## 

## 

## **6. Introduction to OpenGL Programming**

We aim to acquire very basic programming skills on CG using OpenGL.

[OpenGL is](http://www.opengl.org/) a programming library for displaying CG in real-time.

This is probably your first opportunity to use a large library other than the math library.

The OpenGL library is a well-made library in itself, but it still has the problem that it is difficult to program by itself.

Therefore, GL [Utility (GLU)](https://www.opengl.org/resources/libraries/) library is prepared so that the OpenGL library can be used more easily.

However, even with the help of the GLU library, there is still the problem that it is difficult to display the window and render, so the [OpenGL Utility Toolkit (GLUT)](https://www.opengl.org/resources/libraries/) library is also available.

You will be using three libraries.

1. OpenGL: A library for simply drawing CG and making effective use of hardware
2. GLU: A library with more abstract (easy-to-use) functions by combining OpenGL functions
3. GLUT: From OpenGL to window creation and control Library to make the accompanying parts easy to use

(GL is required to use GLU. Both GL and GLU are required to use GLUT.)

**Reference link**

-Description of [OpenGL functions](https://www.khronos.org/registry/OpenGL-Refpages/gl4/)

-[OpenGL Programming Guide Book (ver1.1, “red book”)](http://glprogramming.com/red/index.html)

・[GLUT --The OpenGL Utility Toolkit](http://www.opengl.org/resources/libraries/glut/)

・[The OpenGL Utility Toolkit (GLUT) Programming Interface API Version 3](http://www.opengl.org/resources/libraries/glut/spec3/spec3.html)

### **06.01. The first OpenGL**

When you use OpenGL / glut library (hereinafter simply abbreviated as OpenGL) to create a CG drawing program, you need to follow [the programming style recommended by glut library](http://www.opengl.org/resources/libraries/glut/spec3/spec3.html).

For programs that use the glut library, event-callback programming centered on the glutMainLoop() function is recommended.

**Event processing (event loop)**

Programs that interact with the user through the window often use event-driven algorithms.

The normal program that we have learned so far has been processed in order from the beginning of the main() function, went to the end, and ended (called a flow-driven type).

On the other hand, in the execution by event-driven programming, before entering the event processing loop, first of all,

"what kind of event should occur and what kind of processing should be performed"

for various events that can be assumed, the promise of. ，in**advance**Determine everything.

After meeting the conventions, the program goes into an infinite loop waiting for the event.

After starting the infinite loop, it does nothing but performs the specified operation according to the event.

Programming with the glut library is premised on programming according to this method.

**Program specifications**

This program is simple.

1. Preparation before starting the infinite loop.

2. The infinite loop start

The program is shown below.

code \ 06-01 \ 06-01-Preparation.c

ic2\_BootWindow() function is used for initialization.

After that, the function name defined in this lesson should start with ic2\_.

The most important role of ic2\_BootWindow() is to launch a window using glutCreateWindow().

Since the window can be named, the 0th argument (= argv [0] = command name) on the command line is specified.

In addition, glutInitDisplayMode() specifies the CG drawing style.

GLUT\_DOUBLE declares that two buffers are prepared for drawing.

(This is called double buffering.)

Nothing is specified in this program before entering the event waiting infinite loop called glutMainLoop(). If you want to do something, you need to specify an event/action pair.

Therefore, when the infinite loop starts, the program does nothing.

It doesn't even clear the buffer reserved for the window because it doesn't do anything. Just the window frame appears and you're in an infinite loop.

**The concept of compiling and linking**

OpenGL libraries are actually large-scale, and you should be careful about compiling and linking.

However, since it is also a commonly used library, it is being organized.

Compile time:

All you need to do is include GL / glut.h as #include <GL/glut.h>.

When linking:

You need to specify OpenGL, GLU, GLUT, and all the libraries they use. It is necessary to explicitly indicate three external libraries, libGL, libGLU, and libglut, at the time of linking.

In other words, you need -lGL -lGLU -lglut as arguments to gcc.

**Compiling and linking work**

[1] Command line

You can generate an executable file on the command line as follows.

