Optimizing ARINC 661 Rendering for OpenGL with Hardware Support in the JetOS Aviation Operating System

Boris Barladian [1], Lev Shapiro [1], Nikolay Deryabin [1], Yury Solodelov [2], Alexey Voloboy [1] and Vladimir Galaktionov [1]

[1] *The Keldysh Institute of Applied Mathematics RAS, Miusskaya sq. 4, Moscow, 125047, Russia*
[2] *State Research Institute of Aviation Systems, Victorenko 7, Moscow, 125167, Russia*

Abstract

This paper denotes to the problem of the pilot display visualization speed. The software used in avionics has to follow strict rules prescribed by many standards. The studies used OpenGL Safety Critical (SC) with hardware support for Vivante GPU running in the aircraft real time operating system JetOS. One of the avionics standards – ARINC 661 – defines the application rendered in a cockpit display system. It raises the issue of efficient OpenGL SC using to ensure the acceptable visualization speed. Due to the specific of application prepared by the ARINC 661 server the visualization speed for the prospective aircraft platform (i.MX6 processor with Vivante GPU) is too slow to meet aviation requirements. An efficient visualization speed acceleration algorithm has been proposed and implemented. Firstly the OpenGL calls were optimized. But this optimization cannot be directly integrated into the ARINC 661 server. So a special intermediate module was designed and elaborated. The proposed approach makes it possible to achieve a visualization speed acceptable for an aircraft pilot display.

Keywords

Pilot display, visualization speed, real-time operating system, OpenGL Safety Critical, GPU acceleration, ARINC 661 server

1. Introduction

The Cockpit display system (CDS) provides the visible and audible information on aircraft and environment and gets control commands from aircrew. In such a way aircrew manages the modern Glass cockpit and thus interacts with the aircraft avionics. The primary goal of CDS interface is to minimize the cost of acquiring new aircraft systems, to add new cockpit display functionality over the life of an aircraft, and to manage equipment obsolescence in rapidly evolving technologies. CDS provides graphical and interactive services to user-defined applications (UAs) in the cockpit. The UA sends graphical information to the CDS in accordance with the ARINC 661 standard [1]. The crew controls the UA's behavior using commands sent from the CDS to the UA. The interaction between the UA and the CDS is shown in Figure 1. Data exchanging between UA and CDS is implemented via network.

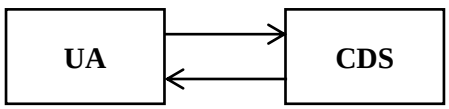

Figure 1: Interaction scheme between UA and CDS

GraphiCon 2021: 31st International Conference on Computer Graphics and Vision, September 27-30, 2021, Nizhny Novgorod, Russia
EMAIL: bbarladian@gmail.com (B. Barladian); pls@gin.keldysh.ru (L. Shapiro); dek@keldysh.ru (N. Deryabin); yasolodelov@2100.gosniias.ru (Yu. Solodelov); voloboy@gin.keldysh.ru (A. Voloboy); vlgal@gin.keldysh.ru (V. Galaktionov)
ORCID: 0000-0002-2391-2067 (B. Barladian); 0000-0002-6350-851X (L. Shapiro); 0000-0003-1248-6047 (N. Deryabin); 0000-0001-5891-7645 (Yu. Solodelov); 0000-0003-1252-8294 (A. Voloboy); 0000-0001-6460-7539 (V. Galaktionov)

The CDS architecture should be robust with sufficient integrity, availability, reliability, and capability to support different display types. Typical examples are:

- Primary Flight Display (PFD);
- Navigation Display (ND);
- Head-Up Display (HUD).

It is obvious that the rendering speed of pilot display must be acceptable to ensure reliable and timely control over the aircraft.

An essential application (UA) is the Airport Movement Map (AMM) which provides a moving map with the aircraft's current location shown on airport taxiways. A typical example is shown in Figure 2 [2]. We tested the rendering speed of the AMM application developed by ARINC 661 server [3]. We rendered it in the pilot display visualization system [4] elaborated for the perspective low power consuming processor i.MX6 with hardware support for Vivante GPU, running under the JetOS aircraft real time operating system [5]. The speed demonstrated by tests is 2-3 frames per second. This visualization speed is unacceptable for avionic applications. The minimum acceptable speed in most cases should be 10-20 frames per second.

