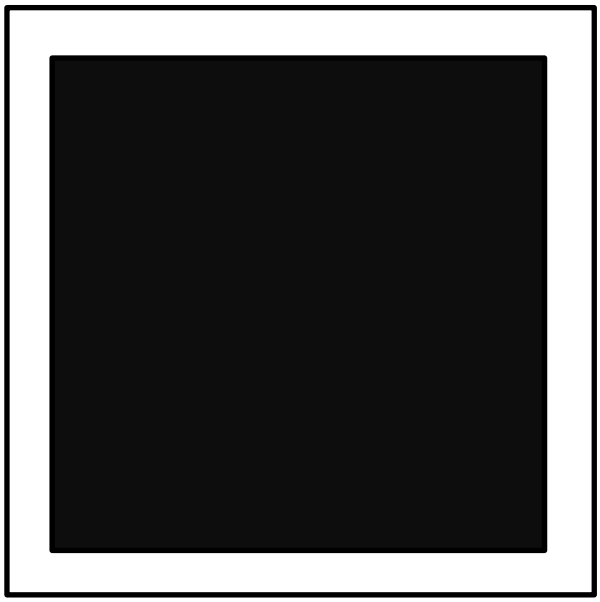
SFU CMPT 361
Computer Graphics
Assignment 2

For this assignment, you are to interpret a 3D graphics specification language, along with doing some preparatory work. The preparatory work is (1) to incorporate lerping and blerping into your line and polygon renderers, and (2) to implement z-buffering. The graphics interpretation will involve manipulating and transforming polygons and lines, and rendering using depth cueing.

These rendering components should be shown in a program that displays eight pages. Each of the required pages displays 1 panel, as shown in the following diagram:



The drawing area should be 650 x 650 pixels, the the white margins are all 50 pixels wide or tall. Thus, the total display area is 750 x 750 pixels. Use white and black for the backgrounds, as shown above.

You will be using and modifying your line drawing and polygon filling from assignment 1. You only need one line drawer for this assignment; I recommend using the DDA.

Color Interpolation
Modify your line drawer to linearly interpolate (lerp) RGB colors between its endpoints. Assume that each endpoint of a line is given to you with a color attached. For each component (R, G, and B), you must interpolate that component.

Interpolating a color component is exactly like interpolating the y-value of the line when working in the first octant. You get the (say) red component r_1 for p_1, and the red component r_2 for p_2, and compute $dr/dx = (r_2-r_1) / (x_2-x_1)$. Start the line with $r = r_1$ and $x = x_1$, and each time you advance one unit in x, you add dr/dx to r (and round). Do the same for g and b, and when writing the pixel with the current x and y, use the current r, g, and b. You should get a smooth gradation of color along a line.

Similarly, modify your polygon renderer to do bi-linear interpolation (blerping) of RGB colors between its vertices.

For **pages 1 and 2** of your assignment, use the same setup as the perturbed mesh test from assignment 1 (page 4, third panel). Almost fill the (650 x 650) drawing area, not the smaller panel from assignment 1. On **page 1**, draw the mesh in *wireframe*: draw the edges (lines) of the mesh, not the filled polygons. On **page 2**, draw the mesh as solid: don't use any lines, just fill in the polygons. On each page, assign a random color to each vertex of the mesh. These pages are for showing me that your color interpolation works.

To implement wireframe rendering, I recommend that you make a class that satisfies the same interface as the polygon renderer, but that class simply calls a line renderer (which one could be a constructor argument) to render the edges of a polygon.

Z-buffering

You should create a z-buffering drawable (decorator) that allocates an array (called a z-buffer) that is the size of its delegate drawable. Look up how to create a dynamic-memory 2D array in java if you've never done it before. Set every entry in the entire array to -200 (which will be our back clipping plane), or to the maximum negative value that a double can hold, at the start of each page. Whenever you are about to write a pixel (x, y) with world-space coordinate z, first compare z to the zBuffer value at (x, y). If z > zBuffer(x, y) then go ahead a write the pixel, and write the z value to the zBuffer. Otherwise, do not write the pixel.

For **page 3** of your assignment, I want you to draw 6 triangles (filled). Each triangle should be given a color of (v, v, v) for v=1, .85, .7, .55, .4, .25, respectively, for triangles 1 through 6. Each triangle should be regular (all three sides the same length), with vertices on the circle centered at the center of you drawing area, with radius 275. Give each triangle a random rotation about that center, say, in the interval from 0 to 120 degrees. Also give each triangle a random z-coordinate chosen between -1 and -199 inclusive. (Either floating random numbers or integer random numbers is okay.) Render the triangles, in order of the v used for the color, using z-buffering.

Graphics file reading

You are required to be able to read in a file in "simp" format, which is as follows. Simp format files always have a ".simp" extension. Note that you are given an interpreter for (a superset of) this simp format; all you have to do is fill in the actions to take when each command is seen. **Do not** write your own interpreter.

Whitespace means spaces, tabs, carriage returns, and newlines.

A line containing whitespace only is ignored.

A line starting with "#" (before any whitespace or other characters) is a comment and is also ignored.

The only other valid lines are ones containing one command from the commands shown in the following table.
Any of these lines may start with an arbitrary number of whitespace characters.

All numbers are double precision.
Anything in angle brackets or double-quotes in the table is called a parameter. All parameters that are points (denoted with <pN> below, where N is a number) are of the format "(x, y, z)" where each of x, y, and z is a number. To simplify interpretation, each comma must immediately follow the number before it (no spaces or tabs inbetween), and a space after each comma is mandatory. Adjacent parameters must be separated by tabs and/or spaces (and no commas).

