SFU CMPT 361
Computer Graphics
Assignment 3

For this assignment, you are to expand your renderer from assignment 2 with perspective, a polygon clipper, ambient lighting, and obj file reading.

The required display is 1 panel, set up just as in assignment 2: the drawing area should be 650 x 650 pixels, and the white margins are all 50 pixels wide or tall. Thus, the total display area is 750 x 750 pixels. Use black for the background of the viewing area.

Command-line argument

Your program must accept a command-line argument, which is the filename of the simp file that you are to read and render. The filename should be specified without the ".simp" extension. You will have to research how to interpret java command-line arguments if you do not know already. If there is no argument present, the program should display the ten pages *pageA, pageB, pageC, … pageJ,* in that order. You are provided with pageA.simp, pageB.simp, … pageI.simp, but must create your own pageJ.simp.

Graphics file expansion

We will add some commands to the "simp" format files, and modify others. All of the previous commands which are not included here are still accepted as part of a simp file. There are five new commands, and two slightly modified ones. Again, all numbers are floats or doubles (your choice). The camera command is written here on two lines but should be written on one line in a simp file. All colors are in [0, 1] coordinates.

Line format	**Meaning**
camera <xlow> <ylow> <xhigh> <yhigh> <hither> <yon>	Place a perspective viewpoint at the origin in the coordinate system defined by the current transformation matrix (henceforth the CTMCS). The y-axis is the "up" vector for the viewpoint, and the viewing direction is negative z direction. The viewing plane is at z = -1, and the window from (xlow, ylow) to (xhigh, yhigh) on the viewing plane defines the sides of the viewing frustum. Hither and yon are the z-values for the near and far clipping planes. There may only be one camera command in a simp file, and it must be before all primitives. No commas between the values.
line <p1> <p2> polygon <p1> <p2> <p3>	As before, except that each of p1, p2, and p3 has the format (x, y, z) **or (x, y, z, r, g, b)**. If any point has r,g,b coordinates, then both of them must. The r, g, and b coordinates are expected to be in the range between 0 and 1.
obj "filename"	Read in the .obj file from the filename, and treat it as being in the current coordinate system. See below for details. Filename should not have the ".obj" extension on it. Thus, *obj "pyramid"* would read a file named pyramid.obj.

ambient (r, g, b)	Everything from this point forward will have the ambient light intensity set as specified by (r, g, b). At the start of the file, the ambient light intensity is pure black: (0, 0, 0).
depth <near> <far> (r, g, b)	Render everything from this point forwards using a depth effect based on camera-space z (csz). This is like depth cueing. If csz >= near, then no effect is applied (the lighting-calculation pixel color is used). If csz <= far, then the depth color (r, g, b) is used for the pixel. If near >= csz >= far, then lerp between the lighting-calculation color and the depth color. This is also called *fog* or *atmospheric perspective*. At the start, set near to the maximum negative z-value for your program (the most negative floating-point value) so that all csz you encounter are less than or equal to near (no effect is applied). This depth effect is very much like depth cueing except that you are using the lighting-calculation color rather than a fixed color, and the effect has a limited range in z.
surface (r, g, b)	Everything from this point forward will have default surface color k_d = (r, g, b), At the start of the run, this default color starts as pure white: (1, 1, 1).

None of the "this point forward" properties are affected by push and pop (**{** and **}**).

Camera Command

If the window defined is not square, then draw on a rectangle of the same aspect ratio as the defined window. Center this rectangle in the central panel. (See pageH for an example.)

Object (or .obj) files

We will read in a subset of the Wavefront .obj file format. This description is adapted from the wikipedia article on this format, https://en.wikipedia.org/wiki/Wavefront_.obj_file, and the fileFormat article http://www.fileformat.info/format/wavefrontobj/egff.htm . You will be given the basic outline of a parser for these files, but you will have to fill in the code that does the interpretation.

Any line beginning with a hash character (#) is a comment, and blank/whitespace-only lines are permitted.

A vertex can be specified in a line starting with the letter "v" followed by whitespace. This is followed by x, y, z, and possibly w coordinates. W is optional and defaults to 1.0. Vertex colors may be present, with the red, green, blue values after the x, y, z, and possibly w. The color values range from 0 to 1.

```
# A geometric vertex with x y z [w] coordinates.
# Numbers are space (or tab)-separated (no commas)
v 0.123 0.234 0.345 1.0

# w is optional and defaults to 1.0
v 0.678 0.901 0.235 1.5

# colors may be present: these lines are {x y z r g b} and {x y z w r g b}
v 0.15 0.39 0.04 0.7 0.12 0.5
v 0.25 0.41 0.33 2.0 0.7 0.42 0.42
```

A vertex normal can be specified on a line beginning with the letters "vn" followed by whitespace. This is followed by the x, y, and z components of the normal. The normals might not be unit vectors. Note that normals are specified independently of the vertex locations. There is no optional homogeneous coordinate on vectors.

```
# A vertex normal with x y z components.  Numbers are space(or tab)-separated (no commas)
vn 0.707 0.000 0.707
vn 0.000 0.707 0.921
```

Faces can be specified on a line starting with the letter "f", followed by whitespace, and then by a listing the indices of the vertices on the face in counterclockwise order. We will support positive and negative indexing; both are 1-based. In other words, the index 1 refers to the first vertex defined in the file, 2 to the second, etc. The index -1 refers to the most-recently-defined vertex, -2 to the second most-recently-defined vertex, etc. A face may consist of any number of vertices.

```
# polygonal faces
f 1 2 3
f 2 3 4 5 6 7
```

To use a vertex normal, you must append its index to the vertex index with two slashes ("//") between them. They are indexed the same way vertices are (using a separate numbering).
The v, vn, and f commands may be freely intermixed in the file.

```
# polygonal faces with vertex normals
f 1//1 2//1 3//2
f 2//1 3//2 4//3 5//4
```

All of the above is the basic .obj format. For this assignment, I want you to treat any line that does not begin with the token v, vn, or f as a comment. These lines are for more complex features. If you examine .obj files on the web, you'll find that there are many other tokens that can start a line.

