Performance Overhead of Haxe Programming Language for Cross-Platform Game Development

Preliminary Communication

Domagoj Štrekelj
J. J. Strossmayer University of Osijek,
Faculty of Electrical Engineering,
Department of Software Engineering
Cara Hadrijana 10b, Osijek, Croatia
dstrekelj@etfos.hr

Hrvoje Leventić
J. J. Strossmayer University of Osijek,
Faculty of Electrical Engineering,
Department of Software Engineering
Cara Hadrijana 10b, Osijek, Croatia
hleventic@etfos.hr

Irena Galić
J. J. Strossmayer University of Osijek,
Faculty of Electrical Engineering,
Department of Software Engineering
Cara Hadrijana 10b, Osijek, Croatia
igalic@etfos.hr

Abstract *– Video game industry has become the largest entertainment based industry, surpassing both the film and the music industry in terms of revenue. Costs of game development are rising with a number of platforms one has to support. In today's competitive industry, it is necessary to support as many platforms as possible to remain profitable. One way to cut down on time spent on porting the game to other platforms, is to use cross-platform programming languages and development frameworks. Even though such frameworks allow drastic reduction of development time spent on making sure games work on all intended platforms, they are not without cost. The cost in this case is mainly in reduced performance, compared to games developed in their native development environments. This paper evaluates performance overhead of a relatively new programming language (less than a decade old) called Haxe, which is built especially for cross-platform development. We have implemented the same game in both its native environment and in the Haxe programming language, from which the game is cross-compiled to run in the native environment. The authors tested developed games on three different hardware configurations, with three different complexity settings, and the results show that even though performance overhead of cross-compilation is not insignificant, the overall reduction in development time attained by developing in Haxe presents a viable option for cross-platform game development, with positive aspects outweighing the negatives.*

Keywords *– cross-platform development, cross-compilation, game development, Haxe, performance.*

1. INTRODUCTION

Video games represent the largest entertainment based industry today, surpassing both the film and the music industry in terms of revenue. Video game market revenue in 2013 was $66 billion, where the mobile games segment of the industry was $8 billion. Global video game revenue is expected to grow to $78 billion in 2017 [1]. This huge amount of revenue is spread across a multitude of gaming platforms, including, but not limited to, PC, consoles, smartphones and tablets. Online revenue, including digital delivery, subscriptions or Facebook games, is not included in the aforementioned revenue data, and it is valued at additional $24 billion in 2013 [1]. It is clear from the given data that video game development provides significant business opportunities for newcomers to the market.

One of the main decisions newcomers have to make upon entering the global video game market is which platforms to support and which platforms will generate the highest return on investment [2]. Supporting multiple platforms often requires multiple development teams specialized in specific platforms [3]. Donnellan [4] showed that, depending on the platform the game was initially developed for, the costs of adding support for an additional platform can range from 15% (adding support for Mac to the game developed for Windows) to 158% of additional cost (adding support for PS3 to the game developed for Windows), measured in hours a programmer spends programming. He also stated that differences in costs of supporting an additional platform are correlated to technical differences between platforms. Duc et al. [5] identified four primary challenges that increase costs of managing diverging codebases while supporting multiple platforms:

- a diverged codebase increases growth of technical debt (extra development that arises when the code is implemented in a manner that will require additional maintenance in the long run),
- a diverged codebase encourages redundant development of features,
- in addition to redundant development, a diverged codebase also encourages redundant test effort, and
- developers and testers tend to become experts on the version of the codebase for their platform, but their expertise is often not portable to other supported platforms.

Overcoming the above challenges requires investment of additional effort in coordination with development between platforms.

Game developers often try to combat increasing costs of multiplatform development by using cross-platform toolkits, languages and frameworks. It is possible to differentiate such toolkits and frameworks between general purpose cross-platform application development tools and tools specific to game development.