	Line format	Meaning
transformations	{	save the current transformation on a stack (a.k.a. push, gsave)
	}	restore the current transformation from the stack (a.k.a. pop, grestore)
	scale <sx> <sy> <sz>	sx, sy, and sz are fl scale factors in axis directions. *Example:* scale 2.1, 1, -1.7
	rotate <axis> <angle>	Axis is X, Y, or Z. Angle is in degrees (counterclockwise when looking at the origin in the direction opposite the indicated axis). This is known as right-handed rotation. *Example:* rotate X 45
	translate <tx> <ty> <tz>	tx, ty, and tz are translation amounts in axis directions. *Example:* translate 1, 2.2, 3.4
primitives	line <p1> <p2>	A line from p1 to p2 in the current coordinate system. *Example:* line (2.2, 3.3, 4.4) (5.5, 6.6, 7.7)
	polygon <p1> <p2> <p3>	A triangle with vertices p1, p2, and p3 in the current coordinate system. *Example:* polygon (2, 3.3, 4) (19, 4.1, 1.7) (-5.5, -6.6, 8.2)
directives	file "filename"	Read in the simp file **filename.simp** from the current directory, and treat it as being part of this input file. *Example:* file "cube"
	wire	Every polygon from this point forward is to be rendered in wireframe. This is not part of the information saved and restored by **{** and **}**.
	filled	Every polygon from this point forward is to be rendered as filled polygons. This is not part of the information saved and restored by **{** and **}**.

The language **is** case-sensitive. Use "rotate Z 45" and not "ROTATE Z 45" or "rotate z 45". Also, there is only one "command" per line. Only whitespace is allowed on a line after the command and arguments (if any). (There are no trailing comments).

Please indent lines between { and }, as you would in C++ or java.

The first section of lines/commands are things that affect the current coordinate system or current transformation matrix (CTM). The second section are our *primitives*: the shapes that our renderer should know about. The third section are *renderer directives*, telling the renderer to open another file, or how to render the primitives that we have specified.

If you encounter any malformed file, you may abort your program or allow it to misbehave however it likes. We will not be testing behaviour on malformed files.

When you get a primitive, you should transform it from the current coordinate system to the world coordinate system by multiplying the vertices by the CTM.

The CTM starts as the identity matrix; this represents world space. The rendering style starts as "filled."

We will be viewing only a small part of the world (world space). The panel we are using should map to the interval -100 to 100 in both X and Y in world space. We are looking down the z-axis, and we will use parallel projection to view objects.

This means that to convert from world space to window space, we simply forget about the Z coordinate. To find screen space from the window space, scale by the appropriate amount (to convert 200 to 650) and translate it so that the origin is in the middle of the screen.

The range of Z coordinates that we are interested in are from 0 to -200. Anything with positive Z or Z less than -200 should not be seen. The coordinate system is right-handed.

If a polygon or a line is completely outside any of the six planes defining the viewing volume, then cull it (do not draw it). Otherwise (it is partially or fully inside, or it is outside the viewing volume but not outside any one plane), you may use **pixel clipping** (scissor testing) to keep the image in the specified volume. Do not use pixel clipping on primitives that are entirely outside the volume.

We will be using *depth cueing* or *depth shading* to render our primitives from pages 3 on. In this technique, we give an object a color that indicates how deep it is into the scene. Typically, the back (or *far*) clipping plane (in our setup, Z=-200) is given the color black and the front (or *near*) clipping plane (Z=0) is given white or some other bright color. Color is linearly varied through the Z range. For your renderer, when you find the world-space Z coordinate of a vertex, then that point/vertex gets the corresponding color. (For instance, assume your near clipping plane color is white (1, 1, 1). If a point is at (32, 49, -150) it would get a color of (.25, .25, .25). [-150 is a quarter of the way from -200 (black) to 0 (white)]. Send this color along with your vertex to your renderer, and have your renderer interpolate the colors between its vertices or endpoints. Note that you can create a depth-cueing drawable decorator that substitutes in the depth color.

To create simp files, you may hand-edit them and/or you may use programs (that you have written) to create them. You are provided with a file *cube.simp* that contains a simple unit cube (one having every coordinate on every vertex either 0 or 1).

Page 4. In simp format, make a scene that contains several scaled boxes with some boxes (or parts of boxes) as far back as Z=-200, and some as close as Z=0. Render with depth cueing, from full green (0, 1, 0) for the near color to full black (0, 0, 0) for the far color. Show rotations, translations, and scales of boxes. Render filled polygons.

Page 5. In simp format, and using at least one *file* command, make a scene that shows off the technical quality of your work. Use your own colors (or white) for the near color of the depth cueing. Use interesting shapes, including at least two different types from: spheres, cylinders, mountains, circular cones, pyramids, block letters.

Pages 6 to 8. Read and display the following files from the current directory, with the black-to-white depth cueing and setup as above: **page6.simp, page7.simp,** and **page8.simp.** These files have been included in the assignment zip file.

Implementation notes

To perform the interpretation of a file, you will need the following objects[1]:

1. A boolean or enum indicating whether you are rendering wireframe or filled polygons.
   Initially, this should be set to "filled".

2. A 4 x 4 matrix CTM, the Current Transformation Matrix.
   Initially, this should be set to the identity matrix, which means world space.