04.03. Please review the section.

Prepare the directory ~ / workspace-ip3 / 06-01 for the work.

$ gcc -Wall -o 06-01 06-01-Preparation.c -lGL -lGLU -lglut

$. / 06-01

[2] eclipse

The difference from the previous section is that you need to tell eclipse the argument of the link. ..

Let's say the project name is 06-01.

|  |  |
| --- | --- |
| 06-01-A | New project registration (see 05-04-02-A) |
| 1 | Menu: File> New> C / C ++ Project |
| 2 | [New C / C ++ Project]  Left: All  Right: C Managed Build  > Next |
| 3 | [C Project]  Project name: 06-01  Project type: Executable> Empty Project  Toolchains: Linux GCC  > Next> Finish |
| 4 | Put 06-01-Preparation.c in the "06-01" project folder (at outside of eclipse) |
| 5 | Refresh the "06-01" project |
| 6 | Change the building method of "06-01" project  On selecting "06-01" project, Menu: File> Properties  [Properties for 06-01]  Left: C / C ++ Build [Unfold]> Settings  Right: Tool Settings> GCC Linker [Unfold]> Libraries  Right: At Libraries (-l), add GL, GLU, and glut respectively (Case sensitive)  > Apply and Close |

After registering the project, create an executable file by compiling.

If you fail on compiling, debug it. This is static debugging.

The procedure is the same as 05-04-04-A.

Once you have a good executable without static bugs, you can go for runtime debug then.

First, set the configuration for runtime debugging. Then start runtime debugging.

At this time, you need to know the name of the desktop environment.

The name of the desktop environment is the contents of the environment variable DISPLAY.

The env command is used to check environment variables. Execute it on the terminal.

$ env

...

DISPLAY=:0.0

...

There were a lot of environmental variables. Only DISPLAY is needed to be checked here. Use the grep command.

$ env | grep DISPLAY  
DISPLAY =:0.0

A string after = is the name of the desktop environment.  
If you are in the computer room, DISPLAY is ":0.0". Don't forget the leading colon. Please note that the display on the remote desktop may be slightly different, such as ":10.0" or ":11.0".

This value will be used in the next Step 4.

|  |  |
| --- | --- |
| 06-01-B | Debug configuration and debug start (Ref: 05-04-05-A) |
| 1 | Select the project to debug at the project windowpane |
| 2 | Menu: Run > Debug Configurations |
| 3 | [Debug Configurations]  Left: Doubleclick “C / C ++ Application ”  Left: Select“ 06-01 Debug ”which will appear under“ C / C ++ Application ”  Right: Environment > Environment variables to set: > Add:: |
| 4 | [New Environment Variable]  Name: DISPLAY  Value: :0.0 (Four letters! colon, zero, comma, and zero)  > OK |
| 5 | [Debug Configurations]  > Apply > Debug |

After that, if you step-execute in the same way as 05-04-07-A, it will proceed smoothly, but it will not proceed when glutMainLoop() is reached.

This is because glutMainLoop() started an infinite loop.

|  |  |
| --- | --- |
| 06-03-C | Terminate the runtime debug |
| 1 | Menu: Run > Terminate |

**Relationship between executable file and dynamic library**

There are many external libraries that are not specified. Most of them are combined with executable files as dynamic libraries.

$ cd ~/workspace-ipd/06-01

$ ls -l 06-01 Debug/06-01

$ ldd 06-01

$ ldd Debug/06-01

**Exercise**

**06-01-ex1**:

Give your window a name. Notate in the alphabet.

**06-01-ex2**:

Create two executable files for 01-02-HelloESYS-Better.c and 06-01-Preparation.c. List the dynamic libraries of the executable files. Clarify the difference. Explain the reason for the difference.

**06-01-ex3**:

Suppose you run step execution after 06-03-B. Step execution will not be possible after glutMainLoop(). Explain the reason.

**06-01-ex4**:

Explain the programming style of event-driven programming.

**06-01-ex5**:

Explain in principle that double buffering is an effective mechanism for CG drawing.

**06-01-ex**:

Suppose there is triple buffering. Consider whether it is effective for CG drawing.

