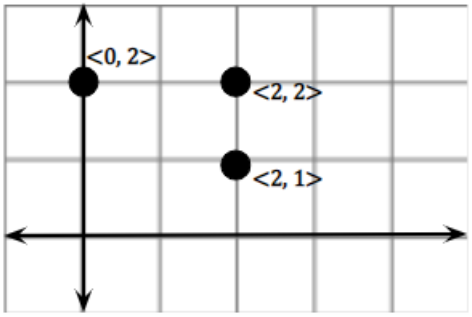
1 Three Paradigms for Rendering

We first begin by discussing how to render points, lines, and triangles.

1.1 Drawing Points

Consider the following graph:



In this class, we represent points in the form $\langle x, y \rangle$, or $\begin{bmatrix} x \\ y \end{bmatrix}$. In C++, we can represent these points like so:

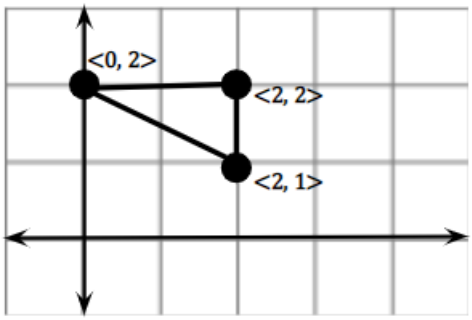
```
float verts[][2] = {
    {2, 1},
    {2, 2},
    {0, 2}
};
```

After loading the array into the GPU, which we'll discuss later, we can use a command like:

```
glDrawArrays(GL_POINTS, 0, 3);
```

1.2 Drawing Lines

Consider the following graph:



In JavaScript, we can make use of the Canvas API to draw this like so:

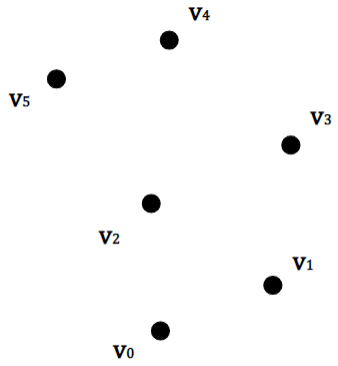
```
moveTo(2, 1);
lineTo(2, 2);
lineTo(0, 2);
lineTo(2, 2);
stroke();
```

In OpenGL, we would do something like:

```
glDrawArrays(GL_LINE_LOOP, 0, 3);
```

The `GL_LINE_LOOP` means that we're making a closed loop of edges, using 3 vertices in total.

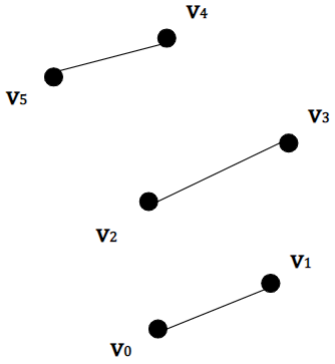
There are several other modes that we can use. To see how these differ, we'll use the following set of vertices as an example:



- `GL_LINES`: If we have the following code segment

  ```
  glDrawArrays(GL_LINES, 0, 6);
  ```

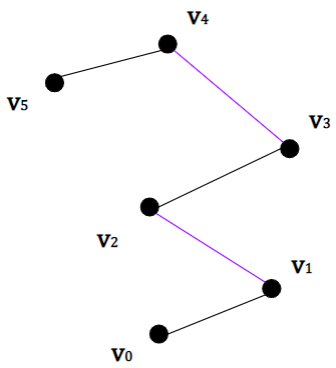
  Then, we would get the following drawing:



- `GL_LINE_STRIP`: Now, if we were to include the following code segment in addition to the one above

  ```
  glDrawArrays(GL_LINES_STRIP, 0, 6);
  ```

