CS3388B: Lecture 3

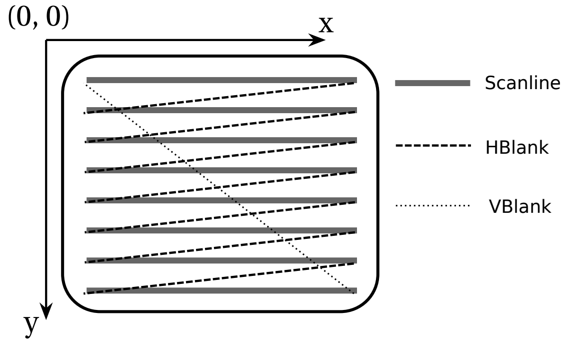
January 17, 2023

3 Basic Building Blocks: Windows, Primitives, and Paradigms

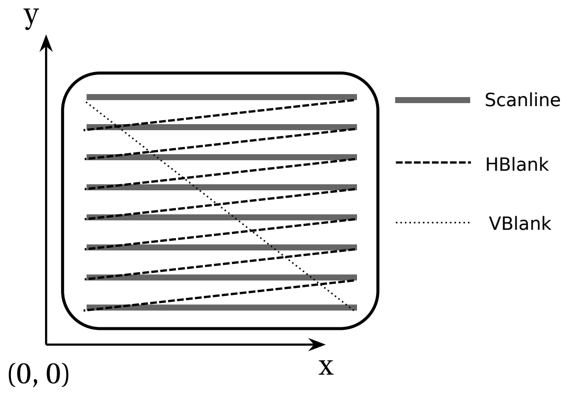
3.1 Displays and Windows

Each graphical display has its own inherent coordinate system. Desktop environments and windowing systems may expose their own coordinate system to the programmer and then translate that system to the layout expected of the display and graphics drivers.

Left-Hand Coordinate Systems. In a left-hand system, the origin is the top-left of the screen and positive x extends right, and positive y extends down. This paradigm is used by Direct2D, Nintendo, Tkinter (GUI in Python), among others.



Right-Hand Coordinate Systems. In a right-hand system, the origin is the bottom-left of the screen and positive x extends right, and positive y extends up. This paradigm is used by Direct2D, Nintendo, among others.



Note: It is also possible for a left-hand coordinate system to a bottom-left origin. The key is to consider the z-axis. A window itself does not have a z-axis, so the definitions remain as above. A three-dimensional system is left-handed if $\hat{\imath} \times \hat{\jmath} = \hat{k}$ points in the direction thumb following the left-hand rule. If $\hat{k}$ following the direction of the right hand, it is a right-handed system.

It's important to keep in mind which coordinate system your display or windowing system is using. Why? Because the definitions of different *transforms* is different depending on if its left-hand or right-hand. For example, clockwise rotations become counter-clockwise rotations.

Most often, a graphical library internally maintains two different coordinate systems. **User space** is defined by the coordinates the API exposes to the user/programmer. **Device space** is defined by the coordinates expected of the output display (or windowing system, if your graphics library is a middleware).

Windowing systems are softwares that manage and maintain multiple render targets for different programs simultaneously. Rather than a graphical program drawing directly to the output device, the program would draw to a particular **frame** or **window**, with its own coordinate system, and the windowing system takes care of layering and combining multiple windows to give a unified frame to render for the display.

When you write a graphical program, you typically only work with one window at a time. Each window has its own **context** which maintains different information about the render settings and render target. Once you create a window and get a hold of its context, you typically only interact with the context, rather than the window, allowing the graphics library to act as middleware.

Because of all these challenges of device-dependent coordinate systems, video drivers, aspect ratios, etc. unless you're hard-core, you are only ever going to use a graphics library as middleware.

3.2 Graphics Primitives

A **primitive** is a basic graphical object: points, lines, triangles, text, etc.

An **attribute** is an additional parameter, settings, etc. that affects how a primitive is drawn. We'll get back to attributes next day.

1. A **vertex** is a point in space used to describe positions of more complex primitives. Vertices are often grouped together to create larger and higher-dimensional objects.