**06-01-ex7**:

Show the include file that defines GLUT\_DOUBLE in the program. Extract the definition and show it.

**06-01-ex8**:

The argument of the glutInitDisplayMode() function is specified by connecting macros with |. Explain the reason.

**06-01-ex9**:

Compare the executable file 06-01 created on the command line with the executable file 06-01 created on Eclipse. Check the size of each file. Consider the reason for the difference in file size. A Unix command called strings may be useful.

### **06.02. First OpenGL --Let's switch the buffer-**

Next, specify the event/action pair.

From this section onward, the project name on eclipse will be the section number.

In this section, "06-02" is the project name.

Extract the source file in the directory with the same name.

**Program description**

code \ 06-02 \ 06-02-SwapBuffers.c

code \ 06-02 \ 06-01\_06-02.html

In this program, a window is created using ic2\_BootWindow(). After that, specify the event/action pair for the opened window.

Here we use a function called glutIdleFunc(). Specify "If you have free time, call the function ic2\_DrawFrame() to work".

Specify the operation using the function name.

There are no plans to accept other events in this program. Nothing else is registered.

You will always be free at runtime.

This means:

・ Start the infinite loop

・ Nothing to do (idle)> Execute ic2\_DrawFrame )

・ Coming back to the infinite loop

・ Nothing to do (idle)> Execute ic2\_DrawFrame()

Then go around again and again. With

The final thing to do at ic2\_DrawFrame() is to draw one frame of CG.

Here, only the undercoat (initialization) of the buffer is done.

Use glutSwapBuffers() to instantly switch between the buffer that was being drawn and the buffer that was being displayed.

As you can see by running it, the inside of the window just turns black.

However, the burden on the system will increase.

The CPU usage does not increase with the program in the previous section.

In the program in this section, the CPU usage rate will increase.

Launch System monitor and check the CPU resource. Or use "top" command in the terminal.

**Exercise**

**06-02-ex1**:

In the program in section 06.01. Explain why the CPU utilization does not change before and during execution.

**06-02-ex2**:

In the program in section 06.02., the CPU usage rate increases during execution compared to before execution. Explain the reason. What is the CPU doing while this program is running?

**06-02-ex3**:

What is the process that puts the most load on the system while monitoring the CPU load? Examine and report what it is the process of doing.

**06-02-ex4**:

glutIdleFunc() passes the "function name" as an argument. Explain from the mechanism of C language that this is the specification of the operation. Explain how function names and variable names are similar.

### **06.03. OpenGL for the first time --Confirmation of buffer switching --**

It is too simple that there is no visual change for the user even though the CPU usage is high.

Let's try to visualize how it is moving.

Change the fill color from time to time.

Paint in green once every three times. Paint the remaining two times in black.

**Program description**

code \ 06-03 \ 06-03-SwapBuffersCheck.c

code \ 06-03 \ 06-02\_06-03.html

[Caution] To open html files, download the code folder and open the html file on your own desktop.

Prepare a static integer variable in ic2\_DrawFrame(). Count and memorize how many times you have drawn.

By the way, as expected, you should see the color change of green → black → black → green → black → black → green → black → black → green → black → black →.

Hmm? Can you see it? Not really?

If you are not confident in your dynamic vision, try running it at command line on theand repeating Ctrl-z and fg.

Ctrl-z pauses the program.

Type fg to resume execution of the paused program. Stop is Ctrl-c.

**Exercise**

**06-03-ex1**:

Change the above program so that it turns green once every five times.

**06-03-ex2**:

Change the above program so that it turns red once every 10 times.

**06-03-ex3**:

Change the above program to green → black → black → black → white → black → black → black.

**06-03-ex4**:

Check the refresh rate (frame per second, fps) of the monitor in the desktop environment you are currently using.

**06-03-ex5**:

Change the above program and improve it so that it ends when the ic2\_DrawFrame() function is executed at the rate of 06-03-ex4 x 100 times (6000 times at 60Hz).