3. A stack of 4 x 4 matrices.
   Initially, the stack is empty.

[1] You can alternatively keep (2), the CTM, on the top of (3), the matrix stack. Similarly, you could keep (4), the current file-reading object, on top of the stack (5). You may also keep the stack (5) implicitly as the call stack by making recursive calls when you see a file command. However, do not do this for the matrix stack (3); you must maintain an explict stack for this. Because I said so.

4. A current file-reading object (ifstream, or whatever) to read commands from.
   Initially, set this to the file you are supposed to read.

5. A stack of file-reading objects
   Initially, this is empty.

6. A z-buffer attached to your (650 x 650) drawing area (in a DrawableDecorator).
   Initially, set all values to 200.

7. A line and a polygon renderer, and a wireframe polygon renderer, all with lerping or blerping.

Each command has a simple interpretation in terms of these objects.

{ means "push the CTM onto the matrix stack."
} means "pop the top of the matrix stack, overwriting/replacing the CTM with the popped matrix.

(any transformation)
means "multiply the CTM by the matrix for this transformation."

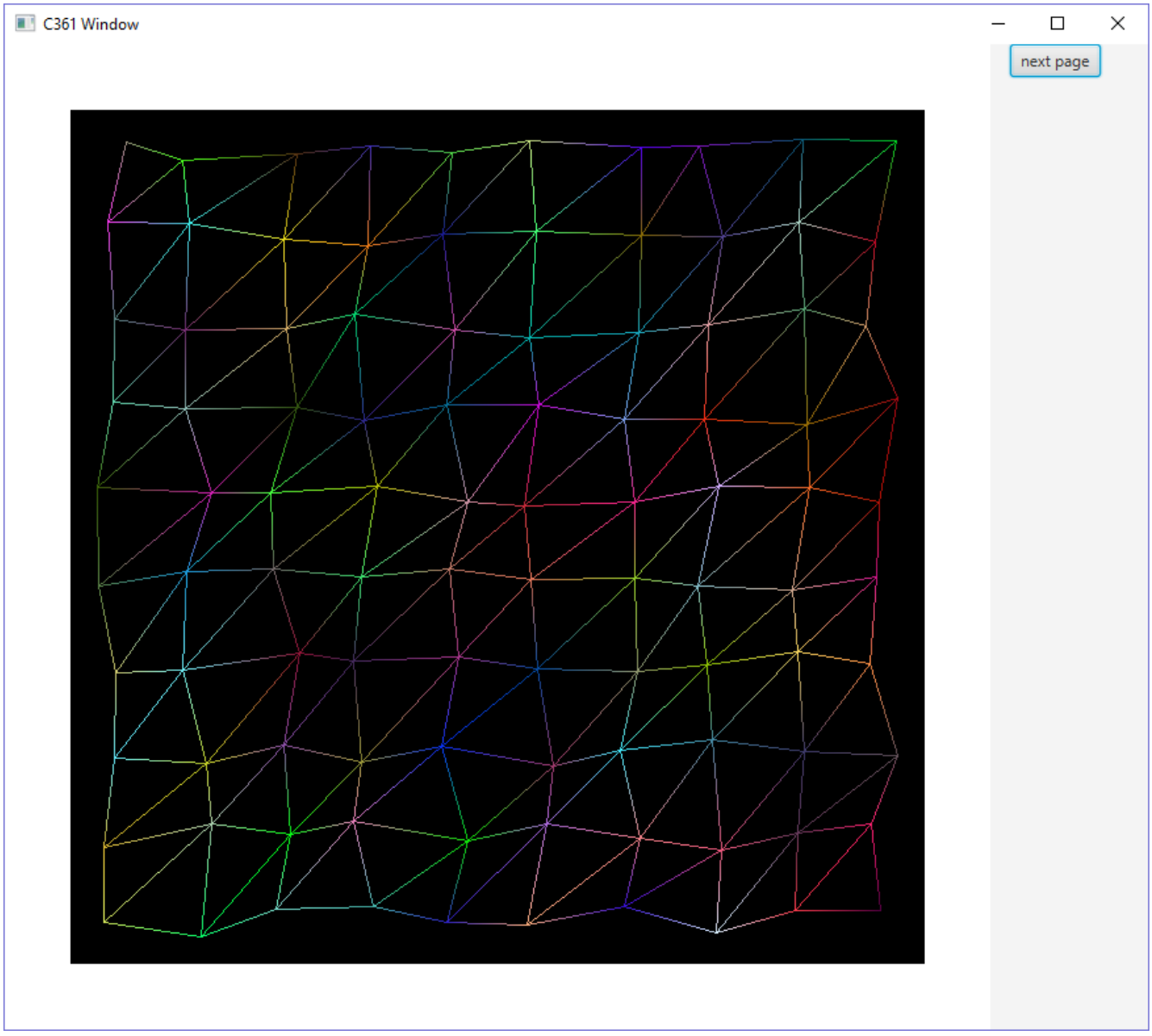
line <p1> <p2>
means "transform p1 and p2 by the CTM and render the line between them using the line-renderer with the z-buffer."