Texture indices
Texturing will be of concern to us not because we implement it, but because texturing coordinates may be present in files that we read. (The "vt" command allows one to define texturing coordinates. We treat it as a comment.) To associate texturing coordinates with a vertex, one puts the texture index second in a slash-separated vertex description. For instance, in the following, the 1-digit numbers are texture indices. The first face has vertex and texture indices, and the second has vertex, texture, and normal indices.

```
# polygonal faces with texture coordinates
f 11/1 12/1 13/3
f 12/2/12 13/3/13 14/4/14 15/5/15
```

Your program must be able to read lines such as these, ignoring the texture indices, as we don't use them.

Negative indices

Relative negative indices are nice in that you can use recently-defined vertices and not have to determine absolute indices.

```
# in this group, the four vertices (in order) are referenced as -4, -3, -2, and -1 in the faces
v -0.500000 0.000000 0.400000
v -0.500000 0.000000 -0.800000
v -0.500000 1.000000 -0.800000
v -0.500000 1.000000 0.400000
f -4 -3 -2
f -3 -2 -1

# in this group, both vertices and normals are referenced with relative negative indices.
# the first vertex & normal are referenced as -3's in the face, the second as -2's, the third as
-1's.
v 0.500000 0.000000 0.400000
vn 0.500000 0.500000 0.000000

v 0.500000 0.000000 -0.800000
vn 0.500000 0.500000 0.200000

v 0.500000 1.000000 -0.800000
vn 0.500000 0.000000 -0.500000

f -3//-3 -2//-2 -1//-1
```

Polygons

If you read a polygon with more than three vertices, then you may want to break it up into triangles with every triangle containing the first vertex. That means if you have a face with vertices v1, v2, v3, … , vk, then you make a face with v1, v2, and v3, a face with v1, v3, and v4, a face with v1, v4, and v5, a face with v1, v5, and v6, etc. This will work if the polygon is planar and convex. It may produce some very odd objects otherwise, but do not concern yourself if that should happen. It is expected, and it is what the model-maker (.obj file author) gets for using a nonplanar or nonconvex polygon. There is no clear geometric interpretation of such an object.

Interpretation

I recommend the following method of interpreting a .obj file. First, create expandable arrays (or lists) for vertices, normals, and faces. The vertex array should have homogeneous (x, y, z, w) points and a color (r, g, b). The normal array has (nx, ny, nz), and the face array has face structures/objects, which are themselves an array or list of vertex descriptions. If you break up any large polygons right away, then each face can have three vertex descriptions. Otherwise, you need an arbitrary number.

A vertex description consists of a vertex index and a vertex normal index. You may set the vertex normal index to 0 if there is no normal specified, or you may compute and use the face normal. (You would then have to insert the normal into a second normal array, so as not to interfere with the indices of the first normal array). When you see a negative index, immediately convert it to a positive index so that the stored vertex description consists of positive indices only.

After reading the whole file like this, you can transform all of the vertices using the CTM. In assignment 4, we will also transform all of the normals, using the inverse CTM (right-multiplied on the homogenized normal, or left-multiplied by the inverse transpose). You can do this on this assignment if you want, but we will not be using the normals until assignment 4. After that, go through the face array, and send the triangles for each face to your renderer (looking up the transformed points in the vertex array).

It is your choice as to whether you keep the transformed vertices in the same data structure as your untransformed ones. (And similarly for normals.)

Syntax or semantic errors
If you encounter any malformed file, be it a simp file or a .obj file, you may abort your program or allow it to misbehave however it likes.

Lighting calculation
The lighting calculation is simple. To get the "lighting-calculation color," multiply the ambient lighting by the object color. You can do this at each vertex (using the vertex color) and then linearly interpolate, as in Gouraud shading. This should produce the same effect as lerping the color and then mulitplying the lerped color by the ambient lighting at each pixel, but the latter is more expensive. It is not necessary to do lighting calculations for lines.

Shading calculation
Use the directions specified above in the **depth** command. Apply at each pixel, using *perspective-correct* interpolation of csz. To do perspective-correct interpolation, linearly interpolate 1/csz. (At each vertex, compute z' = 1/csz, and lerp z'. At each pixel, take the lerped z' and compute csz = 1/z'.)

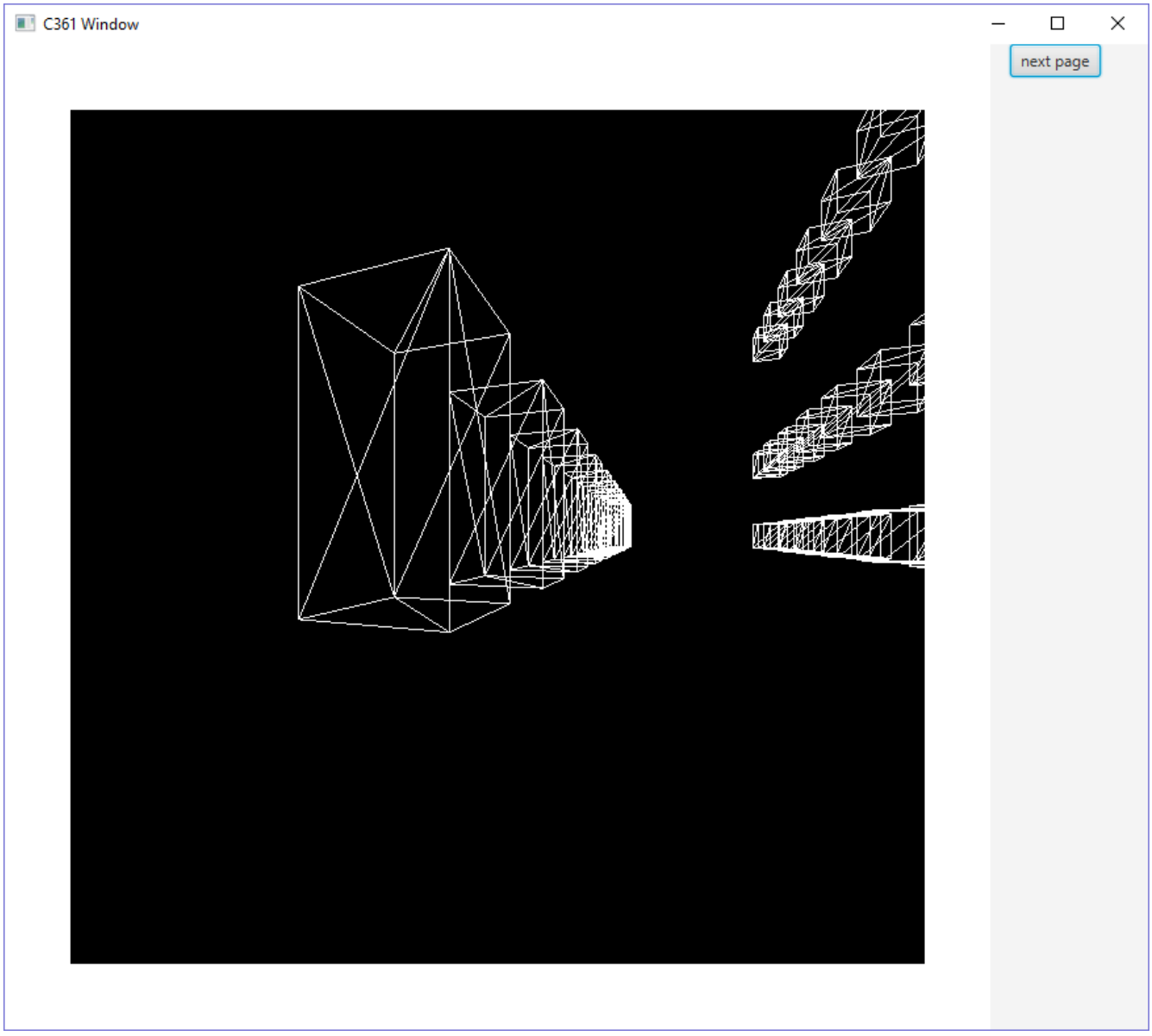
Clipping
You must use a proper polygon clipper for this assignment. You may clip either all in 3D or clip against hither and yon in 3D and the other sides in 2D.

