Multi-windows Rendering Using Software OpenGL in Avionics Embedded Systems

B.Kh. Barladian[1], L.Z. Shapiro[1], K.A. Mallachiev[3], A.V. Khoroshilov[3], Y.A. Solodelov[2], A.G. Voloboy[1], V.A. Galaktionov[1], I.V. Koverninskiy[2]
[1]The Keldysh Institute of Applied Mathematics Russian Academy of Sciences, Moscow, Russia
[2]FGUP «GosNIIAS» State Research Institute of Aviation Systems, Moscow, Russia
[3]Ivannikov Institute for System Programming Russian Academy of Sciences, Moscow, Russia

Elaboration of modern airplane cockpit has tendency to use large displays instead of a lot of separate indicators. The large display should combine information about flight navigation and state of plane equipment. Information coming from a wide variety of devices should be displayed simultaneously. Therefore multi-windows rendering is vitally important here. Its implementation must be embedded in real-time operating system which controls the aircraft. Development of a Safety Critical Compositor for multi-windows rendering for OpenGL SC 1.0.1 software is considered in the paper. It works under the real-time operating system JetOS newly designed for aircraft. Development is based on the use of extensions designed to work in multi-core systems in addition to standard JetOS partitioning services.

Keywords: *aircraft cockpit display, multi-windowing, OpenGL SC 1.0, real-time operating system, embedded systems.*

1. Introduction

Modern airborne systems are designed based on the architecture called Integrated Modular Avionics. A key feature of this architecture is the capability to execute several functional applications implementing the software part of avionics systems on the same computer. The necessary condition for this capability is time and resource sharing between applications. This mode of operation is offered by a real time operating system which is an integral part and most important component of the airborne system.

In the development of modern airplane cockpits there is a tendency to use large displays to combine in it information about the flight navigation and state of plane equipment. Fig. 1 and 2 show the cockpit aircraft evolution from the Sukhoi Superjet to the MS-21.

Fig.1. Cockpit of the Sukhoi Superjet aircraft.

Fig.2. Cockpit of the MS-21 aircraft.

The number of displays in the cockpit of the MS-21 aircraft has been reduced in comparison to the cabin of the Sukhoi Superjet aircraft, but the displays have become much wider and allow displaying more information.

The flight and equipment operation information is generated by numerous flight management systems. This information should be displayed for pilots in the easy-to-read form. The generated information should be displayed graphically on the widescreen displays (so called multi-function displays). The information coming from a wide variety of devices should be displayed simultaneously. In particularly it may be airspeed, attitude indicator, altimeter, turn and slip indicator, vertical speed indicator and so on. At the same time such technical characteristics as engine speed, oil pressure and fuel quantity should be displayed too. In addition, it is useful to visualize a map of the area, various pneumatic, hydraulic and electrical circuits, data from weather radars, various kinds of warnings, etc. This information is usually generated by independent subsystems and should not interfere with each other in accordance with the requirements of ARINC653 [5].

Nevertheless it is often necessary to display images from several subsystems on one screen. Modern operating systems solve this problem by supporting of a multi-window interface when each application's content is rendered into its own window. A simplified approach is to allow each application to open a non-overlapping window onto the display. While the last method allows for faster drawing its implementation for safety critical systems requires significant efforts. A compositor elaboration is needed to support efficient multi-windowing.

Various approaches to implementation of the compositor for safety critical systems are considered in [1]. One of compositor implementations is the CoreAVI's EGL_EXT_compositor extension for EGL for OpenGL SC 1.0.1 and OpenGL SC 2.0 [2]. However its source codes are closed ones and it does not allow to use them for our goal. The OpenGL SC library [3] we are developing is designed to work under the JetOS operating system [4]. This defines development specific and imposes essential requirements to the developed code and algorithms. In particular, the need of OS certification requires full access to the source codes of both the OpenGL SC library and the compositor. On the other hand, when developing the composer we can take advantage of the specific opportunities of the JetOS to improve performance. These features, in particular, include the ability to use several processor cores.