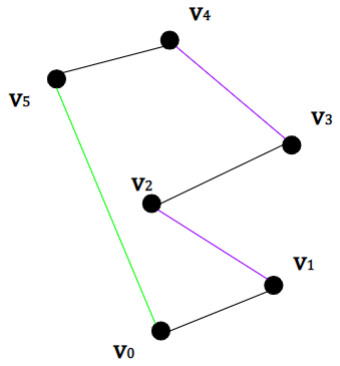
  Then, we would get the following drawing:



- `GL_LINE_LOOP`: Finally, including

  ```
  glDrawArrays(GL_LINES_LOOP, 0, 6);
  ```

  would yield the following drawing:



So, to sumamrize, if we have the vertices $\{v_1, v_2, \ldots, v_n\}$, then:

- `GL_LINES` will draw a line for each pair of vertices; that is, a line will be drawn between v_1 and v_2, v_3 and v_4, and so on.
- `GL_LINE_STRIP` will draw a line for each consecutive pair of vertices, up to and including v_{n-1}; that is, a line will be drawn between v_1 and v_2, v_2 and v_3, v_3 and v_4, and so on. The last line drawn will be from v_{n-1} to v_n.
- `GL_LINE_LOOP` will draw a line for each consecutive pair of vertices, including from the end vertex to the start vertex. So, effectively, this is just `GL_LINE_STRIP` but with a line from v_n to v_1.

1.3 Drawing Triangles

Like with drawing lines, there are three modes for drawing triangles.

- `GL_TRIANGLES`
- `GL_TRIANGLE_FAN`
- `GL_TRIANGLE_STRIP`

Using the same set of 6 points above, we show the following examples.

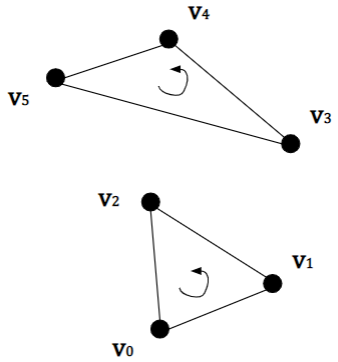
1.3.1 `GL_TRIANGLES` Mode

This mode groups the vertices into groups of three, and then draws a triangle between each group. For example, if you have vertices $\{v_0, \ldots, v_5\}$, then this mode would take vertices $\{v_0, \ldots, v_2\}$ and $\{v_3, \ldots, v_5\}$ and draw a triangle (from $v_0 \to v_1$ and then from $v_1 \to v_2$ and finally $v_2 \to v_0$, while filling it in with a color).

When we use the `GL_TRIANGLES` mode, like so:

```
glDrawArrays(GL_TRIANGLES, 0, 6);
```

Then we get something like:

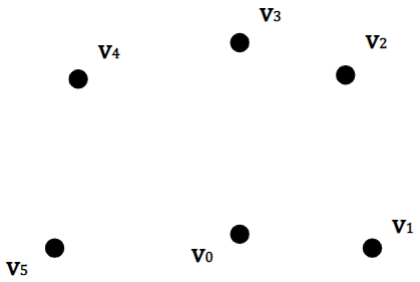


Note that the arrows here indicate that we're looking at the "front" faces of the triangle. The "back" face is the other side.

1.3.2 `GL_TRIANGLE_FAN` Mode

For an array of vertices $\{v_0, \ldots, v_n\}$, v_0 is the common vertex. Then, the rest of the vertices $v_1, \ldots, v_n$ are defining a triangle which shares the initial vertex.

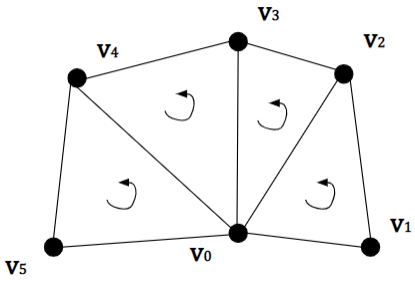
Suppose you're given the following set of vertices like so:



When using this mode, like so

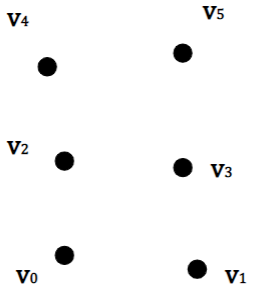
```
glDrawArrays(GL_TRIANGLE_FAN, 0, 6);
```