PageJ.simp
Design an interesting scene showing off your renderer, and using at least one downloaded .obj file from the web. (For instance, *turbosquid.com* has free .obj files.)

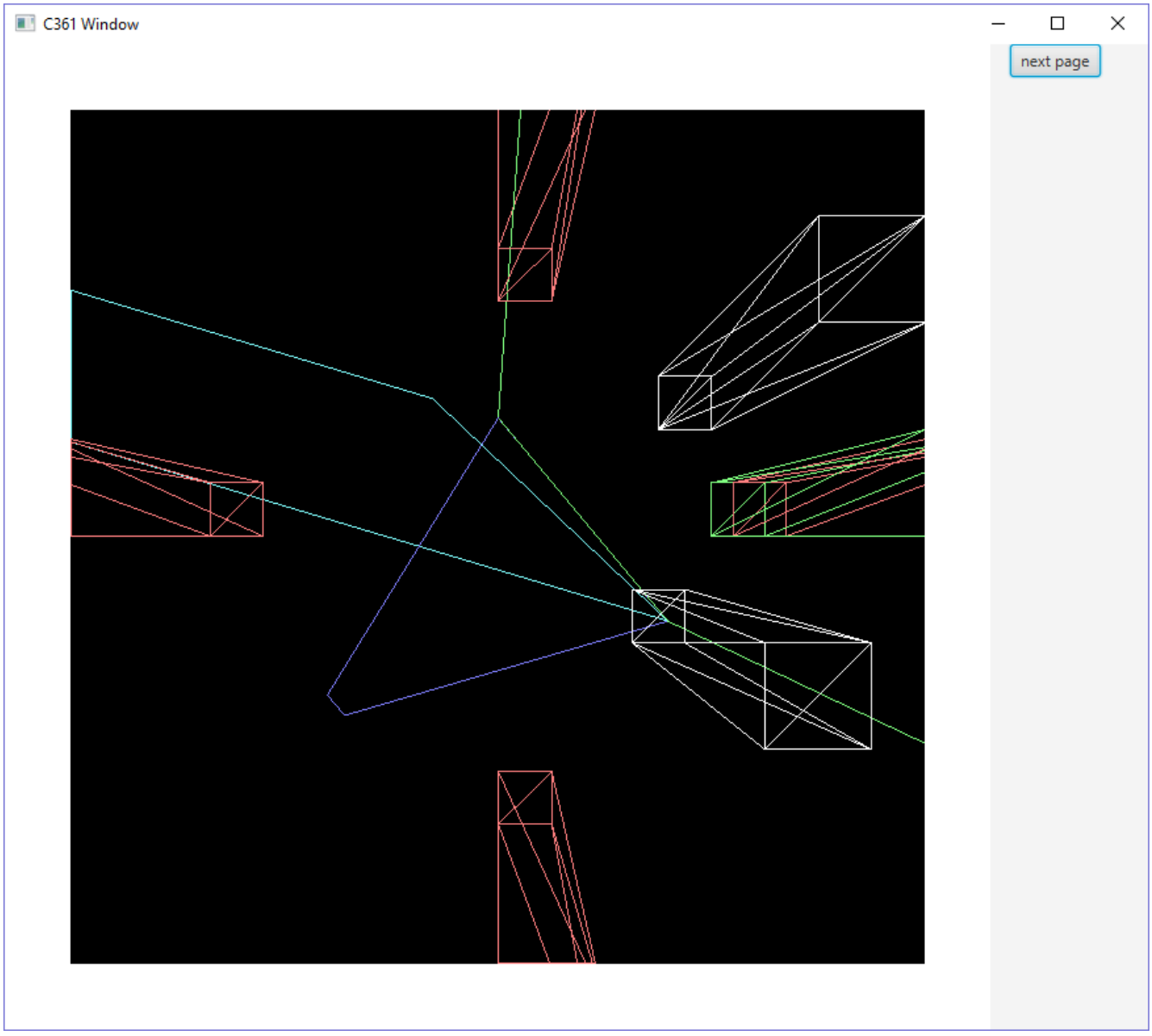
README
Do not forget to include a readme file with your submission, describing your development environment, any problems still existing in the code, instruction on how to run, and any comments or questions for the TA. (Send questions to me directly to me. Don't send me more than a snippet of code—bring your code in to me during office hours if it's a longer question.)

Other details and implementation hints will be discussed in class. It is your responsibility to know all of these details, so do not miss class.

