A quick start with OpenGL and C

This is a very brief tutorial to get you started with writing 3D graphics programs using OpenGL in C. It is written to be accessible for people with a background in Java, so section 1 is a short introduction to C programming. If you have no previous knowledge of C, you will not be a C programmer after reading this, but you will be able to find your way around in existing programs, understand and edit other people's code and write small snippets of code yourself. Section 2 then shifts the focus to OpenGL, starting with its design and concepts and a small subset of its basic functionality. After the general introduction to OpenGL, we finish off with a practical tutorial, written in a very hands-on fashion. Section 3 covers the installation of all the software you need to compile and run the tutorial yourself on a Windows computer, and in section 4 we create a small OpenGL example program from the ground up.

1. A C primer for Java programmers

C is the ancestor of several modern programming languages, including C++ and Java. It is not a particularly pretty language, in fact it allows and even encourages programmers to write very ugly programs, but it was designed to be efficient and easy to use. C is still highly useful and popular for many applications, particularly for hardware related, system-level programming. Learning the basics of C is very helpful also for understanding C++. People switching from Java to C++ are often misled by the similarities between the two languages and forget the important differences. It is of great help to a C++ programmer to know some of the pitfalls of C and to keep in mind that C++ is somewhat of an object-oriented facade built on top of C, not an object-oriented language designed from the ground up like Java.

Compared to Java, C is a relatively inconvenient language to use, but it has one definite advantage: speed. Programs written in C can execute ten times faster or more than exactly the same program written in Java. The reason for that is the large overhead in Java bytecode interpretation, object manipulation and runtime safety checks. C has no such overhead, but also offers much less support for the programmer.

A familiar syntax

C has a syntax that should be immediately familiar to people who know Java. Identifiers are case sensitive, a semicolon terminates a statement, curly braces are used to group statements together, and most control constructs like `if`-statements, `while` and `for` loops and `switch` clauses look and work the same in C as in Java. Primitive types are also mostly similar: `int`, `double`, `float` and the like. Most operators have an identical syntax, an array index is placed within square brackets, and comments are written in the same manner. At the source code level, C and Java look very much alike, and many C statements are valid Java statements as well. Some small and strange quirks in Java like the increment operator (`i=i+1` may be written `i++`) and the rather bizarre syntax of a `for` loop are directly inherited from C.

Functions, and nothing else

So, C and Java may look similar, but they work differently. First of all, everything that Java has in terms of object orientation is entirely lacking in C. There are no classes, no objects, no constructors, no methods, no method overloading and no packages. In C, you have variables and functions, and that's it. You may think of it as if you only had one single class in a Java program. Furthermore, a C function must have a globally unique name. In Java, methods belong to classes and only need to have a unique name within the class, and methods within a class may be overloaded and share a common descriptive name if only their lists of parameters differ. C makes no such distinctions. One result of this is that functions with similar functionality that operate on different kinds of data must be distinguished by their names. For a large API like OpenGL, this is definitely a drawback. A vertex coordinate may be specified with two, three or four numbers, which may be of type `double`, `float`, `int` or `short`. This results in the following twelve function names being used for that single purpose:

```
void glVertex2d(double x, double y)
void glVertex2f(float x, float y)
void glVertex2i(int x, int y)
void glVertex2s(short x, short y)
void glVertex3d(double x, double y, double z)
void glVertex3f(float x, float y, float z)
void glVertex3i(int x, int y, int z)
void glVertex3s(short x, short y, short z)
void glVertex4d(double x, double y, double z, double w)
void glVertex4f(float x, float y, float z, float w)
void glVertex4i(int x, int y, int z, int w)
void glVertex4s(short x, short y, short z, short w)
```

The prefix ”`gl`” in the name of every OpenGL function is due to the lack of classes, separate namespaces or packages in C. The suffixes are required to distinguish the variants from each other. In Java, these functions could all have been methods of a GL object, and all could have been given the same name: `GL.Vertex()`. (In fact, the Java interface to OpenGL, ”JOGL” works that way.)