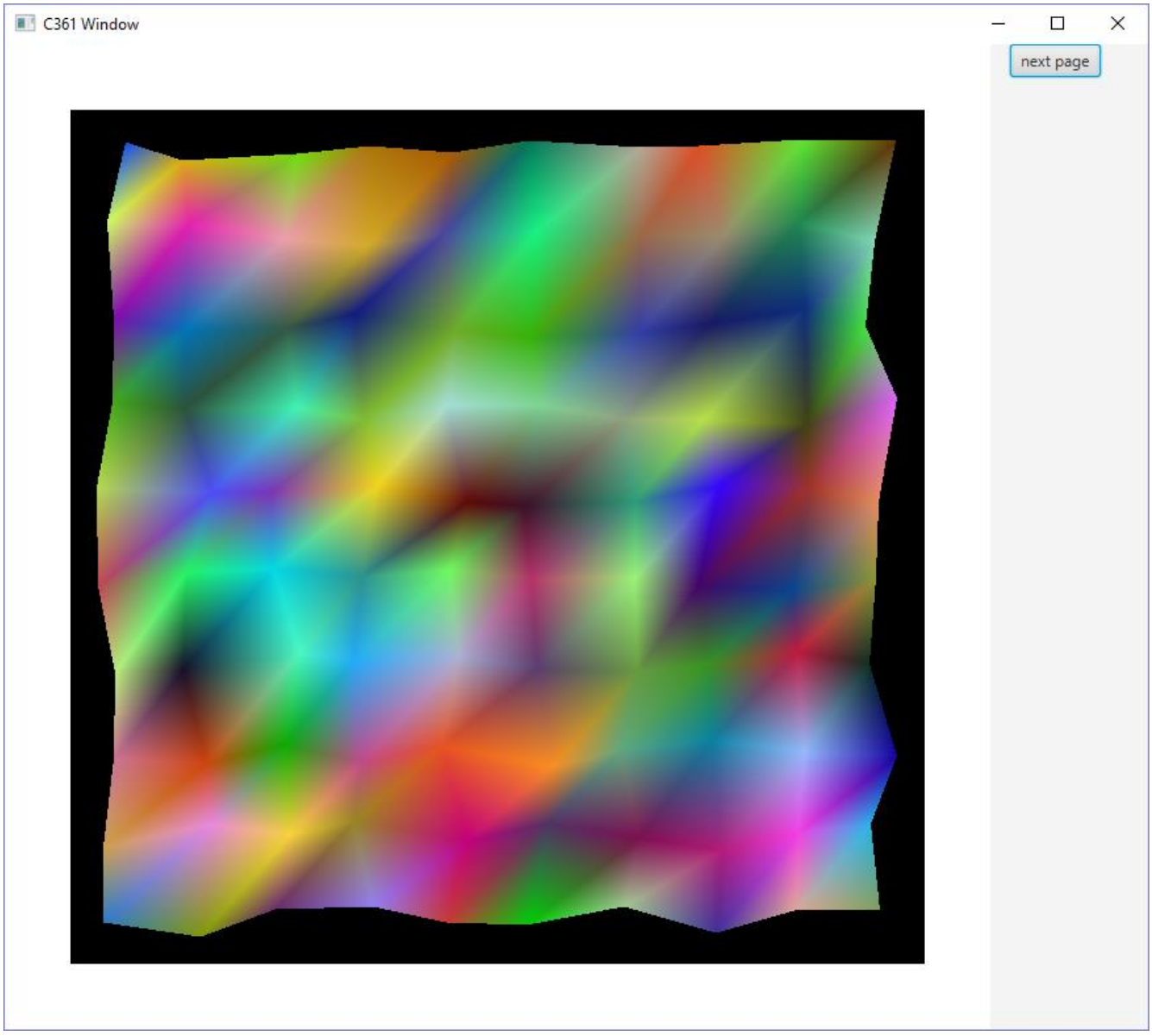
polygon <p1> <p2> <p3>
means "transform p1, p2, and p3 by the CTM and render the polygon defined by them using the polygon renderer with the z-buffer." But check the boolean to determine if you need to render it with the wireframe polygon renderer or the filled polygon renderer.

