ISSN 1330-3651(Print), ISSN 1848-6339 (Online)
DOI: 10.17559/TV-20140312131425

OPTIMISATION OF AUTOMATIC VARIABLE GRAPHIC LAYOUT AND IMPOSITION

Sanja Brekalo, Klaudio Pap, Nikolina Stanić

Original scientific paper
The benefit of automation in graphic prepress lies primarily in an increase of efficiency in the production of layout and imposition of repetitive tasks by shortening lead time. Another benefit is cost reduction, since more work can be done with fewer staff members. One way of automating is by using scripts. Scripts can be written to execute just one task, complex scripts can perform multiple tasks and some scripts automate the entire publishing process. With scripts that execute complex tasks execution period can be time consuming. This paper describes the script that was made to automate variable data layout, imposition and to number digital printing. Quantitative research was carried out in this paper to determine achieved automation and time reduction by using automated processes. This paper also deals with written script optimization in order to shorten the time of its execution and to identify factors that affect the script execution time.

Keywords: *automation; imposition; numeration; optimization; scripting; variable graphic layout*

Optimizacija automatizacije proizvodnje varijabilne grafičke pripreme i montaže

Izvorni znanstveni članak
Prednosti automatizacije u grafičkoj pripremi primarno leže u povećanju efikasnosti u produkciji pripreme za tisak i montaže, posebice u poslovima koji se ponavljaju. Ostale prednosti su smanjivanje troškova pošto se više posla može odraditi s manje zaposlenika. Jedan od načina izrade automatizacije je i skriptiranje. Mogu se izvršavati kratke skripte koje odrađuju samo jedan zadatak, kompleksne skripte koje obavljaju višestruke zadatke, pa sve do onih koje automatiziraju cijeli izdavački proces. Kod skripti koje obavljaju kompleksne zadatke vrijeme izvršavanja može biti dugotrajno. U radu je opisana skripta koja ostvaruje automatizaciju izrade varijabilne grafičke pripreme, montaže i numeriranja za digitalni tisak. Kvantitativno su određene dobivena automatizacija i ušteda u vremenu korištenjem automatiziranog procesa. Osim učinkom automatizacije na stvarni proces grafičke pripreme ovaj rad bavi se i optimizacijom napisane skripte kako bi se skratilo vrijeme njezinog izvršavanja i utvrdili čimbenici koji utječu na skriptna vremena.

Ključne riječi: *automatizacija; montaža; numeriranje; optimizacija; skriptiranje; varijabilna grafička priprema*

1 Introduction

Automatic layout and composition technology is of great value to end-to-end digital publishing solutions because it can relieve or eliminate the bottleneck of creating documents composed of highly customized text and image contents. It is also a very challenging technical problem since it involves 2D optimization of positions and dimensions of multiple types of contents: images, texts and vector graphics [1].

There has been extensive research in this area. One of the earliest efforts may be attributed to the Juno-2 constraint-based drawing editor, developed by Heydon and Nelson in the early 1990s [2]. Jacobs et al. [3] introduced an adaptive document layout system that automatically selects the best template for given contents. Purvis et al. [4] formalized the creation of personalized documents as a multi-objective optimization problem and used a genetic algorithm to automatically assemble such documents. Constrained satisfaction, learning and genetic algorithms are other techniques that have been explored in automated layout systems [5, 6]. Johari et al. [7] created a specialized pagination and layout system for yellow pages. Berkner et al. [8] introduced a method to intelligently scale picture and text portions of an image by utilizing information available in the JPEG2000 file. There were also propositions on image layout algorithms to automate the design of photo albums [9, 10].

Many researchers in the field of automated layout find problematic the fact that many layouts are created manually by a designer [4, 5, 11, 12]. This has led to research in automated layout of information.

The most common technique for doing automated layout is the use of templates. There are also some available software solutions which automate the data flow and creation of page layout according to rules or templates which are produced manually. There are also solutions based on scripts or plug-ins for DTP (Desktop publishing) software.

Some ***advantages of automatic publishing*** are:

- The production will increase by shortening lead time [12].
- The production cost will decrease since more work can be done with fewer working forces.
- There is a potential for automatic generation of personal documents by using digital presses [11].
- Styling a large amount of text and flowing content into templates are examples of work that should be automated since this kind of work is highly repetitive, can easily fail, and is tiresome to be carried out manually.
- Automation can be set on in non-working time of the company [13].

Criteria for job selection in automatic publishing:

- The design is rule or template based with simple or regular design and layout
- The material consists of many pages
- The material is frequently published with data originating from a database, or they can be made by a script
- The work is tedious to be done manually [13].

In this paper authors decided to investigate a different approach to automation. Instead of automation of graphic layout in general, the decision has been made to automate only certain parts of the layout process by identifying jobs that were suitable for automation with scripting.

The goal of the research was to prove that with new proposed models, which use scripting languages, graphic layout can be optimised by increasing speed, reliability and automation level of the process. Hypothesis was that by analysing graphic prepress it is possible to detect jobs suitable for automation by means of scripting. Jobs that need a lot of manual labour and that are repetitive can be optimally automated.

Automation does not always pay-off, since it requires development of many scripts [12].

Decision on job selection was made by observing graphic prepress processes. Criteria for job selection were as follows: the job had to be a repetitive task (not creative) and written script could be used for different jobs of the same nature.