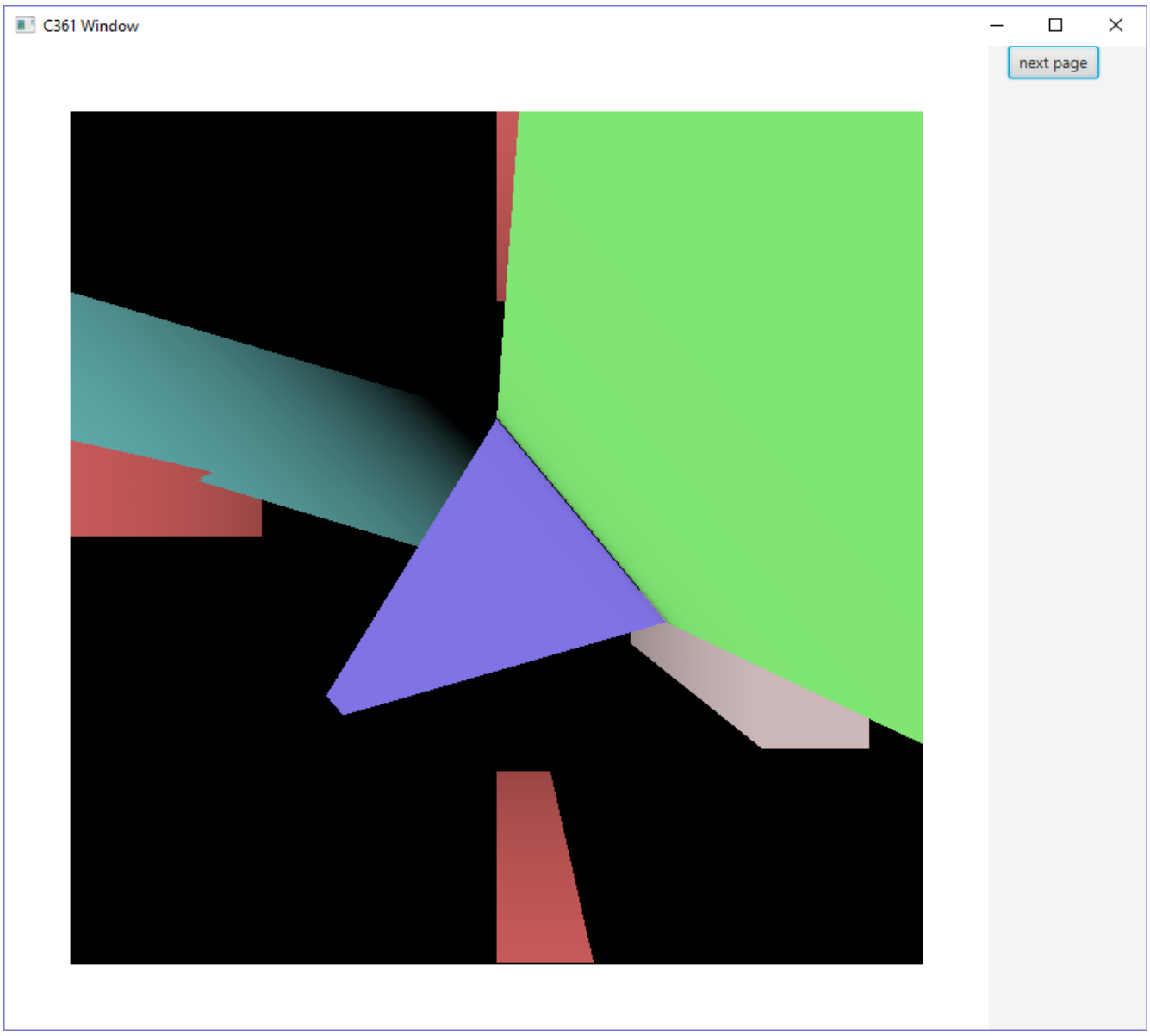
Appearance of completed assignment
The following images show approximately what the pages of your output should be. PageJ must differ significantly; make your own page files and do not use tomsPageJ as your model.