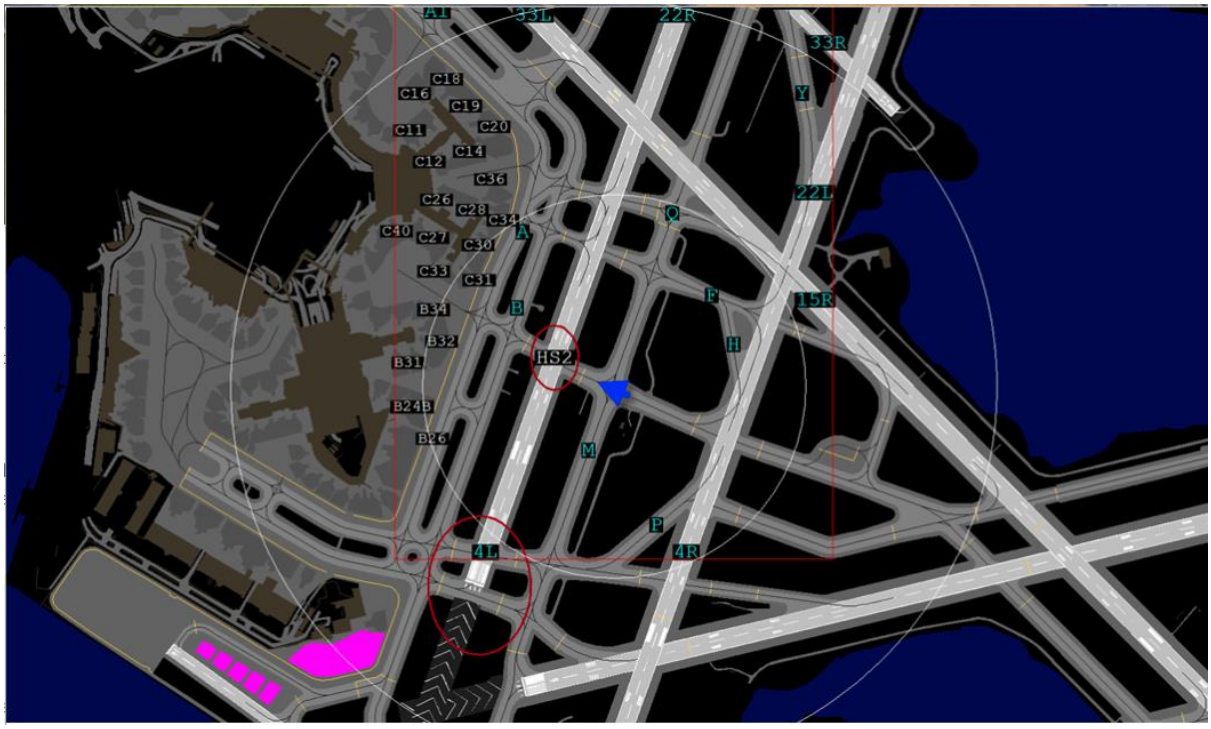

Figure 2: Example of Airport Moving Map

The main reason for this low rendering speed is the specific rendering used in ARINC 661 server. The ARINC 661 demands to use a halo to improve the readability of graphical primitives. Halo is a full outline of a graphic primitive in a contrasting color (typically black). So the visualization of each graphic primitive uses several calls of appropriate OpenGL functions. For example, triangle uses three calls and segment five calls. Therefore to accelerate visualization in ARINC 661 server we try to replace several calls of OpenGL function by a single one whenever it is possible.

Therefore the goal of our study was to investigate the particularities of ARINC 661 rendering and optimize it whenever possible. Ultimate goal is to improve visualization speed till the acceptable level.

2. Related works

The visualization speed of ARINC 661 server is critical for avionic applications. A large number of works are devoted to designing, testing and accelerating the visualization of the ARINC 661 server. Several papers are devoted to the testing of CDS-UA interface [6-8]. The interface has primary influence on speed of display visualization and speed of pilot response on reported situation. Another

article represents the virtual platform for efficient and reliable elaboration of Cockpit Display System for one specific type of aircraft and for domestic aircraft CDS [9].