Then we get something like:



1.3.3 `GL_TRIANGLE_STRIP` Mode

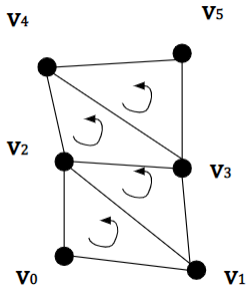
Suppose you're given the following set of vertices like so:



When using this mode, like so

```
glDrawArrays(GL_TRIANGLE_STRIP, 0, 6);
```

Then we get something like:



The way to think about this is that you have your "base" triangle with vertices $\{v_0, v_1, v_2\}$. Then, v_3 is facing the edge formed between v_1 and v_2. Likewise, the vertex v_4 is facing the edge formed between v_2 and v_3 of the triangle with vertices $\{v_1, v_2, v_3\}$.

2 Culling, Hidden Surfaces, Animation

Suppose you have a 3D scene, typically rendered as triangles. These will also typically be rendered with a view, i.e. camera, which has a position and direction.

2.1 Culling

Note that a triangle could be *behind* the viewer. Or, maybe, the triangle is *inside* the sphere. This gives us a definition.

Definition 2.1: Cull

Meaning the same thing as discard, a **cull** is simply the triangles that are not visible to the viewer.

When rendering closed, solid, objects' surfaces, we can also cull back faces.

We can use the following OpenGL commands to enable cull back faces:

```
glEnable(GL_CULL_FACE);
```

We can also specify which faces are culled. By default, the faces that are culled are the back faces. That is, the following is the default setting:

```
glCullFace(GL_BACK); // Default
```

We can also use the following commands:

```
glCullFace(GL_FRONT); // Cull front faces.
glCullFace(GL_FRONT_AND_BACK); // Cull all faces.
```

To disable culling, we can use:

```
glDisable(GL_CULL_FACE);
```

We can also change the conventions on counter-clockwise vs. clockwise, like so:

```
glFrontFace(GL_CCW); // Default
```

We can change this convention to make the opposite face the front face; that is, by using the following command:

```
glFrontFace(GL_CW);
```

Note that:

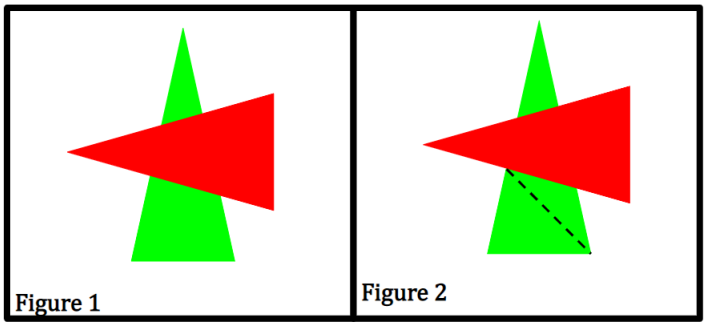
- `CCW` means counter-clockwise.
- `CW` means clockwise.

2.2 Hidden Surfaces Methods

Here, the idea is that if there are some triangles that are in front of the other triangles, the other triangles are hidden. There are three methods for this.

1. Geometric Analysis: Consider all pairs of triangles in the scene, figure out how they overlap with each other, and then render only the visible parts.

   For example, suppose we have a green and red triangle, where the red triangle is partially in front of the green triangle. We would get something like in figure 1.

   With geometric analysis, the bottom part of the green triangle would be broken up into two triangles; in other words, the green triangle would be broken up into three pieces.

   It's not hard to see that there are some significant disadvantages here. In particular:

   - You need to know *all* of the triangles ahead of time.
   - You need a sophisticated algorithm.
   - You need to redo this process when the viewer moves.

