4KB PRESENTATION OF SELECTED 3D GRAPHICS RENDERING ALGORITHMS AS AN EXAMPLE OF OPTIMAL IMPLEMENTATION

Cezary Nalborski

Systems Research Institute, Polish Academy of Sciences
Ph.D. Studies, Warsaw, Poland,
e-mail: *cnalborski@gmail.com*

Abstract.. Progressive miniaturization of devices running programs imposes such requirements as more rapid execution or less resource consumption, for example, memory or battery. The size of the executable program should also be taken into account, especially regarding embedded systems that have programs built into their memory. In this paper, a brief description of the computer software optimization problem is given. A broad definition of the process of optimization and some general aspects are presented by example of a program written in Motorola 68k assembler whose size of the executable file does not exceed 4 kilobytes. The program implements two popular 3D graphics rendering algorithms.

Key words: 3D graphics, rendering algorithms, optimal implementation, assembler, embedded systems

1 INTRODUCTION

1.1 Program optimization

In the software development process, we sometimes face the problem of optimization. In some cases, requirements regarding optimization appear at the beginning of the development process and are given directly, so that the program developed has to meet them, or the target platform imposes the requirements. For example, in embedded systems with built-in programs, because of the high memory cost the maximum available program size is often given. Sometimes, at the final stage of the development process, it appears that the program under development has to be optimized, e.g. in terms of more rapid execution, especially in programs implementing graphics rendering algorithms, or resource consumption e.g. memory usage or computer network usage.

Depending on the moment optimization is considered - whether it is the initial phase of the process of program creation, when we start from the top level design moving to more details, or the final phase when we have a running program, or a phase in the middle when we started without any particular requirements related to performance, power consumption or other resources which may imply that program optimization must be done, the following aspects shown in Table 1 should be considered:

Table 1. The program creation process stack

Layers of program creation process
ARCHITECTURE
ALGORITHMS
DATA STRUCTURES
PROGRAMMING LANGUAGE
COMPILATION
RUN-TIME

1.2 Related work

In recent years, computers have dramatically evolved from big, stationary and isolated structures to sleek and small and portable devices frequently connected to the Internet. Today, we speak about devices running programs, not only computers, as we used to say, which include smartphones, tablets, smartwatches and many others that may be used to bring AI and automation to our lives. A group separate consists of embedded systems.

This progressive miniaturization, portability and inter-networking generates new requirements for modern appliances in which such parameters as program performance, memory usage or resource consumption, such as power or network usage in terms of the volume of transferred data are crucial. In some applications such as embedded systems, the size of the executable program is also important.

The performance of a program is one of the most important aspects discussed here. Depending on the end user to which a program is directed, it may refer to the graphics rendering performance where the graphical user interface is responsible for fast delivery of back-end computations or where it plays the main role, e.g. in computer games entertainment. It may also apply to back-end program computation efficiency in general.

In software optimization at the application level [12] the aim is to increase the performance of a program which, according to the researchers, may be directly affected by power-consumption and heat generated by the processors. The authors studied thermally optimized software where energy and temperature-related optimizations were taken into account. Non optimal software may thermally affect the processor running programs whose optimal temperature is necessary to maintain the expected performance.

As mentioned above, the speed of program execution is affected by power consumption. The direct consequence of miniaturization is smaller power storage efficiency because of the size of the battery that must match the size of a particular device. The evolution of batteries does not go hand in hand with device miniaturization. The smaller the device is, the less power it can deliver without recharging. So it is better for such devices to run programs faster to consume as little power as possible. In [6] some source code-level optimization techniques are examined. The results achieved showed that not all optimizations had significant effect on power consumption, moreover, some platform dependent techniques were found. In [15] new innovative algorithms co-design techniques, architectures and technology for efficient implementation are presented.

Memory usage is also considered an important factor of optimization. Memory size does not necessarily imply greater efficiency. Sometimes memory resources are strictly limited to some portion that may be used by running programs. Under such conditions a running program must be optimized to use less memory. In [14] software optimization techniques for the optimal use of memory were explored. The authors pay attention to memory-aware architecture of the resulting code which may help to reduce power usage. Optimization techniques based on real-life benchmarks report significant reduction of energy consumption and performance improvements.