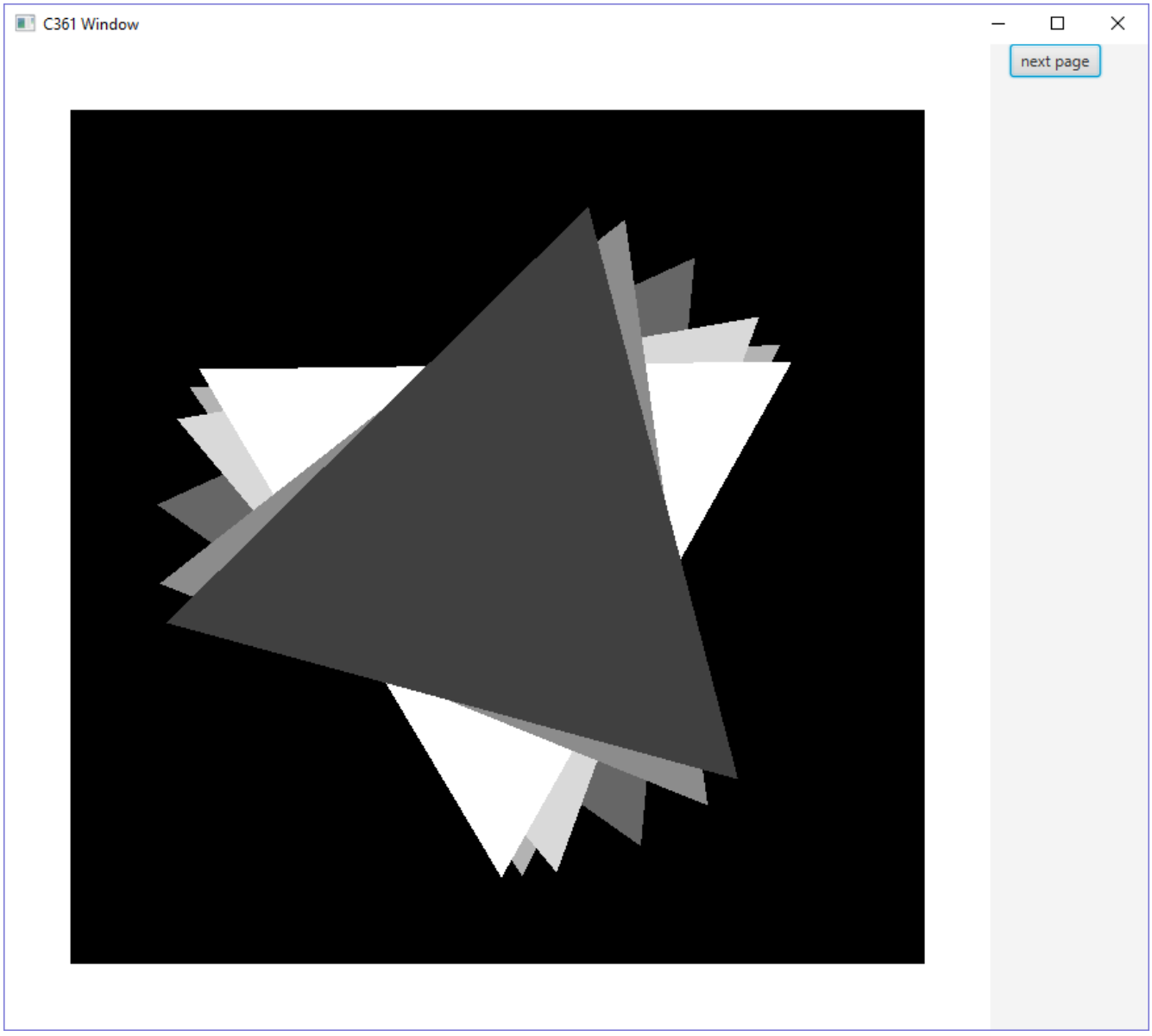
file "filename"
means to push the current file-reading object on top of the stack of file-reading objects, and open a new file-reading object for filename.simp in the current directory. When the end of a file is reached, close the current file-reading and overwrite/replace it with one popped from the stack of file-reading objects.

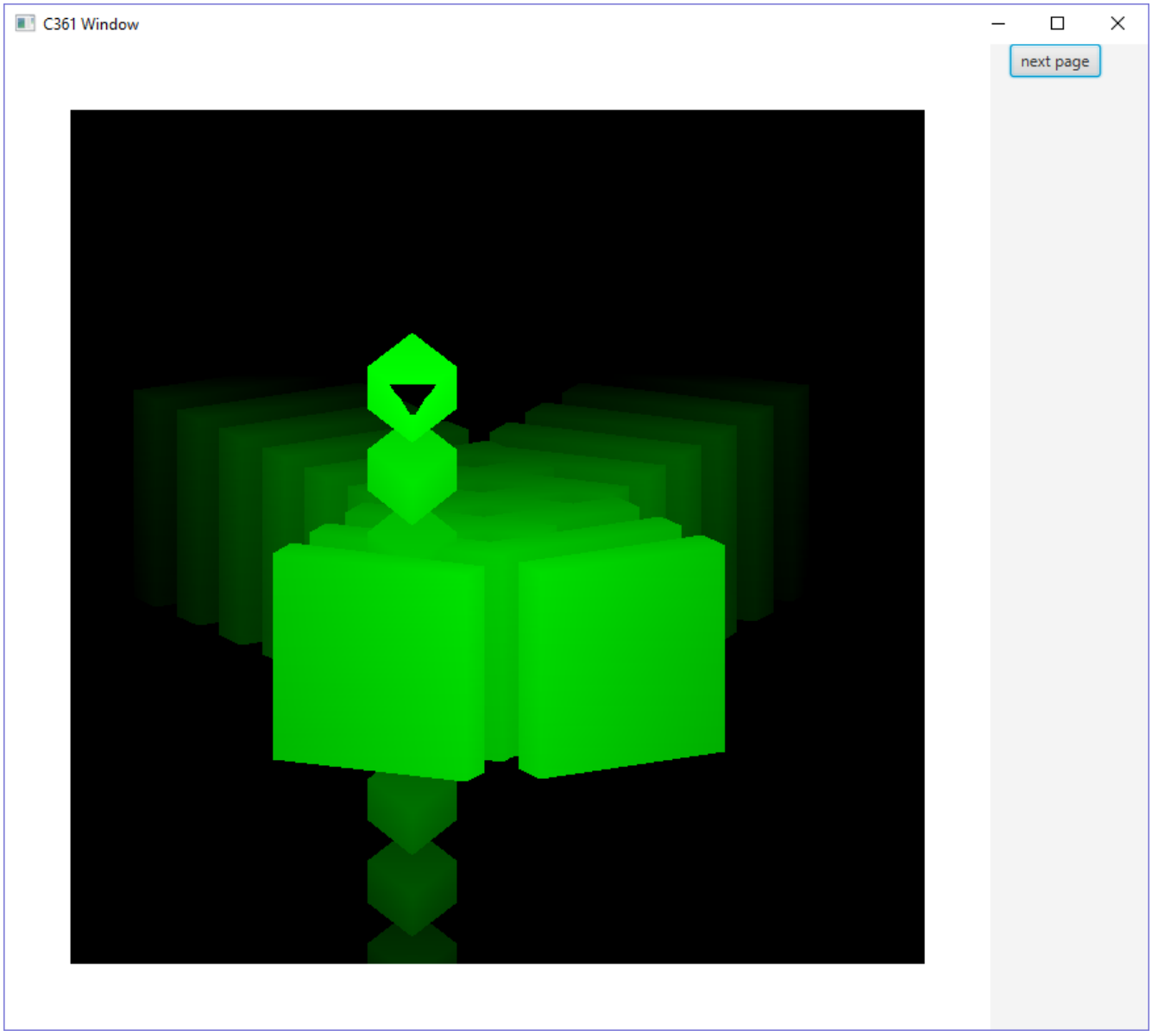
wire or **filled**
Sets the boolean (1) to the corresponding value.