This project aimed at analysing the problem in its real context. The aim was also to give recommendations of practical value. Selected job was a production of automatic variable graphic layout, imposition and numbering. Created automation script was tested and optimised to determine factors that affect script time.

Variable data printing is produced by commercial tools that suffice the needs of digital printing but most of them are either expensive or they are linked to a certain digital printer machine producer. Acquiring such tools for digital printers is often an expense that is difficult to remand. Avoiding such costs can be done by implementing new models [14]. Scripting is an option that could be used to implement new models for printing on demand.

By using scripts in DTP programs graphic prepress can be automated and time savings can be obtained [15]. It is known that certain tasks in graphic layout can be solved by scripting, but it is not known how to use such technologies optimally and whether larger time savings can be achieved by optimising written scripts. Assumption is made that by optimising written scripts larger time savings can be made.

Results from this paper give knowledge that can increase usage of scripting technologies, as well as guidelines on script optimisation.

For short scripts optimisation is not necessary but for scripts that execute with longer time it is advisable to optimise them. With script optimisation execution time can be significantly reduced. Operator waiting time is thus also reduced, and the operator is able to continue working in the program accomplishing other necessary tasks.

2 Scripting in DTP programs

DTP programs that are used today facilitate the production of graphic layout for print preparation and e-publications. One of the implemented features is also the ability to write scripts. A script is a series of statements that instruct an application to perform a set of tasks.

Adobe InDesign is one of the standard and most popular DTP software for professional use today. Scriptable Adobe applications support several scripting languages. In Mac OS AppleScript and JavaScript can be used and in Windows VBScript (Visual Basic or VBA) and JavaScript [16]. Adobe provides an extended implementation of JavaScript, called ExtendScript that is used by many Adobe applications that provide a scripting interface. In addition to implementing the JavaScript language according to the ECMA JavaScript specification, ExtendScript provides certain additional features and utilities [17]. The ExtendScript interpreter conforms to the current, ECMA 262 standard for JavaScript. All language features of JavaScript 1.5 are supported [18].

A problem of JavaScript programs is slow execution speed. That is because JavaScript programs are usually executed by interpreters and JavaScript has many dynamic features which must be checked at runtime. Performance improvement through the use of code optimization is an important method for making JavaScript a proper choice for building high quality software. Because a JavaScript statement executes many machine instructions, slight changes of JavaScript source code can greatly improve the performance. Code optimization can be statically applied by using source level transformation. JavaScript compilers can also adopt code optimization to generate faster target code. Even JavaScript interpreters can utilize code optimization techniques at runtime [19].

Guidelines for writing JavaScript code [20] are known and for writing efficient code one should adhere to them. But with ExtendScript the situation is different. ExtendScript uses JavaScript syntax but there are additional features and utilities that change the language. There is insufficient literature on optimising ExtendScript code even from the manufacturer Adobe System Incorporated. Information is unavailable on Adobe ExtendScript engine interpretation.

The original script that was written for testing had long execution time. Hypothesis was that by altering script code execution time could be improved and tests have been carried out to determine factors that could optimize execution time of ExtendScript code.

3 Experimental research

3.1 Materials and methods

For research Adobe InDesign CS6 version 8.0.1. was chosen. Script was written in ExtendScript version 4.2.12. and DOM (Document Object Model) version 8.0. Scripts were written in Adobe ExtendScript Toolkit CS6 version 3.8.0.12. application used for writing and testing ExtenScript scripts.

Computer characteristics: OS Windows 7 Professional, 64 bit, Intel Core ™ 2 Quad CPU Q8200, 2×2,33 GHz, 4,00 GB RAM.

3.2 Script used for measurements

Script used for testing was a script written to automate production of variable graphic layout, imposition and numbering for digital printing. The final product of the script is a collection of pages, with an imposed PDF document, which has placed numeration on it. Script is used for preparation of sheets for digital printing of numerated tickets, vouchers, coupons and other similar numerated materials.

Operation of the script is as follows:

a) Determining imposition sheet size and a PDF document that will be imposed (user input).
b) Setup of imposed item information: width, height, bleeds size, number of numerated blocks per item, leading zero number for number creation (user input).
c) Positioning blocks for numeration on the imported item (user action).
d) Imposition setup by choosing one of the two options- with item rotation for 90 degrees or without (user choice).
e) Imposition and drawing crop marks (script)
f) Numeration setup by choosing if the numbers are incremented over the sheet or through imposed sheets (for numerated blocks of materials) and the range of numbers (user choice).
g) Numeration (script).

Script execution process can be seen on Fig. 1. Fig. 2 shows the final product after script execution.