In many cases the size of the executable program is an important factor. Especially in embedded systems which were described in [9]. Storing programs on chips of embedded systems is interesting from the cost efficiency perspective. The authors considered techniques of code compression which helps to produce smaller executable files. The smallest possible size seems also worth considering if we look at those devices from a perspective of inter networking. Fast access to a program from the Internet in some conditions may be a big advantage. Smaller size may also reduce the costs of transfer of the program over the Internet especially in cellular networks.

2 IMPLEMENTATION

2.1 Assumptions

For this paper, we assumed that we will create a computer program that will be a real-time presentation of two selected 3D graphics rendering algorithms, such as texture mapping [11] and Gouraud shading [8] with static light source. The size of the executable file cannot be greater than 4kb. It has to render graphics in real time (50 frames per second). As a destination device, the Motorola 68EC020 32-bit platform [4] was selected. It has strictly limited hardware capabilities in terms of the processor computing speed set at 10 MIPS and available RAM memory, set at 2MB. The device was emulated on the WinUAE [19] application running Amiga 1200 standard configuration. The base environment for WinUAE [19] was Intel Core 2 Duo P9400 2,4 GHz PC with 4 GB RAM working on the Windows XP operating system. It allowed WinUAE to emulate Amiga 1200 in real-time conditions.

As a programming language for the program [16], Motorola 68k assembler [13] was selected with AsmOne [2] as a programming IDE. To make the source code and the resulting binary machine code comprehensible and unequivocal in every aspect, no support of Amiga 1200 internal GPU or any internal Amiga operating system framework was used. Only basic arithmetic assembler instructions were used with direct access to video and RAM memory. Thanks to the fact that there is one-to-one correspondence between the instructions called mnemonics and the binary device code that will be produced in the final step of the program creation process, we expect to achieve maximum available rendering performance and the smallest size of the executable program.

The whole program - starting from data preparation at the initial phase, later data transformation such as rotations, 3D points to 2D observer scene projection and then triangular rendering algorithms with linear interpolation were implemented according to the results presented in [3,8,11].

2.2 Aspects of optimization

2.2.1 Architecture This is the top aspect of optimization with the biggest impact on the program creation process. At this step fundamental structural choices of the program have to be made. Especially, we have to decide about other layers of the program creation process lying below in the stack. We have to decide about the algorithms that will be used for all necessary computations, data structures that will store all the data, the programming language that will be used for implementation, the usage of tools in the construction of the executable file phase and all necessary or additional computations that may or will be made at the execution stage.

The basis for considerations related to the architecture is the requirements that the program must fulfill. If there is a strong need in the performance layer, we have to use some low level programming language such an assembler. Some requirements affecting the overall architecture are hidden to the process, e.g. a target device and its hardware architecture, forcing some architectural aspects such as the maximum available memory or the use of some dedicated platform dependent developing tools and languages.

It is always worth taking into account such aspects as scalability or maintainability of the program. Depending on the requirements, the easiest way to increase the performance of the program in terms of its speed and computing power is to run its multiple instances or modify the code to run parallel some threads. On the other hand, maintainability is important if there are no strong opimization requirements but some effort and cost aspects are mentioned

2.2.2 Algorithms This aspect occurs right after the architecture. The choice of efficient algorithms and optimal implementation of these algorithms affects performance more than any other aspect. Selected algorithms should always meet the following: available resources, given requirements, constraints and expected load. Sometimes, if the available environment and given requirements meet the constraints, it is worth selecting less complex algorithms. It will increase readability of the source code and make the maintenance easier or cheaper. In some cases we have to do it that way to be able to meet the expected change request time during the maintenance of the program.

2.2.3 Data structures Data structures are directly connected to previous aspect mentioned in paragraph ‘Algorithms‘. They are as important for the program as wheels are for the car. On the one hand, they must ensure storage of all necessary data for selected algorithms. On the other hand, they must meet the storage constraint given directly or indirectly by the device platform. For example, one of the fastest sorting algorithms called bucket sort or bin sort, mentioned in [17] uses a huge amount of memory. For small data sets, it would be better to use some slower algorithms, consuming less resources such as bubble sort [17], and giving lower complexity or better code readability.

