3D computer graphics with OpenGL

Karin Kosina (vka kyrah)

OpenGL

Open Graphics Library

OpenGL

- a platform-independent API for 2D and 3D graphics applications
- a standard, not a library
  - various implementions (e.g. by graphics card vendors) with varying degrees of optimisation
- input: primitives (polygons, lines, points)
- output: pixels
- low-level
- state-machine
- only does rendering
  - need additional framework for OS integration, image loading,...

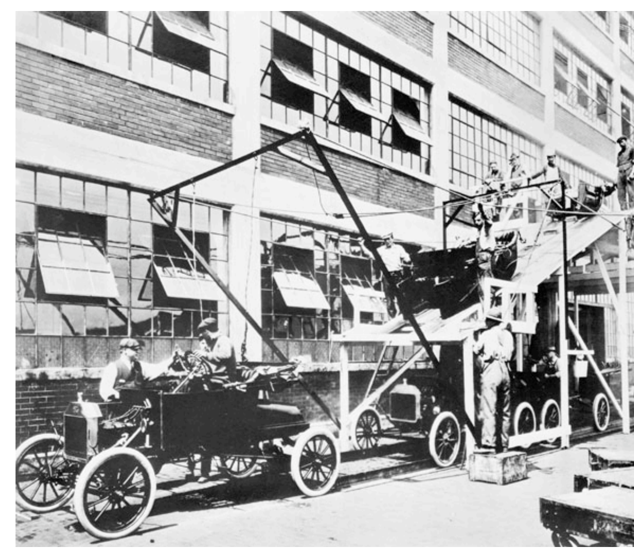
the rendering pipeline

concepts

- Rendering *pipeline*?
    - Think of oil pipelines, assembly lines, ski lifts,...
- Pipelines consist of stages.
    - In an oil pipeline, the oil passes through sequencially.
    - The speed of the pipeline is determined by the slowest part of the pipeline, no matter how fast the other stages may be.
- Ideally, a pipeline of *n* stages should give a speed-up of factor *n*
    - assembly line is a good example

concepts

- Pipeline stages are executed in parallel, but they are stalled until the slowest stage has finished its task.
- cf. a car factory assembly line:
  - attaching the steering wheel takes 3 minutes
  - each other step takes 2 minutes
  - ➙ you can finish one car every 3 minutes
- Slowest stage = "bottleneck"



graphics rendering pipeline

- Function:
  - generate (“render”) a 2-dimensional image given 3-dimensional objects (and a virtual camera, light sources, a lighting model, etc.)
- Rendering speed
  - update speed of images
  - expressed in frames per second (fps)
  - rendering speed is determined by the bottleneck

overview

overview

the application stage

- Fully controlled by application programmer
  - collision detection,
  - input handling (keyboard, mouse, any other devices)
  - animations (updating model transformations)
  - acceleration algorithms (such as hierarchical view frustum culling)
- Output:
  - Geometry to be rendered in the form of rendering primitives (points, lines, triangles)

overview

the geometry stage

- Computes what should be drawn, where it should be drawn, how it should be drawn.
- Handles per-vertex operations.
- Can be subdivided into five functional stages:
    - model & view transform, lighting, projection, clipping, screen mapping.
- With a single light source, each vertex requires approximately 100 individual floating point operations!

overview

the rasterization stage

- Input: transformed and projected vertices, colors, and texture coordinates from the geometry stage.
- Task is to assign correct colors to the pixels on the screen to render a correct image.
- *Rasterization* (aka *scan conversion*):
  - Conversion of 2d vertices in screen space (each with a z-value, one or two colors, and possibly a set of texture coordinates) into pixels on the screen.

the rasterization stage

- Handles per-pixel operations.
- Information for each pixel is stored in the color buffer (a rectangular array of colors).
- Color buffer should contain only the colors of the primitives which are visible from the point of view of the camera.
- This is usually done using the Z-Buffer algorithm.

summary

STL?

~~STL?~~

SDL

Simple Directmedia Layer

SDL

- SDL is a free cross-platform multi-media development API
- abstraction for OS-dependent tasks
    - create window and rendering context
    - handle keyboard, mouse, and joystick events
    - audio
    - thread abstraction
    - ...
- see http://libsdl.org

anatomy of an SDL application

1. Initialise SDL (SDL_Init())
2. Create OpenGL rendering context (SDL_SetVideoMode())
3. Do your own OpenGL and app initialisation
4. Run main loop:
    - rendering
    - event processing
5. Cleanup

brace yourselves

anatomy of an SDL application

```
int main(int argc, char ** argv)
{
  int width = 640, height = 480;

  // Initialize SDL
  if (SDL_Init(SDL_INIT_VIDEO) < 0) {
    fprintf(stderr, "Unable to init SDL: %s\n", SDL_GetError());
    return -1;
  }

  if (!SDL_SetVideoMode(width, height, 32, SDL_OPENGL)) {
    fprintf(stderr, "Unable set video mode: %s\n", SDL_GetError());
    SDL_Quit();
    return -1;
  }

  SDL_WM_SetCaption("SDL/OpenGL intro", NULL);  // window title
  myinit(width, height);  // initialize OpenGL

  // ... continued on next page
```