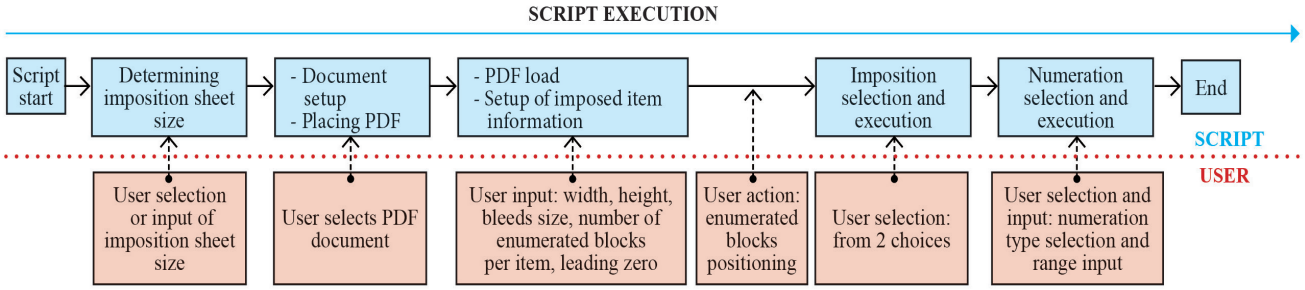

Figure 1 Steps in script execution process

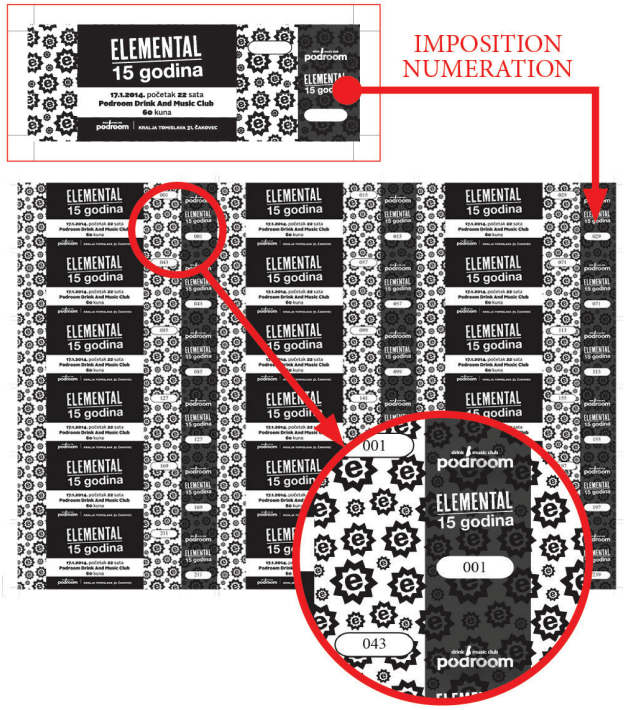

Figure 2 Imposition and numeration on the imported item

3.3 Time measurements

Time measurements were conducted by a script with "$.hiresTimer" – High resolution timer option which gives time in microseconds [21].

Time given in this paper is average time derived from 5 conducted measurements. Script time is given as imposition, numeration and idle time.

All steps of script execution except numeration and imposition are user influenced so when determining script optimization time they were not measured. User time was only included in the final comparison of manual and script time.

Idle time is time that passes after script execution until the moment when the operator can continue his operation in the program (it includes screen redraw and other tasks conducted by the program after script execution). Idle time is also measured by the script with *event listeners*. InDesign *idle tasks* are executed when there are no other tasks in the application processing order to be executed [22].

Script and manual layout time were all measured by doing the same imposition with one PDF document. Because the script is dynamic and execution depends on user selection all measurements were done with this fixed parameters: imposed sheet size 450×320 mm; PDF document size 2,32 MB; PDF imposed item size on the sheet 85×54 mm, bleeds 3 mm, imposition option set to head up – 20 items per sheet, no leading zeroes, numeration options set to increment through the imposed sheets with numeration range from 1 to 1000.

3.4 Measuring automation

Automation was measured as number of clicks and keystrokes with a program WhatPulse 2.3.1 (http://www.whatpulse.org/). The numbers were compared between jobs being carried out as manual process and executed by the script.

4 Results and discussion

4.1 Execution time of the original script

Testing options; number of numerated blocks per item on the imposition: 1, 2, 3, 4 and 5.

Table 1 Original script execution time in seconds

Number of numerated blocks	1	2	3	4	5
Imposition time	0,82	0,95	1,14	1,25	1,40
Numeration time	30,84	73,46	127,44	196,22	273,55
Idle time	26,14	51,09	112,91	192,65	323,28
Total time	57,80	125,50	241,49	390,12	598,23

From the results in Tab. 1 it can be concluded that the imposition time is short and that by adding numeration

frames it doesn't increase significantly. Numeration and idle time are high and both of them should be optimized.

4.2 Testing *doScript* method by setting *UndoModes*

Testing options; number of numerated blocks per item on the imposition: 2.

One of the script optimizations that Adobe advises is using *doScript* method and its optional parameter *UndoModes* which determines how the program will manage its *Undo* option while executing a script.

Adobe claims: "InDesign gives you the ability to undo almost every action, but this comes at a price: for almost every action you make, InDesign writes to disk. Tools presented by the user interface used for normal operation do not present any problem. For scripts, which can perform thousands of actions in the time a human being can blink, the constant disk access can be a serious drag on performance. The doScript method offers a way around this bottleneck performance by providing two parameters that control the way that scripts are executed relative to InDesign's Undo behaviour. " [22]

"UndoModes" [22] options in Table 2 and Fig. 3 are:

1. *UndoModes.autoUndo* - add no events to the Undo queue; **2.** *UndoModes.entireScript* - put a single event in the Undo queue; **3.** *UndoModes.fastEntireScript* - put a single event in the Undo queue; **4.** *UndoModes .scriptRequest*: Undo each script action as a separate event; **5.** *UndoModes* parameter is not set **6.** Original script without *doScript* method.