2.2.4 Programming language The choice of the programming language used during the implementation phase is significant but depends more on given requirements. If the requirements specify the target device or family of devices, we may be limited to a sub-set of programming languages available to that device family. Next, if there is a requirement related to program performance in terms of the execution time or executable file size, it is always better to use as low level a programming language as possible, e.g. an assembler. The main feature of the assembly language is that its mnemonics instruction language stands in one-to-one correlation to the executed binary code. There are also executable file size benefits. The output produced will match exactly the source code, with no extra instructions written in the code. Higher level languages, depending on the compiling options often produce executable file much greater in size than those written in assembler. This happens in the translation phase from higher level language to the binary code level. Those extra add-ons often affect the performance by executing an unnecessary code, however, the higher level the programming language has, the more readable the code is. The time needed for development, especially on the maintenance level is much shorter. Finally, sometimes if the portability of the source code is not so important, a good solution is to use an assembly language that can run only on a particular device but with optimal execution performance. However, where the execution time or the executable file size is not the most important factor, using a higher level language gives us the ability to write a portable code that once written can be compiled for many different platforms.

2.2.5 Compilation During the compilation phase, the compiler tries to translate the written source code to output the executable file. In case of the assembly language, it is always done in correlation between one source code instruction called mnemonic and one binary device code. In case of higher levels, the translation process is carried out using algorithms implemented in the compiler and what is worth mentioning - compiler options. Many compilers, as default compile a less aggressive code in terms of its proper execution or platform specific code, e.g. the same code of a simple ‘while(1)‘ or ‘for(;;)‘ loop may be translated to a binary machine code differently, depending on the selected destination platform for which the binary machine code may be executed. Some of the devices use built-in memory cache which speeds up the repeated portion of code. It may generate errors during the execution phase.

Algorithms built into compilers are very complex multi-pass translators and in some way often try to 'guess' the intention. Especially in case of huge blocks of code written in a high level language, it is worth reconfiguring the compiler according to the expected optimization level. Using an assembly language, we write a code dedicated to a particular device with no extra logic during the translation phase in terms of the produced binary executable.

2.2.6 Run-time In some cases, there is a possibility to optimize at the run-time phase. Especially in case of very high level languages, using virtual machines as the run-time environment, it may be executed by tuning the virtual machine parameters according to our needs.

In case of high level languages, there is a possibility to generate a self-modifying code adopted relative to the requirements. But, this has to be done carefully and with correlation to the target device architecture and such aspects as code caching. The same self-modifying code that runs properly on one device without a built-in code caching will generate errors running on a device equipped with a code cache.

There is always a possibility of generating some data at the run-time. It particularly concerns the memory usage constraints and the size of the executable file. For example, in some 3D graphics programs, textures may be pre-calculated at the run time. It is also possible to pre-generate some back-end data used for computations in mathematical transformations.

Good results are often obtained with the use of the overlay feature of executable files. This feature helps to reduce the amount of RAM needed to run a program. At the run-time phase, only part of the executable, called the root, is loaded into the memory and executed. The rest of the executable file is split into smaller parts loaded or unloaded, according to the situation.

3 RESULTS

Based on the assumptions described in point 2.1 the following architecture was designed (Fig. 1):

