A FRAMEWORK FOR GAME TUNING

Juan Haladjian, Frank Ziegler, Blagina Simeonova, Barbara Köhler, Paul Muntean, Damir Ismailovič and Bernd Brügge

Technische Universität München, Munich, Germany

ABSTRACT
The increasing complexity of current game projects highlights the need for new tools and working methodologies to support the games’ engineering process. We make use of the mobile touch technology as a way to improve communication and encourage collaborative work between the different parts involved in the game project (developers, usability testers, end-users). This paper presents a framework for quickly balancing and tuning games, followed by a quasi-experimental study that demonstrates its advantages with respect to traditional game tuning approaches.

KEYWORDS
Game tuning; game balancing; game prototyping.

1. INTRODUCTION
Game balancing is the process of changing game parameters, scenarios and behaviors in order to avoid the extremes of getting the player frustrated because the game is too hard, or becoming bored because the game is too easy. (Koster, 2005) We introduce the concept of game tuning as the process of changing game parameters, scenarios and behaviors, for any purpose, not necessarily related to the game’s difficulty. One common reason for tuning a game is aesthetics. Experience shows that efforts to tune a game are often underestimated. Different parameters often depend on each other. Changes in one parameter may unbalance other parameters. For example, in a game where many enemies move towards a player, making the enemies move faster may make the game too hard. To balance it again, a solution could be to reduce the spawn frequency of the enemies.

Games are tuned in different ways, ranging from low-level approaches like modifying constants in source code and recompiling the project, to higher level ones like using in-game controls that adapt the game at run time. Lower level approaches allow for major flexibility but require a higher degree of expertise in programming. In an educative game targeting seven-year-old kids, a psychologist should probably determine game parameters related to user interaction, since he better understands kids’ motoric skills. A psychologist, however, does not have programming experience. Higher-level approaches like special GUIs for setting up game parameters can potentially be used by anyone, but add development effort and are game specific.

The contribution of this paper is threefold: First we present a case study that describes how the lack of a tool for tuning affected the development speed of an industrial game project. The second contribution is Schulplattler, a framework for dynamically tuning of games. Finally, we present a quasi-experimental study that shows how tuning a game can be done faster using this framework rather than doing it programmatically.

2. BACKGROUND
In the gaming industry it is common to use scripting languages in order to speed up game tuning. Everything that might change at a later point and is important for game tuning is outsourced into script files which can be reloaded without recompiling or sometimes even without restarting the game (Gregory, 2009). While this might save some time when changing parameters, it still requires a trained person that knows how to program in the chosen scripting language. Furthermore many of the scripting languages introduce some performance penalties (Thorn, 2010). Additionally, for mobile devices, the scripts need to be deployed to the device
manually or over the Internet every time they change, as they cannot be directly edited on the device. On the other hand scripting allows for great flexibility as not only simple values but also game logic can be changed.

Outside of game industry, Wizard of Oz prototyping is used in the design phase to determine some important information about a human interface before it is actually built. In Wizard of Oz prototyping, a human takes over some tasks of the computer that have not been implemented yet. This has for example been applied to systems that are supposed to have a speech recognition subsystem. The tests then help determine which kind of input the users try to give and therefore to limit the abilities the later system should to have. The advantage is that those tests can uncover a considerable amount of information before the system is actually built and therefore save a lot of time.

Game balancing is less general than game tuning, as it focuses on adapting the game’s difficulty. Game balancing is a design problem that faces challenges like when and how should the game be adapted, and up to what degree the game is automatically adapted. Maintaining a challenged and focused player during the game play is extremely difficult in interactive contexts (Hunicke and Chapman, 2003).

Many game balancing techniques rely on measuring the difficulty the user is facing. A heuristic is designed to map a given state into value that specifies how easy or difficult the game feels to the user at a given moment in time.

One approach of dynamic game balancing uses domain specific knowledge in order to define a set of rules, which are typically applied to Non-Player Characters (NPCs). Example of such a rule is: “shoot if opponent is in sight, otherwise chase”. This approach is error prone because the rules are error prone. A huge drawback is that adaptive behavior is hard to reach.

A natural approach to deal with dynamic game balancing is to use machine learning. (Demasi and Cruz, 2002) built intelligent agents that use genetic algorithms that best fit the user level.

3. CASE STUDY

In this section, we analyze a two-year game project that targets the iPad platform and teaches mathematics to preschool children. The game is composed of six mini-games, each of which consists of specific tasks and exercises that need to be solved by the player. Shadowing and interviewing the developers we found out they invested a considerable amount of time tuning the game. In particular, they admit to have found very time consuming positioning game elements and setting 2D coordinates at the source code. As an example, one of the mini-games teaches kids drawing digits from zero to nine. In a first step, a game character uses a flashlight to draw a digit. The user must then follow the light with the finger. The shape of each digit is represented by a series of Bezier curves. A number is represented by an average of five Bezier curves, and each Bezier curve requires four coordinates, which are then interpolated using the cubic Bezier formulae. Representing a number requires therefore, defining 200 coordinates. Other parameters that were tuned in this game were the speed at which the character draws the number, the scale of the numbers, how sensitive the drawing is to errors, etc.