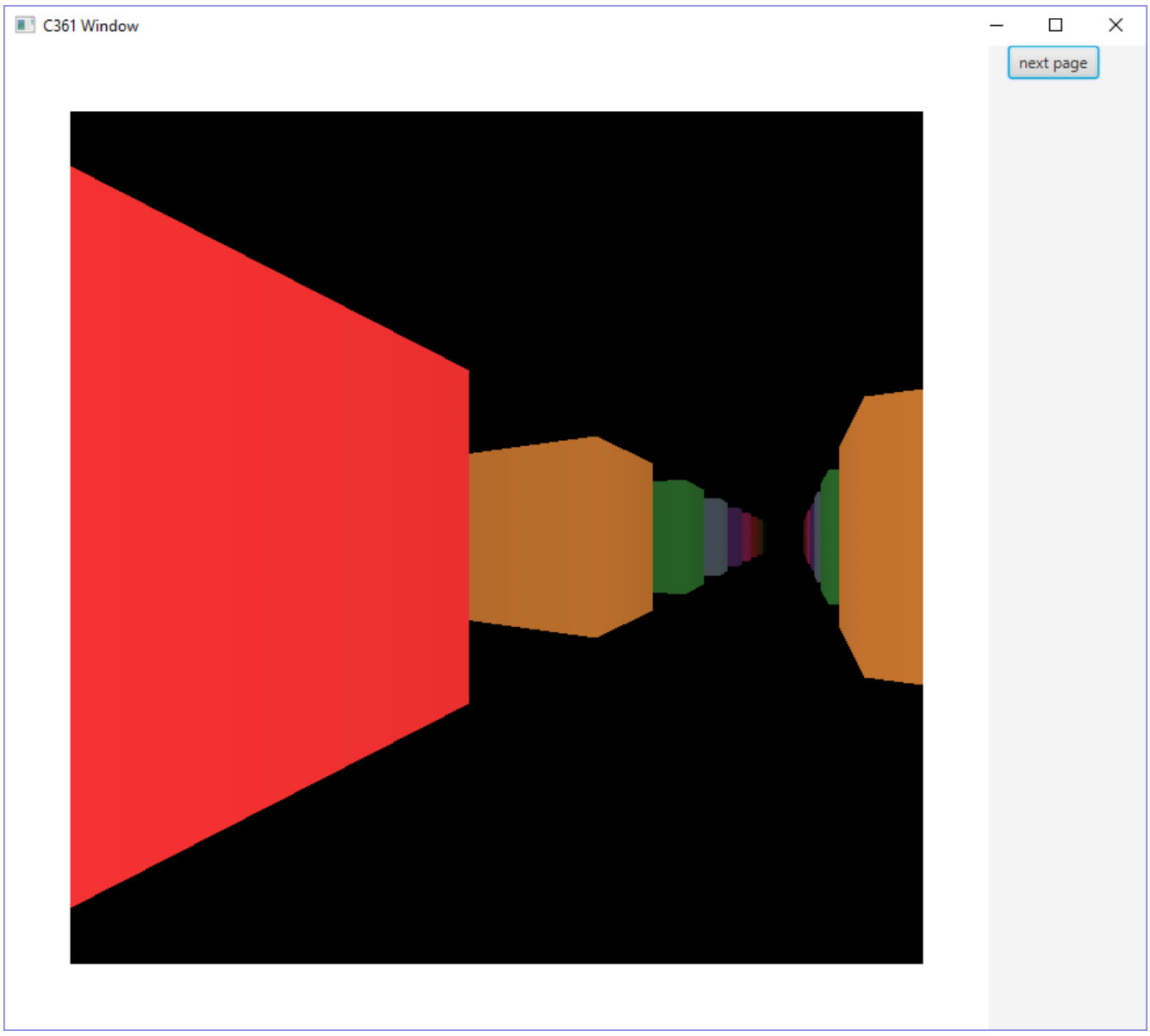
pageA



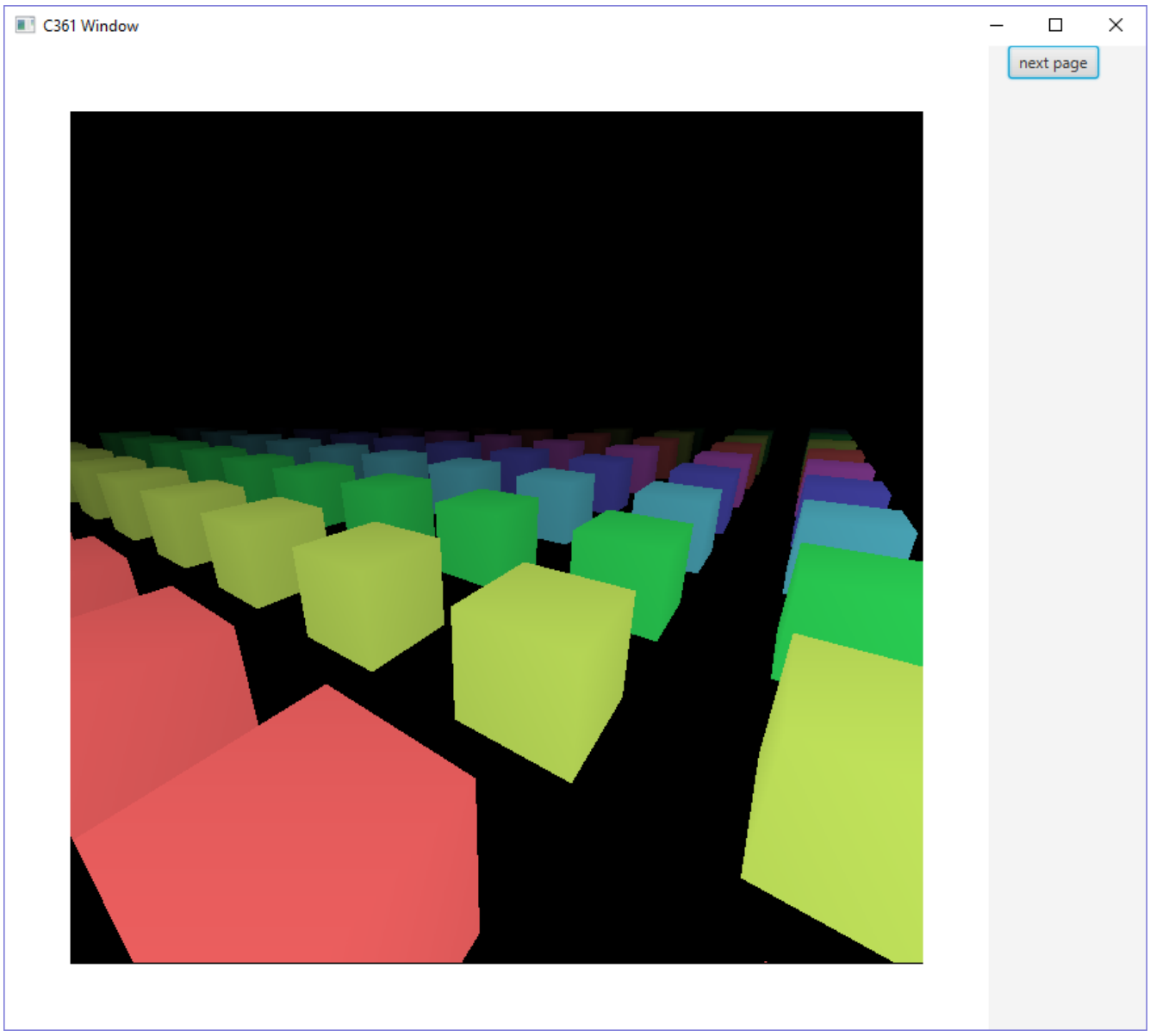