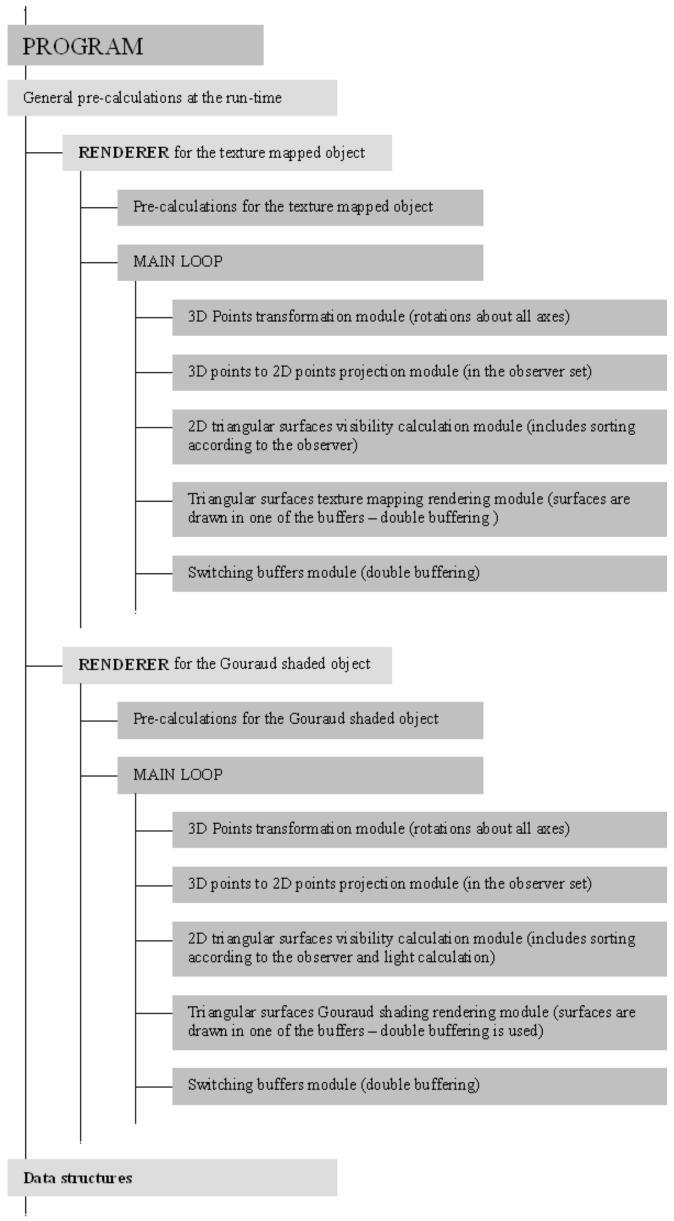
PROGRAM
General pre-calculations at the run-time
RENDERER for the texture mapped object
Pre-calculations for the texture mapped object
MAIN LOOP
3D Points transformation module (rotations about all axes)
3D points to 2D points projection module (in the observer set)
2D triangular surfaces visibility calculation module (includes sorting according to the observer)
Triangular surfaces texture mapping rendering module (surfaces are drawn in one of the buffers – double buffering)
Switching buffers module (double buffering)
RENDERER for the Gouraud shaded object
Pre-calculations for the Gouraud shaded object
MAIN LOOP
3D Points transformation module (rotations about all axes)
3D points to 2D points projection module (in the observer set)
2D triangular surfaces visibility calculation module (includes sorting according to the observer and light calculation)
Triangular surfaces Gouraud shading rendering module (surfaces are drawn in one of the buffers – double buffering is used)
Switching buffers module (double buffering)
Data structures

Fig. 1. Program architecture

To describe objects in 3D space, polygonal 3D mesh modeling was used. In this model, objects are represented as a set of points in 3D space (called vertices) and a set of triangular surfaces. As shown in Fig. 2, the front surface of the cube constructed by four vertices V_0,V_1,V_2,V_3 is represented by two triangular surfaces.

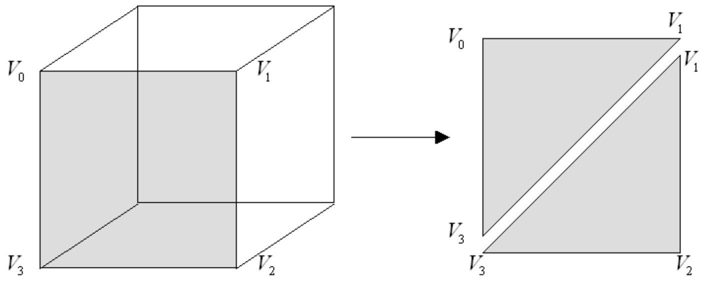

Fig. 2. Polygonal mesh model based on triangular surfaces

According to the selected 3D objects description model, data structures that are part of the program contain the following information:

- definitions of objects in 3D space
  - vertex list, every vertex described by three numbers
  - in case of the texture mapped object, there is additional information ascribed to each vertex - the u,v coordinates in the texture space
  - in case of the Gouraud shaded object, there is additional information ascribed to each vertex - the color and calculated light intensity value
  - list of triangular surfaces (every surface described as a set of three vertex indexes constructing it)
- partial texture used for texture mapping of object 1 (Fig. 3)
- partial sine table (pre-calculated first 90 degrees with 16 bit precision)
- two rendering buffers for rendered objects (only one buffer is used at a time)
- buffers for 3D points after rotations
- buffer for 2D points after projection from 3D
- buffer for average z values for every triangular surface (calculated as arithmetic average from z values of vertices constructing the surface)
- buffer for sorted surfaces indexes (the sorting is based on calculated average z value for the surface)
- buffer for light intensity values for every vertex of the rotated object (used for Gouraud shading [8])
- colour palette

In the next step, algorithms for particular components of the architecture were selected. At the bottom, before the rendering all necessary transformations must be made. These transformations are rotations about all three axes X, Y, and Z used in the same way for both objects presented in the program. The only difference is that in the second rendering module for Gouraud shaded [8] object, after the transformations, light intensity for each vertex of the object is calculated. First, the points were rotated an angle θ for the X axis using equations described in (1). Then, the result of rotation about the X axis became the input for rotation about the Y axis using equations described in (2). Finally, the result of rotation about the Y axis became the input for rotation about the Z axis using equations described in (3).