2. A **point** is a point in space. Technically, points are zero-dimensional and thus have no volume or area to draw. But, when a point is drawn, it is really drawn as a filled circle with some radius. A single vertex may be used to define a point.5

3. A **line** is a straight line *segment*, constructed by joining two vertices in a straight line and using those vertices as the end points of the line segement.

4. A **polyline** is a piece-wise line. It is a series of vertices joined together where intermediate vertices are both the end point of one line and the beginning point of another line.

5. **Text** is text.

6. A **raster graphic** or **raster image** is a 2D grid of pixels, with the color of each pixel stored

explicitly (often as some RGBA value).

7. A **sprite** is a special kind of raster graphic where instead of storing the color explicitly, an index into a *palette* is stored.

Take a moment to think about the following. Think about how you would draw a picture on a piece of paper. This is analogous to creating 2D graphics. When you are drawing on a piece of paper, most of what you draw are lines (segments). Well, curves would be a better word, since drawing *straight lines* (or perfect circles) by hand is very difficult. You draw many lines. You might draw several lines connected together, tip to tail. Over time, an image appears. Our brain *fills in the caps*. You may only draw the four lines which outline a square, but our brain interprets it as a solid square.

So, what's the point? In two dimensions, we often use collections of one-dimensional objects to create an image. In three dimensions, we often use collections of two-dimensional objects to create an image. Most of our world is opaque. You can't see inside, for example, a table. So why would your draw anything but the surface of the table. 3D graphics are really just collections of hollow objects composed of many two-dimensional *polygons*.

3.3 Polygons

Polygons are *closed* shapes created from a series of straight lines joined together.

- If none of the edges defining a polygon intersect, a polygon is called **simple**.

- If any line (which is not also an edge of the polygon) intersects a polygon's surface at exactly two places, then the polygon is called **convex**. Alternatively, all interior angles are less than or equal to 180°.

We have special kinds of polygons which you're already familiar with:

- A **triangle** is a three-sided polygon
- A **quadrilateral** or **quad** is a four-sided polygon

To make things confusing, people often say "polygons" when they really mean "triangles". It's common to hear things like "This scene has 4000 polys". What that actually means it that there are 4000 triangles in the scene.

So what's with this nomenclature? It's because polygons are almost always "rasterized" as collections of triangles. Take any quadrilateral and draw one of its diagonals. Now you have two triangles side by side.

In general, there's lots of different collections of polygons that are graphics primitives.

These polygons are specified using vertices. Based on the type of polygon primitive you want to draw, you give a certain number of vertices. 2 vertices make a line, 3 vertices make a triangle, 4 vertices make a quad, etc. See Figure 1.

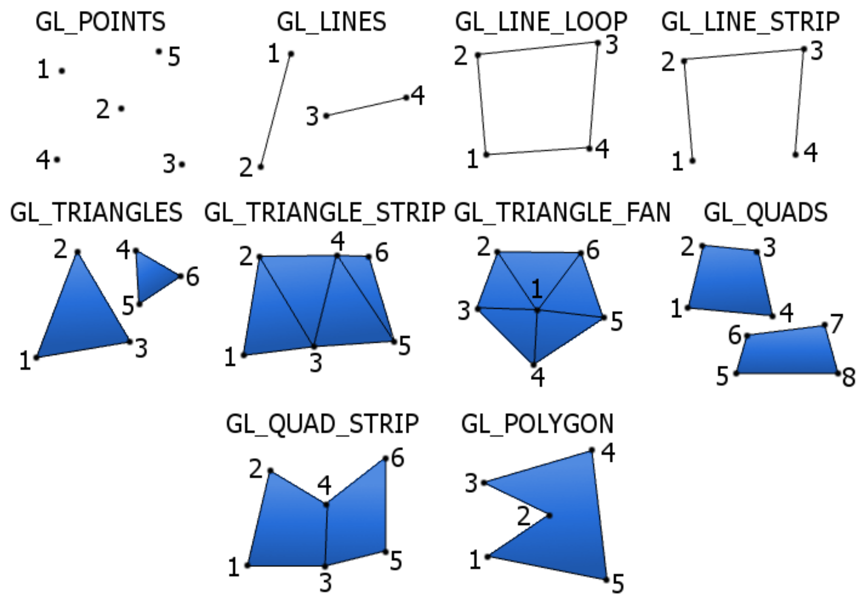


Figure 1:

Quads, quad strips, and polygons are *deprecated*.

