Therefore, the player is more accustomed to the gameplay before the gameplay becomes more complex.

Of course, before we could control the introduction of the effects based on the player’s score, we had to reintroduce the accumulation of points. We then wrote a ScoreKeeper class which updates the scoreboard and enables gameplay effects once a certain score is reached. The disabling and enabling of gameplay effects is very straightforward – we just utilized the enabled property of the GameObject class, which each of the effect scripts inherited from.

Once the levelling system was completed, we implemented the settings menu and thereby enabled the user to toggle the additive effects. The user can set each additive effect to be score dependent, immediate, or off. The work of creating the dropdown menus for each effect was surprisingly difficult. To associate the selected value of the dropdown menu with other scripts, we had to implement delegate methods which neither of us had heard of. So, we learned about delegate methods and then, after implementing those methods, we were able to link the selection of the dropdown menus with the gameplay.

The UI for the settings menu was fairly difficult to arrange in an aesthetically pleasing way. We had to research how Unity’s UI deals with multiple screen sizes – that was the bulk of the difficulty. Other difficulty lied in the resolution of the text. We had to fiddle with the properties of the Text objects to get them to appear clearly, rather than blurry, on the screen.

Following is an interaction flow diagram which illustrates traversal across the various scenes in our project.



Traversing scenes

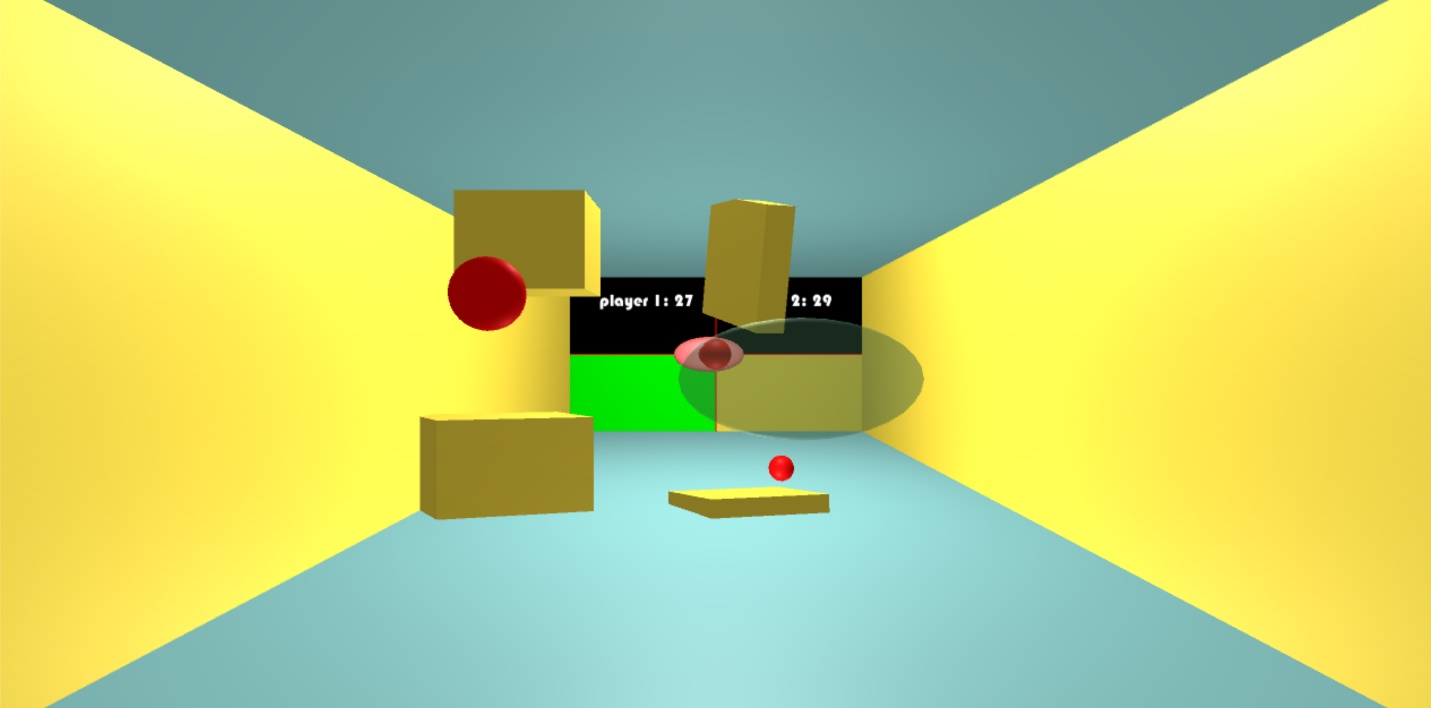
The following screenshots illustrate the scenes in further detail.



