CSCI 444/544 - Lab 7
Friday, March 19, 2021
LAB IS DUE BY **Friday April 09 11:59 PM**!!

Step 0 – Run a POV-Ray Program

This lab package contains three files:

1. `lab7.pdf` – this writeup
2. `lab7-template.pov` – the starter code for this very simple POV-Ray program
3. `povconsole32.exe` – the actual POV-Ray program compiled and built to run on the lab machines (and potentially other Windows machines)

For the lab machines, you'll want the exe file in the same folder as the POV file for your image.

Part I – Generate a POV File

To begin, make a capy of the `lab7-template.pov` file and rename it to be `lab7.pov`. A POV file is a series of instructions that describe the geometry in our scene as well as settings and conditions to render the scene under. It is a text file that any text editor can work with.

The POV language has the ability to define macros and has some concepts we are familiar with (variables, conditionals, loops, arrays).

Part II – Run POV-Ray

To run a POV-Ray program, in a terminal you can type the following command to run POV-Ray from the command line:

```
povconsole32.exe lab7.pov
```

or on your own machine if you set up POV-Ray

```
povray lab7.pov
```

Running the program will generate a png file with the same basename as the POV file you passed as an argument. Open the file, `lab7.png` and check out what the template gives you. A black screen! Let's put some objects into our scene and start the ray tracing fun.

Step 1 – Setup the world

Part I – Place a Camera

We'll begin by placing a camera in our scene so we have a viewpoint to render from – or more accurately, a source to shoot rays from.

The syntax to create a camera is

```
camera { angle a              // full field of view angle in degrees
         location <x, y, z>   // eye point in world space
         look_at <x, y, z>    // look at oint in world space
       }
```

We then have control over each of the 7 parameters. For our camera, which will be placed at TODO #01, we'll give it a field of view of **90** degrees. We'll then place the camera at **(2, 3, -3)** and have it look slightly above the origin at **(0, 1.5, 0)**. This will ultimately give us a nice angled view of our drawing so that we can see three sides.

Part II – Place 2 Lights

Next, in order to be able to see anything in our scene we need for it to be lit by placing lights in our scene. We'll use point lights and we'll create two so that we can see the shadows cast by each of them.

The syntax to create a light is

```
light_source { <x, y, z>     // light position in world space
               rgb<r, g, b>  // light color specified as RGB
             }
```

We now have 6 parameters to vary to change our lights. These two lights will go at TODO #02 and #03. Both lights will be light whites (RGB = **(1,1,1)**).

The first light will be at position **(1500, 2500, -2500)**. This corresponds to being behind the camera.

The second light will be at position **(-1000, 800, 3000)**.

Part III – Draw the ground

We're now ready to draw our first object and be able to see something! At TODO #04, we'll draw a plane that will represent the ground.

The syntax to create a plane is

```
plane { <xn, yn, zn>,          // surface normal ending with a comma
        d                      // distance along normal vector
                               // oriented from origin to plane
      }
```

The surface normal describes the orientation of the plane and the distance *d* describes the distance from the origin to the plane by traveling along the surface normal.

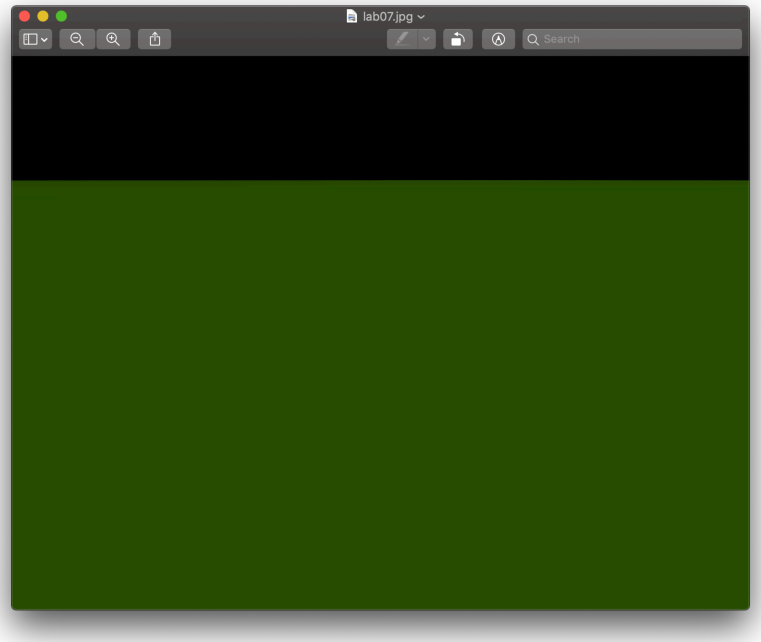
The ground plane will lie in the XZ-plane, so needs to have a surface normal of **`(0,1,0)`**. We want this plane to pass through the origin, so the distance will be **0**.