Table 2 Script execution time in seconds measured while varying values of *UndoModes* parameter of *doScript* method

UndoModes options	1.	2.	3.	4.	5.	6.
Imposition time	0,98	0,95	0,98	0,98	0,98	0,95
Numeration time	457,04	459,98	300,32	73,39	73,81	73,46
Idle time	53,85	45,64	49,22	49,28	49,14	51,09
Total time	511,87	506,57	350,52	123,65	123,93	125,50

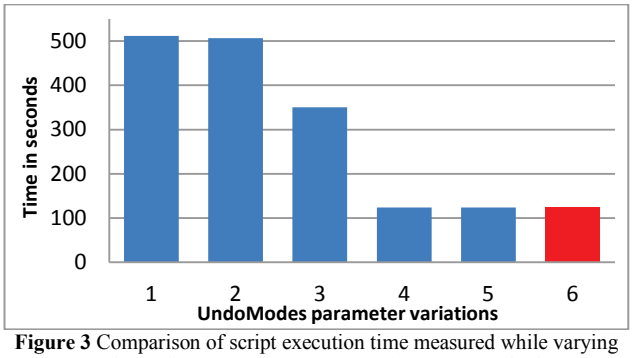

Figure 3 Comparison of script execution time measured while varying values of *UndoModes* parameter of *doScript* method

From Tab. 2 and Fig. 3 it can be concluded that the claims found in Adobe documentation in [22] are not true. While using *UndoModes* parameter in *doScript* method execution time was not optimized. Instead, execution time was increased by 3÷4 times. In scenarios 4. *UndoModes .scriptRequest*: *Undo each script action as a separate event* and 5. *UndoModes parameter is not set* time is the same as in the original script. This is because scripts execute the same way. The predefined option of *doScript* method is to write all *Undo* steps.

4.3 Using linked instead of separate text frames

Testing options; number of numerated blocks per item on the imposition: 1, 2, 3, 4 and 5.

Original script uses separate text frames for number placements. To determine if linked frames have an effect on script execution time the script was changed to use linked frames while placing numbers in text frames.

Original script execution while numerating:

The beginning of the numeration process is when imposition is created on the *master page* with all text frames placed on items (inserted PDF document). While adding new pages to the document, imposition will get copied on new pages and text frames are accessible by *overriding* them from the *master page*.

Numerating starts by adding necessary pages to the document. Number of pages is derived from user input of numeration range and items per sheet. As it is mentioned in *3.3 Time measurements* numerating is done by placing first number on the first page and second number on the second page in the same text frame from master page. Numeration option is set to increment through the imposed sheets (as opposed to numeration incrementing consecutively on a single page). Script uses 2 nested *for loops* to accomplish that. Outer loop is used to loop through frames on the page and inner is used to loop through the pages. In the inner loop selected text frame gets overridden and created number is placed in the frame. No linking of the frames is done.

Changes in the original script:

This script uses extra for loops. First for loop is used to override all text frames in the document. Then it uses extra for loop for linking text frames with consecutive numbering through the pages. Final for loop is used to place all the numbers at once in one linked *story*. This for loop has a nested loop for creating consecutive numbers that will get placed in the outer loop.

Table 3 Script execution time in seconds measured for a script using liked frames

Number of numerated blocks	1	2	3	4	5
Imposition time	0,84	0,98	1,13	1,30	1,40
Numeration time	21,28	51,31	94,61	156,17	233,08
Idle time	1,23	5,13	5,12	11,64	13,08
Total time	23,35	57,42	100,86	169,11	247,56

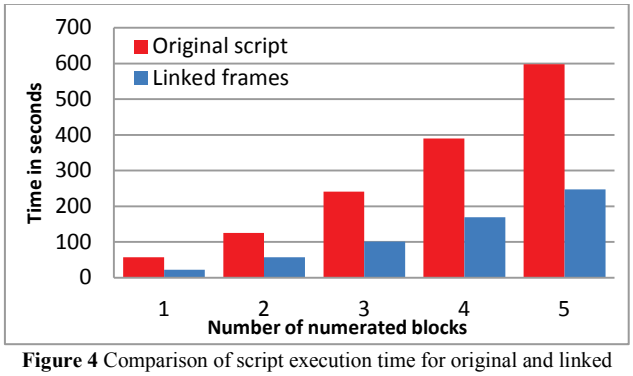

Figure 4 Comparison of script execution time for original and linked frames script

Tab. 3 and Fig. 4 show results after conducting the tests. From the results shown in Fig. 4 and by comparing time values in Tab. 1 and Tab. 3 it can be concluded that using linked frames instead of separate frames for text placement is better. Script using linked frames is in average 2,4 times quicker i.e. total time of script execution is more than halved. The greatest reduction in time was with idle time. Reasons for that could be the size of the file and Adobe InDesign script execution optimization for using linked text frames. Comparison of file sizes of original script and linked frames script is given in Tab. 4 and Fig. 5.