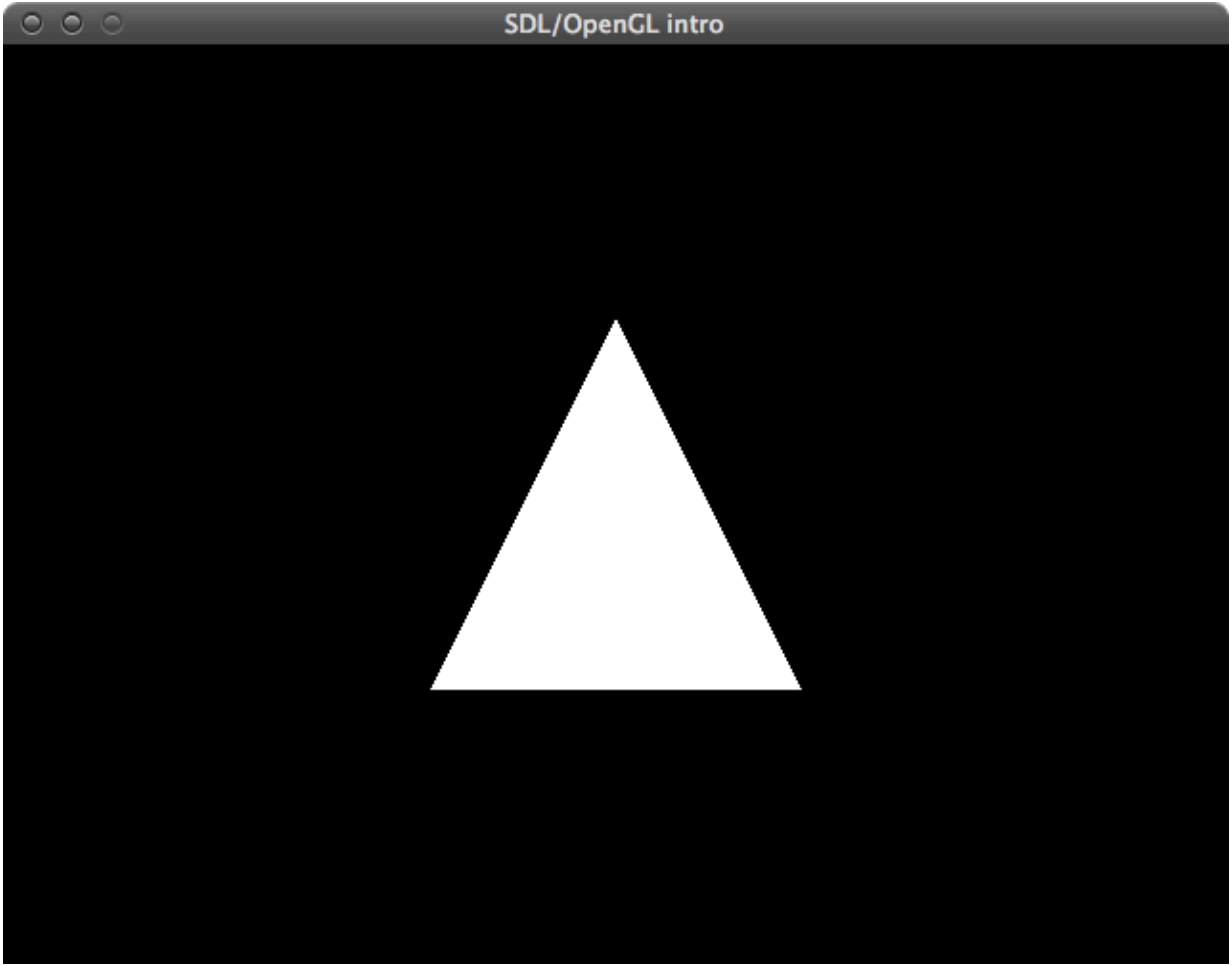
anatomy of an SDL application

```
  // main application loop
  bool done = false;
  while (!done) {
    mydisplay();
    SDL_Event event;
    while (SDL_PollEvent(&event)) {
      if (event.type == SDL_QUIT) done = true;
      if (event.type == SDL_KEYDOWN) {
        switch(event.key.keysym.sym) {
        case SDLK_ESCAPE:
          done = true;
        }
      }
    }
  }

  SDL_Quit();
  return 0;
}
```

basicsdl.cpp

</SDL>

now for some OpenGL fun!