**06-03-ex6**:

Execute the program of 06-03-ex5 and measure the execution time in seconds. A very simple measurement of execution time is possible by doing “date; ./06-03-ex5; date” on bash. It is expected to take 100 seconds to run. Explain why you expect 100 seconds.

**06-03-ex7**:

If the measurement result of 06-03-ex6 is not 100 seconds, try to explain the cause.

### **06.04. First OpenGLbuffer by time**

In the previous section, "ic2\_DrawFrame()" was specified to run only when it is idle by glut, but this "if free time" is actually a bit hard to understand.

The X server used in Linux redraws the entire desktop (depending on the environment) at a frequency of about 60Hz. It is actually the refresh rate of your display.

(The X server is a process on Linux that provides a GUI to you.)

(The OpenGL sample program you are running actually asks the X server to display it when drawing.)

As of this time, when the amount of work in ic2\_DrawFrame() is small, it does not take a long time in its execution.

That means, "I'm free, so I try running ic2\_DrawFrame()" and it is probably less than 1/60 second. There is a possibility that "I try to execute ic2\_DrawFrame() because I am free (one more time)" may occur by 1/60 seconds (maybe one more time or two times?).

Also, with this method, the drawing time interval is not constant.

(Even if the load of the ic2\_DrawFrame() function is the same, if another heavy job starts running on the OS, the number of times the function can be executed in 1/60 seconds will change.)

(Speaking in detail, On the contrary to the above, there is a possibility that 1/60 second or more has passed when it comes to "next free time". It is a state of so-called processing failure. Unfortunately, the speed of display refresh is constant, so If you do not make it within 1/60 seconds, the next chance will be 2/60 seconds later to switch the next drawing.)

(The computer on the 5th floor will instantly draw the amount you want to draw in one frame here. Since it will be completed [in a much shorter time than 1/60 seconds], there is no need to think about the processing failure.)

Therefore, let’s change the specified event type to a timer.

Follow the writing style recommended by the glut library. Do as follows.

(1) Specify the function name with glutDisplayFunc(). In this function, describe the contents to be drawn. Here is the ic2\_DrawFrame() function.

(2) In glutTimerFunc(), specify the function you want to start after t [ms]. Here, specify the ic2\_timerhandler() function).

(3) Describe to call the glutPostRedisplay() function in the ic2\_timerhandler() function. This function tells you to redraw the next time you have a chance.

The "next chance" comes when the OS redraws the entire desktop. This is determined by the refresh rate of the display you are using, not by the OS or program.

**Program description**

code \ 06-04 \ 06-04-SwapByTimer.c

code \ 06-04 \ 06-03\_06-04.html

Declare to use ic2\_DrawFrame() when drawing in the buffer by glutDisplayFunc(ic2\_DrawFrame()). Corresponds to (1).

Reserve glutTimerFunc(250, ic2\_timerhandler, 0) to call the ic2\_timerhandler() function 250 [ms] after that point. Corresponds to (2).

After entering the event-waiting infinite loop, ic2\_timerhandler() starts executing 250 [ms] later.

In the ic2\_timerhandler() function, it is instructed to redraw the next time there is a chance. For this, we call the glutPostRedisplay() function. Corresponds to (3).

After that, I reserve to call myself (ic2\_timerhandler()) again 250 [ms] from now on. Corresponds to (2).

After executing ic2\_timerhandler() and returning to the event-waiting infinite loop, ic2\_DrawFrame() is executed when it is time to draw. This will switch the two buffers of drawing and showing.

This time, as a result, the drawing is updated at 250 [ms] intervals. It will be easier to observe the color change of the buffer.

Also, the CPU usage is much lower than the program in the previous section.

**Exercise**

**06-04-ex1**: Reduction of CPU cost

The CPU usage rate of the above program tends to be lower than that of the program in the previous section. How much will the processing amount be reduced? It would be even better to consider it together with the execution results of the programs 06-02 and 06-03.

**06-04-ex2**: Change the drawing interval according to the monitor

Change the drawing time interval of the above program according to the refresh rate of the monitor you are currently using.

**06-04-ex3**: Relationship between drawing time interval and CPU load

Try changing the drawing time interval of the above program and show the relationship between the drawing time interval and the CPU load. To what extent is it possible (is it meaningful) especially for those who narrow the interval?