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}, \tag{1}$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}, \tag{2}$$

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{3}$$

Then, 3D points were projected to 2D viewer plane. Assuming that the normal vector of the viewing plane was parallel to one of the axes, the projection of 3D point $(v_x,v_y,v_z\ )$ to 2D point (p_x,p_y) with scale vector **s** and offset vector **t** was made using the equation described in (4).

$$\begin{bmatrix} p_x \\ p_y \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & 0 & s_z \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} + \begin{bmatrix} t_x \\ t_z \end{bmatrix}. \tag{4}$$

After that calculation, additional information such as the average Z value for every triangular surface was computed. It was calculated as an arithmetic average of all vertices constructing the triangular surface. This information was then used for surface sorting, but only surfaces visible to the observer are taken to sort. The visibility was calculated using the Back-face culling algorithm described in [7]. The dot product (5) of each triangular surface‘s normal vector and the vector from the camera to the surface is checked. If it was greater than or equal to zero, the analyzed surface was removed from the sorting and rendering process.

$$(V_0 - \mathbf{P}) \cdot \mathbf{N} \geq 0. \tag{5}$$

P is the point of view and V_0 is the first vertex of a triangular surface. **N** is the normal of the analyzed triangular surface and was calculated using the equation described in (6).

$$\mathbf{N} = (V_1 - V_0) \times (V_2 - V_0). \tag{6}$$

The point of view was assumed to be (0,0,0), if points were already in the view space. For the Gouraud shaded object, additionally, for each vertex constructing the analyzed triangular surface, the color and the light intensity value, were calculated. To calculate the light intensity value for a given vertex, the dot product of its normal vector and the vector from camera to the vertex was calculated. All vectors were normalized before calculations. The colour at a given vertex was calculated as the product of the colour ascribed to this vertex and, earlier calculated light intensity. Objects consisting of triangular surfaces were drawn from the farthest to the closest relative to the observer.

The first object (Fig. 3) was rendered using the texture mapping algorithm [11]. The renderer worked for every visible triangular surface. The rendering algorithm rasterized the whole triangular surface, beginning from the top-most vertex and moving down to the bottom-most one. At every raster line, it started from the far-left edge constructing the rendered surface, and stopped at the far-right edge constructing the surface. At the edges, the texture coordinates (u,v) were calculated by linear interpolation of texture coordinates assigned to the vertex constructing the edge. Then, texture (u,v) values calculated at the edges were interpolated through the whole raster. Each pixel of the raster was taken from the texture at the calculated u,v coordinates. The following process is illustrated in Fig. 4.

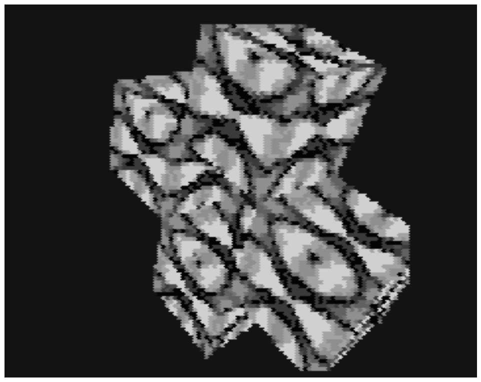

Fig. 3. The texture mapped object with leopard pre-rendered texture

To obtain a smooth animation, at the end of each iteration of the renderer the rotation angle θ was increased by value 1 until it reached 360 degrees. Then the value was zeroed. The results of the following rendering process are illustrated in Fig. 3.