We can now render our scene and look at the plane!

Hmm, it's all black. Ah that's right. We need to give our plane a color! We do this by adding inside the `plane` body (denoted by the braces `{}`) what the pigment color is. We'll set it to be a greenish color. Your ground plane block should be as follows now

```
plane { <0,1,0>, 0              // set the plane properties
        pigment { color rgb<0.25, 0.47, 0.00> }   // set the color
}
```

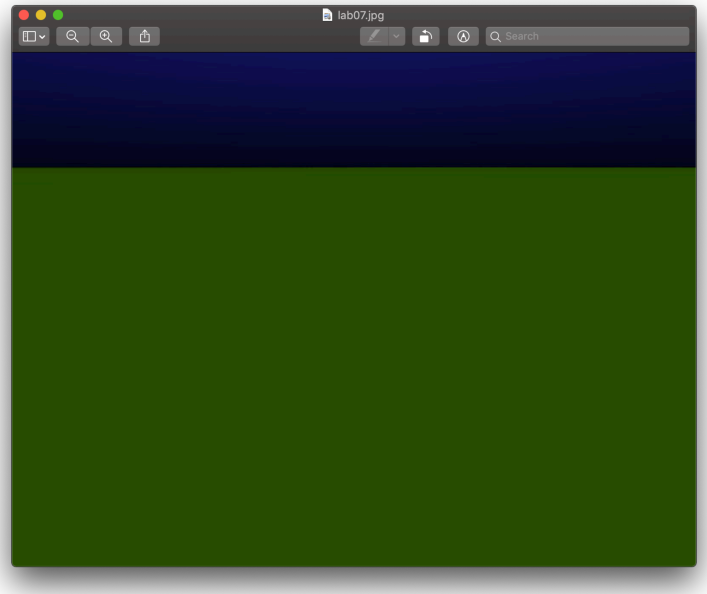
Now rerun the program and take a look at your ground. You should see the following:



Part IV – Draw the sky

Now, we'll turn our dark in to light. Add a second plane at TODO #05. This time the surface normal will be pointing downward **`(0,-1,0)`** and the distance will be **`-10000`**. (Self-check: why is the distance negative?) Finally, set the color to be a nice shade of blue **`(0.2, 0.2, 1.0)`**.

Run the program once more and gaze into the heavens.



Step 2 – Make the M

Let's begin to draw our object in the sky. This object will be our representation of the Mines M.

Part I – Draw an M Prism

We'll make up our M using three objects – two boxes for the legs and a prism for the middle V part.

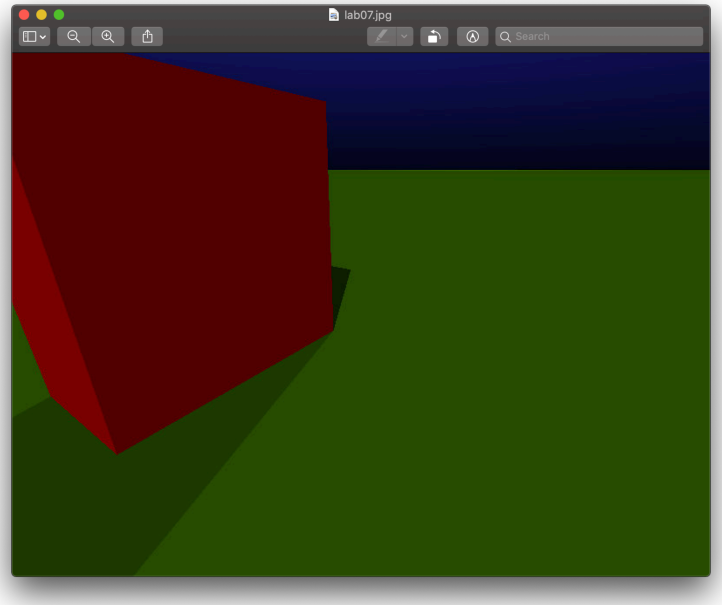
We'll begin at TODO #06 with our first box. The syntax for a box is

```
box { <x1, y1, z1>, // point 1 representing corner
      <x2, y2, z2>  // point 2 representing corner along opposite diagonal
      //-- optional transform information
      scale <sx, sy, sz>      // optional scale information
      rotate <rx, ry, rz>     // optional angles in degrees
                              //   to rotate around each axis
      translate <tx, ty, tz>  // optional translation information
     //-- optional material information
     texture {
             //-- texture properties
             }
}
```