**06-04-ex4**: Significance of timer interrupts

From this, try to explain what happened when a program that does not use a timer handler. Explain the cause of the high CPU load factor at that time. If the execution result is not as expected from (06-03-ex5) to 06-03-ex7).

**06-04-ex5**: Accuracy limit of a timer interrupt

Strictly speaking, there is no guarantee that the drawing interval will be 4Hz (every 250ms) in the above program. Describe the reason.

**06-04-ex6**: Asymptotic change of drawing time interval

Improve the above program and create a program that gradually widens the drawing time interval.

**06-04-ex7**: Specifying drawing time interval

Improve the above program so that the period can be specified from the command line in the form of bpm. bpm is the number of frame switching per minute (beat per minute).

**06-04-ex8**: Background gradation

Improve the program in this section so that the logo expands smoothly. Divide 20 or more from the minimum to the maximum. At the same time, change the background from black to white over 3 cycles. Improve the logo so that it is not difficult to see.

### **06.05. First OpenGL --Parallel projection camera --**

To draw CG, it is necessary to align the camera, light source, and object.

In this section, we will prepare only the camera and the object.

First, set the camera.

Projection conversion from 3D to 2D is performed by a parallel projection camera (also called orthogonal projection or orthographic projection).

**Concept of projection**

First, we will learn mathematical expressions.

The coordinate system for expressing the three-dimensional world in which projection is performed on a plane is called a camera (three-dimensional) coordinate system, and the coordinates are expressed by *Pcamera* (Large P).

The plane of the projection destination is called the imaging plane, the two-dimensional coordinate system is used to express the imaging plane (two-dimensional) coordinate system, and the coordinates are expressed by *pcamera* (Small p).

The projection from 3D to 2D is represented by the projection matrix **P.**

*pcamera* = **P** *Pcamera*

Here, originally *Pcamera* has three elements and *pcamera* has two elements. But for the simplification of the calculation structure, the homogeneous coordinate representation is applied.

(If the original is 3 elements, increase 1 element, if the original is 2 elements, increase 2 elements so that it becomes a 4-dimensional vector at any time.)

(What value should be put in the increasing element? There is a rule about this. You can see it later by looking at the actual program, but you do not need to know it in the content of this lesson [because it can be left to the OpenGL library].)

By this, **P** is 4x4. It becomes a matrix.

Mathematically, the above is sufficient, but when actually projecting a 3D world onto a 2D plane, it is necessary to define some detailed but absolutely necessary items.

Here, I will explain according to the promise in OpenGL.

- OpenGL uses a right-handed system as the three-dimensional coordinate system.

- In the camera 3D coordinate system, the focus of the camera is fixed at the origin.

- In the camera 3D coordinate system, the optical axis of the camera is directed to the direction vector (0,0, -1).

- In the camera 3D coordinate system, the vertical direction of the camera is positive on the Y axis and the upper side is positive, and the horizontal direction of the camera is the X axis and the right side is positive.

Reference: Red Book [Chapter 3](http://glprogramming.com/red/chapter03.html), Figure 3-9: Object and Viewpoint at the Origin

**Parallel projection orthographic**

Parallel projection is an orthographic projection of the coordinates in space onto a plane with a constant Z value (a plane parallel to the XY plane).

Naturally, the size of the imaging surface needs to be the size of the object.

(If you want to obtain an image of a 170 cm tall person by parallel projection, you need to prepare imaging planes with a size of 170 cm above and below(!) .)

In principle, you can set the coordinate points of any Z value. It is possible to project, but in reality, it limits the range of Z values ​​of the coordinates of the points that can be projected.

(This is because if the range is too wide, the accuracy of perspective judgment will decrease when using the perspective judgment method called Z-buffer method, which is usually used by OpenGL hardware.)

Reference: Red Book [Chapter 3](http://glprogramming.com/red/chapter03.html), Orthographic Projection

**Program explanation**

code \ 06-05 \ 06-05-OrthogonalCamera.c

code \ 06-05 \ 06-04\_06-05.html

Set the camera at the beginning of ic2\_DrawFrame() which draws one frame.