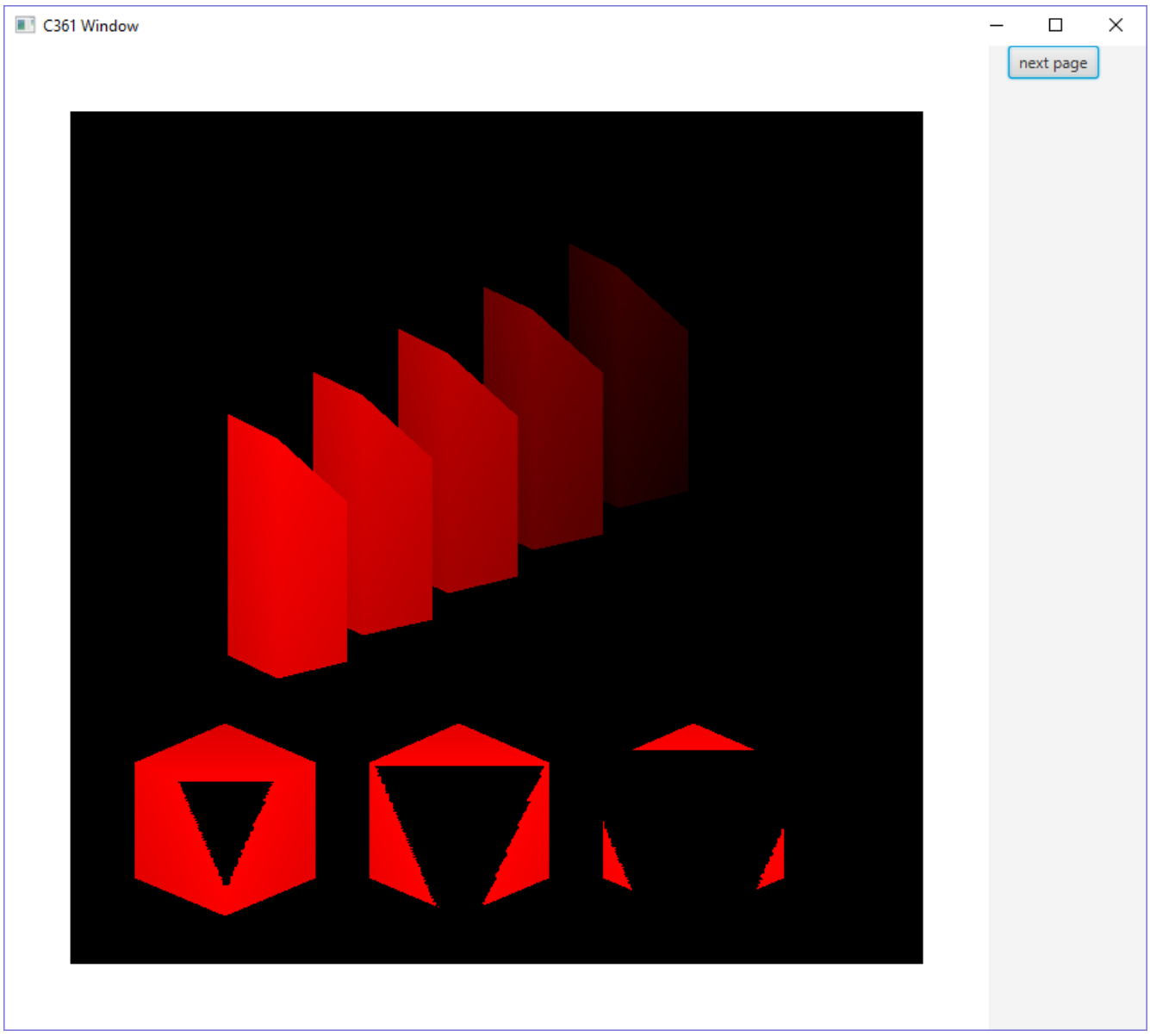
Appearance of completed assignment: the following images show approximately what the pages of your output should be. Random perturbations and colors will differ. Pages 4 and 5 must differ significantly; make your own page files and do not use tomsPage4 and tomsPage5 as your model.