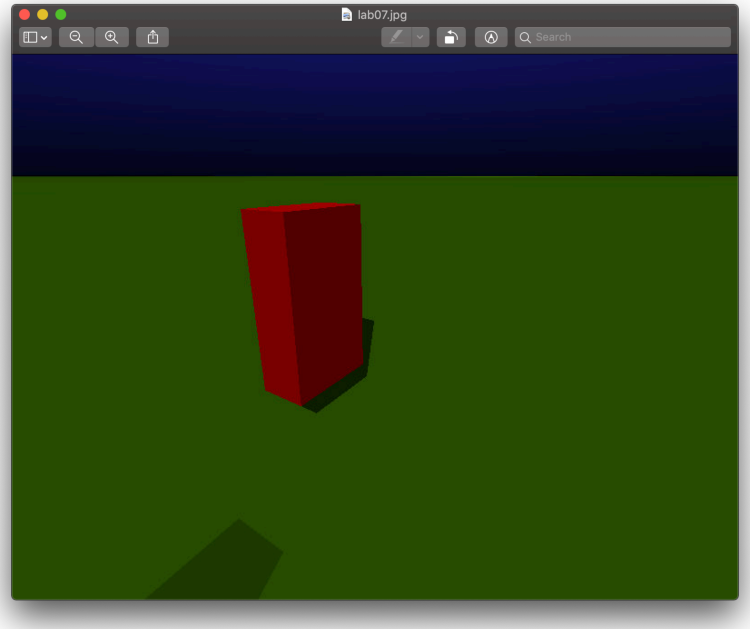
We specify a box by its outside corners. These correspond to the (x,y,z) min and (x,y,z) max points. We can then optionally apply any transformation properties and apply a material as well.

For our first box, it will range from the point (-4, -3, -2) to (-2, 4, 2). We won't add any transformation properties (yet). The texture properties we will set as we did with the plane, and make it red (1,0,0).

Run the program, check your image, and you should see a red block floating in the air with two shadows being cast.

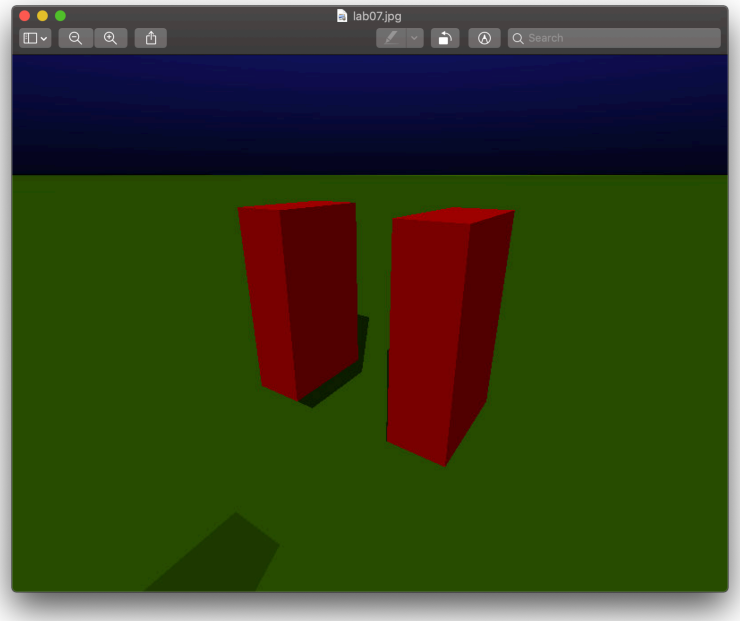


That's pretty big. Let's first scale it to be smaller by 0.3 along every dimension. Then translate the block up 1.5 along Y (and nothing along X or Z).



A more manageable block.

Add a second block, this time ranging from (2, -3, -2) to (4, 4, 2). Apply the same transformation and coloring properties.



Voila! Two blocks.