Winding Order

While not strictly needed for 2D graphics, and thus the first part of this course, we're talking about polygons so we may as well talk about winding order.

Remember that any plane has two possible normals, one pointing "up" and another pointing "down". You can think of this as the "front side" and "back side" of a plane. A polygon is just a subset of a plane.

Winding order determines which of these normals to use. More precisely, it determines which side of the polygon is the "front side". Normals are assumed to extend out of, and away from, the front side.

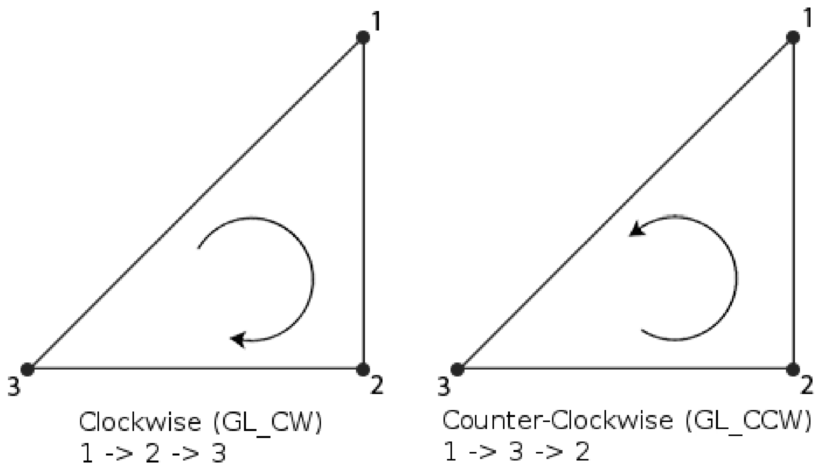


Figure 2:

Consider Figure 2. The left triangle is drawn in the order [1, 2, 3]. With a clockwise winding order, this triangle's normal is facing *toward* the viewer, and thus out of the screen. The right trangle is drawn in the order [1, 3, 2]. With a counter-clockwise windiner order, this triangle's normal is again facing *toward* the viewer. **Counter-clockwise is the default**.

3.4 Graphics Library Paradigms

OpenGL is a very common graphics library. It's the focus of this course after all.

OpenGL follows two paradigms that can make programming in it confusing. So we better get accustomed to it right away.

- **Global state**. For each *OpenGL context* there is one global state machine which holds all of the information and render settings. If you are trying to draw two different things, the render settings from the previous will apply to the next, unless explicitly changed between draw calls.

- **Client/Server**. The OpenGL interface follows a client/server architecture. This does not mean that there is necessarily networking between client and server. Rather, that the program which makes calls to the API is on the "client-side" and all data and requests must be passed through the API to the internal "server-side". The server itself then does the rendering.

  You must explicitly send data from client to server (or at least send a pointer to the data on the client side) in order to render it.

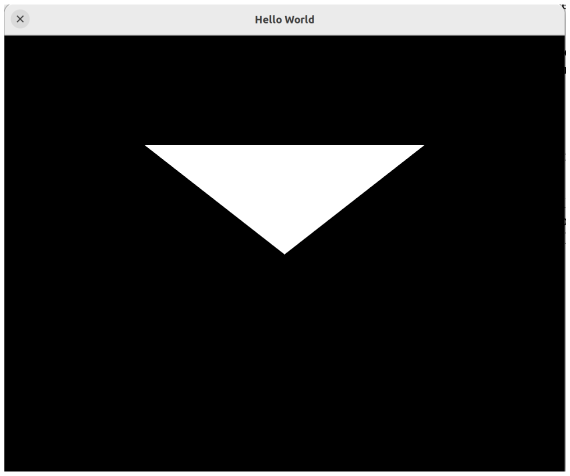
The other main paradigm of graphics libraries is **object-oriented**. There is no global state, but rather objects encapsulate the required data and state, just like object-oriented programming. DirectX 12 and Vulcan are object-oriented.