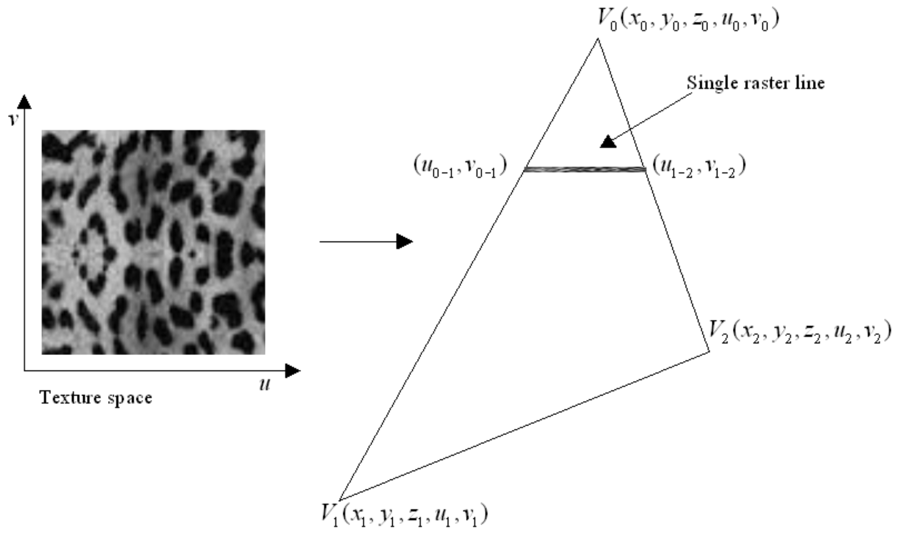
$V_0(x_0, y_0, z_0, u_0, v_0)$
Single raster line
v
(u_{0-1}, v_{0-1})
(u_{1-2}, v_{1-2})
$V_2(x_2, y_2, z_2, u_2, v_2)$
Texture space
u
$V_1(x_1, y_1, z_1, u_1, v_1)$

Fig. 4. Single surface rasterization of the texture mapped object

The second object (Fig. 5) was rendered using the Gouraud shading [8] algorithm. The renderer worked for every visible triangular surface. The rendering algorithm rasterized the whole triangular surface beginning from the top-most vertex and moving down to the bottom-most one. At every raster line, it started from the far-left edge constructing the rendered surface, and stopped at the far-right edge constructing the surface. At the edges, the colour of the pixel was calculated by linear interpolation of pixel colour intensity assigned to the vertex constructing the edge. Then, colour values calculated at the edges were interpolated through the whole raster and were taken as colours of pixels constructing the raster. The following process is illustrated in Fig. 6.

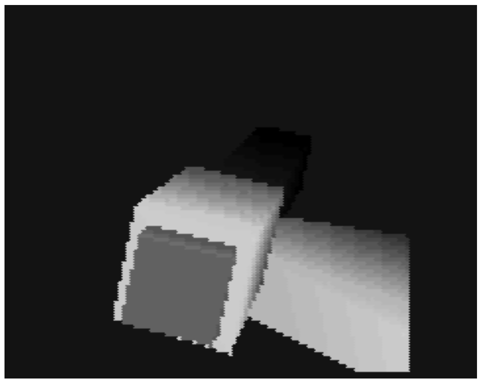

Fig. 5. The 3D Gouraud shaded object (Axe) with constant light source

To obtain a smooth animation at the end of each iteration of the renderer, the rotation angle θ was increased by value 1 until it reached 360 degrees. Then the value was zeroed. The results of the following rendering process are illustrated in Fig. 5.

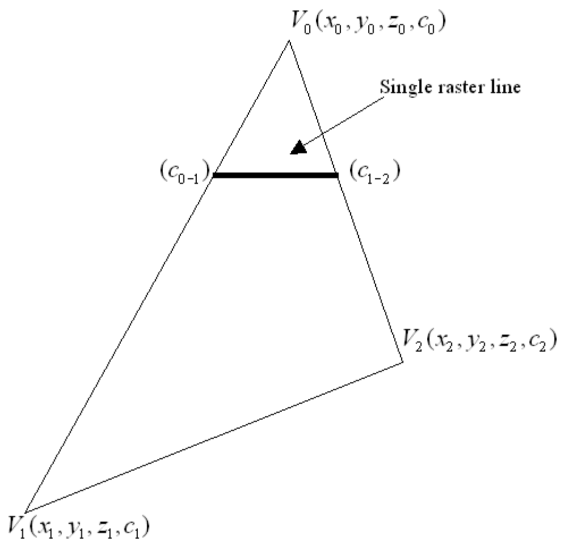
$V_0(x_0, y_0, z_0, c_0)$
Single raster line
(c_{0-1})
(c_{1-2})
$V_2(x_2, y_2, z_2, c_2)$
$V_1(x_1, y_1, z_1, c_1)$