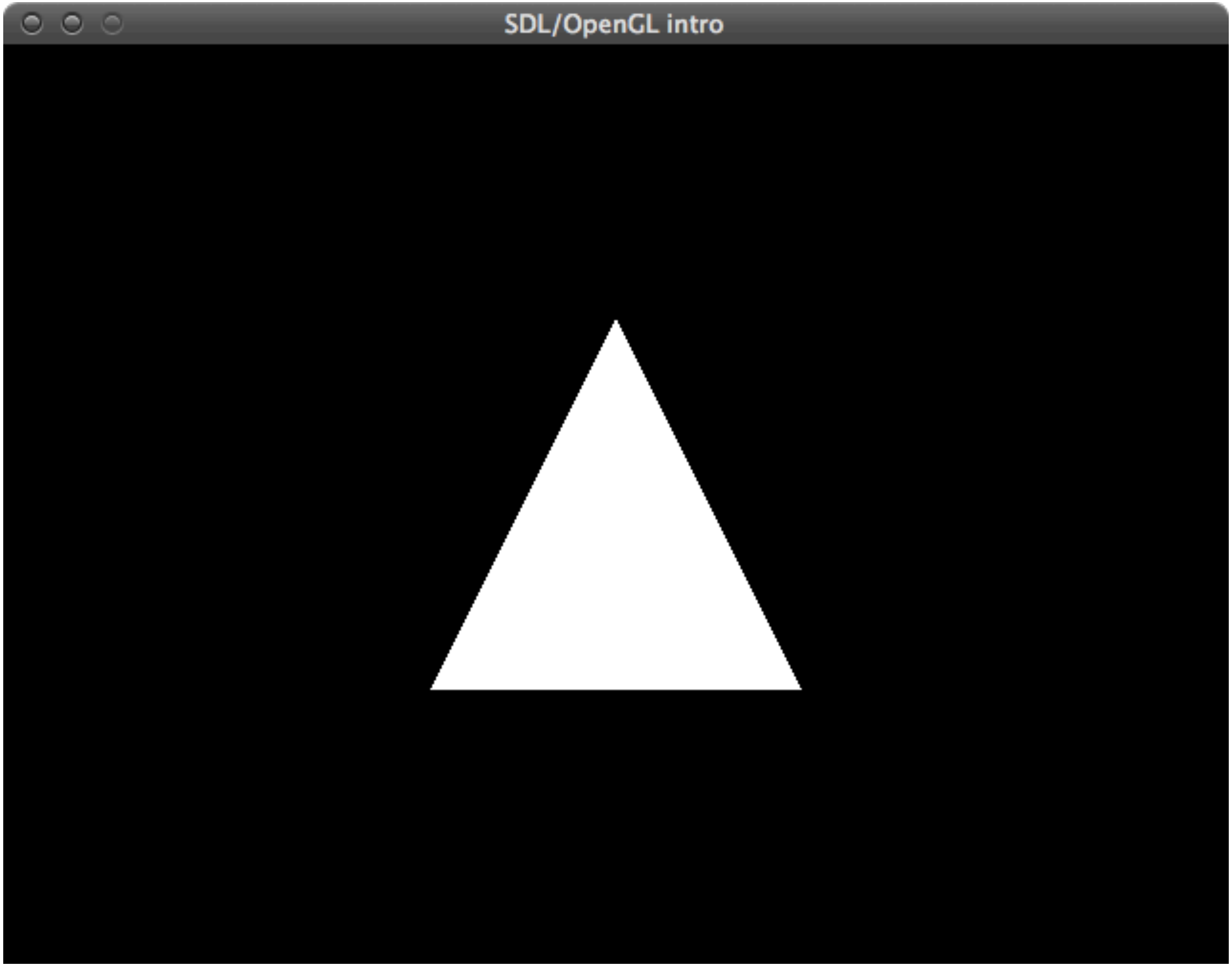
triangle.cpp

OpenGL initialisation

```
void myinit(int width, int height)
{
  glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
  glEnable(GL_DEPTH_TEST);
  glViewport(0, 0, width, height);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(45.0, (float)width/(float)height, 0.1, 100.0);

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt(0.0, 0.0, 4.0,  // eye
            0.0, 0.0, -1.0, // center
            0.0, 1.0, 0.0); // up
}
```

drawing

```
void mydisplay()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glBegin(GL_TRIANGLES);
  glVertex3f( 0.0f, 1.0f, 0.0f);
  glVertex3f( 1.0f,-1.0f, 0.0f);
  glVertex3f(-1.0f,-1.0f, 0.0f);
  glEnd();

  SDL_GL_SwapBuffers();
}
```


SDL/OpenGL intro

OpenGL initialisation

```
void myinit(int width, int height)
{
  glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
  glEnable(GL_DEPTH_TEST);
  glViewport(0, 0, width, height);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(45.0, (float)width/(float)height, 0.1, 100.0);

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt(0.0, 0.0, 4.0,  // eye
            0.0, 0.0, -1.0, // center
            0.0, 1.0, 0.0); // up
}
```

OpenGL initialisation

```
void myinit(int width, int height)
{
  glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
  glEnable(GL_DEPTH_TEST);
  glViewport(0, 0, width, height);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(45.0, (float)width/(float)height, 0.1, 100.0);

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt(0.0, 0.0, 4.0,  // eye
            0.0, 0.0, -1.0, // center
            0.0, 1.0, 0.0); // up
}
```

OpenGL initialisation

```
void myinit(int width, int height)
{
  glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
  glEnable(GL_DEPTH_TEST);
  glViewport(0, 0, width, height);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(45.0, (float)width/(float)height, 0.1, 100.0);

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt(0.0, 0.0, 4.0,  // eye
            0.0, 0.0, -1.0, // center
            0.0, 1.0, 0.0); // up
}
```

the z-buffer

- The Z-buffer is the same size as the color buffer and stores the z-value from the camera to the closest primitive.
- When a primitive is rendered to a certain pixel, the z-value of the primitive at that pixel is computed and compared to the contents of the Z-buffer at the same pixel.
  - If the new z value is smaller than the z value in the Z-buffer, the primitive is closer to the camera ➙ the z value and the color of that pixel are updated.
  - If the new z value is greater, color and z are not changed.