Now to connect the towers. We will use a prism to make the custom V shape. A Prism is described as a cross-sectional polygon in the XY-plane along with the range to sweep through Z. The syntax looks like

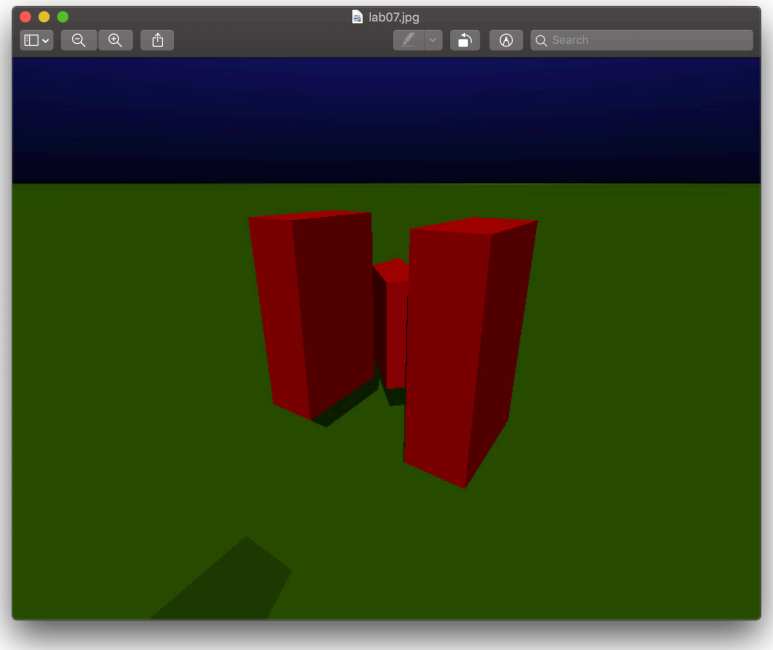
```
prism { zMin, zMax,       // z-min and z-max to sweep through
                          //  end with comma
        n                 // number of points that make up a closed loop
                          //   describing the polygon.  Polygon needs to
                          //   be simple but not necessarily convex.
        <x1, y1>,         // comma separated list of (x,y) coordinates
        <x2, y2>,
        ...
        <xn, yn>          // the last point == the first point
        //-- optional transformation information
        //-- optional material properties
      }
```

We'll make a prism now. The prism will sweep from -2 to 2 and will be made up of 7 points. The 7 points are

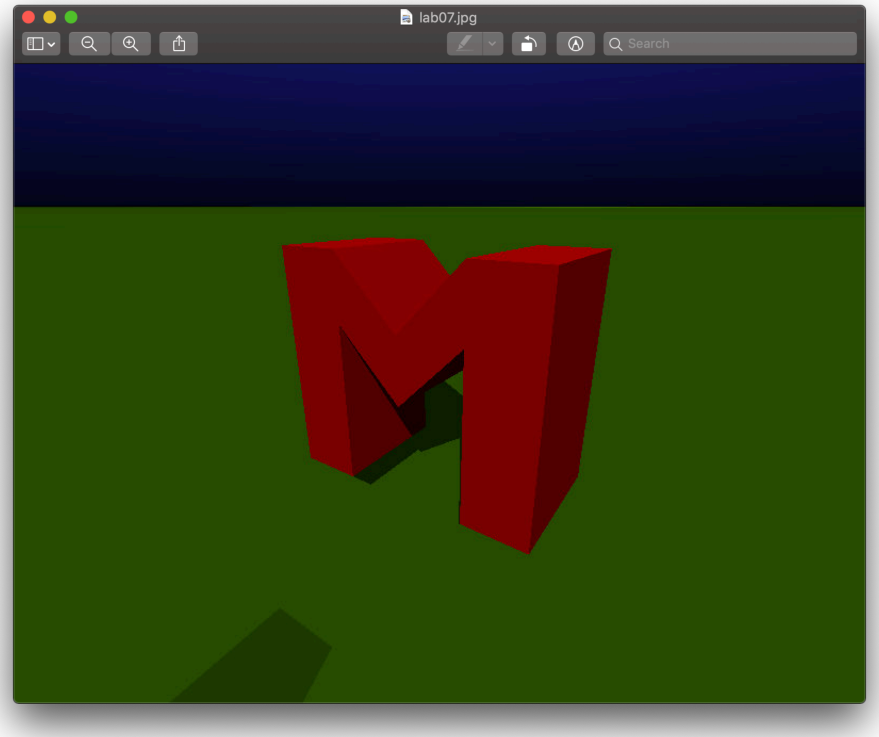
```
(-2, 2)
( 0, 0)
( 2, 2)
( 2, 4)
( 0, 2)
(-2, 4)
(-2, 2)
```

Additionally, apply the same transformations we had been before (scale then translate).

We can now run the program to check out the prism.