2. Painter's Algorithm: The idea is that you *sort* the triangles so that the furthest triangles from the camera/viewpoint get rendered first, and then you successively render the closer triangles, thus letting the closer triangles overwrite the farther triangles. The algorithm can roughly be written like so:

```
Sort triangles in (reverse) order from viewpoint.
    Farthest triangles from view are first in list of triangles
    Closest triangles from view are last in list of triangles
Render farther triangles first and closer ones last.
    The closer ones overwrite the portions of the ones they hide.
```

For example, suppose we have three triangles where the green triangle is the farthest from the viewer, the red triangle is the next farthest, and the blue triangle is the closest.

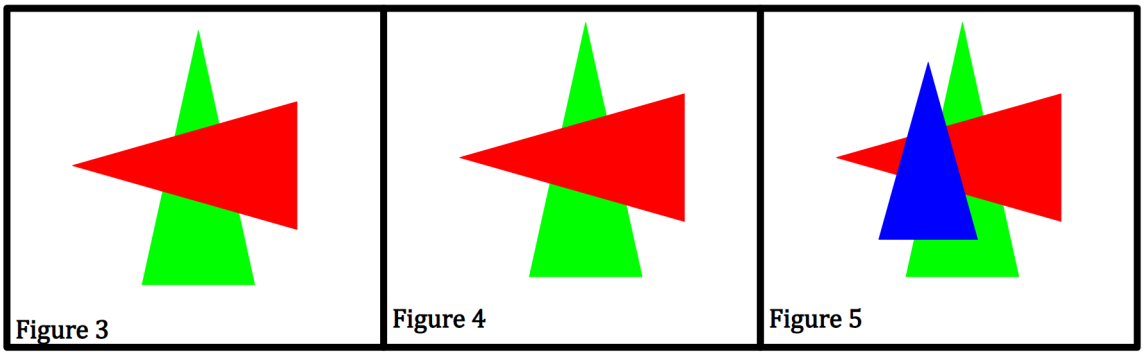
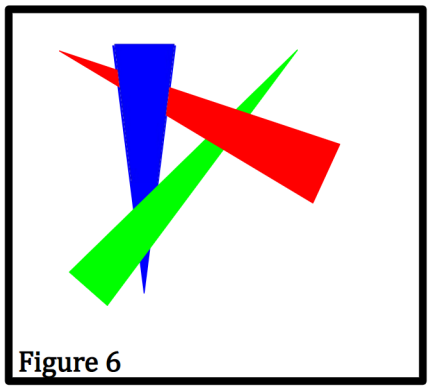


Figure 3 shows the green triangle being rendered first. Figure 4 shows the red triangle being rendered next, thus hiding portions of the green triangle's pixels. Figure 5 shows the blue triangle being rendered next, hiding portions of the green and red triangle's pixels.

One obvious advantage is that it's elegant. All you need to do is sort and render. However, like with Geometric Analysis, there are some disadvantages:

- You need to know *all* of the triangles ahead of time so you can sort them.
- You need to redo this process when the viewer moves.
- It may not be possible to sort consistently by distance. In particular, suppose you have interleaved triangles, like seen in figure 6.



3. Depth Buffer: This takes advantages of the fact that we're drawing into pixels. The idea is that you give each pixel two values; you give the pixel a color value and a **depth** value, or (essentially) the distance from the viewer. A rough algorithm is as follows:

```
Render triangles to pixels in any order. For each triangle:
    Compare its depth to the previously written depth.
    Keep the closer pixel color/depth values.
```

The advantages are:

- It's simple.
- You can render triangles in any order (so no need to sort).
- This can be done in parallel. Amenable to parallel computing.

The disadvantages are:

- You need extra memory for the depth values[1].
- Aliasing problems.

2.3 Animations

To give the appearance of motion, we show a succession of still images changing very rapidly.

In OpenGL, we maintain two frame buffers[2]:

- Front buffer: The current image being displayed.
- Back buffer: The next image to be displayed.

So, while the user is watching the front buffer, the back buffer is being processed (the next image that is being created). Then, you can just swap the buffers to switch the back buffer and the front buffers.

[1] Memory is cheap, so this isn't too bad.
[2] A buffer where we write an image.