OpenGL initialisation

```
void myinit(int width, int height)
{
  glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
  glEnable(GL_DEPTH_TEST);
  glViewport(0, 0, width, height);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(45.0, (float)width/(float)height, 0.1, 100.0);

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt(0.0, 0.0, 4.0,  // eye
            0.0, 0.0, -1.0, // center
            0.0, 1.0, 0.0); // up
}
```

OpenGL initialisation

```
void myinit(int width, int height)
{
  glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
  glEnable(GL_DEPTH_TEST);
  glViewport(0, 0, width, height);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(45.0, (float)width/(float)height, 0.1, 100.0);

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt(0.0, 0.0, 4.0,  // eye
            0.0, 0.0, -1.0, // center
            0.0, 1.0, 0.0); // up
}
```

OpenGL initialisation

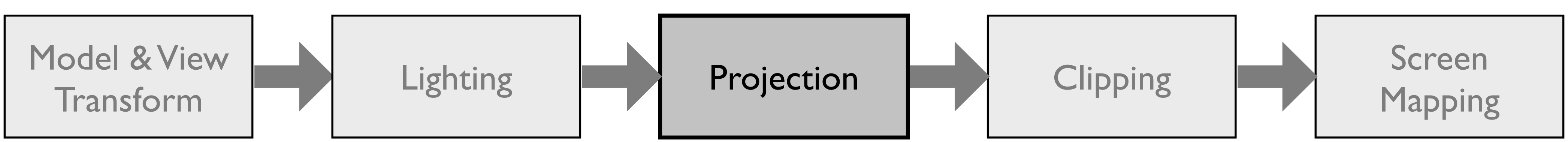
```
void myinit(int width, int height)
{
  glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
  glEnable(GL_DEPTH_TEST);
  glViewport(0, 0, width, height);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(45.0, (float)width/(float)height, 0.1, 100.0);

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt(0.0, 0.0, 4.0,  // eye
            0.0, 0.0, -1.0, // center
            0.0, 1.0, 0.0); // up
}
```

OpenGL initialisation

```
void myinit(int width, int height)
{
  glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
  glEnable(GL_DEPTH_TEST);
  glViewport(0, 0, width, height);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(45.0, (float)width/(float)height, 0.1, 100.0);

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt(0.0, 0.0, 4.0,  // eye
            0.0, 0.0, -1.0, // center
            0.0, 1.0, 0.0); // up
}
```

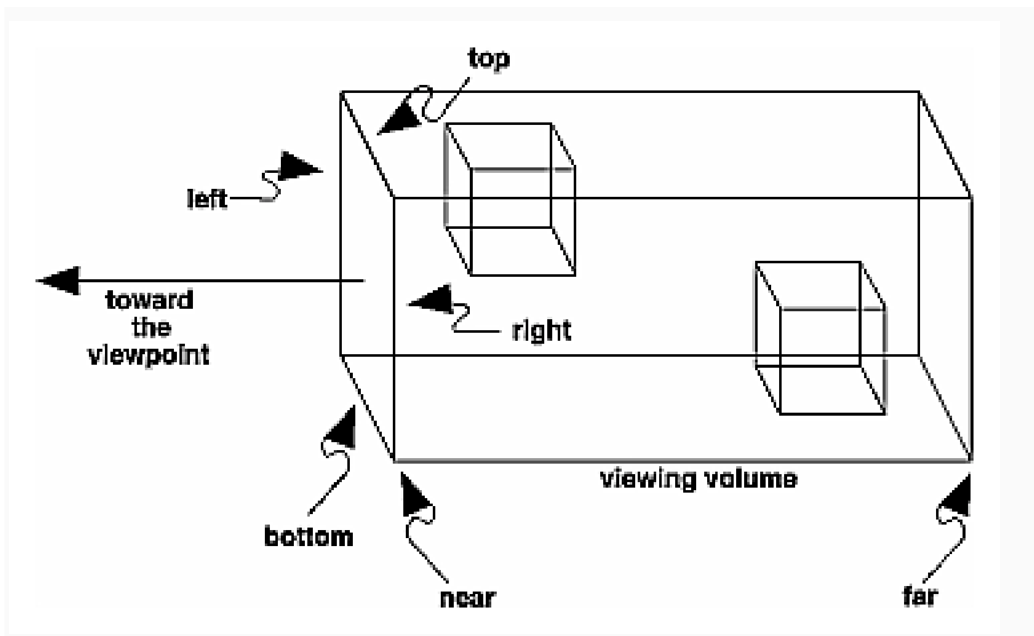
projection

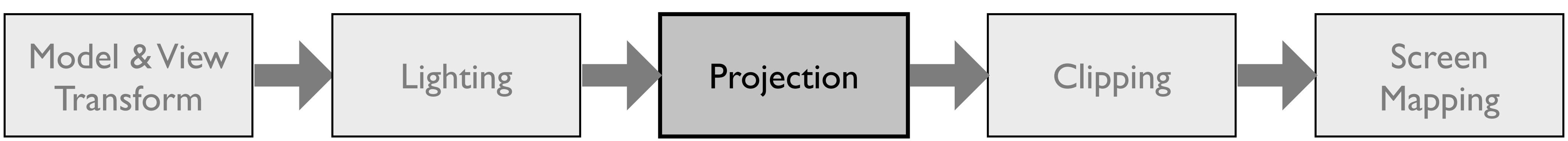
- Two projection methods:
  - orthographic vs. perspective projection
- Orthographic projection:
  - View volume is a rectangular box.
  - Parallel lines remain parallel after the transform.