The works [10, 11] deal with the issues of rendering speed. In these works the OpenVG library is proposed to use for visualization, instead of using the 3D OpenGL library. OpenVG is a standard of 2D vector graphics API defined by the Khronos Group. OpenVG is designed for embedded system GUI rendering, and its features are appropriate to implement most of ARINC661 widgets. However, in our case this approach is inapplicable, since the ARINC 661 server developed by Ansys [3] uses the OpenGL library including its three-dimensional capabilities.

3. Particularity of the ARINC 661 application rendering

We investigated the real AMM applications created according to ARINC 661 standard. The taxiways are represented by line segments there. A concrete pavement of taxiways is represented by set of grey triangles. The three specific test applications were studied. They are shown in Figures 3, 4, and 5. The first application (Figure 3) contains only concrete pavement represented by triangle mesh. The second application (Figure 4) contains taxiways only. And the third application (Figure 5) represents the map of taxiways and concrete pavement. The OpenGL SC 1.0.1 was used by applications for visualization.

These three test applications allow to explore the main rendering problems encountered in an AMM application – rendering a triangular mesh, an array of segments, and a combination of them. The buttons at the bottom of the screen in these figures allow to control the level of details of the rendered objects and their rotation. Direct use of OpenGL in these tests results in too slow visualization speed for these examples: 3 frames per second for the triangular mesh test, 2.3 frames per second for the array of segments, and 1.7 frames per second for its combination.