Memory management

Java has a very convenient memory management system that takes care of allocating and de-allocating memory without the programmer having to bother about it. C, on the contrary, requires all memory allocation and de-allocation to be performed explicitly. This is a big nuisance to programmers, and a very common source of error in a C program. However, the memory management in Java is not only a blessing. The lack of automatic memory management and other convenient functionality is one of the reasons why C and C++ are much faster than Java.

Pointers and arrays

Java by design shields the programmer from performing low level, hardware-related operations directly. Most notably, direct access to memory is disallowed in Java, and objects are referenced in a manner that makes no assumption regarding where or how they are stored in the computer memory. C, being an older, more hardware oriented and immensely less secure language, exposes direct memory access to the programmer. Any variable may be referenced by its address in memory by means of *pointers*. A pointer is literally a memory address, denoting where the variable is stored in the memory of the computer. Pointers pointing to the wrong place in memory are by far the most common causes of error in C programs, but no useful programs can be written without using pointers in one way or another. Pointers are a necessary evil in C, and unfortunately they are just as essential in C++.

Arrays in Java are abstract collections of objects, and when you index an array in Java, the index is checked against upper and lower bounds to see whether the referenced index exists within the array. This is convenient and safe, but slow. C implements arrays as pointers, with array elements stored in adjacent memory addresses. This makes indexing very easy, because if you have an array `arr` with a number of elements and want to access `arr[4]`, all you have to do is to add 4 times the size of one array element to the address of the first element and look at that new address. Unfortunately, no bounds checking is performed, so an index beyond the last element of an array, or even a negative index, is perfectly allowed in C and will often execute without error. Indexing out of bounds of an array could cause fatal errors later in the program, and the end result will most probably be wrong. Indexing arrays out of bounds is a very common pointer error in C, and forgetting to check for incorrect bounds in critical code is the most common security hole in C programs. Once again, C++ does not make things any better in this respect.

Source files

The source code for a C program may be split into several files, but it is not a simple task to decide what goes where, or how many files to use. In Java, the task is extremely straightforward: use one file for each class. In C, there are no classes, so you could theoretically write even a very large C program in one single file. This is inconvenient, and not recommended for anything beyond the size of the small programs in this tutorial, but different programmers split their code somewhat differently into several files. There are guidelines on how to organise C source code to make it easy to understand, but no there is no single set of guidelines that everyone uses. To compensate for the lack of packages in C, there is a concept called *header files* or *include files*, which is a sort of half-baked solution that works OK most of the time. Include files are inserted by the compiler into the source code at compile time, and included header files with function prototypes can be used to provide access to extra libraries and functions that are defined in other, separately compiled source files. Unfortunately, include files can be used inappropriately, as there is nothing in the C language itself that actually mandates what should go where. Self-taught and undisciplined C programmers sometimes re-invent the wheel badly or learn from bad but ”clever” examples, and develop a programming style of their own that can be very hard to read.

main( )

A stand-alone C program must have a function called `main`. It is defined like this:

```
int main(int argc, char *argv[])
```

When a C program starts, its `main` function is invoked. Where to go from there is entirely up to the programmer. The program exits when the main function exits, or when the statement `exit` is executed.

Example

A very short but complete C program is given below. It calculates a function value for ten numbers from 0.0 to 0.9, stores them in an array and exits. It is of no particular use, but it shows the syntax for some common tasks.

```
/* A program that does nothing much useful */

double myfunc(double x) {
      double f;
      f = 3*x*x - 2*x + 8;
      return f;
} // End of myfunc()

int main(int argc, char *argv[]) {int i;
      double x;
      double values[10];
      for (i=0; i<10; i++) {
            x = 0.1*i;
            values[i] = myfunction(x);
      } // End of for loop
      return 0;
} // End of main()
```

A program that does something useful probably needs some input and output. We can include a standard library header file `<stdio.h>` that provides this for us. There are many standard library functions in C, and not much real programming can be done without them. However, it is far beyond the scope of this brief introduction to present any of them. We just use the `printf` function as an example below.

```
/* A program that prints a countdown and exits */

#include <stdio.h>

int main(int argc, char *argv[]) {
      int i;
      for (i=10; i>0; i++) {
            printf("%d, ", i);
      }
      printf("Launch!\n");

}
```