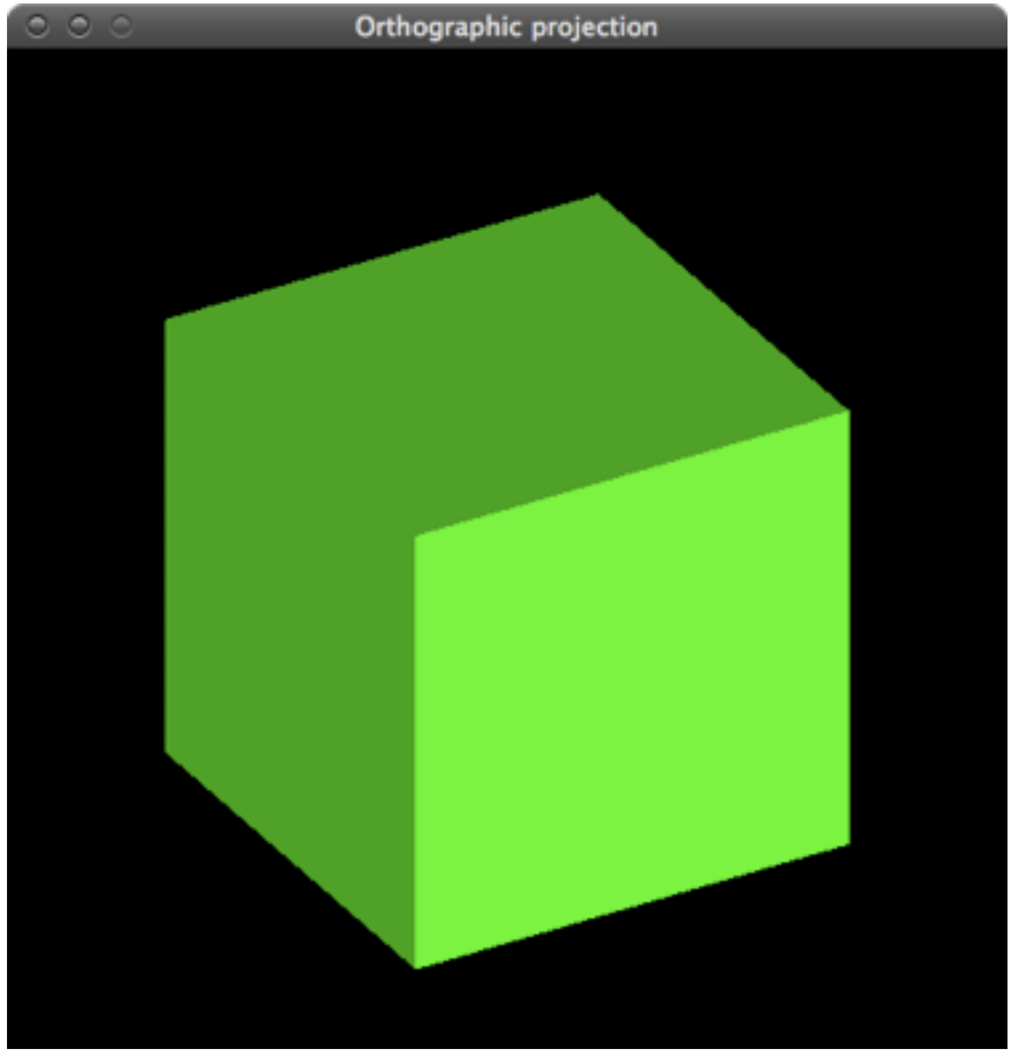
projection

```
glOrtho(float left, float right,
        float bottom, float top,
        float near, float far);
```