Figure 3: Triangle mesh visualization

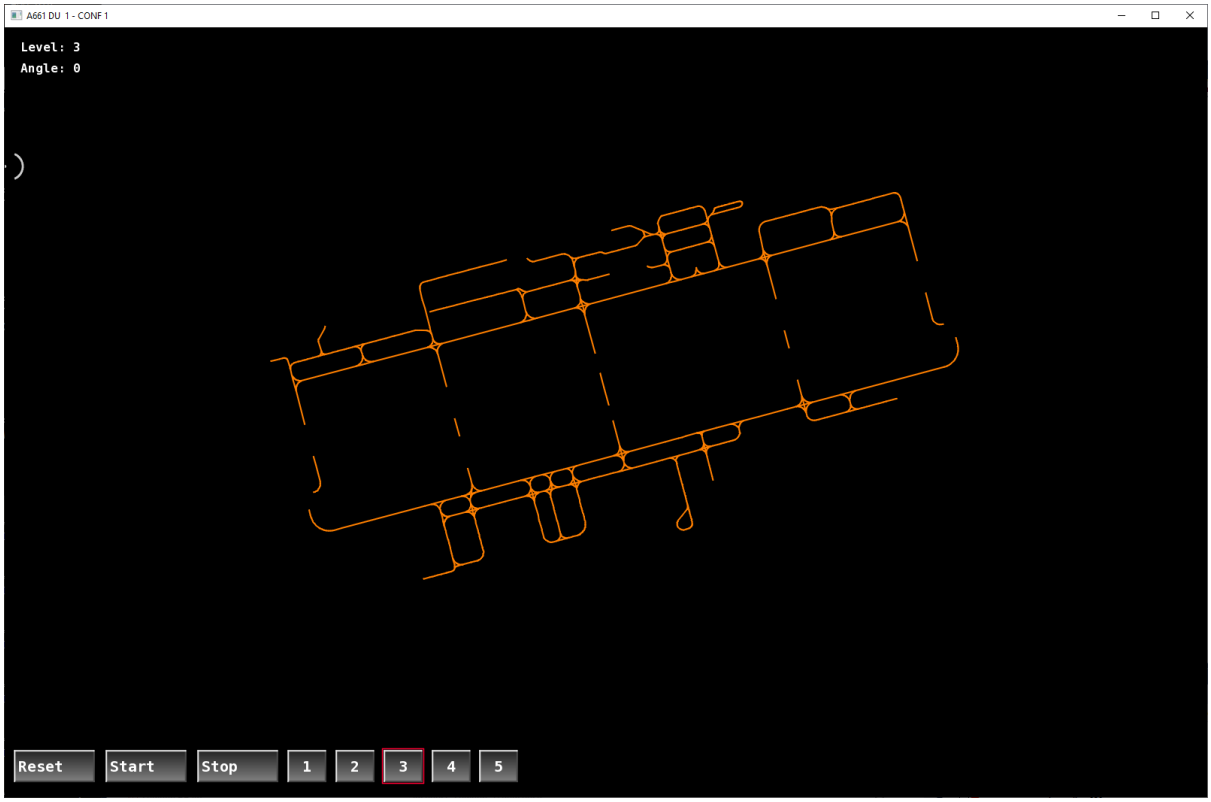

Figure 4: Array of segments visualization

Figure 5: Triangle mesh and array of segments visualization

In these examples, rendering a triangular mesh uses 7381 *glDrawArrays()* calls per frame and an array of segments 5031 such calls. Each *glDrawArrays()* call breaks the OpenGL pipeline, data is written from CPU memory to GPU memory. Thus, using hardware OpenGL becomes ineffective.

4. Optimization of OpenGL calls

The ARINC 661 server developed by Ansys [3] uses special intermediate OGLX library to call appropriate OpenGL functions. So the first our implementation was made directly in OGLX library by some code optimization. The OGLX library uses for geometry visualization the *glDrawArrays()* function only. The following types of OpenGL primitives in *glDrawArrays()* function are used: GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_TRIANGLES, GL_LINE_STRIP, GL_LINE_LOOP and GL_LINES. To decrease the number of *glDrawArrays()* calls we use in the optimized version the GL_TRIANGLES and GL_LINES primitives only. Other types of primitives are simply converted to them. Additionally to reduce the number of *glDrawArrays()* calls instead of setting current color by *glColor4f()* we set the color in vertices by *glColorPointer()* function. To replace several calls of *glDrawArrays()* by single one in our extension we need to unite arrays of vertex coordinates and their attributes – colors and texture coordinates of different graphic primitives for sequentially called drawing functions *glDrawArrays()*.

4.1. OpenGL state

Combining vertex attributes is only valid if successive calls to the drawing function match the same OpenGL state. The state of OpenGL is determined by the OpenGL drawing parameters that have been set when the *glDrawArrays()* function was called. The following parameters define the OpenGL state in OGLX library:

1. stencil mask set by *glStencilMask()* function;
2. clear value for the stencil buffer, set by *glClearStencil()* function;
3. clear buffer mask, set by *glClear()* function;
4. width of rasterized lines, set by *glLineWidth()* function;
5. frame buffer color components for writing, set by *glColorMask()* function;
6. stencil functions and parameters set by *glStencilFunc()* and *glStencilOp()* functions;
7. scissor box parameters, set by the *glScissor()* function;
8. various parameters allowing and disallowing the use of different data, scissor box, stencil buffer, texture, depth test and so on set by *glEnable()* and *glDisable()* functions;
9. Name (identifier) of used texture set by *glBindTexture()* function;
10. Type of current primitive – triangle or segment.

4.2. Work algorithm

The algorithm of the optimized OGLX library can now be described as follows:

1. Calls of such OpenGL functions as *glClearColor(), gltencilMask(), glClearStencil(), glClear(), glColor4f(), glEnable(), glDisable(), glEnableClientState(), glDisableClientState(), glLineWidth(), glColorMask(), glStencilOp(), glStencilFunc(), glViewport(), glScissor(), glBindTexture()* are replaced by appropriate functions with *Set* prefix, i.e. *SetClearColor(), SetStencilMask(), SetClearStencil()*, and so on. These functions instead of passing parameters to OpenGL store them in internal private structure. This structure, in particular, defines current OpenGL state.
2. Original call of *glDrawArrays()* in OGLX library is replaced by special *ProcessPrimitives()* function developed by us. When it is called the parameters of the previous and current states of OpenGL are compared. If these states are the same (or it is the first call of the *ProcessPrimitives()* function), then the drawing data (vertex and texture coordinates, vertex colors) is added to the previous one. Using the *glLoadMatrix()* function requires separate calls of *glDrawArrays()* function for different matrices. To avoid these multiple calls the vertex coordinates are multiplied by current matrix before adding. The triangles primitives GL_TRIANGLE_STRIP and GL_TRIANGLE_FAN are replaced here by GL_TRIANGLES one. Segment primitives GL_LINE_STRIP and GL_LINE_LOOP are replaced by GL_LINES one. This primitive transformation is necessary to provide using one *glDrawArrays()* function for triangles and one for line segments to draw different types of primitives.