pageB



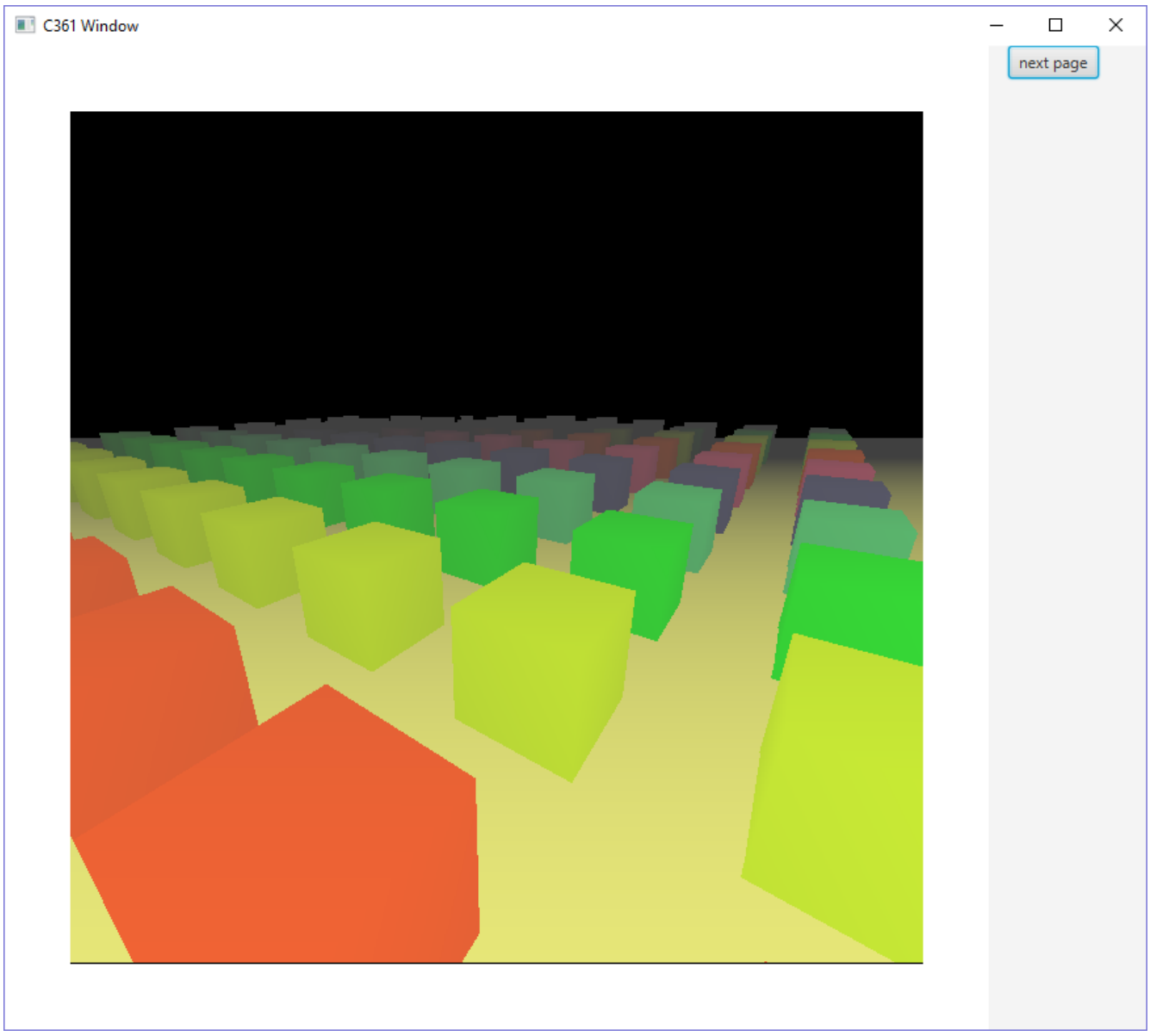
pageC



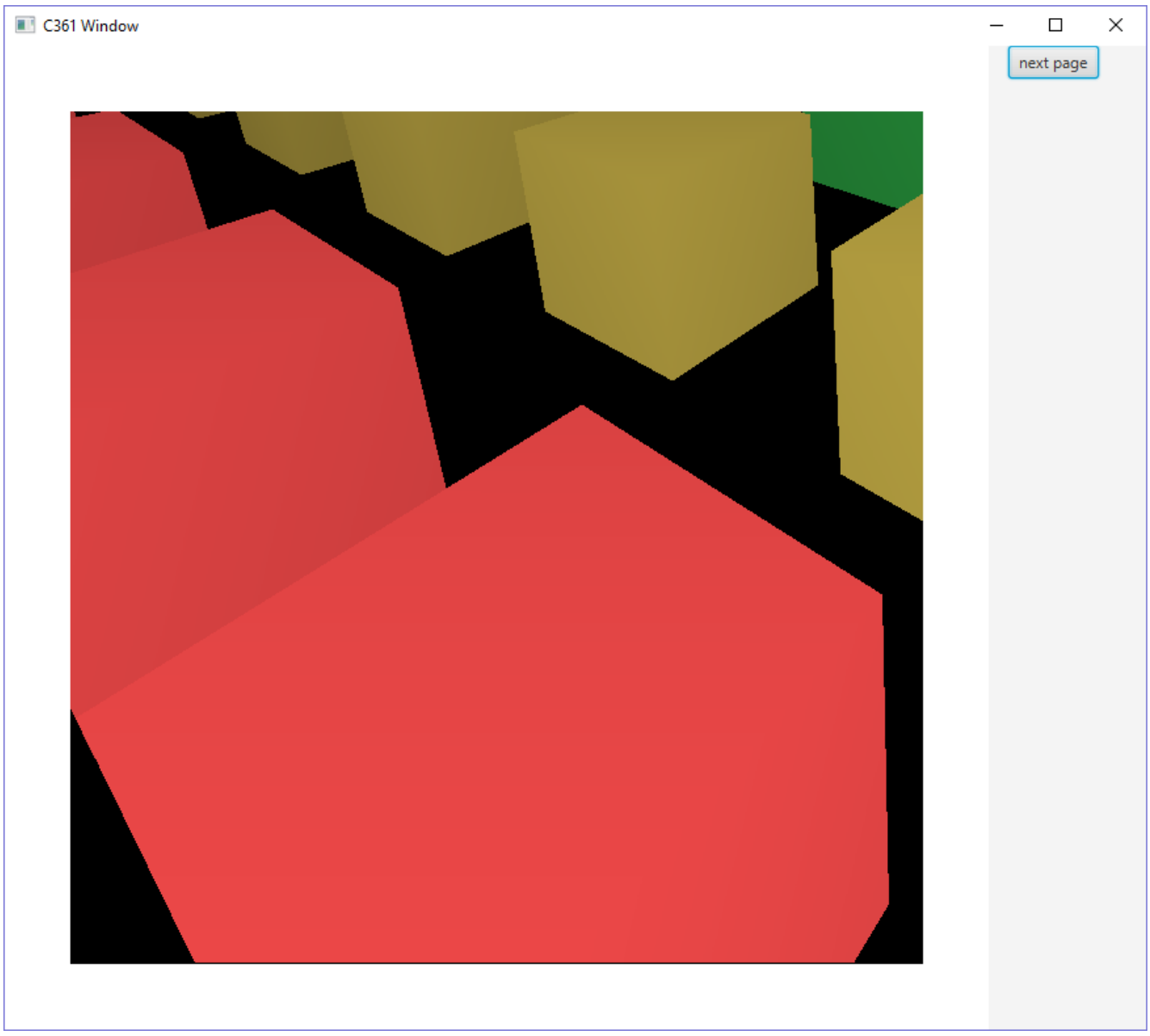