2. OpenGL: design and concept

OpenGL is a *direct rendering* API, which means that everything is drawn immediately after it is specified. OpenGL has no memory of what has already been drawn, apart from what is painted in the rendering window. The opposite to this behaviour is *indirect rendering* or retained rendering, where the entire scene is first specified as some kind of data structure, often a tree, and only then sent away to be drawn. In OpenGL, there is no memory of the last frame, so animation has to be performed by erasing the screen and redrawing all objects at new positions. Timing of the animation is entirely up to the programmer. This makes an OpenGL program full of small, low level details, and it is necessary to move these details to separate functions for the program to be well structured and readable. Everything within OpenGL is performed by a large collection of functions starting with ”`gl`”. The header that defines most of these functions is named `<GL/gl.h>` on most platforms.

Primitives

There are very few graphics primitives in OpenGL. Strictly speaking, there are only four: pixel images, points, lines and triangles. Triangles are the most general and most often used 3D primitive. Several triangles can be combined to construct any planar polygon, and any surface can be approximated by a mesh of triangles. A triangle is specified by three vertices in a 3D space. A vertex is specified by one of the many functions whose names start with ”`glVertex`”.

Rendering context

Apart from vertex coordinates, OpenGL has functions for specifying normal vectors, colors, texture coordinates, texture images, lights and a few other pieces of information regarding the things to be drawn. Everything about the drawing is saved in a ”graphics state”, often called *rendering context*, that keeps track of the current color, the current normal vector, the currently active texture image and so forth. Without going into any details, we should mention that textures and lights play a very prominent part in OpenGL. Textures in particular have become very important as a means for compensating for the lack of geometric detail and the simplistic local lighting model which is still used in most real-time 3D rendering.

Transformations

Primitives alone do not make a good graphics API. Transformations are also required to make it general enough to be really useful. Transformations in OpenGL, as in most 3D graphics APIs, are described by 4x4 matrices operating on homogeneous coordinates. There are several transformation matrices in OpenGL, the most used ones being the *modelview matrix* and the *projection matrix*. The modelview matrix takes care of all the transformations from local (user) space to camera (view) space, and any transformations concerning camera motion or scene motion should be performed by changing the modelview transformation matrix. The projection matrix is responsible for the transformation from view space to screen space, most importantly the perspective projection and the transformation from general view coordinates to a pixel-oriented window coordinate system suitable for rendering. Hierarchical transformations play an important part in most 3D graphics applications. OpenGL supports hierarchical transformations by a *matrix stack*, where matrices may be pushed and popped to perform and undo transformations as required. Keep in mind that OpenGL is a direct rendering API, so only one transformation is active at a time. All that is needed is the ability to rapidly switch back and forth between different transformations, and the matrix stack makes this simple.

That’s it!

Primitives, the existence of a rendering context and knowledge of transformation matrices are the key components to understanding the underlying concept of OpenGL. It takes a lot more details to explain it in depth, but at an overview level, this is all there is to it. Now we are ready for some real programming.

3. Tools for OpenGL programming

Before you can start coding, there are a few things you need, but don't be alarmed. The hardware you need is available in almost any modern PC, and the software can be downloaded free of charge and is easy to install.

The hardware

You should try to find a computer with a reasonably good and modern 3D graphics card to run this tutorial. Any Windows system supports OpenGL, but some less capable graphics cards will force the system to do everything in software. OpenGL is designed for hardware acceleration, so software emulation will be slow, and sometimes ugly too. Most stationary PCs sold in recent years have fairly good 3D graphics hardware, but beware of some of the cheapest graphics chips that are integrated directly onto the motherboard. Many laptops, even very modern and otherwise good systems, can also be very bad when it comes to 3D graphics. On the other end of the scale, stationary computers designed for gaming often have outstanding 3D performance, often quite good even when compared to professional level 3D workstations.

The software

To write programs in C, you need at least a simple text editor to write the source code, and a compiler to create the executable files. A number of options exist, some of which are better than others. The tools we will use here are very good, widely used and absolutely free. The compiler is ”gcc” from the GNU project, and the editor is a full-fledged development environment for the Windows platform called “Dev-C++” from Bloodshed Software. Both are available together in a single installation package from various places over the Internet. We have placed one version where you got this document from:

`http://www.itn.liu.se/~stegu/OpenGLquickstart/`

The file there was the most recent version from the developers as of on March 3, 2009. If you want to look for a later version, you can have a look at `http://www.bloodshed.net/devcpp.html`.