Table 4 File sizes in MB of InDesign documents (extension .indd) produced by original and linked frames script

Number of numerated blocks	1	2	3	4	5
Original script	8,53	15,40	22,21	28,91	35,62
Linked frames	5,76	9,75	13,76	17,79	21,79

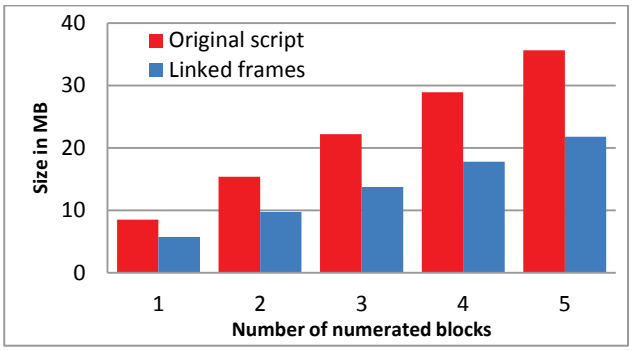

Figure 5 Comparison of file sizes in MB of InDesign documents produced by original and linked frames script

4.4 Using *everyItem()* for retrieval of data in *Collection objects*

Testing options; number of numerated blocks per item on the imposition: 1, 2, 3, 4 and 5.

Time measurements in this test were done on prepared documents with 50 pages. The same *master pages* were used like in the previous tests; with number of frames varying from 1÷5 per PDF item. Script was only unlocking, *overriding*, frames from the master. Used scripts are shown in Fig. 6.

```
//1. everyItem()
for (pg = 0; pg < 50; pg++) {
    page = myDocument.pages.item(pg);
    masterImposition.textFrames.everyItem().override(page);}
//2. item()   and for loop + extra DOM access
for (pg = 0; pg < 50; pg++) {
    page = myDocument.pages.item(pg);
    for (i = 0, broj= masterImposition.textFrames.length; i <broj; i++) {
        masterImposition.textFrames.item(i).override(page);}}
//3. item()   and for loop
var numFrames= masterImposition.textFrames.length;
for (pg = 0; pg < 50; pg++) {
    page = myDocument.pages.item(pg);
    for (i = 0; i < numFrames; i++) {
        masterImposition.textFrames.item(i).override(page);}}
```

Figure 6 Scripts used for tests on working with *Collection objects*

Fig. 7 shows results of time measurements. From the tests a conclusion can be derived that if a script should change properties of all items in a *Collection* script will execute in less time if *everyItem()* is used. From Fig. 7 it can be concluded that if a *Collection object* is big (it has a greater number of objects in it) the difference in execution time is also bigger as opposed to working with single object access. From this test it can be also concluded that the number of DOM accesses also has great impact on execution time. By using variables instead of extra DOM accesses execution time can be reduced.

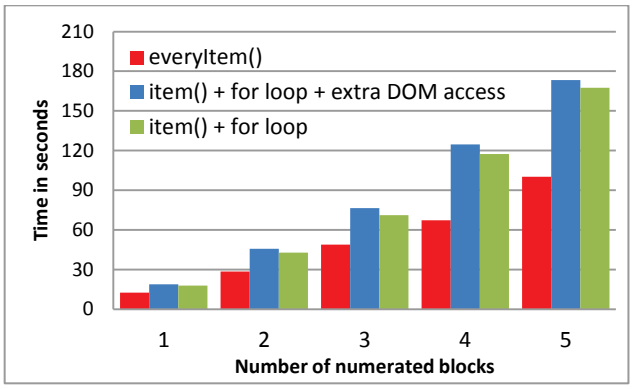

Figure 7 Execution time comparison of scripts using *everyItem*() to work with *Collection objects,* and scripts that use single and extra DOM object access

4.5 Testing the execution speed of *while* and *for* loops

Tests to determine the execution speed of *while* and *for* loops were done with a specially written script for the tests. Used script can be seen in Fig. 8 and results are shown in Fig. 9.

```
//variable number is set in tests to values:
//100.000, 1.000.000, 10.000.000, 100.000.000
a = b = c = num2 = 0; num1 = number;
//1. for
for (num = 0; num < number; num++) {a++;}
//2. while –
while (num1--) {b++;}
//3. while +
while (num2 < number) {c++; num2++;}
```

Figure 8 Scripts used for tests on *for* and *while* loop execution speeds

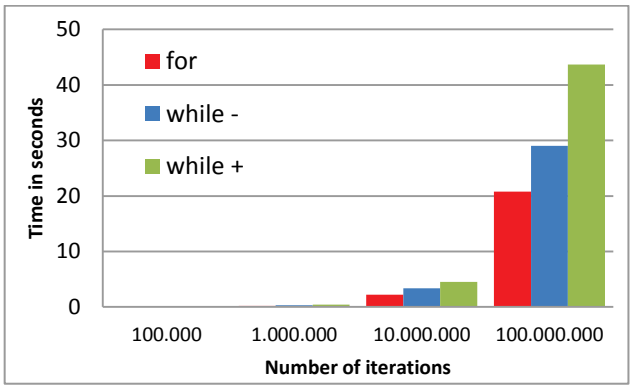

Figure 9 Comparison of results on *for* and *while* loop execution speeds

From the results it can be concluded that *for* loop executes in shorter time than *while* loop. Execution time of *while* loop is dependent on the way the loop executes, if it executes by incrementing or decrementing the variable in the condition. If the *while* loop uses decrementing it will execute in shorter time. These tests of possible optimization of the code were a JavaScript optimization. The script did not have any access to the DOM. Time savings were relatively small, except in the last example with 100 000 000 iterations of the loops, a situation that is not so common. By applying this

optimization to the script really small time savings can be obtained. Because of that it can be concluded that by optimizing functions or tasks that do not access the DOM larger time savings in script execution time can't be obtained.