projection

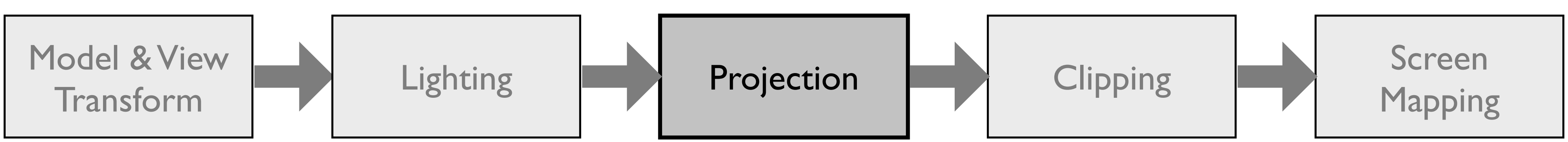


proj_ortho.cpp

Model & View Transform → Lighting → **Projection** → Clipping → Screen Mapping

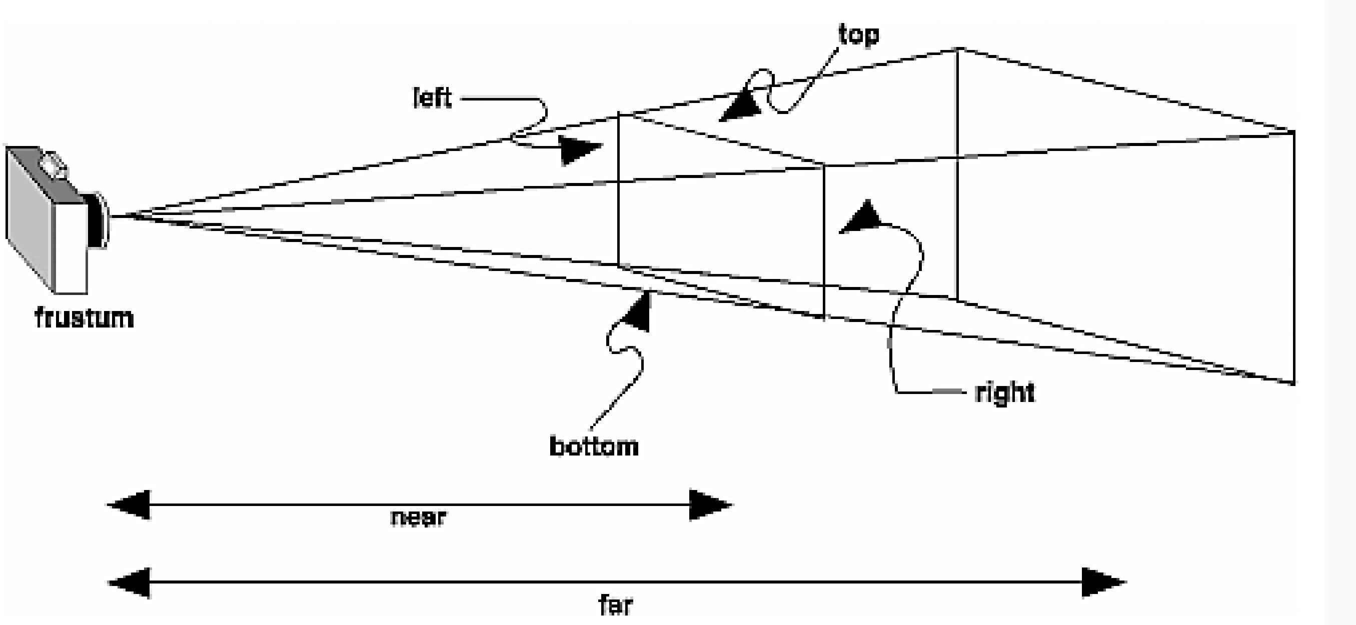
projection

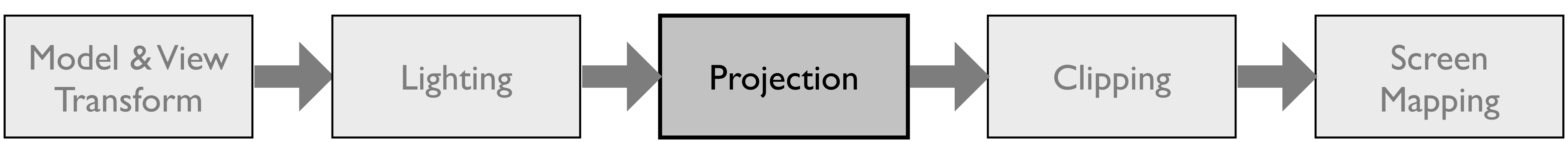
- Perspective projection:
    - The farther away an object lies from the camera, the smaller it appears after projection.
    - Parallel lines converge at the horizon.
    - View volume (called frustum) is a truncated pyramid with a rectangular base.



projection

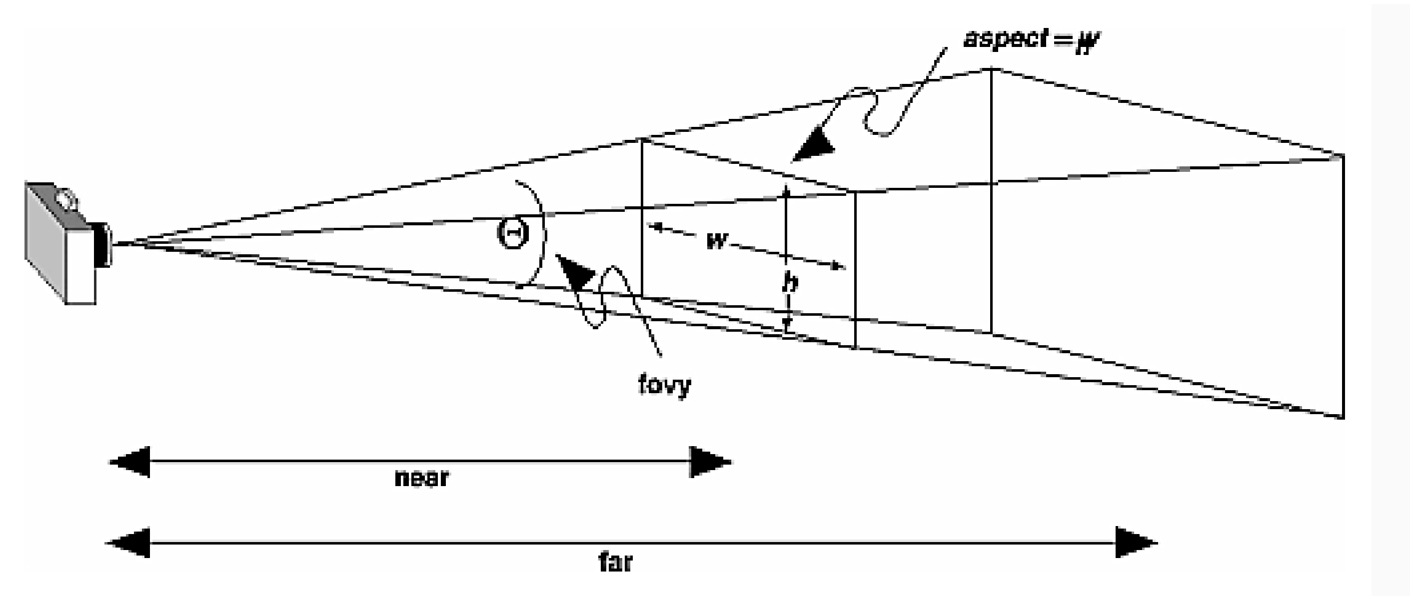
```
glFrustum(float left, float right,
          float bottom, float top,
          float near, float far);
```