Download and install the compiler tools by using the all-in-one installation package.

The libraries

The essential libraries you need to write OpenGL programs are already included with the Dev-C++ installation, but we will install an additional free library that makes OpenGL programming under Windows a lot easier: GLFW by Marcus Geelnardt and Camilla Berglund. You can download the source code for GLFW and compile it yourself with the `gcc` compiler, but to make installation easier, you can use a Dev-C++ ”development package” that will install the few files required for GLFW in the correct locations. Download the file `glfw2.6-3.devPak`, start Dev-C++, select `Tools->Package manager`, click the `Install` icon, locate the file `glfw2.6-3.devPak` and select `Open`, click `Install`, `Next` and `Finish` to perform the installation. You are now prepared to start programming with GLFW. Finally, there are a few C example files and two texture images for this tutorial, named `planets1.c` to `planets7.c`, `earth.tga` and `random.tga`. Download and save those files somewhere convenient. Now you're all set!

4. OpenGL tutorial

The project

Start Dev-C++. Before you start coding, you need to create a new project and set it up for our particular purpose of programming in C under Windows using OpenGL and GLFW. Please follow the instructions below to do this. All the steps are required.

`File->New->Project...` click ”Empty project”, select ”C project”, name the project ”planets”, click ”OK” and name the project file ”planets.dev”. Save the project file in a fresh folder somewhere convenient.

`File->New->Source File`, click ”Yes” to add the file to the project.

`File->Save`, name the file ”planets.c” and save it in the same folder as the project file.

`Project->Project Options...`, select tab “Parameters”, in text field “linker”, type `”-mwindows -lglfw -lopengl32 -lglu32”` and click OK.

Our first OpenGL program

Your new and empty C source file ”planets.c” is where the C code should go. Open the file ”planets1.c” from Dev-C++, cut and paste the entire contents (see below) into your file ”planets.c” and save the file. (Alternatively, you may include the file ”planets1.c” in the project instead of your own file, but you will want to make edits later, so it is probably best to do all your editing in ”planets.c”.)

```
#include <GL/glfw.h> // GLFW used for convenience

int main(int argc, char *argv[])
{
      int running = GL_TRUE;
      double t, t0, fps;

      // Initialise GLFW
      glfwInit();

      // Open the OpenGL window
      if( !glfwOpenWindow(640, 480, 8,8,8,8, 32,0, GLFW_WINDOW) )
      {
            glfwTerminate(); // glfwOpenWindow failed, quit the program.
            return 1;
      }

      // Main loop
      while(running)
      {

            // Swap buffers, i.e. display the image and prepare for next frame.
            glfwSwapBuffers();

            // Check if the ESC key was pressed or the window was closed.
            if(glfwGetKey(GLFW_KEY_ESC) || !glfwGetWindowParam(GLFW_OPENED))
            running = GL_FALSE;
      }

      // Close the OpenGL window, terminate GLFW and exit.
      GlfwTerminate();
      return 0;
}
```

(Above: The entire content of the file planets1.c.)

`Execute->Compile` (or ctrl-F9, or use the “compile” button in the toolbar) to compile the program. If any errors occur, you did something wrong, probably in specifying the project options. Find the error and try again until the compilation succeeds.

`Execute->Execute` (or ctrl-F10, or use the “execute” button in the toolbar) to execute your newly compiled file. Because what you create is a standard Windows executable file, you can also execute the program without any special tools at all. Open a file explorer window, locate the program file ”planets.exe” in your project folder and double-click on it.

If an empty black window is showing, everything works OK and you’re ready to continue! Close your newly created application by pressing the ESC key or by closing its window.

Make a habit of always closing your applications when you are done testing. A typical OpenGL program uses a lot of processing and graphics power, and having more than one such application running, either as a visible window or minimized in the task bar, will slow down your other programs a lot.