The goal of this paper is to evaluate feasibility of using the Haxe programming language in development of cross-platform games. This goal is accomplished by development of a simple 2D game in the Haxe programming language, development of the same game in a native environment of a chosen platform and the comparison of the performance of the two.

1.1. RELATED WORK

Most of the papers examining cross-platform development tools and techniques are based on cross-platform development for mobile applications and not specifically on games [6]–[8]. Papers that deal with cross-platform game development investigate development tools for either smartphone games [9], or "big" games (PC and console games) [4]. No study was found which evaluates the tools supporting all or most of the platforms, including both "small" platforms (smartphones, tablets and web games) and "big" platforms (PC and consoles).

The authors of this paper managed to find only one study which specifically acknowledges the Haxe programming language [10]. The study describes cross-platform development as one of the more recent trends in computing and explains that Haxe was created to help overcome the challenges of cross-platform development, but has no information on performance or viability of the language in game development.

2. HAXE PROGRAMMING LANGUAGE

Haxe is an open source toolkit for cross-platform development which allows for compilation of programs to multiple target languages or platforms. It consists of the Haxe language, the Haxe compiler, and the Haxe standard library.

The Haxe language is a high-level programming language, supporting both functional programming (e.g., type inference, nested functions, recursion) and object-oriented programming principles (e.g., classes, interfaces, enumerators, getters and setters). Haxe is statically typed; however, it retains the flexibility of dynamically typed languages by employing a Dynamic data type to represent untyped data at design-time [11]. The Haxe language provides a mixture of features from languages supported by the Haxe compiler. Language-specific differences are abstracted away through conditional compilation, allowing for compilation of a specific code depending on compilation parameters. As such, conditional compilation is instrumental in cross-platform development. The Haxe language syntax follows the ECMAScript standard, though deviates therefrom where necessary. Unlike other ECMAScript languages, such as Javascript, Haxe is a compiled language.

The Haxe cross-compiler is a command-line tool which compiles Haxe source code into source code or bytecode of a target language or platform. As of Haxe 3.1.3, supported languages and platforms include Flash, Neko, JavaScript, ActionScript 3, PHP, C++, Java and C#, with Python support confirmed for version 3.2. [12]. The compiler lexically scans the base class of a Haxe program, beginning at the entry point represented by a static main function. During the scanning process, the compiler checks for occurrences of new class names, which are also lexically scanned. The code is then parsed and type-checked before macros, optimizations and transformations are applied, resulting in a typed abstract syntax tree (AST). The abstract syntax tree is then translated to either source code or bytecode, depending on the language or platform targeted by the compiler.

The Haxe standard library consists of a general purpose API, a system API and target specific APIs. The

general purpose API contains classes describing data types, data structures and algorithms which can be used on all targets. The system API contains file system and database APIs, and can therefore only be accessed when compiling to targets supporting such operations. These targets include C++, C#, Java, Neko, and PHP. Target specific APIs contain operations unique to the targeted language or platform, and can only be accessed when compiling to the chosen target.

3. GAME DEVELOPMENT AND IMPLEMENTATION

Two versions of the game were developed for the HTML5 platform. One was developed in the platform's native environment using JavaScript. The other was developed in Haxe and compiled for the HTML5 platform into Javascript source code. Both versions were built around game development frameworks, with the Javascript version using Phaser, and the Haxe version using HaxePunk.

The game was developed to utilize the 2D HTML5 canvas rendering context at a 640px by 480px resolution, rendering at a fixed frame rate of 60 frames per second.