3. If the previous and current OpenGL states do not match then the data united for drawing with the necessary settings in OpenGL from the previous state is drawn by one call of *glDrawArrays()*.
4. Only the changed OpenGL parameters of the previous state from 3.1 are set in OpenGL.
5. The array of vertex coordinates and colors are set in OpenGL by using *glEnableClientState(), glVertexPointer()* and *glColorPointer()* functions.
6. If the textured triangles are drawn, then the array of vertex texture coordinates is set by using *glEnable(GL_TEXTURE_2D), glEnableClientState(GL_TEXTURE_COORD_ARRAY)* and *glTexCoordPointer()* functions. If triangles are not textured or segments are drawn then the *glDisableClientState(GL_TEXTURE_COORD_ARRAY)* and *glDisable(GL_TEXTURE_2D)* functions are called before *glDrawArrays()* calling.
7. The *glDrawArrays()* function is called respectively with the GL_TRIANGLES or GL_LINES parameter.

Note. Several calls of OpenGL functions – *glTexImage2D(), glViewport(), glGenTextures()* and *glTexImage2D()* – are called from OGLX once during its initialization. We have omitted these details here for the sake of simplicity

In result of these optimization the number of calls of the glDrawArrays() function for the application in figure 3 (triangles) has decreased from 7398 to 46, for taxiways application on figure 4 (line segments) from 5049 to 46 and for the application on figure 5 (triangles and line segments) from 12436 to 47. Appropriately speed was increased to 24.1 frames per second for the first application, till 21.2 for the second one and till 15.2 frames per second for the third application.

5. The OpenGL server

The solution described in previous section provides acceptable visualization speed for ARINC 661 server rendering. But it has essential drawback – it requires notable changes in OGLX library which is a part of ARINC 661 server. So this requires valuable additional efforts during avionic software certification. More suitable approach is isolating developed approach in separate module. The most reasonable is to place this code directly into OpenGL library or in some intermediate layer between ARINC 661 server and OpenGL. For the sake of effectiveness we implemented OpenGL library as Asymmetric Multi-Processing (AMP) module in JetOS terms.

5.1. OpenGL server

For a multi-core system JetOS supports the ability to run multiple modules (or JetOS instances) on a single device. These modules work independently on different processor cores. This functionality in JetOS is called Asymmetric Multi-Processing (AMP). It is an extension of the ARINC 653 standard and allows more efficient use of processor resources. We use this technology to run the OpenGL library on separate processor core as a OpenGL server. The interaction between ARINC 661 server and OpenGL server is implemented through shared between modules memory blocks. Interaction scheme between UA, CDS and OpenGL server is shown in Figure 6.

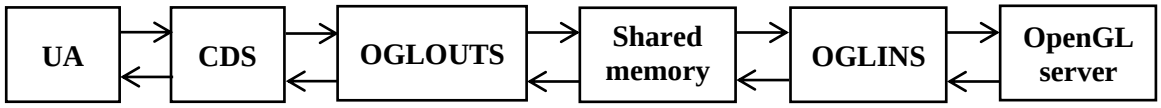

Figure 6: Interaction scheme between UA and CDS using OpenGL server

We have implemented a solution close to the approach used in [4] for multi-window display visualization. The interaction between CDS and the OpenGL server is as follows:

1. In CDS all used OpenGL functions are replaced by special ones implemented in OGLOUTS library. No changes done directly in CDS.
2. These functions mainly record all parameters set passed by these functions and their identifiers in arrays in memory shared by CDS and the OpenGL server. Their actual execution by the OpenGL server will start after the call the *SwapBuffers()* function which starts render of the whole frame.