A less boring OpenGL program

Our first OpenGL program is really boring, because it doesn't draw anything. Let's add some graphics to the window. We need to set up a camera projection, and draw at least one primitive. This can be accomplished by the code in "planets2.c". Cut and paste, compile and have a look at the result. Now take a closer look at the code. The drawing code that was added compared to "planets1.c" is in `main` and consists of two parts: first the window is prepared for drawing by erasing it and setting up a camera projection, then a single triangle is drawn by a sequence of vertices enclosed between a `glBegin` and a `glEnd`. Before each vertex is specified, the graphics context is changed by `glColor` to specify a color. When a color is specified, subsequent vertices will have that color if no new color is specified. (If no color is ever specified, OpenGL starts with white as the current color.) Note that the OpenGL rendering performs smooth color interpolation over the surface to set the color between vertices.

An FPS counter

The image in the example above does not change over time, but it is actually redrawn many times per second. To find out exactly how many times per second it is updated, we can add a frame counter to the program and use the built-in timer function in GLFW to calculate the number of frames per second (FPS). We have no convenient text output in our application, so we print the FPS counter information in the window title bar. This is performed by a couple of extra variables and the function showFPS in "planets3.c" Try it out. The display is exactly the same as before, but now the program keeps track of how fast it is updated. The FPS reading is probably some number reasonably close to 60, 75 or 85 FPS, because OpenGL by default updates the graphics at most once per display refresh. If you want to see how fast your simple scene can be drawn, uncomment the following function call immediately before the main loop:

```
glfwSwapInterval(0);
```

This will make OpenGL redraw the display as fast as possible, without waiting for a new display scan. You will probably see a great increase in FPS reading when you do this change. The scene we are drawing so far is extremely simple and can be updated thousands of times per second on a good computer. If you make the window smaller, the FPS reading will increase, and if you make it larger, the FPS will decrease. This is because OpenGL primitives take longer to render the more pixels they cover, and erasing a large window also takes longer than a small window. One of the many limiting factors to OpenGL hardware rendering performance is how fast pixels can be written to the frame buffer memory.

An animated object

We are constantly redrawing the image, so it makes sense to animate the scene to make use of all that rendering power. All you need to do is make some aspects of the rendering dependent on time. We can use the GLFW timer to keep track of time, and rotate the object by a transformation. Uncomment the following line before the call to glBegin and you should see a rotating triangle instead.

```
glRotatef(90.0f*glfwGetTime(), 0.0f, 0.0f, 1.0f);
```

The rotation speed is 90 degrees per second, and the rotation is performed around the axis (0,0,1), i.e. The Z axis. Note that, contrary to many other programming APIs, OpenGL has chosen to measure angles in degrees rather than radians.

A more complicated object

Our program has grown since we started, and it is time to move some stuff out of the main function to make it more readable. It is also time to draw something in true 3D - the triangle we have drawn up until now is only a flat object in the (x,y) plane. The file "planets4.c" contains a more complicated object: a colorful cube with six faces. Try it out and browse through the code.

Hierarchical transformations

The cube in ”planets4.c” rotates around the Z axis, and the camera is looking along the Y axis. To make the view a little more interesting, you can add a static rotation immediately before the animated rotation, by simply adding the following line before the existing call to `glRotatef`:

```
glRotatef(30.0f, 1.0f, 0.0f, 0.0f);
```

Hierarchical transformations are a key to efficient 3D graphics programming. Let’s try that in OpenGL. To make a planetary system with a sun and a planet, we could replace the drawing commands with this:

```
float t = (float)glfwGetTime();
glRotatef(30.0f, 1.0f, 0.0f, 0.0f);
glPushMatrix();
glRotatef(90.0f*t, 0.0f, 1.0f, 0.0f);
drawColorCube(1.0);
glPopMatrix();
glRotatef(30.0f*t, 0.0f, 1.0f, 0.0f);
glTranslatef(3.0f, 0.0f, 0.0f);
glRotatef(180.0f*t, 0.0f, 1.0f, 0.0f);
drawColorCube(0.5);
```

The order in which the transformations are applied matters. A rotation followed by a translation will translate the object along the new, transformed coordinate axes. A translation followed by a rotation will rotate the object at a new position, but keep the rotation axis in the same place relative to the object. Now try to extend the small planetary system with acubical moon orbiting around the planet. You may have to adjust the size, position and rotation speed of the objects to make the scene look good.