4.6 Final script optimisation

Testing options; number of numerated blocks per item on the imposition: 1, 2, 3, 4 and 5.

To prepare the script for use all the conclusions from tests and literature research were implemented in the final script. These optimizations were performed:

Managing variables scope;

When JavaScript code is being executed, execution context is created. The execution context (sometimes called the scope) defines the environment in which code is to be executed. A global execution context is created upon script start, and additional execution contexts are created as functions are executed, ultimately creating an execution context stack where the topmost context is the active one.

Identifier resolution performance is directly related to the number of objects to search in the scope chain. The farther up the scope chain an identifier exists, the longer the search goes on and the longer it takes to access that variable. If scopes are not managed properly, they can negatively affect the execution time of the script. [23]

Because of the above mentioned reasons all variables used in the script were created as local variables as to shorten the script execution time because of execution context and smaller scope chain while resolving identifiers.

Working with Collection objects;

JavaScript regards *everyItem()* as a single object, even though any property or method being invoked on that entity sends a command that multiple actual objects will receive. This mechanism is known as a *verb-first command*: instead of invoking the specified method on each receiver a single method performs the action (or verb) on any number of objects [24].

Saving resolved references to the DOM in variables;

Script was changed to save in local variables resolved references to the DOM that will be used multiple times. Thus once resolved references do not have to get resolved multiple times and by saving resolved references number of DOM accesses is smaller [24].

The best option for working with Collection object references was function *getElements()*. This function resolves references to *Collection objects* and saves them to a *field*.

Using only for loops;

Final script uses only for loops that were optimized not to access DOM while testing conditions.

Doing script tasks with DOM access locality

One of the biggest optimizations in script execution time was gained by rewriting the code in a way that it utilizes DOM access locality. The principle behind this optimization is shown in Figure 10.

In the script used for testing this was done by joining 2 *for* loops that were used in the linked frames script. In the linked frames script first *for* loop was used only to override frames on pages by using *everyItem()* and the second *for* loop was used for linking frames. When this two *for* loops were joined in one optimal script execution and significant time savings were gained. This is shown on Fig. 11.

From the above statement it can be concluded that when using DOM access time and space locality in a way that all tasks are performed on near objects in the DOM, or on objects that were recently selected, great time savings can be obtained. Non-optimized way of accomplishing tasks would be to solve tasks one at a time on every object as it can be seen in Fig. 10.

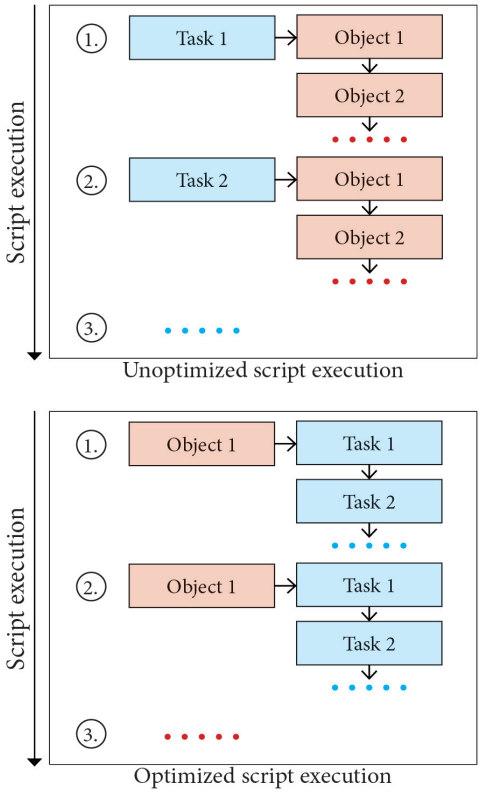

Figure 10 Principle for optimising script execution time with DOM access locality

```
//overriding text frames
for (pg = 1; pg < pgCount; pg++) {
    page = myDocument.pages.item(pg);
    masterMontaza.textFrames.everyItem().override(page);}
//linking
for (pg = 2; pg < pgCount; pg++) {
    frameNum = 0;
    while (frameNum < masterMontaza.textFrames.length) {
        txt1 = myDocument.pages.item(pg - 1).textFrames.item(frameNum);
        txt2 = myDocument.pages.item(pg).textFrames.item(frameNum);
        txt1.nextTextFrame = txt2;
        frameNum++;}}

//optimisation
var framesMaster = masterMontaza.textFrames.everyItem();
for (pg = 2; pg <= numPg; pg++) {
    framesMaster.override(pgDoc.item(pg));
    page1 = pgDoc.item(pg).textFrames.everyItem().getElements();
    page = pgDoc.item(pg - 1).textFrames.everyItem().getElements();
    for (frameNum = 0; frameNum < num; frameNum++) {
        page[frameNum].nextTextFrame = page1[frameNum];}}
```

Figure 11 Linked frames script- overriding and linking frames and optimised script example

By optimization, the script execution time can be shortened from 3 to 4 times. With longer scripts greater time savings can be achieved by optimizing. From Tab. 5 and Fig. 12 it can be concluded that by optimizing ExtendScript code significant time savings can be gained, especially for scripts that execute with long time.

Table 6 shows results after script optimization. Total time for all measurements was reduced by average 70 %, that is optimized script runs in only 30 % of original script time.

This proves the hypothesis that by optimising written scripts larger time savings can be made.