Hmm, it seems the prism description is actually sweeping through Y and a cross-section in the XZ-plane. No problem, let's transform the prism to match our intended use. Before doing the scale, let's first rotate our prism around the X-axis. Apply a rotation transformation with the angles (-90, 0, 0) to only rotate around X.



Hooray! An M.

Before going further, let's make an organizational change to our object. All of these individual components are building up a larger object and should therefore be treated as one object. We don't want to individually transform each component, we'd much rather transform the entire object as a whole unit. We'll follow principles of Constructive Solid Geometry to make this happen.

We want all the components to form one solid object – this corresponds to a union operation to join components together.

At TODO #07A, open a union block

```
union {
```

and then close it at TODO #07B

```
}
```

We can now cut the duplicated information from every block and paste it one time at TODO #07C. The three pieces of information we want are the `scale`, `translate`, and `texture` blocks (be sure to leave the `rotate` inside the `prism` block).

We should receive the same image as above.

While organizational, as we now add some trim along the edges and corners of our M, we would not want to duplicate all of the same information a total of 58 more times!

Part II – Add Some Ornamentation

We'll first add the trim to our M. This will be made up of two additional objects: spheres at the corners and cylinders along the edges. We place the spheres to hide the joints between cylinders.

We'll start by adding a cylinder along every edge. The syntax for a cylinder is

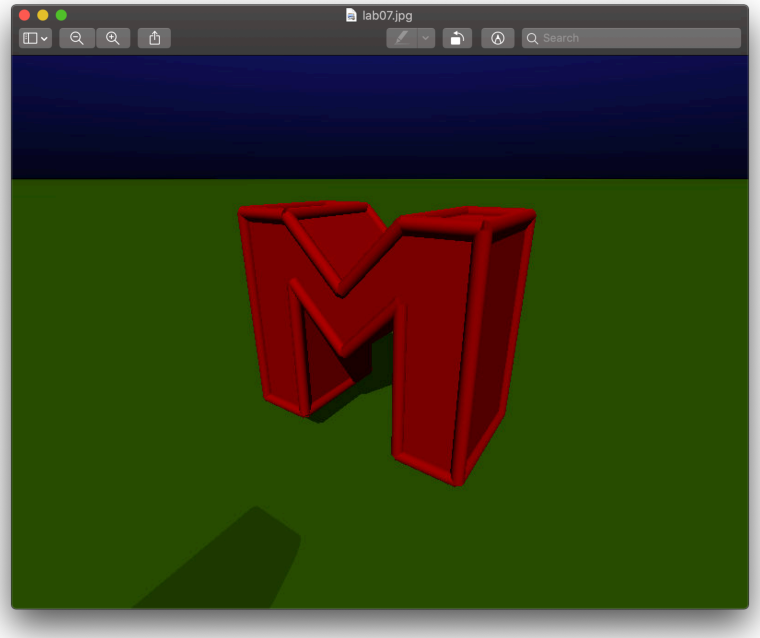
```
cylinder { <bx, by, bz>,      // bottom x,y,z point
           <tx, ty, tz>,      // top x,y,z point
           r                  // radius
         }
```

To save you some typing, here are 36 cylinders you would need at TODO #08:

```
cylinder { <-4,-3, 2>, <-4,4, 2>, 0.2 }
cylinder { <-4,-3,-2>, <-4,4,-2>, 0.2 }
cylinder { < 4,-3, 2>, < 4,4, 2>, 0.2 }
cylinder { < 4,-3,-2>, < 4,4,-2>, 0.2 }

cylinder { <-4,-3, 2>, <-2,-3, 2>, 0.2 }
cylinder { < 4,-3, 2>, < 2,-3, 2>, 0.2 }
cylinder { <-4,-3,-2>, <-2,-3,-2>, 0.2 }
cylinder { < 4,-3,-2>, < 2,-3,-2>, 0.2 }
cylinder { <-4, 4, 2>, <-2, 4, 2>, 0.2 }
cylinder { < 4, 4, 2>, < 2, 4, 2>, 0.2 }
cylinder { <-4, 4,-2>, <-2, 4,-2>, 0.2 }
```

```
cylinder { < 4, 4,-2>, < 2, 4,-2>, 0.2 }
cylinder { <-2,-3,-2>, <-2, 2,-2>, 0.2 }
cylinder { <-2,-3, 2>, <-2, 2, 2>, 0.2 }
cylinder { < 2,-3,-2>, < 2, 2,-2>, 0.2 }
cylinder { < 2,-3, 2>, < 2, 2, 2>, 0.2 }

cylinder { < 4, 4,-2>, < 4, 4, 2>, 0.2 }
cylinder { <-4, 4,-2>, <-4, 4, 2>, 0.2 }
cylinder { < 4,-3,-2>, < 4,-3, 2>, 0.2 }
cylinder { <-4,-3,-2>, <-4,-3, 2>, 0.2 }
cylinder { <-2,-3,-2>, <-2,-3, 2>, 0.2 }
cylinder { < 2,-3,-2>, < 2,-3, 2>, 0.2 }
cylinder { <-2, 4,-2>, <-2, 4, 2>, 0.2 }
cylinder { < 2, 4,-2>, < 2, 4, 2>, 0.2 }
cylinder { < 0, 0,-2>, < 0, 0, 2>, 0.2 }
cylinder { < 2, 2,-2>, < 2, 2, 2>, 0.2 }

cylinder { <-2, 2,-2>, < 0, 0,-2>, 0.2 }
cylinder { < 0, 0,-2>, < 2, 2,-2>, 0.2 }
cylinder { <-2, 4,-2>, < 0, 2,-2>, 0.2 }
cylinder { < 0, 2,-2>, < 2, 4,-2>, 0.2 }
cylinder { <-2, 2, 2>, < 0, 0, 2>, 0.2 }
cylinder { < 0, 0, 2>, < 2, 2, 2>, 0.2 }
cylinder { <-2, 4, 2>, < 0, 2, 2>, 0.2 }
cylinder { < 0, 2, 2>, < 2, 4, 2>, 0.2 }
```