Execute ic2\_SetUpCamera\_Ortho() to find the 4x4 matrix for orthogonal projection.

*glOrtho (-1, 1, -1, 1, -1, 1)*

In this program, the horizontal (X coordinate) is from -1.0 to 1.0. The vertical (Y coordinate) is -1.0 to 1.0. This range is the target range for orthogonal projection. This value becomes the 1st to 4th arguments.

In this program, the depth (Z coordinate) -1.0 to 1.0 is the range of orthogonal projection. (5th argument, 6th argument)

Reference: Red Book [Appendix F. Orthographic Projection](http://glprogramming.com/red/appendixf.html)

**Attention**

The aspect ratio of the imaging surface given by glOrtho() must match the aspect ratio of the window actually displayed.

Otherwise, the drawing will be distorted (geometrically incorrect).

Since glutCreateWindow() prepares a square window by default, this program specifies that the aspect ratio of glOrtho() is 1.0 square.

To tell the truth, unfortunately, even if it is executed correctly, it looks the same as before ... because it does not draw the object. Then to the next section.

**Exercise**

**06-05-ex1**: Parallel (orthographic) projection

For a parallel projection camera, describe the elements of the projection matrix with the valuables given to the glOrtho(). Here, the projection destination is a plane with Z = constant.

Reference: Red Book [Chapter 3, Orthographic projection](http://glprogramming.com/red/chapter03.html)

Reference: Red Book [Appendix F. Orthographic projection](http://glprogramming.com/red/appendixf.html)

**06-05-ex2**: Meaning of Z-value element

06-05-ex1 above, please explain what the Z-value element in the converted camera coordinate system vector means.

**06-05-ex3**: Z-axis

According to the OpenGL convention, explain which direction the Z-axis is positive concerning the screen you are looking at.

**06-05-ex4**: glOrtho() function

Show the actual values of the projection matrix (MODELVIEW matrix) of *glOrtho(-1, 1, -1, 1, -1, 1)* prepared when the program is executed. Also, show the process of deriving the value. Show all 4x4 components numerically.

Reference: Red Book [Chapter 3, Orthographic projection](http://glprogramming.com/red/chapter03.html)

Reference: Red Book [Appendix F. Orthographic projection](http://glprogramming.com/red/appendixf.html)

**06-05-ex5**: Actual projection plane

Where is the projection plane actually in the glOrtho() setting of the program in this section? Examine the material and explain why.

Reference: Red Book [Chapter 3, Orthographic projection](http://glprogramming.com/red/chapter03.html)

Reference: Red Book [Appendix F. Orthographic projection](http://glprogramming.com/red/appendixf.html)

**06-05-ex6**: Drawing range in the Z-axis direction

You can set the drawing range (positive subspace) in the Z-axis that contradicts the basic explanation of camera projection. Is there any good reason to allow to do so?

(The program in this section is just such an example)we have

### **06.06. First OpenGL --Drawing a line art object --**

Since we prepare a camera of orthographic projection, if you give the shape of the object, a geometrically correct drawing will be possible.

In the program in section 06.05, the space that can be drawn is -from 1.0 to 1.0 for all X, Y, and Z axes.

Therefore, let's prepare a drawing of a line art object that falls within this range.

**Program description**

code \ 06-06 \ 06-06-LogoOpenGL.c

code \ 06-06 \ 06-05\_06-06.html

ic2\_OpenGLLogo() function draws a line art within the range of -1.0 to 1.0 in all X, Y, and Z axes.

The scale factor can be taken as an argument so that the display size can be changed later.

**Program description (drawing lines)**

OpenGL provides a set of functions for drawing some typical CG objects.