3. The OpenGL server processes the data stored in shared memory by using special OGLINS library. Functions in this library read data from shared memory, retrieve the corresponding OpenGL function identifier and its parameters stored by OGLOUTS library and call the native OpenGL function.
4. CDS uses several functions that are used during the initialization and should be executed immediately. These are the *glViewport(), glGenTextures(), glBindTexture()* and *glTexImage2D()* functions. In case *glGenTextures()* result should be returned to the CDS for future use by the *glBindTexture()* function. This is provided by a synchronization mechanism.
5. When all the data required for OpenGL execution – geometry parameters, various attributes, OpenGL state parameters, and appropriate sequence of OpenGL function calls – are prepared and the *SwapBuffers()* function is called then synchronization mechanism starts the OpenGL server. It processes all data stored in shared memory.

5.2. Synchronization between CDS and OpenGL server

Synchronization of the CDS and the OpenGL server is done using special objects called events which are implemented via small shared blocks of memory between modules [12]. Two events are used in our case:

StartOgl – is set to the signaled state by CDS in OGLOUTS library when the data stored in the shared memory is ready for processing by the OpenGL server;

EndOgl – is set to signaled state by the OpenGL server when a specified piece of data stored in shared memory is being processed and the OpenGL server is ready to process the next piece of data.

Initially StartOgl event is set to non-signaled state and EndOgl is set to the signaled state.

The synchronization on CDS level is implemented in OGLOUTS library. It should be noted that synchronization is only necessary for functions which require immediate execution on OpenGL server. These are the functions pointed in section 4.1 and SwapBuffers() function which is called for frame rendering completion. The rest of the functions in OGLOUTS library implement the algorithms described in section 4.2, but results are now stored in appropriate structures in shared memory instead of using internal arrays in the OGLX library.

The pseudocode of the algorithm for each function from the OGLOUTS library which executes OpenGL functions through an OpenGL server can be represented as follows:

1. The name and interface of each function is the same as the corresponding function from the OpenGL standard. Only functions used in CDS have been implemented.
2. Wait while the **EndOgl** event will be set to the signaled state. That means that previous portion of data already was processed by OpenGL server.
3. Store the specified function identifier and all passed parameters in the corresponding arrays in the shared memory.
4. Set **EndOgl** event to the non-signaled state.
5. Set **StartOgl** event to the signaled state.
6. Wait while **EndOgl** event will be set to the signaled state.
7. In case when some data needs to be returned, retrieve it from shared memory. This is only the glGenTextures() function in the current implementation,.
8. In case of *SwapBuffers()* function the entire set of the OpenGL function calls with corresponding data prepared using the algorithms described in section 4.2 will be passed to the OpenGL server for execution. The data arrays passed by the *glVertexPointer(), glColorPointer()* and *glTexCoordPointer()* functions which are stored in special separate arrays also pass through shared memory.

The synchronization on the OpenGL server level is relatively simple. The corresponding pseudo-code of the algorithm can be represented as follows:

```
while(TRUE)
{
   Wait while the StartOgl event will be set to the signaled state.;
   Set StartOgl event to the non-signaled state.
```

Call the *process_all_ogl_commands()* function.
Set the **EndOgl** event to the signaled state.
}

The *process_all_ogl_commands()* function sequentially processes all OpenGL functions stored in shared memory one after the other. The corresponding array contains function identifiers and basic parameters. The end of processing in that array is indicated by special function identifier. It should be noted that in case of the several functions called during initializing (specified in section 4.1) and required immediate execution, the OpenGL server will process them in one call. While during *SwapBuffers() function* processing the entire set of OpenGL functions used for given frame will be done.

6. Results and conclusion

The visualization speed using the original OGLX library and two proposed approaches is shown in the Table 1.

Table 1
Visualization speed using the original OGLX library and two suggested approaches. Speed is measured in frames per second.