pageD



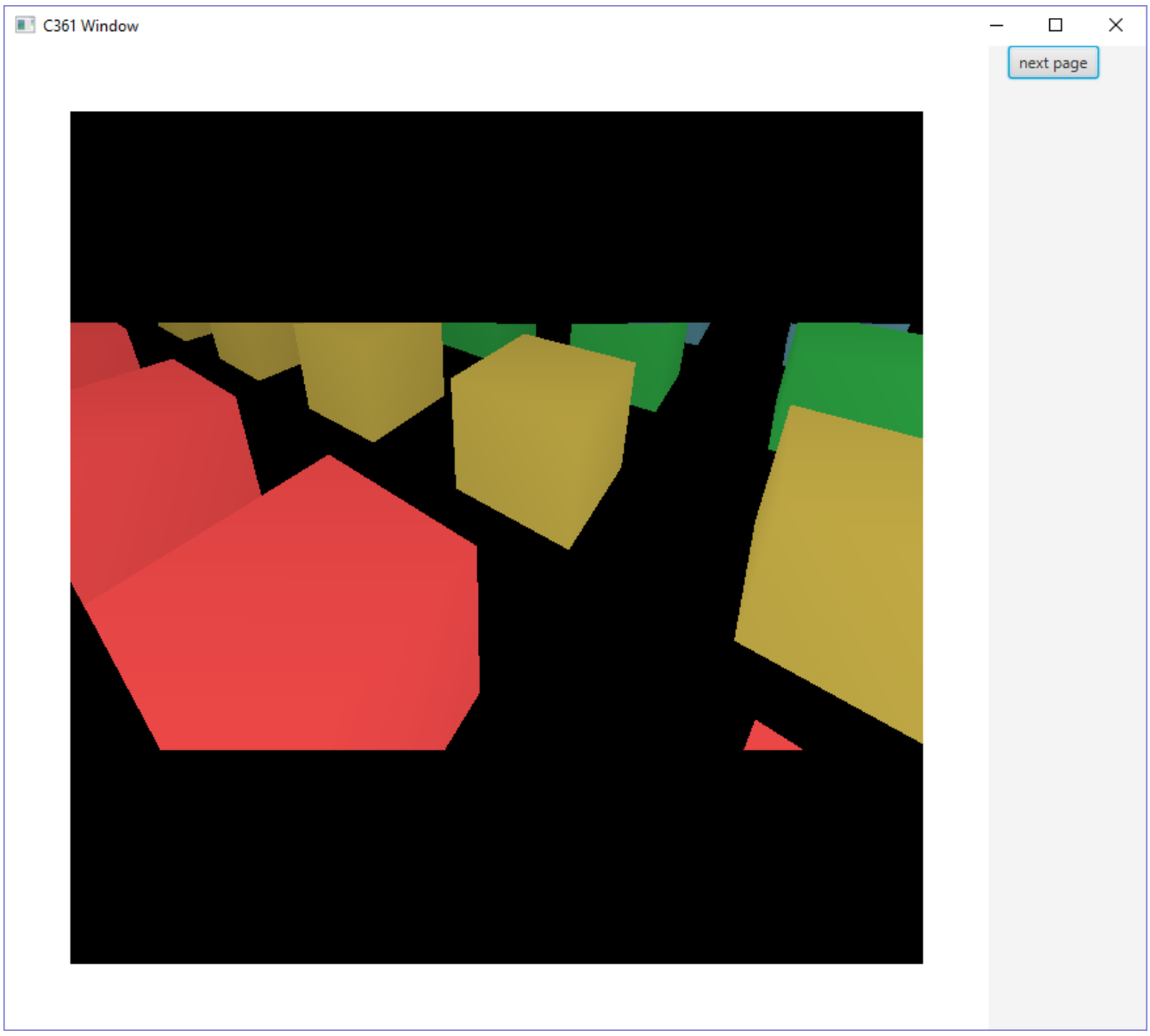
pageE



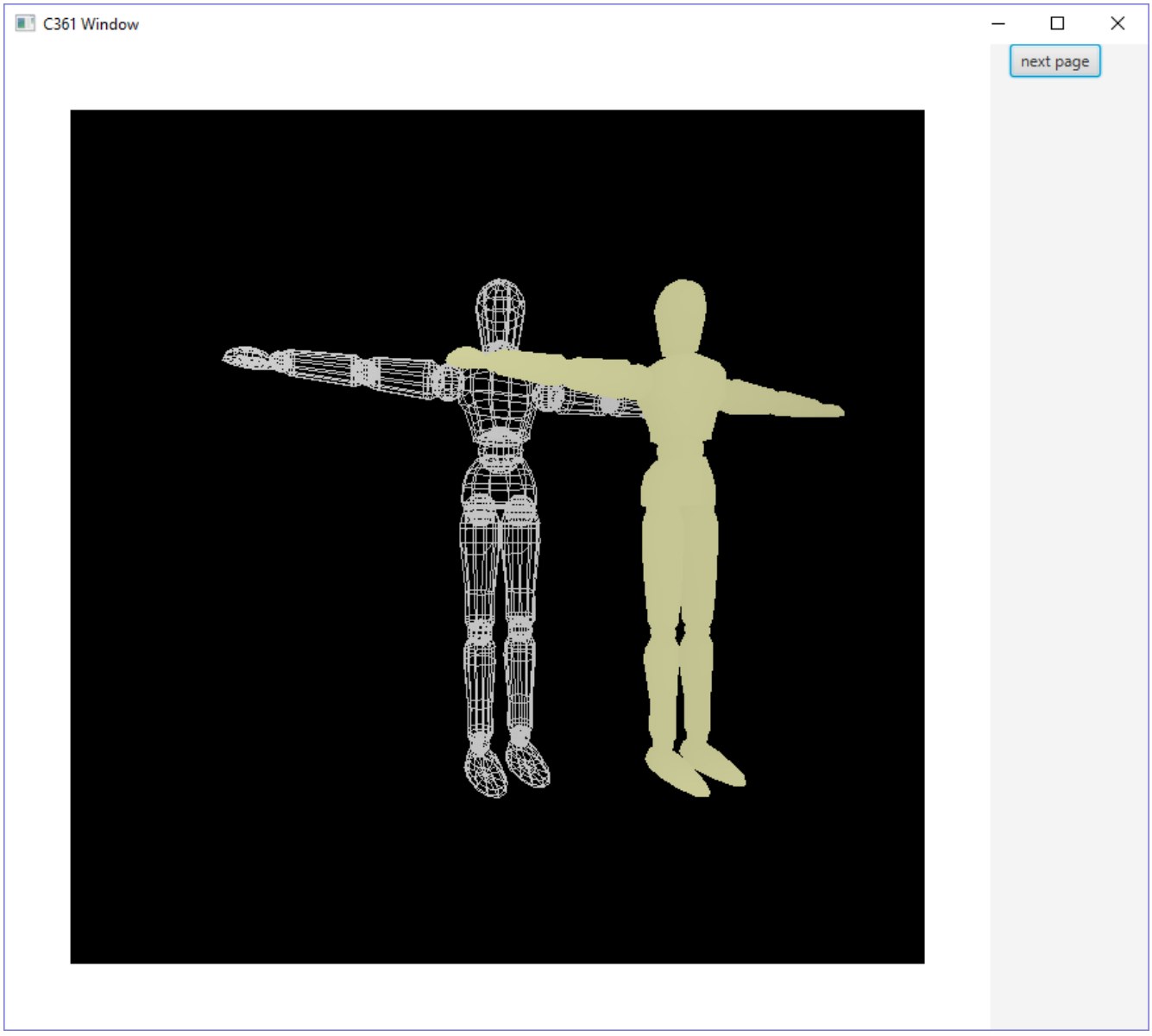