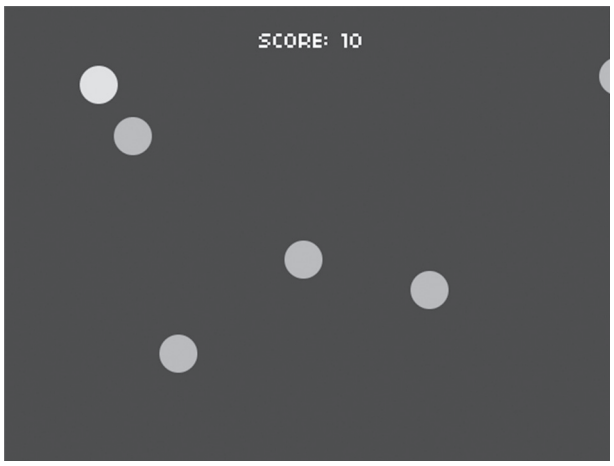

Fig. 1. Game screen during gameplay

In the game, the player is tasked with juggling a ball in mid-air while avoiding collision with obstacles. A point is gained for every second the ball is kept in play. The ball's position is fixed on the abscissa, but variable on the ordinate. This is due to gravitational acceleration affecting the body of the ball. Player input gives the ball an upward velocity for the frame it passed, resulting in the ball's jumping motion. Obstacles are created at the far right side of the game window at varying points along the ordinate. As the game progresses, they move along the abscissa at varying speeds towards the left side of the screen where the player-controlled ball resides. Colliding with an obstacle or leaving the bounds of the game window removes the ball from play. This stops further creation of obstacles, thus ending the game. Once all existing obstacles leave the bounds of the game window, the game can be restarted. Figure 1 shows the game screen during gameplay, where the white ball is controlled by the player, while the blue balls are obstacles that player's ball must not collide with.

While the two game versions are functionally the same, disparity between frameworks called for different implementations of specific game elements. Phaser requires the Javascript version to use external raster graphics to display the ball and obstacles, while the Haxe version uses HaxePunk to create necessary bitmap images. Unlike HaxePunk, Phaser features a built-in physics system which handles the required calculations regarding the player-controlled ball. Because of this, variables used in the calculations needed to be adjusted in order to achieve a similar playing experience in both versions of the game.

4. RESULTS

Game performance was evaluated by comparing frame rates achieved in both game versions during an average gaming session. Higher and consistent frame rates are more desirable, offering a smoother and more responsive gameplay experience. Lower and inconsistent frame rates result in instabilities and difficulties while playing.

Performance was analysed by comparing the minimum, the maximum and an average frame rate attained throughout the course of a single gaming session. Population variance of the recorded frame rate data set was calculated to determine frame rate consistency. Lower variance values indicate a more consistent frame rate due to observed data points close to the determined average, while higher variance values indicate frame rate instability. Due to the fast-paced nature of the game, testing sessions lasted up to two minutes in length. Both versions were tested on three different computer hardware configurations, each representing a high-end, an average and a low-end PC. All tests were performed using the latest stable version of the Google Chrome web browser (37.0.2062.124 m).

Performance test results across different computer hardware configurations for the Javascript version and results for the Haxe version are presented in Table 1 and Table 2, respectively.

Table 1. Haxe version performance test results

Haxe version	Observed performance of the Haxe game version across different PCs		
Observed values	**High-end PC results**	**Average PC results**	**Low-end PC results**
Minimum frame rate	58.82	58.48	55.87
Maximum frame rate	60	60	60
Average frame rate	59.65	59.64	59.36
Variance	0.0937	0.1111	0.2942

Table 2. Javascript version performance test results

Javascript version	Observed performance of the Javascript game version across different PCs		
Observed values	**High-end PC results**	**Average PC results**	**Low-end PC results**
Minimum frame rate	60	59	59
Maximum frame rate	60	60	60
Average frame rate	60	60	60
Variance	0	0.0528	0.0751

By comparing the results shown in Table 1 and Table 2, it is clear that the Haxe version of the game has a minimally lower frame rate across all computers it was tested on. This is evident from the observed average frame rate and variance. In contrast, the Javascript version of the game has a stable and consistent frame rate across all computers, as evidenced by the observed average frame rate in Table 2. Some slight deviation from the mean frame rate did occur on average and low-end computers. However, it was not frequent enough to affect the average frame rate, making it unobtrusive during the testing session.