Check it out!

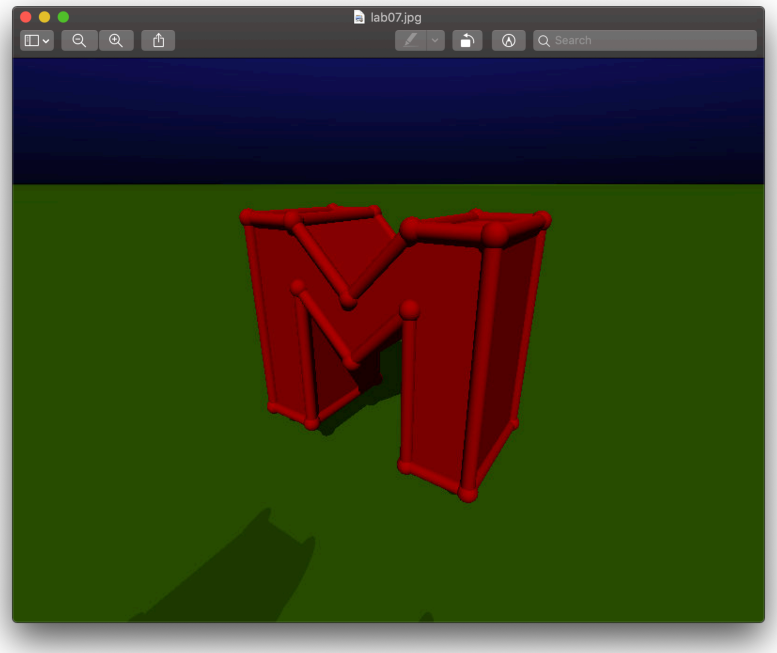


Notice the gaps at every corner? Let's cover those up with a sphere. Here are 22 of them :)

```
sphere { <-4,-3, 2>, 0.3 }
sphere { <-4,-3,-2>, 0.3 }
sphere { <-4, 4, 2>, 0.3 }
```

```
sphere { <-4, 4,-2>, 0.3 }
sphere { < 4,-3, 2>, 0.3 }
sphere { < 4,-3,-2>, 0.3 }
sphere { < 4, 4, 2>, 0.3 }
sphere { < 4, 4,-2>, 0.3 }
sphere { <-2, 2,-2>, 0.3 }
sphere { <-2,-3,-2>, 0.3 }
sphere { < 2,-3,-2>, 0.3 }
sphere { < 0, 0,-2>, 0.3 }
sphere { < 2, 2,-2>, 0.3 }
sphere { < 2, 4,-2>, 0.3 }
sphere { < 0, 2,-2>, 0.3 }
sphere { <-2, 4,-2>, 0.3 }
sphere { <-2, 2, 2>, 0.3 }
sphere { <-2,-3, 2>, 0.3 }
sphere { < 2,-3, 2>, 0.3 }
sphere { < 0, 0, 2>, 0.3 }
sphere { < 2, 2, 2>, 0.3 }
sphere { < 2, 4, 2>, 0.3 }
sphere { < 0, 2, 2>, 0.3 }
sphere { <-2, 4, 2>, 0.3 }
```

We make these slightly larger to add some contrast to our view.