Table 5 Final script execution time in seconds

Number of numerated blocks	1	2	3	4	5
Imposition time	0,83	0,98	1,11	1,25	1,38
Numeration time	19,16	42,25	67,01	96,42	130,84
Idle time	1,11	1,77	2,63	6,16	8,51
Total time	21,10	45,00	70,75	103,83	140,73

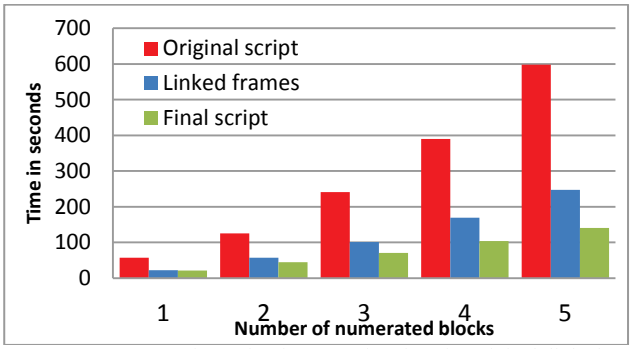

Figure 12 Comparison of script execution time for original, linked frames and final script

Table 6 Total time savings with script optimisation

Number of numerated blocks	1	2	3	4	5
Total time savings with script optimisation	64 %	64 %	71 %	73 %	76 %

4.7 Automation results

Testing options; number of numerated blocks per item on the imposition: 1, 2, 3, 4 and 5.

In these final tests comparison between manual and script job execution is made. For determining automation the number of clicks and keystrokes necessary for job execution were counted. The numbers were compared between jobs carried out as manual process and when executed by the script. Tab. 7 shows test results. The results were varying around 96 % for user input reduction and automation level was quite high.

The results supported the hypothesis that by analysing graphic prepress it is possible to detect jobs suitable for automation by means of scripting. Jobs that need a lot of manual labour and that are repetitive can be optimally automated.

Comparison between measured time of manual and script execution are given in Tab. 8.

This proves the former hypothesis that by using scripting savings in operational time can be obtained. Results show that the manual time of job production can be reduced for about 94 % by using script automation. This statement is only true when time to write and optimize scripts is not added to the production time. Time for script creation can be dismissed if the scripts are written for repetitive jobs and on different jobs of the same nature.

It is important to mention that manual time and user input by clicks and keystrokes are highly dependent on the program operator and his experience and knowledge in program and layout process. Results given in this paper are obtained by 5 measurements from 5 different experienced operators. Single results for one job can vary more than 50 % in click and keystroke count and 30 % in time measurements.

Table 7 Script and manual job execution; difference between user inputs by clicks and key strokes

Number of clicks and keystrokes	Number of numerated blocks				
	1	2	3	4	5
Manual	568	994	1182	1370	1556
Script	27	33	41	52	58
User input reduction	95 %	97 %	97 %	96 %	96 %

Table 8 Script and manual job execution; difference between average total time of job execution in seconds

Average total time	Number of numerated blocks				
	1	2	3	4	5
Manual	980	1113	1327	1482	1637
Script	28	55	84	116	154
Time reduction	97 %	95 %	94 %	92 %	91 %
Reduction factor	1:35	1:20	1:16	1:13	1:11

5 Conclusion

With scripting graphic prepress processes can be automated and time savings in layout processes can be achieved. By identifying jobs that are suitable for automation with scripting and automating only parts of the layout processes high automation level in graphic prepress can be obtained. Scripts should be written so that they can be used in different jobs of the same nature.

Scripts used for graphic prepress automation can execute with long time. While the script is running the operator cannot operate the program and execute other manual or script tasks. When scripts have long execution time they should be optimized. By script execution optimization in certain situations smaller document sizes can be achieved.

Tests conducted in this paper show that script optimization can lead to high time savings in script execution time (in the tested example time saving were 70 % better than the original script time). Greatest optimizations can be obtained when optimizing parts of the script that access the DOM. Higher time savings are obtained with long run scripts.

From conducted tests following conclusions and guidelines for script optimization have been made;

- When using *UndoModes* parameter of *doScript* method script execution time can be prolonged, so time tests should be made before using this option.
- When working with a lot of frames on pages it is desirable to work with linked frames instead of separate frames. This ensures better script execution time and smaller files sizes.
- Working with *Collection objects* is recommended whenever possible (using *everyItem*() to set the properties of *Collection objects*)
- DOM accesses should be as little as possible, because they take longer time to execute.
- *For* loops with fixed conditions (do not access the DOM in every condition check) should be preferred.

- All variables should be set to local variables as to shorten the script execution time because of execution context and smaller scope chain while resolving identifiers.
- Saving resolved references into local variables (preferred usage of *getElements*() method or saving other references in variables and reusing them when necessary).
- When using DOM access time and space locality should be used; tasks should be performed on near objects in the DOM, or on objects that were recently selected. It is better to select one object from DOM and execute all tasks on that object than executing one task at a time on all objects.

6 References

[1] Lin, X. Active layout engine: Algorithms and applications in variable data printing. // Computer-Aided Design. 38, 1(2006), pp. 444-456. DOI: 10.1016/j.cad.2005.11.006

[2] Heydon, A.; Nelson, G. The Juno-2 constraint-based drawing editor. Technical Report 131a, Digital Systems Research Center, Palo Alto, California; 1994.