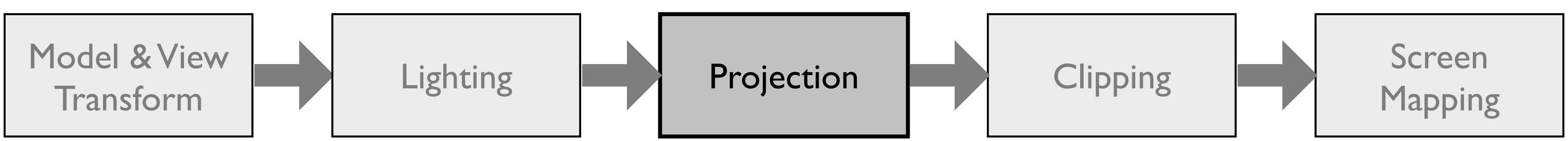




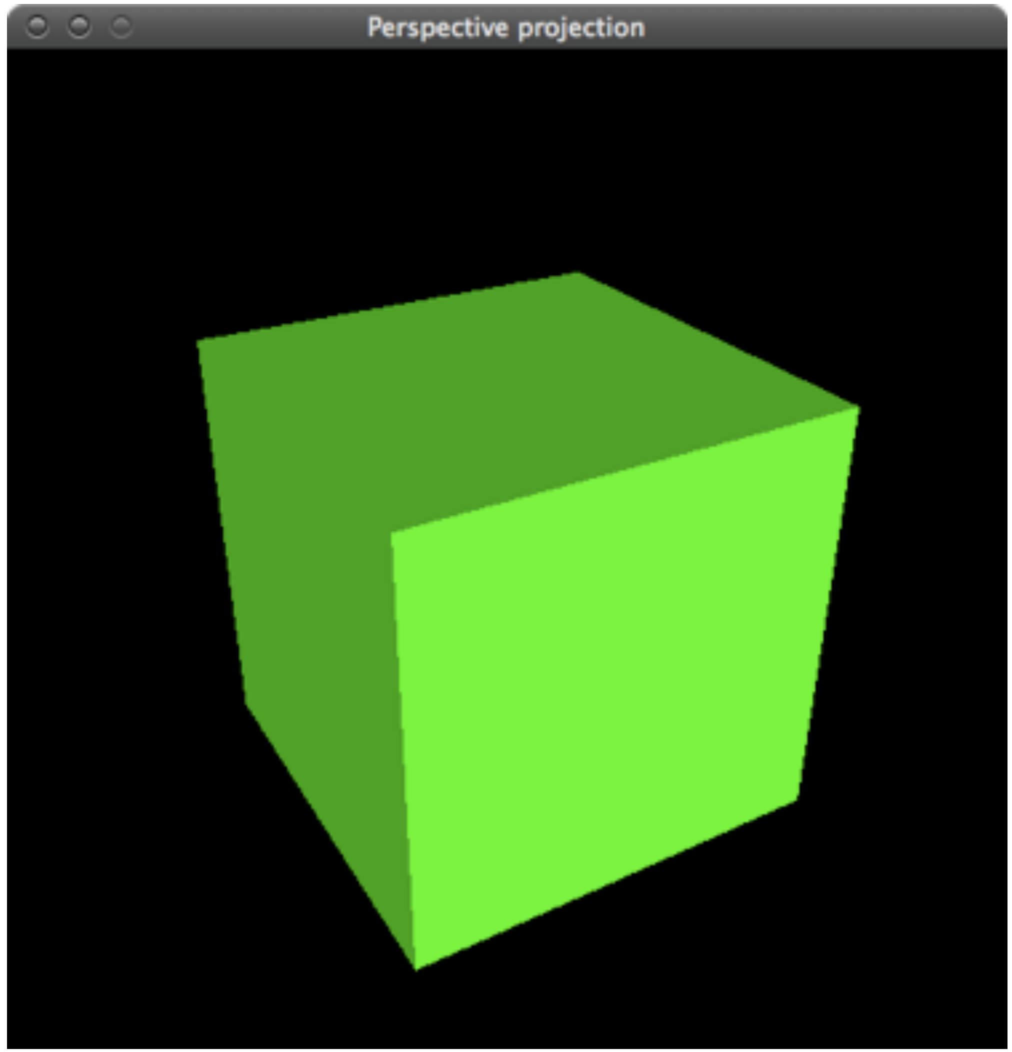
projection

```
gluPerspective(float fovy, float aspect,
               float near, float far);
```