2. Types of Composition

In the paper [1] two main types of graphical composition are considered – composition into the hardware level and composition into a framebuffer.

While the benefits of the hardware level composition include good performance, power conservation and efficiency in dealing with a large number of updates, this approach requires additional bandwidth to display all windows. It also requires specific support for framebuffer driver which is not accessible for us in currently used hardware.

The composition into the framebuffer approach combines elements from multiple applications and off-screen buffers into a single framebuffer. The framebuffer then renders the data to the display. The composition into the framebuffer requires only one layer to display all buffers. In fact it is the only available approach in our case. Data visualization scheme used is shown in Fig. 3. Each application renders the data using OpenGL SC in own off-screen buffer. These buffers then are passed to the compositor. It forms from them the single framebuffer layer and visualizes it on display by using frame buffer library. The main problem here is effective synchronization of independently running applications and the compositor. The implementation of synchronization depends on means provided by the operating system.

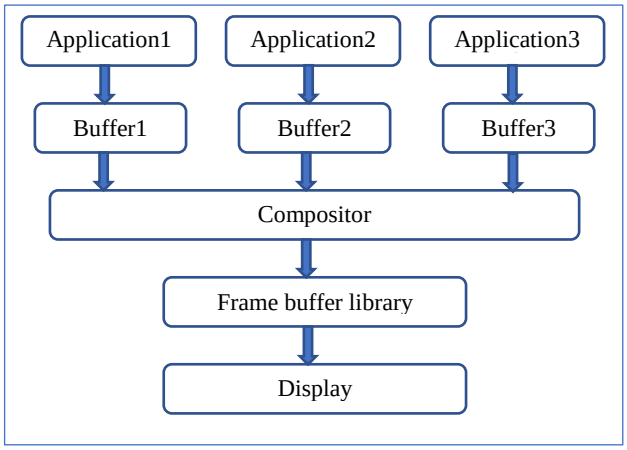

Fig. 3. Compositor - Information flow to Display

3. Solution via JetOS partitions

First implementation of compositor was based on using standard JetOS tools developed according ARINC 653 standard [5]. In this case several applications and compositor work in a single processor. Each application and compositor work on their own JetOS partitions. The JetOS provides memory and time partitioning in accordance to the ARINC 653 requirements.

Partitions are scheduled on a fixed cyclic basis. To assist this cyclic activation, the OS maintains a major time frame of fixed duration, which is periodically repeated throughout the module's runtime operation. Partitions are activated by allocating one or more "partition windows" within this major time frame. Each "partition window" is defined by its offset from the start of the major time frame and expected duration. The order of partition activation is defined by the system integrator using configuration tables. This provides a deterministic scheduling methodology whereby the partitions are furnished with a predetermined amount of time to access processor resources. A module may contain several partitions running with different periods.

The rendered images are passed from the application to the compositor by using special shared memory blocks. Each application uses the own memory block for image rendering. This memory block has read-only access for the compositor.

Synchronization between applications and compositor is provided by queuing messages transmitted between partitions via special communication channels. Two channels are used between each application and the compositor. First channel is used by application to inform the compositor that image is ready to display. The second channel is used by the compositor to inform the application that the image was displayed and the application can use the image buffer again. The application synchronization scheme can be represented by the algorithm shown in Fig. 4.

```
not_first = false;
While (true)
{
  If (not_first)
  {
    Wait the message that image was displayed
}
not_first = true;
  Render image
  Sent message that image is ready for display
}
```

Fig. 4. Application synchronization algorithm.

Appropriately the processing by the compositor images rendered by applications can be represented by the following algorithm (Fig. 5):

```
for (int ic = 0; ic < application number; ic++)
{
  Wait not more than 1 millisecond the message that the
rendered image from ic-th application is ready to display
  If the message came
  {
    Display image from the appropriate shared memory
block by using frame buffer library
    Send the message that image from the given application
was displayed
  }
}
```

Fig. 5. Compositor synchronization algorithm.