[3] Jacobs, C.; Li, W.; Schrier, E.; Bargeron, D.; Salein, D. Adaptive grid-based document layout. // ACM Trans Graph. 22, 3(2003), pp. 838-847. DOI: 10.1145/882262.882353

[4] Purvis, L.; Harrington, S.; O'Sullivan, B.; Freuder, E. C. Creating personalized documents: an optimization approach. // ACM symposium on Document engineering / Florence, Italy, 2003, pp. 68-77.

[5] Lok, S.; Feiner, S. A Survey of Automated Layout Techniques for Information Presentations. // Proceedings of the Smart Graphics Symposium / Hawthorne, NY, USA, 2001, URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.9.8736&rep=rep1&type=pdf (13.09.2013).

[6] Geigel, J.; Loui, A. Using Genetic Algorithms for Album Page Layouts. // IEEE MultiMedia. 10, 4(2003), pp. 16-26. DOI: 10.1109/MMUL.2003.1237547

[7] Johari, R.; Marks, J.; Partovi, A.; Shieber, S. Automatic yellow-pages pagination and layout // Journal of Heuristics. 4, 2(1997), pp. 321-342. DOI: 10.1007/BF00132503

[8] Berkner, A.; Schwartz E. L. SmartNails: Display- and image-dependent thumbnails. // SPIE conference on document recognition and retrieval XI / San Jose, 2004, pp. 54-65.

[9] Geigel, J.; Loui, A. Using Genetic Algorithms for Album Page Layouts // IEEE MultiMedia. 10, 4(2003), pp. 16-26. DOI: 10.1109/MMUL.2003.1237547

[10] Atkins B. Adaptive photo collection page layout. // International conference on image processing / Singapore, 2004, pp. 2897-2900

[11] Goldenberg, E. Automatic layout of variable-content print data. MSc Dissertation. School of Cognitive & Computing Sciences, University of Sussex, Brighton, UK, 2002. URL: http://www.hpl.hp.com/techreports/2002/HPL-2002-286.pdf (13.09.2013).

[12] Grahn, K. J. Efficient production of uniform layout. // Proceedings of the 1st international IARIGAI student conference on print and media technology / Chemnitz, Germany, 2005, pp. 144-148.

[13] Grahn, K. J. Catalogue Production Automation – Case Studies, Master's Thesis in Publishing Technology at the School of Computer Science and Engineering, Royal Institute of Technology, Stocholm, Sweden, 2006. URL: http://kiosk.nada.kth.se/utbildning/grukth/exjobb/rapportlistor/2006/rapporter06/grahn_karl-johan_06063.pdf (18.01.2013).

[14] Zhao, Y.; Sun, W. Practice of imposition and illustrator variable data plate making with barcode. // 17th IAPRI World Conference on Packaging / Tianjin, China, 2010, pp. 251-253.

[15] Brekalo, S.; Knok, Ž.; Breslauer, N. Optimizacija procesa prijeloma publikacija korištenjem skriptiranja // Međunarodni znanstveni skup Tiskarstvo & Design / Tuheljske toplice, 2013.

[16] Adobe Systems Incorporated, Adobe Introduction to Scripting. Adobe Systems Incorporated, California, USA, 2012.

[17] Adobe Systems Incorporated, JavaScript Tools Guide. Adobe Systems Incorporated, California, USA, 2012.

[18] Adobe Systems Incorporated, Adobe InDesign CS6 Scripting Tutorial. Adobe Systems Incorporated, California, USA, 2012.

[19] Dongseok, J.; Kwang-Moo, C. Points to Analysis for JavaScript. // SAC '09 Proceedings of the 2009 ACM symposium on Applied Computing / Honolulu, 2009, pp. 1930-1937

[20] Herczeg, Z.; Lóki, G.; Szirbucz, T.; Kiss, Á. Guidelines for JavaScript Programs: Are They Still Necessary? // Proceedings of the 11th Symposium on Programming Languages and Software Tools (SPLST'09) and 7th Nordic Workshop on Model Driven Software Engineering (NW-MODE'09) / Tampere, Finland, 2009, pp. 59-71

[21] Autret M. Comparing the Performance of ExtendScript Snippets. 2011. URL: http://www.indiscripts.com/post/2011/06/comparing-the-performance-of-extendscript-snippets. (19.09.2013).

[22] Adobe Systems Incorporated, Adobe InDesign CS6 Scripting Guide: JavaScript. Adobe Systems Incorporated, California, USA, 2012.

[23] Souders S. Even Faster Websites. O'Reilly Media Inc., Sebastopol, USA, 2009.

[24] Autret, M. On 'everyItem()', 2010. URL: http://www.indiscripts.com/post/2010/06/on-everyitem-part-1. (19.09.2013)

Authors' addresses

Sanja Brekalo, M. Sc.
Međimurje University of Applied Sciences in Čakovec
Bana Josipa Jelačića 22a, 40000 Čakovec, Croatia
E-mail: sbrekalo@mev.hr

Klaudio Pap, PhD
University in Zagreb, Faculty of Graphic Arts
Getaldićeva 2, 10000 Zagreb, Croatia
E-mail: klaudio.pap@grf.hr

Nikolina Stanić Loknar, PhD
University in Zagreb, Faculty of Graphic Arts
Getaldićeva 2, 10000 Zagreb, Croatia
E-mail: nikolina.stanic@grf.hr