Test/approach	Original	Optimized OGLX	OpenGL server
Concrete pavement only (Fig. 3)	3	24.1	20.2
Taxiways only (Fig. 4)	2.3	21.2	21.8
Taxiways with concrete pavement (Fig. 5)	1.7	15.2	15.2

It should be noted that research has shown that direct use of shared memory for computations is slower than regular RAM. So to obtain the results shown in the table in OpenGL server we use for them regular RAM and copy the results to the shared memory before OpenGL server invoking.

Both proposed approaches produce results that are acceptable for use in aeronautical applications. But the OpenGL server approach is more preferable from a certification and maintenance point of view since the additional code is isolated from both the OGLX and the OpenGL libraries

The proposed approach made it possible to preserve the specifics of the ARINC 661 server operation, in particular, the use of halo effects to ensure better readability of widgets on the pilot's display and at the same time ensure an acceptable rendering speed.

7. References

[1] E. Barboni, S. Conversy, D. Navarre, P. Palanque, Model-Based Engineering of Widgets, User Applications and Servers Compliant with ARINC 661 Specification. In: Doherty G., Blandford A. (eds) Interactive Systems. Design, Specification, and Verification. DSV-IS 2006. volume 4323 of Lecture Notes in Computer Science,. Springer, Berlin, Heidelberg, 2007. doi:10.1007/978-3-540-69554-7_3

[2] GE Aviation. Airport Surface Moving Map, 2021. URL: https://www.geaviation.com/systems/avionics/navigation-guidance/airport-surface-moving-map.

[3] Ansys SCADE Solutions for ARINC 661 Compliant Systems, 2021. URL: https://www.ansys.com/products/embedded-software/solutions-for-arinc-661.

[4] B.Kh. Barladian, N.B. Deryabin, A.G. Voloboy, V.A. Galaktionov, L.Z. Shapiro, High speed visualization in the JetOS aviation operating system using hardware acceleration, in: Proceedings of the Graphicon-2020 conference, CEUR Workshop Proceedings, 2020, Vol. 2744, pp. short3:1-short3:9. doi:10.51130/graphicon-2020-2-4-3.

[5] K.M. Mallachiev, N.V. Pakulin, A.V. Khoroshilov, Design and architecture of real-time operating system, Proceedings of the Institute for System Programming 28(2) (2016) 181-192. doi:10.15514/ISPRAS-2016-28(2)-12.

[6] H. Sartaj, M.Z. Iqbal, M.U. Khan, Testing cockpit display systems of aircraft using a model-based approach. Softw Syst Model (2021). doi:10.1007/s10270-020-00844-z.
[7] W. Yang et al., An Efficient Approach for Monitoring and Analyzing Real-Time ARINC 661 Events, in: Proceedings of the IEEE/AIAA 37th Digital Avionics Systems Conference (DASC), 2018, pp. 1-5. doi:10.1109/DASC.2018.8569329.
[8] M.Z. Iqbal, H. Sartaj, M.U. Khan, F. Ul Haq, I. Qaisar, A Model-Based Testing Approach for Cockpit Display Systems of Avionics, in: Proceedings of the ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS), 2019, pp. 67-77. doi:10.1109/MODELS.2019.00-14.
[9] Y. Zheng, X.Y. Lei, Research and Implementation of Virtual Cockpit Panel Development Platform Based on ARINC 661. in: Proceedings of the IEEE Chinese Guidance, Navigation and Control Conference (CGNCC), 2014, pp. 1357-1361.
[10] J. Yoon, N. Baek, H. Lee, Graphics Rendering Based on OpenVG and Its Use Cases with Wireless Communications. Wireless Personal Communications 94(2) (2017) 175–185. doi:10.1007/s11277-015-3163-y.
[11] J. Yoon, N. Baek, H. Lee, ARINC661 graphics rendering based on OpenVG, in: Proceedings of the 5th International Conference on IT Convergence and Security (ICITCS), Aug 2015.
[12] B.K. Barladian, L.Z. Shapiro, K.A. Mallachiev, A.V. Khoroshilov, Yu.A. Solodelov, A.G. Voloboy, V.A. Galaktionov, I.V. Koverninskii, Visualization Component for the Aircraft Real-Time Operating System JetOS, Program. Comput. Software 46(3) (2020) 167–175. doi:10.1134/S0361768820030020.