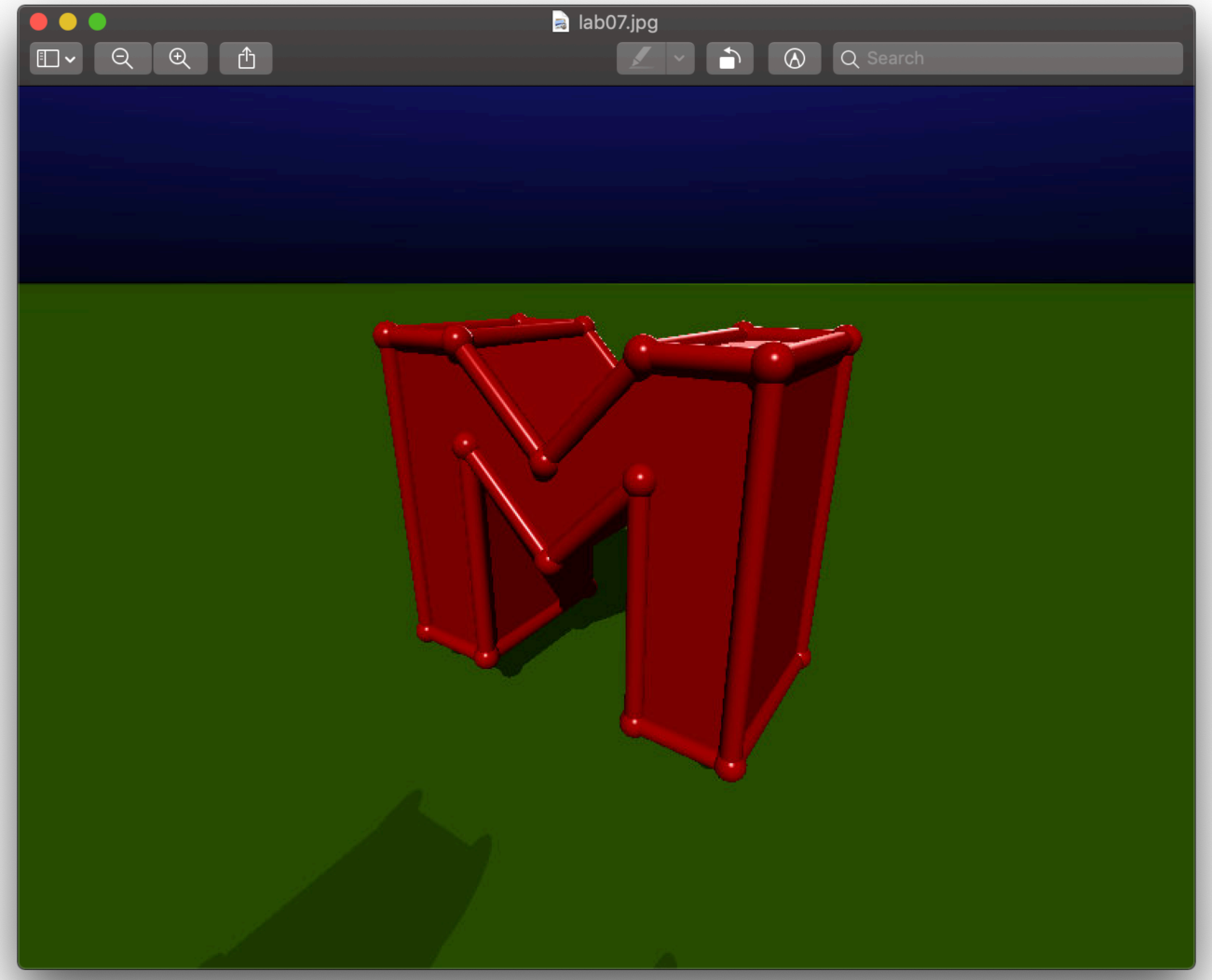
We'll add two more "decorations" to our M. The first will be to apply Phong Shading to our M. This is done by adding to the texture block the following

```
finish { phong 1 }
```

Now our full texture block, inside of the union block, is

```
texture { pigment{ rgb<1.0, 0.0, 0.0> }
          finish { phong 1 }
        }
```

Until now, we were only applying Flat Shading. When we run the program this time, we see the specular highlight on top of the M that is from the sky light. We'll also see specular highlights on each of the spheres from the sun light placed behind the camera.