Fig. 6. Single surface rasterization of the Gouraud shaded object

For the sorting algorithm, bucket sorting [17] was selected because of a relatively small amount of data to be sorted (a few average Z axis values for a visible triangular surface), low complexity, fast implementation and a small volume of the executable code produced. Double buffering [10,5] mentioned in the architecture model, was a simple solution for providing a stable picture of drawn objects. The aim was to show to the user already rendered surface and hide the process of rendering surface by surface. This was achieved by using two rendering buffers. While one was used for presenting the final rendering results, the other was used for current rendering operations. They were switched after every rendering iteration. At the final stage of the implementation, to reduce size of the executable file an overlay feature, combined with compression, was used. A special program was used to reduce the size of the original executable file by compressing it. But the compressed file could not be executed without a special small header (root) responsible for launching. During the run-time, only the root header loads into memory and executes. Then it loads the rest of the executable file which is compressed original executable file. After decompression in memory, it is executed. This process is illustrated in Fig. 7.

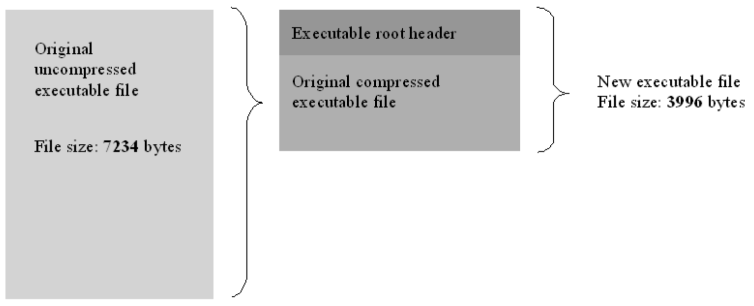

Fig. 7. Executable file size reduction with overlay feature

It is worth mentioning that some pre-calculations at the run-time were prepared (Fig. 8). First, to achieve the biggest possible and acceptable size of the texture used for texture mapping [11] a set of copy-paste and mirror-reverse transformations was performed. Second, for the rotation transformations about the three axes, the calculation for 360 degrees was made based on the source data, calculated only for the first 90 degrees of the sine function. The above pre-calculations allowed us to reduce the size of the original executable file.

The resulting executable file size was 3996 bytes [20] which gives over 80% of efficiency. The rendering frame rate per second did not exceed 50 frames per second which is a very good result. Although the selected rendering algorithms were quite complex, it was done smoothly, in real-time

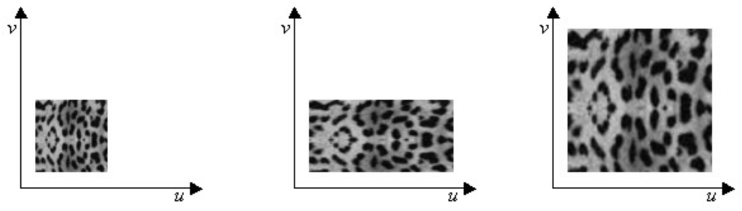

Fig. 8. Pre-calculations of texture

with nice presentation for the end user [18]. The above results were, for sure, best possible, thanks to use of assembler language, pre-calculations at the run-time and compression of the executable file with overlay feature. The development of this program including the design phase took 320 working hours.

4 CONCLUSIONS

The aspects of optimization presented in this article and the results achieved by the sample optimal implementation bring us close to a conclusion that it is rare to produce an optimal program that ‘fits all‘. It is rare to produce one program that will be optimal with regard to execution and fulfillment of all requirements.

Many conditions have to be considered at the stage of program optimization. On the one hand, the program must meet all requirements given at startup. On the other hand, it often transpires at a different stage of program development or even at the maintenance phase that it has to be optimized. The inclusion of all aspects of optimization, such as platform dependence or independence, potential bottlenecks or ‘in-return‘ solutions does not mean that we obtain a truly optimal program that works on any platform under any conditions. Sometimes, it may not even be possible. In some cases, it is better to optimize only selected aspects of program implementation to fulfill given constraints.