Draw a sphere

Real celestial bodies like stars and planets are not cubical, but approximately spherical. Drawing a sphere in OpenGL means approximating it with polygons. It is not difficult to do this, but it takes quite a bit of work to get it right. Instead of asking you to do the sphere drawing function yourself, we present you with a finished function with the name `drawColorSphere` to illustrate the principle. You can find it in the file ”planets5.c”. Compile and run the file, and have a good look at the new code in the function `drawColorSphere`.

Like `drawColorCube`, the size of the sphere is one parameter to the function, and an additional integer parameter makes it possible to choose the detail level of the polygonal approximation. Try varying the detail level from around 2 to around 20 to see what happens with the shape of the “sphere” object. If you ask for a detail level of 100, how many polygons will be drawn for each sphere? How high can you go in the number of polygons before your computer starts to drop the frame rate below an acceptable speed and make the animation jerky? (Note that the main workload of the drawing is not the graphics, but rather the large amount of calls to sin() and cos() functions for each frame. There are smarter ways of drawing a sphere in OpenGL.)

Use the sphere to create your own planetary system with a sun, a planet and a moon. Spend more triangles for large spheres, and fewer for small ones to make the display look nice without using an unnecessarily high number of polygons.

A texture

Without textures for the object, the scene looks rather dull. In OpenGL, it is easy to add a texture to an object. All it takes its that you enable texturing, load a texture into memory, activate a texture and make sure to supply texture coordinates with your vertices. The hardest part to implement yourself is the reading of pixel data from an image file of some sort.Once that is taken care of, activating and using a texture is easy. Texture images stored in TGA bitmap files can be read in by GLFW, so we will use the convenience function glfwLoadTexture2D to load and activate a texture. (Without the aid of GLFW, loading and activating a texture will require a few more lines of code, but will also be more flexible.) Code to activate and use a texture can be found in the file planets6.c. Use that code as a starting point for doing your own planetary system as defined below.

Assignment

Create an animated planetary system with a sun, a planet and a moon, all rendered as textured spheres of reasonable roundness.

Optional assignment: Light sources

Lighting is an important part of 3D graphics, and OpenGL has built-in support for three types of lights: directional lights, point lights and spotlights. Until now, we have kept all lighting disabled, and then objects are simply rendered in the current vertex color, as if there was a full strength white ambient light. When lighting is enabled, vertex normals and the positions and properties of currently active light sources also affect the color. If texturing is also enabled, the texture color is applied to the object first, and then multiplied by the result from the lighting calculations.

Code to activate and place a point light source can be found in `planets7.c`. Modify it to suit your needs and make your solar system appear as if it were all lit by the sun in the middle:

Activate and place a point light source in the middle of the sun, set a white base color for all objects, use textures for the sun, the planet and the moon. Draw the sun with lighting disabled (which will yield full intensity), but draw the planet and the moon with lighting enabled. Hint: you can use some ambient light to make the shadow side of the planet and the moon appear somewhat lighter than pitch black. Ambient light can be set by the following statements:

```
float color4f[4]={0.1f, 0.1f, 0.1f, 1.0f};
glLightfv(GL_LIGHT0, GL_AMBIENT, color4f);
```

More information

If you want to go further and explore more of the endless possibilites with OpenGL, there is a lot of information on the Internet. One very popular set of tutorials is NeHe, `http://nehe.gamedev.net`. They are easy to follow and thorough, but they are based on an older and outdated OpenGL framework called GLUT, so you might want to adjust them somewhat to use GLFW instead. If you are serious about learning more about OpenGL, a highly recommended source of knowledge is “OpenGL Programming Guide”, published by Addison-Wesley. There is also the “OpenGL Reference Manual” from the same publisher, if you want a full alphabetical list of all OpenGL functions. Before you buy either of those books, however, have a look at `http://www.opengl.org`, where you can find older versions in HTML and PDF format. Contrary to Direct3D, OpenGL has not undergone any dramatic architectural changes over the years, so older versions of OpenGL documentation are still perfectly useful for learning the basics. Everything written about OpenGL since version 1.1 from the early 1990's is still a source to learn from. Have fun!