Let's consider an example of the compositor work. Resultant image generated from buffers produced by two applications is presented in Fig. 6.

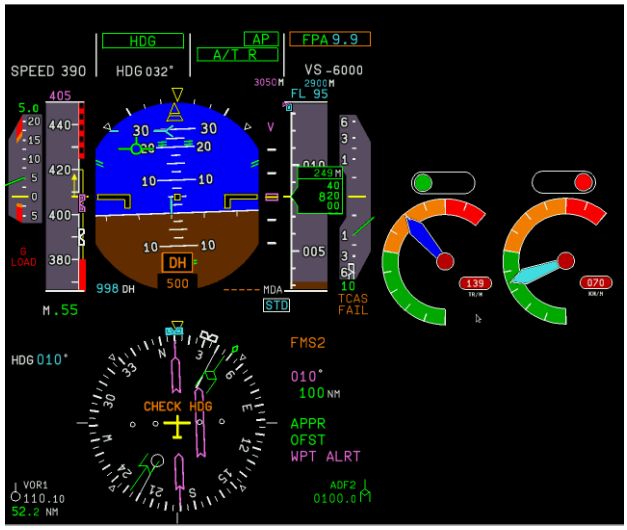

Fig. 6. Composition of two application buffers.

Two applications – the Primary Flight Display (PFD) on the left side of image and the Counter on its right side – work simultaneously (in fact in line due to the requirements of ARINC 653) and images produced by applications are visualized by the compositor. Suggested approach works correctly but visualization speed in given example is insufficient for avionic applications. Both applications work with speed ~5 frames per second. There are several reasons for such behavior. First at all the typical avionic processor PowerPC [6] has low performance. The second reason is that all partitions work on a single processor core and have predetermined amount of time to access its resources. We can slightly optimize this time subdivision only

taking into account real applications needs. In the given example the frame time was subdivided in the following manner:

1. PFD – 45ms
2. Counter – 15ms
3. Compositor – 16ms

This schedule provides more or less balanced access to the processor resources for applications with essentially different resource requirements.

But future acceleration is possible by using of all processor cores only. Processor PowerPC (P3041) [6] used by us has four cores while perspective PowerPC (P4080) [7] can have eight cores.

4. Multicore solution

In case of multi core system JetOS supports ability to run multiple modules (instances of JetOS) on one device. These modules operate independently on different processor cores. This JetOS feature is called AMP – Asymmetric Multi-Processing. This AMP feature makes possible to use processor power in significantly more efficient way. Appropriate project configuration using AMP feature is called AMP project. AMP project supports shared memory blocks which we use for images passing between modules. But it does not provide currently queuing messages which we used for synchronization in solution via JetOS partitions. Due to this reason we decided to use small shared memory blocks for synchronization in multicore solution. As in the case of previous solution we need to support two events for synchronization of interaction of each application with the compositor:

1. **Start_copy** – when the image is prepared by application and is ready for display by the compositor;
2. **End_copy** – when the image was already displayed by the compositor and the appropriate memory block can be used again by application for rendering the next frame.

```
#define AMP_UP        1
#define AMP_DOWN   0
typedef int*  AMP_EVENT;
/// Get event state.
int AMP_GetEventState(AMP_EVENT ev)
{
  return ev[0];
}
/// Set the event in the  state "up".
void AMP_SetEvent(AMP_EVENT ev)
{
  ev[0] = AMP_UP;
}
/// Set the event in the  state "down".
void AMP_ResetEvent(AMP_EVENT ev)
{
  ev[0] = AMP_DOWN;
}
/// Infinitely wait while event is in state "down".
void AMP_WaitEvent(AMP_EVENT ev)
{
  RETURN_CODE_TYPE ret;
  while (ev[0] == AMP_DOWN)
  {
    TIMED_WAIT(MILLISECOND, &ret);
  }
```

Fig. 7. Event emulation

This pair of events is implemented by using a 16-byte memory block shared between modules. The first half of this block is used for **Start_copy** event and the second half – for **End_copy** event. For convenience and a more intuitive interface we implemented the set of functions which emulates work with these memory blocks as with events (Fig. 7). As an argument all these functions use a pointer either to the first half or to the second half of the appropriate 16-byte shared memory block.