As evidenced by data in Table 1 and Table 2, game performance was not found to be noticeably affected by the disparity between computer hardware configurations used. However, this was expected due to game's simplicity.

With both versions of the game achieving the game's maximum frame rate of 60 frames per second, the authors concluded that such simple game is not adequate for testing performance loss resulting from cross-compilation.

In order to improve the experiment, the authors increased the complexity of the game by adding extra objects to the game. Extra objects were added to both native and Haxe versions of the game.

The levels of added complexity will be referred to as low, medium and high complexity levels for 1,000, 5,000 and 10,000 extra objects added, respectively. Extra objects added to the game had the same properties as the obstacles in the game, with the exception of their bitmap image being transparent, making them invisible to the player. Also, the game was configured such that collisions of a player object with extra objects did not result in the "Game Over" screen. This configuration enabled the authors to artificially increase computational complexity of the game, thus making performance loss of cross- compilation evident, while not affecting the gameplay itself.

Tables 3 and 4 show the results of testing the gameplay with added complexity levels for Haxe and Javascript versions, respectively. The metrics shown in the aforementioned tables are the same metrics used in the previous experiment: the minimum and the maximum frame rate, an average frame rate and variance. Additionally, for each complexity level of the game, Tables 3 and 4 state the authors' subjective evaluation of whether the game is playable at that combination of hardware configuration and complexity level. The authors rated the game as *playable* if it averaged above 24 frames per second, without perceivable dips in the frame rate or lags during gameplay.

Table 3. Haxe version performance test results with added complexity

Haxe version		Extra Objects		
		1000	**5000**	**10000**
Computer	**Metric**	**Frames Per Second**		
Low-end	**MIN**	47.34	16.67	7.97
	MAX	60	18.94	9.44
	AVG	59.45	17.70	8.99
	VAR	1.3437	0.1129	0.0308
	NOTE	Playable	Unplayable	Unplayable
Average	**MIN**	54.05	13.26	7.08
	MAX	60	14.71	8.95
	AVG	59.75	13.62	8.19
	VAR	0.2155	0.0788	0.2514
	NOTE	Playable	Unplayable	Unplayable
High-end	**MIN**	59.17	25.28	13.12
	MAX	60	29.41	17.54
	AVG	59.8	28.57	13.53
	VAR	0.0341	0.8907	0.3833
	NOTE	Playable	Playable	Unplayable

Table 4. Javascript version performance test results with added complexity

JavaScript version		Extra Objects		
		1000	**5000**	**10000**
Computer	**Metric**	**Frames Per Second**		
Low-end	**MIN**	56	46	28
	MAX	60	60	60
	AVG	58.84	57.45	57.01
	VAR	0.3561	3.1547	15.7804
	NOTE	Playable	Playable	Playable
Average	**MIN**	58	47	30
	MAX	60	60	60
	AVG	59.51	57.14	54.44
	VAR	0.2734	2.816	12.1574
	NOTE	Playable	Playable	Playable
High-end	**MIN**	60	55	46
	MAX	60	60	60
	AVG	60	58.82	58.03
	VAR	0	0.4387	3.1387
	NOTE	Playable	Playable	Playable

Generally speaking, the Javascript version (Table 4) of the game performs much better than the Haxe version, running at almost consistent 60 frames per second on average across all three different computer configurations used for testing, on all complexity levels. The authors'

subjective evaluation of game performance on all combinations of hardware configuration and complexity levels is that the game is highly playable. The minimum frame rate shown in the table occurs at the beginning of each round, while the game is still loading assets and creating objects, and it does not affect gameplay. After the initial dip in the frame rate, the game reaches a stable frame rate that is very close to a maximum of 60 frames per second.