Immediate Mode and Retained Mode

Not to be confused with the Client/Sever paradigm of OpenGL, there is yet another difference with *how* data gets to the server.

- In **Immediate Mode**, the client-side program calls functions *every frame* which tells the server how to render the scene. This includes specifying vertex positions, what primitives to draw, and other attributes. Immediate mode is characterized by `glBegin()` and `glEnd()`

```
glBegin(GL_TRIANGLES);
glVertex2f(0, 0);
glVertex2f(0.5, 0.5);
glVertex2f(-0.5, 0.5);
glEnd();
```



- In **Retained Mode**, the server-side *retains* the data to be rendered. The client uses API calls to "initialize" data containers which are internal to the library/server. Those containers only need to be filled once (assuming the data itself is not changing). Then, when attempting to draw, the client only needs to supply the (index of) the server side object containing the data to render.

Retained mode is more "advanced" but *much* more useful and practical then immediate mode. We will revisit retained mode when we start dealing with 3D graphics.

3.5 The Hello World of OpenGL

Nearly every graphical program follows these general steps:

1. Create an **OpenGL context**. Each context holds an entire OpenGL "runtime" (e.g the client/server setup).
2. Create a window to display the rendered image.
3. Specify the data to render.
4. Draw the objects to render in a loop, once per frame, until the program terminates (e.g. the window is closed).

Create the Window

Steps 1 and 2 are often quite challenging and *very* specific to the particular operating system. Instead of dealing with that, we have GLFW!

GLFW (Graphic Library FrameWork) is a simple library the encapsulates the creation of windows and OpenGL contexts on a variety of operating systems. In programs using GLFW all we have to do is:

- `glfwInit()`
- `window = glfwCreateWindow(...)`
- `glfwMakeContextCurrent(window)`

Done. We have an OpenGL context and a window, and they're tied together.

Draw stuff

Since we're starting with immediate mode, steps 3 and 4 above are tied together. Here's a simple loop to draw a triangle every frame until the GLFW window is closed.

```c
/* Loop until the user closes the window */
while (!glfwWindowShouldClose(window))
{
    /* Poll for and process events (mouse, keyboard, etc.) */
    glfwPollEvents();

    /* Color the whole view as glClearColor. By default, black. */
    glClear(GL_COLOR_BUFFER_BIT);

    /* Use immediate mode to try a triangle */
    glBegin(GL_TRIANGLES);
        glVertex2f(0., 0.);
        glVertex2f(0.5, 0.5);
        glVertex2f(-0.5, 0.5);
    glEnd();

    /* Swap front and back buffers */
    glfwSwapBuffers(window);

}
```

Note: Yes, we have to do this swap buffers thing. Why? We'll get their later. For now, just make sure you do it at the end of each draw loop.

Note: It's a common code style to indent everything one extra level between `glBegin()` and `glEnd()`

HelloWorld.cpp

```cpp
#include <GLFW/glfw3.h>

int main(void)
{
    GLFWwindow* window;

    /* Initialize the library */
    if (!glfwInit())
        return -1;

    /* Create a windowed mode window and its OpenGL context */
    window = glfwCreateWindow(800, 500, "Hello World", NULL, NULL);
    if (!window)
    {
        glfwTerminate();
        return -1;
    }

    /* Make the window's context current */
    glfwMakeContextCurrent(window);


    /* Loop until the user closes the window */
    while (!glfwWindowShouldClose(window))
    {
        /* Poll for and process events (mouse, keyboard, etc.) */
        glfwPollEvents();

        /* Color the whole view as glClearColor. By default, black. */
        glClear(GL_COLOR_BUFFER_BIT);

        /* Use immediate mode to try a triangle */
        glBegin(GL_TRIANGLES);
        glVertex2f(0., 0.);
        glVertex2f(0.5, 0.5);
        glVertex2f(-0.5, 0.5);
        glEnd();

        /* Swap front and back buffers */
        glfwSwapBuffers(window);

    }

    glfwTerminate();
    return 0;
}
```

To compile this with g++:

```
g++ HelloWorld.cpp -lOpenGL -lglfw
```