It is always worth considering optimization aspects in terms of the accepted effort which encompasses the time that is consumed by program optimization and/or later in the maintenance phase. The effort may be increased by an unreadable but highly optimized code. The above aspect is important if we look at the process bearing in mind cost-effectiveness, which may increase if the effort spent on optimization or maintenance is also significant.

References

1. Agner Fog, (2014) *Optimizing subroutines in assembly language*, Technical University of Denmark
2. Rune Gram-Madsen, AsmOne macro assembler for the Amiga computer and Motorola 680x0 processor, *http://www.amigacoding.com/index.php/680x0:AsmOne*, 1991
3. Bresenham, J. E., (1965) *Algorithm for computer control of a digital plotter*, IBM Systems Journal, vol. 4, no. 1, p. 1-29
4. Commodore Business Machines, (1991) *The AmigaDOS Manual Third Edition*, Bantam Books, United States
5. Daniel Sánchez-Crespo Dalmau, (2004) *Core Techniques and Algorithms in Game Programming*, New Riders, vol. 1, no. 1, p. 64-98
6. Rainer Leupers, (2013) *Code Optimization Techniques for Embedded Processors: Methods, Algorithms, and Tools*, Springer Science & Business Media, Dordrecht, Netherlands
7. David H. Eberly, (2006) *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics, p. 69.*, Morgan Kaufmann Publishers, Boca Raton, United States
8. Gouraud, Henri, (1971) *Continuous shading of curved surfaces*, IEEE Transactions on Computers, vol. C-20, issue 6, pages 623-629
9. Robert Oshana, Mark Kraeling, (2013) *Software Engineering for Embedded Systems: Methods, Practical Techniques, and Applications*, Elsevier, Waltham, United States
10. JungHyun Han, (2011) *3D Graphics for Game Programming*, CRC Press, Boca Raton, United States
11. Marek Domaradzki, Robert Gembara, (1993) *Tworzenie realistycznej grafiki 3D*, LYNX-SOFT, Warsaw, Poland
12. Florin Balasa, Dhiraj K. Pradhan, (2011) *Energy-Aware Memory Management for Embedded Multimedia Systems: A Computer-Aided Design Approach*, CRC Press, Boca Raton, United States
13. Motorola Inc., (1992) *MOTOROLA M68000 FAMILY Programmer's Reference Manual*, Denver, United States
14. M. Verma, P. Marwedel, (2007) *Advanced Memory Optimization Techniques for Low-Power Embedded Processors*, Springer, Dordrecht, Netherlands
15. Robert Fasthuber, Francky Catthoor, Praveen Raghavan, Frederik Naessens, (2013) *Energy-Efficient Communication Processors: Design and Implementation for Emerging Wireless Systems*, Springer Science & Business Media, New York, United States
16. Cezary Nalborski, Source code of the 4kb presentation, *http://www.czarek.nalborski.com/pliki/4kb_intro.lzx*, 2010
17. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, (2009) *Introduction to Algorithms, Second Edition.*, MIT Press and McGraw-Hill, United States
18. Recorded video of the 4kb presentation, *https://youtu.be/zYElGwE6ewY*, 2010
19. WinUAE Amiga computer emulator, *http://www.winuae.net/*, 1995
20. Cezary Nalborski, 4kb executable file, *http://ftp.amigascne.org/pub/amiga/Groups/F/Freezers/Freezers-4kIntro*, 2010

4KB PREZENTACJA WYBRANYCH ALGORYTMÓW RENDEROWANIA GRAFIKI 3D JAKO PRZYKŁAD OPTYMALNEJ IMPLEMENTACJI

Streszczenie. Postępująca miniaturyzacja urządzeń wykonujących programy, narzuca takie wymagania jak szybsze wykonywanie lub mniejsze wykorzystania zasobów na przykład pamięci lub baterii. Rozmiar pliku wykonywalnego programu, także powinien zostać wzięty pod rozwagę, w szczególności w przypadku systemów, w których programy są wbudowane w pamięć. W pracy, przedstawiono krótki opis optymalizacji programu komputerowego. Główne definicje procesu optymalizacji i pewne podstawowe aspekty zostały zaprezentowane na przykładzie programu napisanego w asemblerze Motorola 68k, którego rozmiar pliku wykonywalnego nie przekracza 4 kilobajtów. Program implementuje dwa popularne algorytmy renderowania grafiki 3D.