pageF



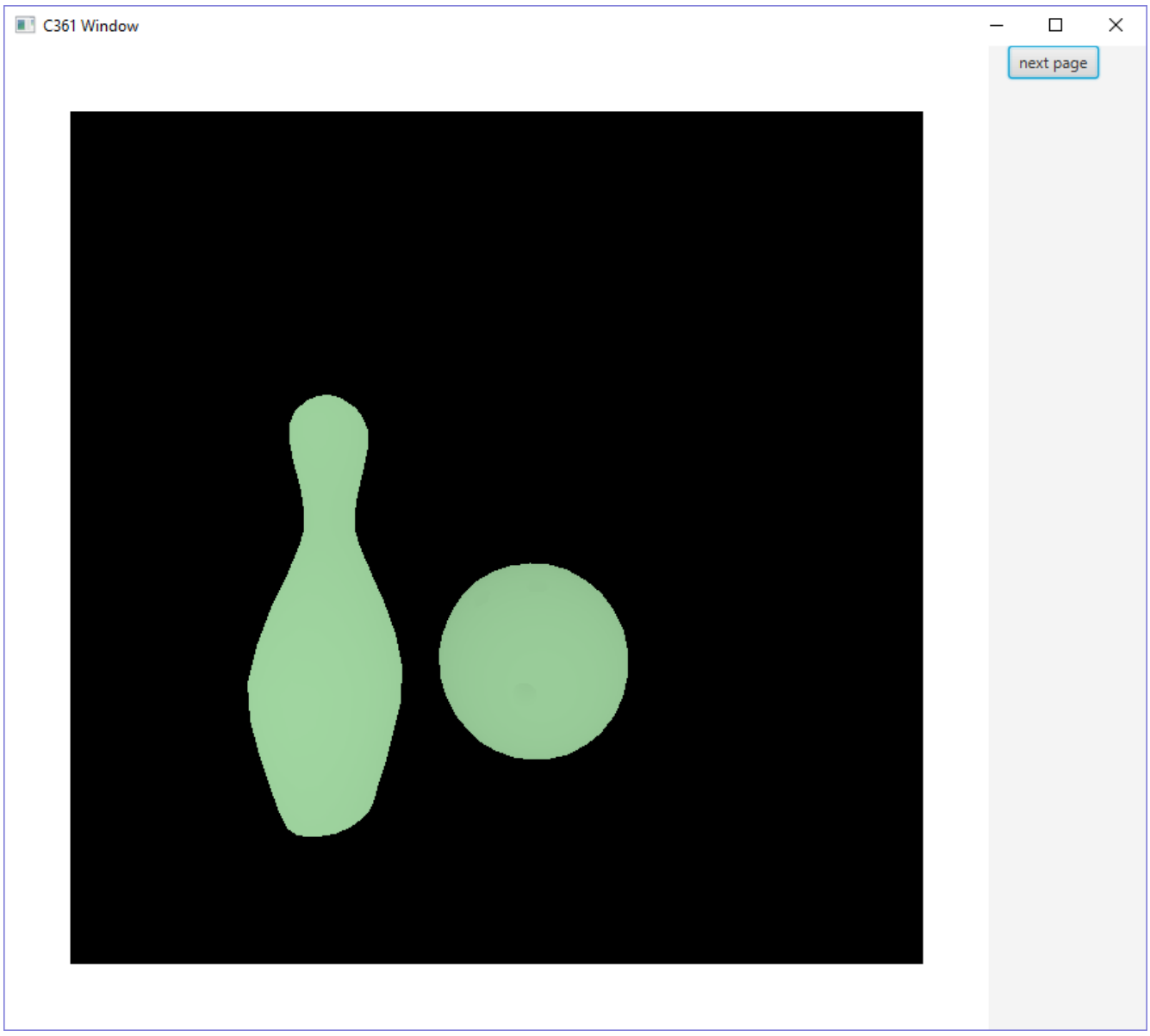
pageG



pageH



pageI



pageJ