To pass rendered image from application to the compositor each application uses memory block shared with the compositor.

Now the work of the application can be represented by the following algorithm (Fig. 8):

```
While (true)
{
AMP_WaitEvent(End_copy);
AMP_ResetEvent(End_copy);
Render image
AMP_SetEvent(Start_copy);
}
```

Fig. 8. AMP application algorithm.

Appropriately the processing of images rendered by applications by the compositor can be described by the following algorithm (Fig. 9):

```
for (int ic = 0; ic < application number; ic++)
{
  If (AMP_GetEventState(Start_copy[ic]) = AMP_UP)
  {
    AMP_ResetEvent (Start_copy[ic]);
    Display image from the appropriate shared memory
block by using frame buffer library
    AMP_SetEvent(End_copy[ic]);
  }
}
```

Fig. 9. AMP compositor algorithm.

At the initialization state all **End_copy** events are set to *AMP_UP* state while **Start_copy** are set to *AMP_DOWN* state.

The proposed technology of using multi-core processor under JetOS allows to increase the speed of visualization for the example on Fig. 6 till 8.8 frames per second for PFD application and till 44 frames per second for Counter application. It should be noted that the rendering speed for application PFD is still insufficient. Even with one application running in the JetOS partition solution the rendering speed does not exceed 7.4 frames per second. When using multi-core technology the speed slightly increases due to the work of the frame buffer library in a separate core. Partially it is due to the fact that the PFD application is overcomplicated itself. Later this application is modified to increase its efficiency.

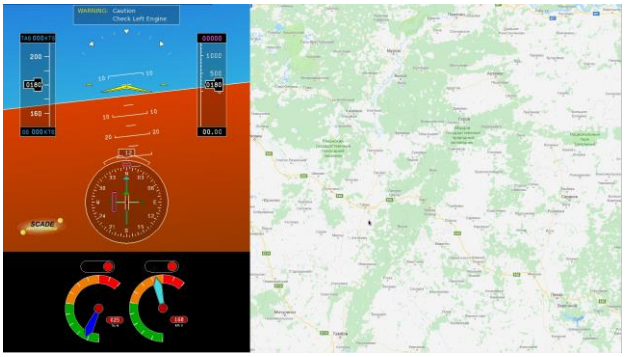

Fig. 10a. Example of AMP solution for the compositor. Visualization of PFD + map + Counters.

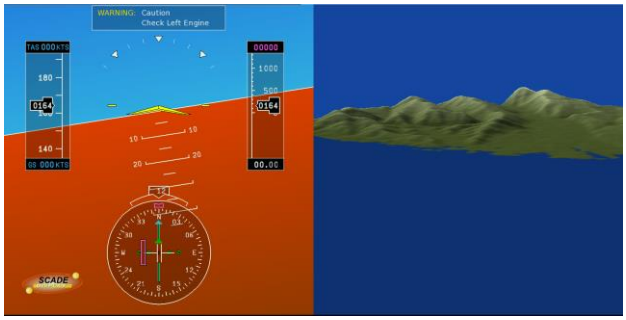

Fig. 10b. Example of AMP solution for the compositor. Visualization of PFD + relief.