We counted directly from the source code, for each of the mini-games, how many variables were tuned. The results are shown in table 1. We distinguish between position coordinates and other parameter types.

A mini-game had on average 98 variables that needed to be adapted. Adapting variables (e.g. coordinates) from source code is a tedious work, since there is no immediate visual feedback of their effects on the game. It is hard to determine where positions will exactly be on the screen. In order to observe the effect of each change, the game needs to be recompiled and restarted many times. We observed a developer recompiling and re-running the game up to ten times in order to precisely position a game object. Not only this is time consuming, but also limited to programmers. The psychologist involved in the project could not directly tune the game in this way. Instead, he reported usability issues he observed during testing, which developers used later to tune the game afterwards, which caused some communication overhead. This was the main reason why we realized the need for a tool that would allow anyone to tune a game.

<table>
<thead>
<tr>
<th>Mini-Game</th>
<th>#1</th>
<th>#2</th>
<th>#3</th>
<th>#4</th>
<th>#5</th>
<th>#6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coordinates</td>
<td>53</td>
<td>43</td>
<td>230</td>
<td>10</td>
<td>30</td>
<td>10</td>
</tr>
<tr>
<td>Other Variables</td>
<td>43</td>
<td>63</td>
<td>36</td>
<td>48</td>
<td>20</td>
<td>3</td>
</tr>
</tbody>
</table>
4. A FRAMEWORK FOR GAME TUNING

In (Haladjian, 2012) we presented TangoPhysics, a tool for quickly prototyping mobile games. In a first step the game would be edited using predefined objects and behaviors, and assets quickly imported from the device’s libraries, camera, microphone, internet, etc. Users could then start the simulation mode and play the game. One of the main advantages of the tool with respect to other mobile game prototyping tool was its efficiency at creating mobile game prototypes. This was mainly due to the fact that the tool already runs at the end-device. A simple shake gesture would switch between the game edition mode and the game simulation mode, making it possible to quickly test changes done to the game. But still, the game needed to be restarted, thus loosing the game’s state every time.

The main goal of Schuhplattler is to dynamically alter the game without the need to restart it. This makes it possible for players to keep playing while someone else is balancing their game. Our idea is to integrate Schuhplattler as a component of TangoPhysics with the purpose of quickly and intuitively prototyping and balancing games.

The Schuhplattler framework consists of two parts: the client library and the manipulation tool. The client library is used by applications to register variables for tuning. Registering a variable is done with a single line of code. The manipulation tool is the user interface to tune the game.

We designed the manipulation tool to run on a mobile touch device, for two main reasons. The first one is that user interaction of a mobile touch device is more direct, improving the intuitiveness of the balancing process. This couples well with the fact that often non-programmers take care of the balancing. The second reason is the easiness to transport the device and use it anywhere. It may be easier, cheaper, or more natural to test end-users in their own environment rather than at the developer’s environment.

The manipulation tool is currently implemented in iOS and targets the iPad platform. It connects to any application using the Schuhplattler client library via Wi-Fi or Bluetooth. After the connection is established, the manipulation tool fetches every variable registered by the application. When the manipulation tool receives a variable, it displays a controller for it. Figure 1 (center) shows the manipulation tool with five floating point and integer variables.

![Figure 1. (Left and right) The Bug-Game and (center) the manipulation tool](image)

Variables in the application are kept synchronized with variables in the manipulation tool. Changes to variables done in the manipulation tool are immediately sent to the game, and vice-versa.

5. EVALUATION

In this section we present a study where we measure the time needed for tuning a game with and without our framework. We describe how we tested it, we show the results, and we mention our framework’s current limitations.

5.1 Methods

We performed a study to measure the time needed by users to adapt one of the mini-games from the case study. The game, illustrated in Figure 1 consists of many bugs spawning into the screen. There are good bugs and bad bugs. Good bugs should be dragged to the exit point, bad ones need to be smashed by tapping them with the finger. We modified the game so that eight parameters could be dynamically adaptable. Some of these parameters were the speed and scale of the bugs, their touch radius and the font size of the numbers shown by the game characters. We gave the testers an iPad and showed them a reasonably balanced version
of the game. We then gave them a version of the game containing random parameters and told them to adapt it until they were satisfied. The balanced version was running on another device they could constantly look at. We gave the same random parameters to every tester.

We created two test groups, A and B. Group A was given a laptop with a constants file containing the variables to balance, their min and max value, and a short written description of what the variable is used for. They were also told how to run the game from the development environment (pressing the “play” button on the XCode IDE). Group B was given an iPad running Schuhplattler, and the same written description of the different variables. We made sure the testers understood every variable before starting with the tuning. Three individuals were tested per group.