In comparison, the Haxe version suffers from an unstable, albeit high frame rate which is capable of causing visible latency issues. On a low complexity level, the game was highly playable on all hardware configurations, reaching a high stable frame rate, which was very close to a maximum of 60 frames per second. On a medium complexity level, the game was playable only on high-end configuration, reaching an average frame rate of 28.57 frames per second. On medium and low-end configurations, the game was rated as unplayable, with average frame rates well below the 24 frames per second threshold. On a high complexity level, the game was rated as unplayable on all hardware configurations, with average frame rates well below the 24 frames per second threshold.

5. CONCLUSION

In this paper, the authors have developed a simple 2D game in both the Haxe programming language and the native environment and evaluated the difference in performance between the two. This paper shows that the Haxe programming language is a viable alternative to development of 2D games in native environments, and performance loss attained through cross-compilation is not big enough to justify the time that would be required to develop the game in each of intended platforms.

The results show that the difference in performance of Haxe and a native version of the game is minimal for a game of low to medium complexity. It is evident that the real difference in performance starts with games of medium complexity (with around five thousand moving objects on the screen at one time). Positive aspects of cross-platform development in Haxe far outweigh the negative ones observed through performance analysis. In the opinion of the authors, the difference in performance is worth using Haxe for cross-platform development instead of developing a separate version in a native environment. Future work consists of developing the game in all of the platforms the Haxe programming language is capable of exporting to and evaluating if performance differences are as small as in the case of the Javascript version, while taking into account the cost of development of the game for all targeted platforms.

REFERENCES

[1] M. Nayak, "FACTBOX - A Look at the $66 Billion Video-Games Industry", Reuters, 10-Jun-2013, http://in.reuters.com/article/2013/06/10/gameshow-e-idINDEE9590DW20130610 (accessed: September 28, 2014).

[2] V. Landsman, S. Stremersch, "Multihoming in Two-sided Markets: An Empirical Inquiry in the Video Game Console Industry", Journal of Marketing, Vol. 75, No. 6, 2011, pp. 39–54.

[3] J. Babb, N. Terry, "Comparing Video Game Sales by Gaming Platform," Southwestern Economic Review, Vol. 40, No. 1, 2013, pp. 25–46.

[4] C. Donnellan, "An Empirical Study on Cross-Platform Game Development", 2010, http://www.smu.edu/~/media/Site/guildhall/Documents/Theses/Donnellan_FinalThesis.ashx?la=en (accessed: September 28, 2014).

[5] A.N. Duc, A. Mockus, R. Hackbarth, J. Palframan, "Forking and Coordination in Multi-platform Development: a Case Study", Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Torino, Italy, September 18-19, 2014, p. 59.

[6] G. Hartmann, G. Stead, A. DeGani, "Cross-platform Mobile Development", Tribal Linc. House Paddocks, Technical Report 2011.

[7] H. Heitkötter, S. Hanschke, T.A. Majchrzak, "Evaluating Cross-platform Development Approaches for Mobile Applications", Web Information Systems and Technologies, Springer, 2013, pp. 120–138.

[8] M. Palmieri, I. Singh, A. Cicchetti, "Comparison of Cross-platform Mobile Development Tools", Proceedings of 16th International Conference on Intelligence in Next Generation Networks, Berlin, Germany, October 8-11, 2012, pp. 179–186.

[9] A. Puder, I. Yoon, "Smartphone Cross-Compilation Framework for Multiplayer Online Games", Proceedings of 2nd International Conference on Mobile, Hybrid, and On-Line Learning, Saint Maarten, Netherlands, Antilles, February 10-16, 2010, pp. 87–92.

[10] S. Ortiz Jr., "Computing Trends Lead to New Programming Languages", Computer, Vol. 45, No. 7, July 2012, pp. 17–20.

[11] Toolkit Introduction, http://haxe.org/documentation/introduction/toolkit-introduction.html (accessed: September 29, 2014).

[12] F. Ponticelli, L.M. Sylveste, "Professional HaXe and Neko", Wiley, 2008.