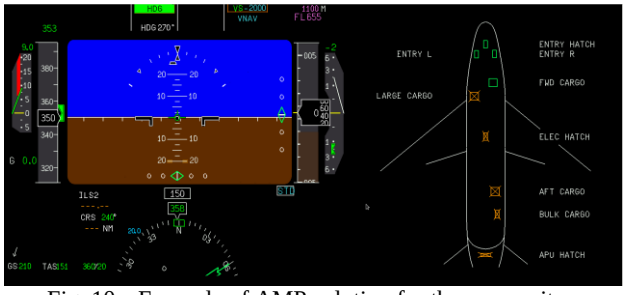

Fig. 10c. Example of AMP solution for the compositor. Visualization of PFD + state of doors.

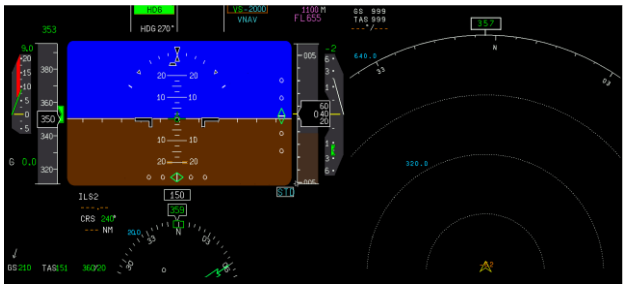

Fig. 10d. Example of AMP solution for the compositor. Visualization of PFD + navigation display.

Additional examples of images produced by suggested multi window approach are shown in Fig. 10. The following rendering speed was reached for these examples:

Fig. 10a:
PFD – 16 frames per second
Map – 16 frames per second
Counter – 16 frames per second

Fig. 10b:
PFD – 16 frames per second
Relief – 9.2 frames per second

Fig. 10c:
PFD – 10.3 frames per second
State of doors – 21.7 frames per second

Fig. 10d:
PFD – 10.3 frames per second
Navigation display - 21 frames per second

5. Conclusion

Analysis of visualization algorithms for various data used in embedded avionics systems shows that JetOS partitioning services alone do not secure the required performance. The use of extensions for work in multi-core systems provided by JetOS improves the performance. However, the rendering speed is still not sufficient in some cases. In order to further increase performance the possibility of using multi-core processor options directly in the OpenGL SC library should be considered.

6. References

[1] A Safety Critical Compositor for OpenGL SC 1.0.1 and OpenGL SC 2.0. *http://www.coreavi.com/sites/default/files/compositor_whitepaper_final.pdf.*

[2] EGL_EXT_compositor. *http://www.coreavi.com/sites/default/files/coreavi_product_brief_-_egl_ext_compositor.pdf.*

[3] B.Kh. Barladian, A.G. Voloboy, V.A. Galaktionov, V.V. Knyaz', I.V. Koverninskii, Yu.A. Solodelov, V.A. Frolov, and L.Z. Shapiro, Efficient Implementation of OpenGL SC for Avionics Embedded Systems. *Programming and Computer Software, 2018, Vol. 44, No. 4, pp. 207–212. DOI: 10.1134/S0361768818040059.*

[4] K.A. Mallachiev, N.V. Pakulin, A.V. Khoroshilov, Design and architecture of real-time operating system. *Proceedings of the Institute for System Programming, vol. 28, issue 2, 2016, pp. 181-192.ISSN 2220-6426 (Online), DOI: 10.15514/ISPRAS-2016-28(2)-12.*

[5] ARINC Standards Store: https://www.aviation-ia.com/product-categories.

[6] Universal Data Processor Module MUPD/P3041-VPX 3U, http://www.nkbvs.ru/products/elektronnie-modyli/vpx-3u/moduli-universalnogo-protsessora-dannix-mypd-p3041/.

[7] QorIQ® Processors Based on Field Proven Power Architecture Technology P-Series. *https://www.nxp.com/products/processors-and-microcontrollers/power-architecture-processors/qoriq-platforms/p-series:QORIQ-POWER-ARCHITECTURE-P-SERIES.*