We measured the time the original developer of the game required to integrate the framework into the game in order to adapt eight variables. Additionally we measured how many times the tester modified each variable and the time each tester needed to adapt the entire game.

5.2 Results

The developer of the game required around 50 minutes to make the game adaptable. He did not know anything about the Schuhplattler framework before.

Table 2. Average time and number of changes users needed to adapt a game with and without our framework

<table>
<thead>
<tr>
<th>Group</th>
<th>Number of Changes to Variables</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A - Source Code</td>
<td>16</td>
<td>12' 37&quot;</td>
</tr>
<tr>
<td>B - Schuhplattler</td>
<td>31</td>
<td>5' 37&quot;</td>
</tr>
</tbody>
</table>

Table 2 shows how many times the testers modified each variable on average and the total time needed on average to adapt the entire game. The results indicate that users needed less time for the tuning when using the Schuhplattler framework, although they performed more changes to the variables. We observed that users adapting from source code avoided randomly changing variables. They instead analyzed the game and decided what the most appropriate parameter should be. When using Schuhplattler, they were curious how the game would look like by making for example bugs be huge. Since this took them only around five seconds, they tried the different variables actively.

The logics behind two parameters were a bit complicated to grasp by some users. The bugs do not move like linearly like billiard balls, but they follow a sinusoidal path. We made the frequency and amplitude of the sinusoid adaptable. Most users could not foresee the consequences of changes to these variables. Users changing from source code could not reason about optimal value for these variables, like they did with other variables, and were confused. Users testing with Schuhplattler were also confused, but immediately started modifying their values and observed how the movement was affected. They managed to make bugs move reasonably in considerably less time. One tester dealing with source code could not achieve reasonable bugs movements and refused to continue testing.

In general, we observed the quality of the produced game was better when Schuhplattler had been used for tuning, as when tuning was done directly from source code.

5.3 Limitations

The framework does not yet support adaptivity of every kind of variable, only Boolean, integer and floating-point variables are currently available. It is expected in the future to let it support string variables and to model game variables using a composite pattern. Variables would then be composed of other variables hierarchically, thus potentially allowing any object to be adaptable.

Issues of scale need to be addressed, like how to navigate and find a variable when tuning games with hundreds of variables. Grouping variables into categories could be a basic solution. But variables and groups would still only be found with their name. Variables metadata could be introduced to better describe them. A variable that is related to for example some NPC that moves around could have as metadata the shape and position of the object it belongs to. The tuning tool could then display the shape in its current position, letting the user immediately find the object for which he would like to adapt its properties. Once the object is selected, the tuning tool could display all its variables.
Whether the final version of the game will suffer performance loss is another issue to be studied. Ideally, two versions of the game would be created, the one for tuning, the one for delivery. This can be accomplished using compiler flags. The problem is that dynamically changing some variables may require more than simply setting the value of the variable. When dealing with physics engines for example, doing changes to physical objects may require resetting them before the changes have effect. Using compiler flags in many places in the code would make it less readable.

6. CONCLUSION

We demonstrated that using today’s mobile technology, the fine-tuning of games should not be a task limited to programmers. Using Schuhplattler, game balancers can observe end-users while they play, analyze their reactions to different game variables, and determine the optimal values for these variables. In the traditional approach, game balancers would observe the game and inform the developers about changes to be done. The developers would then tune the game and again give back again for testing. The process would then be repeated until either the balancers were satisfied with the game, or there was no more budget for tuning.

We showed that simple variables can be tuned directly by balancers with Schuhplattler. We also demonstrated that this can be done quickly and that balancers explore more the effects of changes when using Schuhplattler, thus increasing the quality of the game. This, however, is not supposed to replace the traditional approach, but to complement it. Schuhplattler only supports tuning simple variables. More complex game changes would still require programming.

Another advantage of Schuhplattler is its reusability. High-level game tuning approaches (e.g. scripts or in-game menus) are game-specific; new scripts or in-game menus need to be implemented for different games. Schuhplattler offers a higher-level application independent interface usable by any application for which adaptable Boolean, integer or floating-point variables are needed.

Tuning the game is not only useful for finding out the optimal parameters before delivering the game. Our tool can also be used after delivery for different purposes. In general purpose games, a human can sit a few steps away from the player and play against him by modifying the game trying to make it easier or harder. In educative games, tutors can control the game according to the learner, in order to improve the learning experience (Ayad, 2010). In the future, this framework may be used for improving automatic game balancing. We know from psychology research that humans understand better how to adapt game parameters to increase the game’s fun in each situation (Ismailovic, 2011). Changes done by the game tester could be stored and learned by the game, which could try reproducing them later.

REFERENCES