Here, we use a format called "GL\_LINES" that draws a line segment by connecting a sequence of points.

|  |
| --- |
| GL\_LINES object |
| glDisable(GL\_LIGHTING); // Coloring assuming that the value range is 0.0 to 1.0 if RGB color is specified thereafter  glColor3f(1.0, 1.0, 1.0); // R, G, B  glLineWidth(1.0); / / Line thickness (unit is [pixel])  glBegin(GL\_LINES); // Start of drawing (draw line segment AB, line segment BC)  glVertex3f(s \* -0.8, s \* 0.8, 0.0); glVertex3f (s \*) -0.8, s \* 0.2, 0.0); //segment AB  LineglVertex3f(s \* -0.8, s \* 0.2, 0.0); glVertex3f (s \* -0.8, s \* 0.2, 0.0); //segment BC  LineglEnd( ); // End of drawing |

(By the way, use GL\_LINE\_STRIP to connect point clouds to draw a line segment)

Reference: Red Book [Chapter 2](http://glprogramming.com/red/chapter02.html), Describing Points, Lines, and Polygons → Lines

**Exercise**

**06-06-ex1**: Thickness of lines

Let's change the the thickness of the lines by a factor of two.

**06-06-ex2**: Coordinate system

How are the 3D coordinates in the above source code arranged concerning the currently displayed window? Show the origin, X-axis, Y-axis, and Z-axis, respectively. Also, explain where the unit (1 unit for each axis) corresponds.

**06-06-ex3**: Original logo

Prepare your own graph paper-like worksheet and create your own logo. Rewrite the program and show your logo. Make the scale adjustable.

### **06.07. First OpenGL --Periodic operation --**

Since we are drawing OpenGL LOGO, we would like to see some movement here.

The scale factor specified when calling the ic2\_OpenGLLogo() function is changed in synchronization with the background color switching.

**Program explanation**

code \ 06-07 \ 06-07-Periodic.c

code \ 06-07 \ 06-06\_06-07.html

**Exercise**

**06-07-ex1**:

Modify the above program and write split variable splitnumber (positive integer). Write the program that makes the logo look like it is expanding smoothly. It is important to operate the loopcounter value and wait time of the timer handler.

(06-07-Periodic.c corresponds to split number = 10 in period 2500 [ms])

**06-07-ex2**:

Describe the relationship between scale factor (argument s of ic2\_OpenGLLogo()) and frame count (loopcounter) value in the program. Read the program and analyze it accurately. The loopcounter should be analyzed by taking a value from 0. Find the value of s for each value of loopcounter.

(It doesn't matter whether it is for the original 06-07-Periodic.c or for the program of 06-07-ex1. Please specify which one is adopted.)

**06-07-ex3**:

As examined in 06-07-ex2, the relationship between loopcounter and s is linear. Express this linear relationship as an equation.

**06-07-ex4**:

When loopcounter and s are in direct proportion, the logo does not seem to be approaching at a constant speed (it seems to move faster when approaching far and slower when approaching close). Explain the reason using the formula of projective geometry.

**06-07-ex5**:

Derive a formula for calculating the scale factor from the frame count so that the logo appears to approach at a constant speed concerning the passage of time (frame count).

**06-07-ex6**:

Modify the program based on the above formula of ex5 and check its operation subjectively.

**06-07-ex7**:

In 06-07-Periodic.c, the logo cannot be drawn when the green background is reached. At that time, change the logo so that it is drawn at 1.0 times (original size).

### **06.08. First OpenGL --Measurement of microsecond order --**

You can check the system time in microsecond order by using the gettimeofday() function which is a system function of Unix.

(The content of this section has nothing to do with OpenGL, so it can be applied to other Linux programs in general.)

Code \ 06-08 \ 06-08-CountTime.c

**Exercise**

**06-08-ex1**:

Modify 06-04-ex7 program so that it can measure the cycles correctly.

**06-08-ex2**:

In OpenGL programming, the method shown here is NOT actually recommended as a timekeeping method. Think about the reason.

(It is recommended to use the glutGet (GLUT\_ELAPSED\_TIME) function instead)

### **06.09. First OpenGL --Code Cleaning --**

The CG object is hard to see when the background color changes, so let's organize it before proceeding to the next section.

code \ 06-09 \ 06-09-LogoOpenGL.c

code \ 06-09 \ 06-06\_06-09.html

**Exercise**

**06-09-ex1**:

Explain how the background color changed in the program up to section 06.06.