Finally, we'd like to get rid of those awful jaggies along the edges. The last is to use anti-aliasing. This is not done in the code itself, but instead done when we run the program by telling POV-Ray to apply supersampling as its anti-aliasing technique.

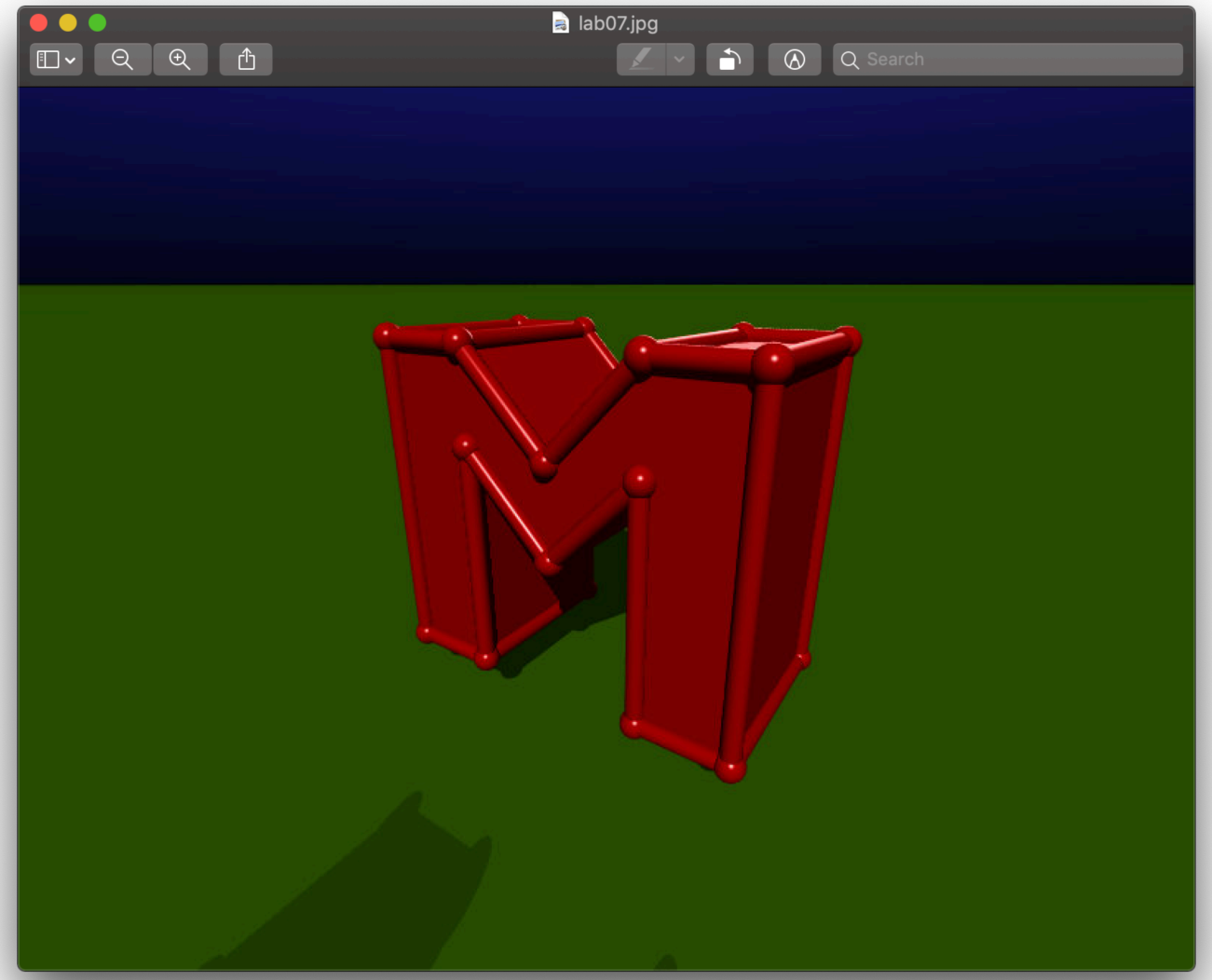
When you run the program, the command should now be

```
povconsole32.exe +A lab7.pov
```

or on your machine

```
povray +A lab7.pov
```

Run and get your final smoothed image.



Step 3 – Submit!

All you need to submit for this lab is your lab07.pov file.

You now have the starting piece for A4. Continue to experiment with POV-Ray and apply as many cool effects as you can!

LAB IS DUE BY **Friday April 09 11:59 PM**!!