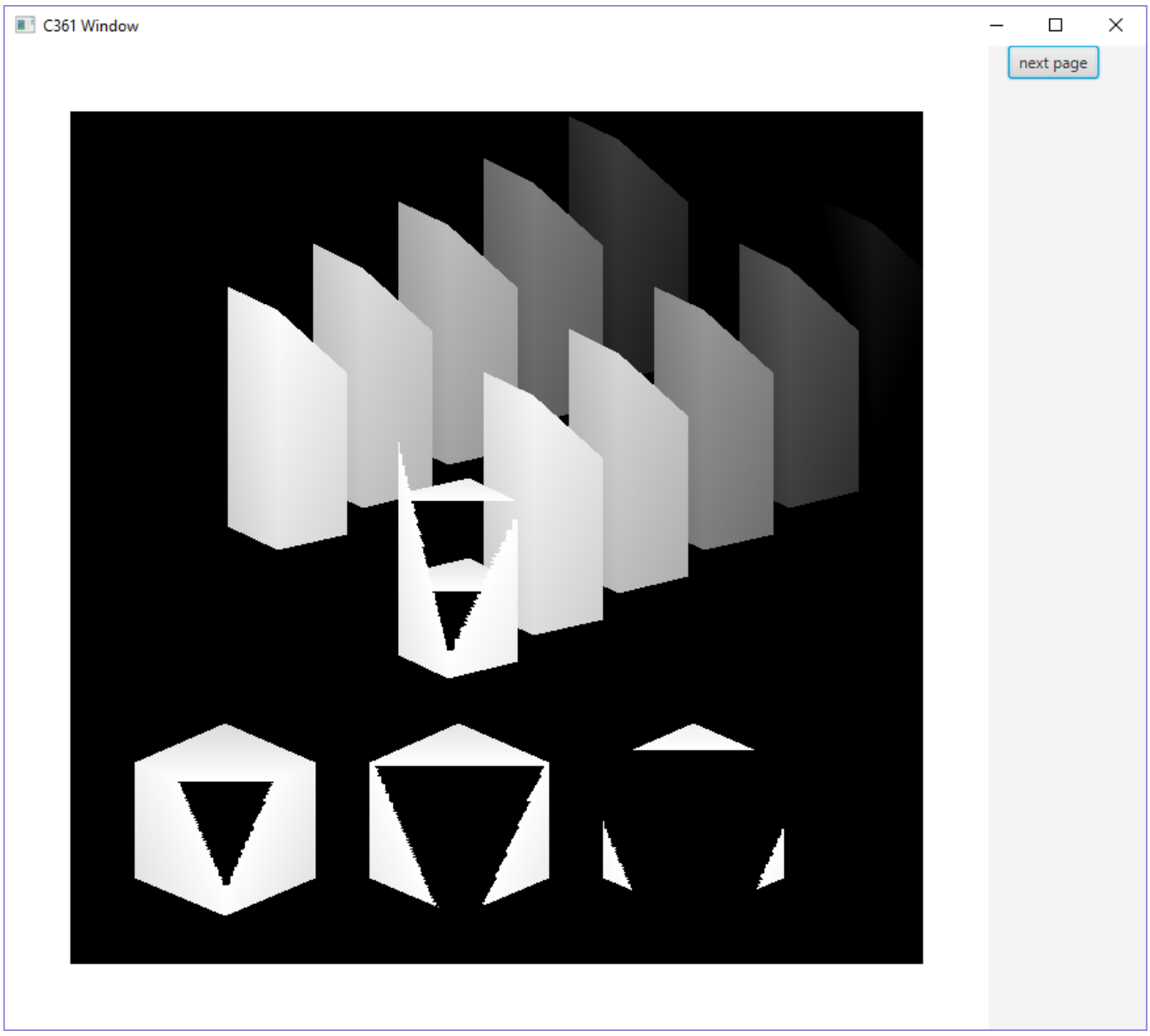

C361 Window
next page

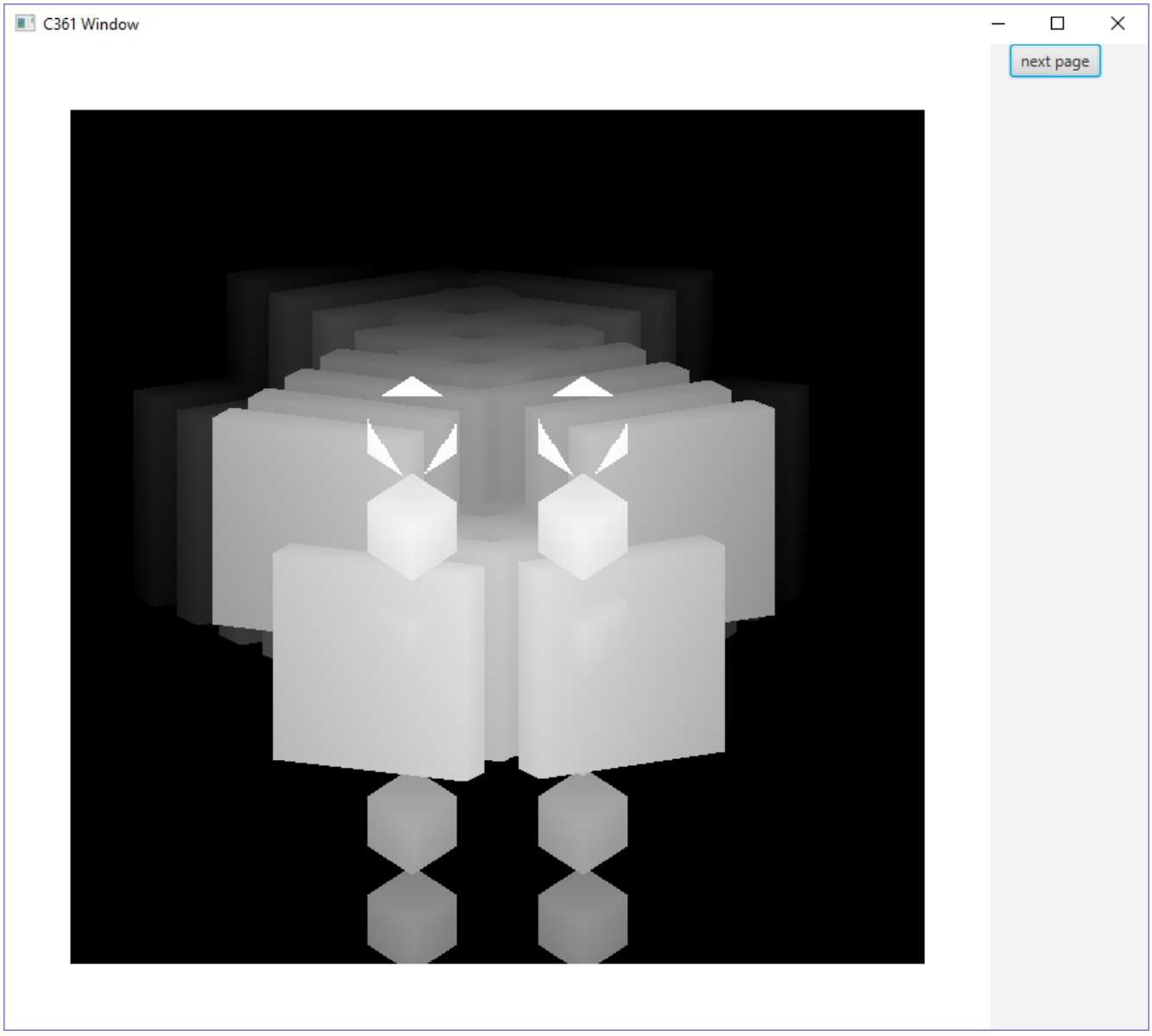

C361 Window
next page

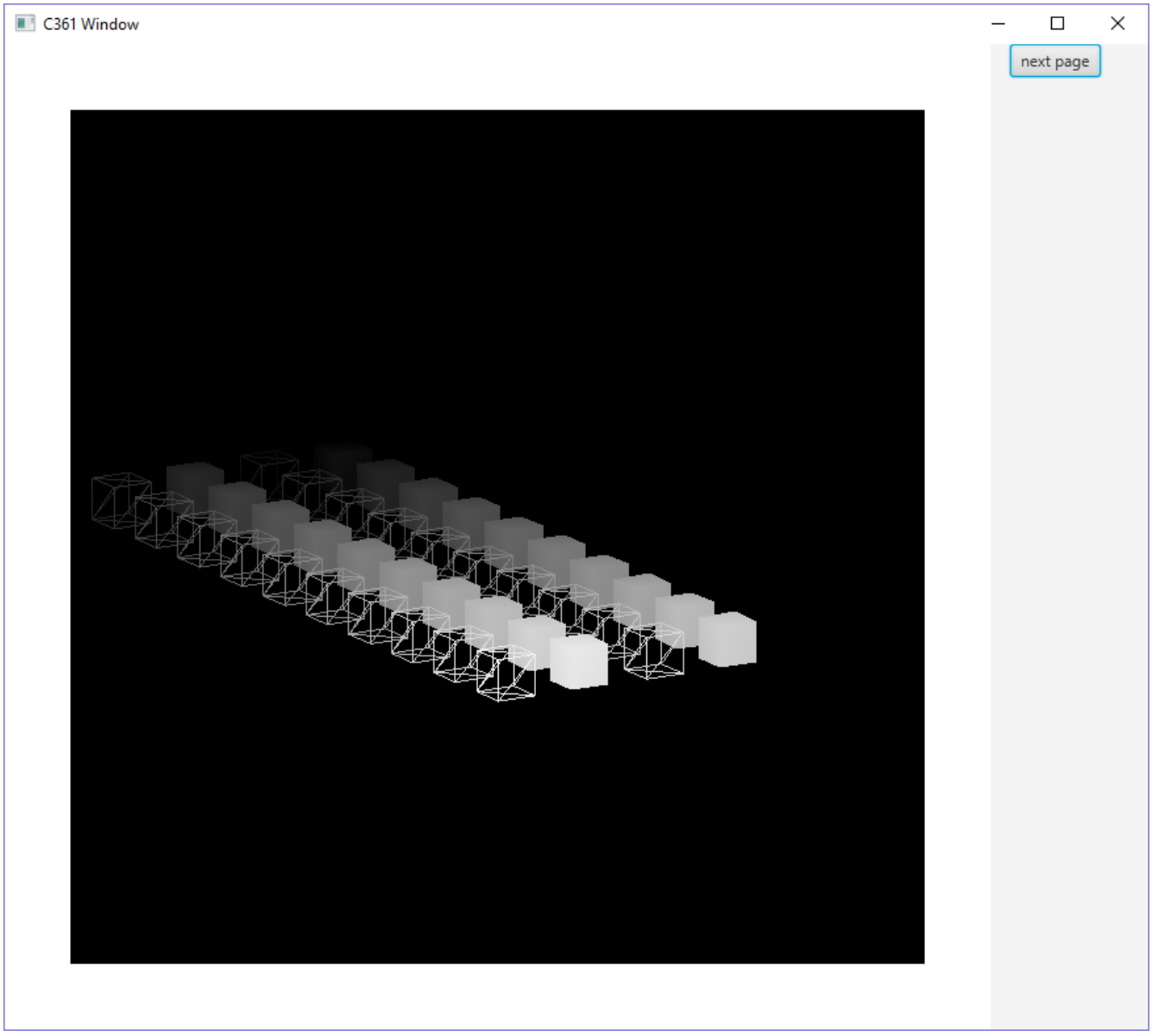

C361 Window
next page


C361 Window
next page


C361 Window
next page


C361 Window
next page


C361 Window
next page


C361 Window
next page