The main menu scene



The settings menu scene



The play scene

The software that we have built could easily be multi-platform given that we built it upon Unity which features the ability to deploy applications to nearly any platform. The caveat is that our technology for interpreting the user’s gestures, the Xbox Kinect, is not compatible with many platforms. It would be simple, though, to receive user input via some other means because our implementation loosely couples user interactions with the user input technology. The user interactions were programmed using the C# scripting language. We used C# to program everything included in our project, however, given more time, we would reprogram the collection and persistence of the statistics using SQL – we researched, but did not incorporate a SQL database solution for our project because the implementation of such was too demanding given that it was of a lower priority than the substantially more demanding implementations of the gameplay and graphical menus.

We completed the gameplay, however, it could be extended further. The artificial intelligence could also be extended. Currently, the AI decides to pursue a ball based on when the ball approaches the AI paddle. The AI then pursues that ball, until it has struck or missed the ball, regardless of other factors. We could extend the AI to evaluate situations and choose the best option. For example, the AI could be programmed to pursue a ball which is rapidly advancing before a ball which is incessantly ricocheting between the arena walls.

We could implement gestural control of the menus via the Kinect. This would require further research into the Kinect API so the difficulty of such a task depends on how quickly we can find what we are looking for in that API.

The game does not end after a set amount of goals. Implementing this feature would be simple since we had implemented it last semester.

Currently, the game does not feature sound effects or background music. Such additions would further engage the user, bettering the user’s experience significantly (Wei and Zheng). Primarily, sound effects would be produced upon collisions and the activation of effects.

The game is not multiplayer, but we could implement local multiplayer as well as online multiplayer. Local multiplayer would require a large amount of refactorization since we did not design the program loosely enough to accommodate more than one player. Online multiplayer would require us to handle the complications involved in concurrent access and server hosting, but it would be very worthwhile since such a feature would drastically increase the likelihood of a user to play the game – users would be able to socialize with each other as well as experience the greater challenge associated with competing against a human opponent (Yamakami).

We could reimplement the statistics using an in-memory database system. The time to access in-memory databases is far shorter than that of disk-based databases (Pingfu, He, Sadiq, Zheng, and Zhou). Furthermore, it is far easier to perform complex operations on data via SQL than via helper functions written in C#.

Instead of triggering additive effects via time, we could trigger them via collisions with trigger objects. This would greatly extend the challenge of the game and the potential to strategize since players would be given incentive to aim for objects other than goal zone segments.

We could implement many more additive gameplay effects in addition to what we have already implemented. For example, we could introduce wind and arena transformations – both would greatly increase the complexity of the game, but would not take away from the intuitive gameplay.

Aside from extending the gameplay, we could improve the maintainability and reliability of our application. We could write unit tests which test for 100% code coverage (every line of code is executed) and all possible values, sequences of events, and concurrency. We could also make the code as loosely coupled as possible so that different modules could be independently changed with ease.

This project could be extended in a very large number of ways. The extensions that we have thus made since last semester and the effort involved has progressed our skill with 3D graphics, scripting, software development, gestural control, API utilization, and game design. More importantly, we have furthered our ability to study and apply good design while developing working software modules under time constraints – an ability that is foundational to the creation of advanced software.

Works Cited

Pingfu Chao, Dan He, S. Sadiq, Kai Zheng and X. Zhou, "A performance study on large-scale data analytics using disk-based and in-memory database systems," 2017 IEEE International Conference on Big Data and Smart Computing (BigComp), Jeju, 2017, pp. 247-254.

T. Wei and H. Zheng, "Sound Effect of Physical Engine in Game Design," 2011 International Conference of Information Technology, Computer Engineering and Management Sciences, Nanjing, Jiangsu, 2011, pp. 148-151.

T. Yamakami, "Anomaly of mobile social games: Lessons learned in the hype of Japanese mobile social games," 2012 7th International Conference on Computing and Convergence Technology (ICCCT), Seoul, 2012, pp. 198-202.