projection



proj_persp.cpp

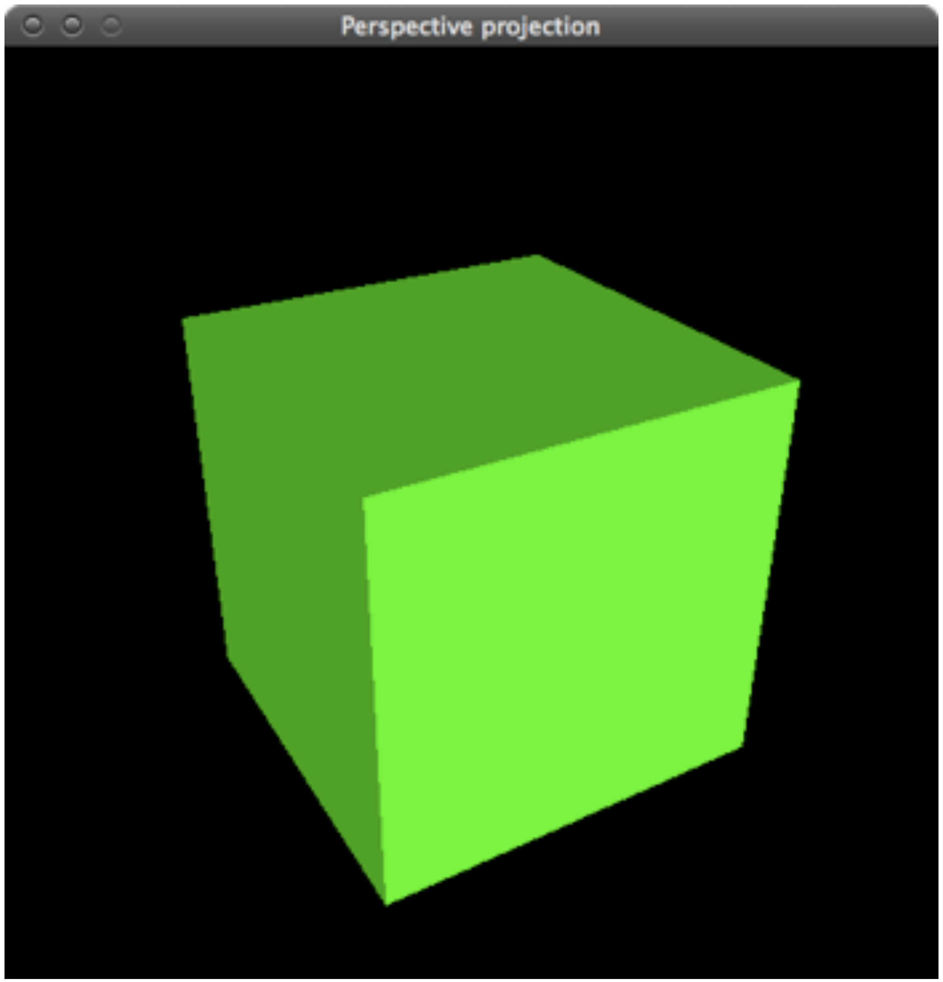
Model & View Transform → Lighting → **Projection** → Clipping → Screen Mapping

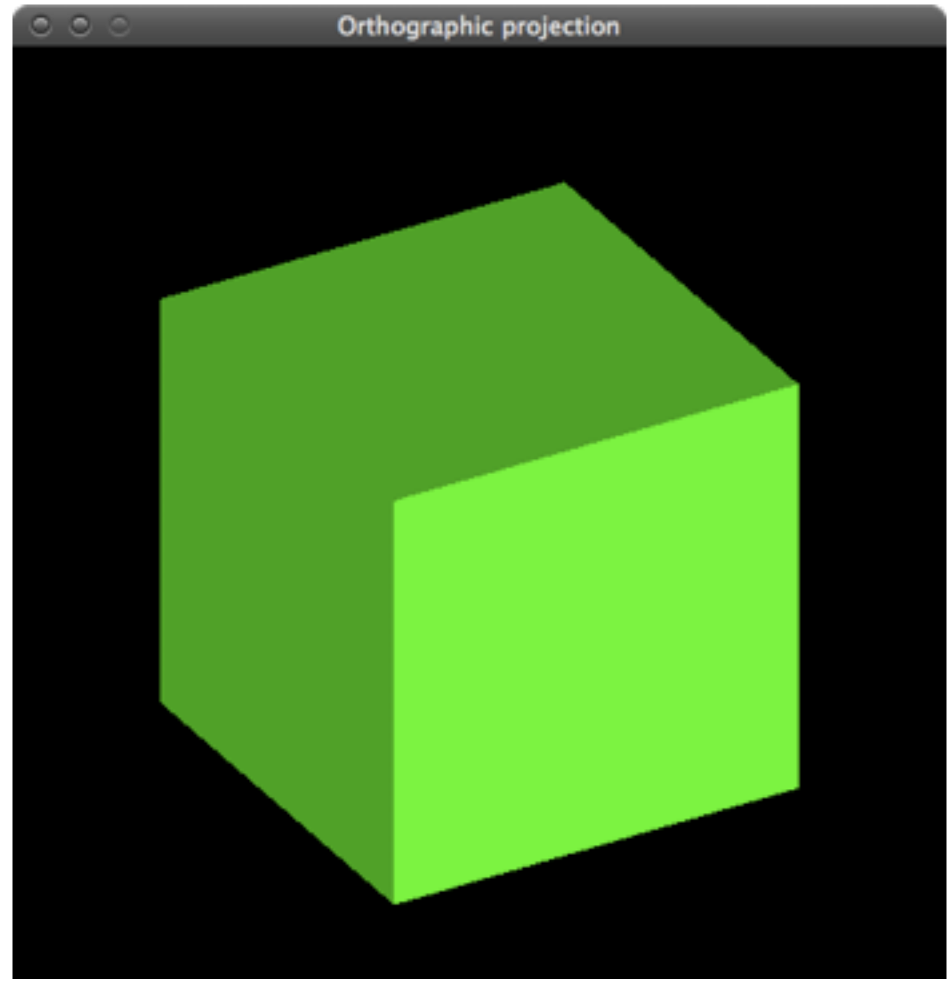
OpenGL initialisation

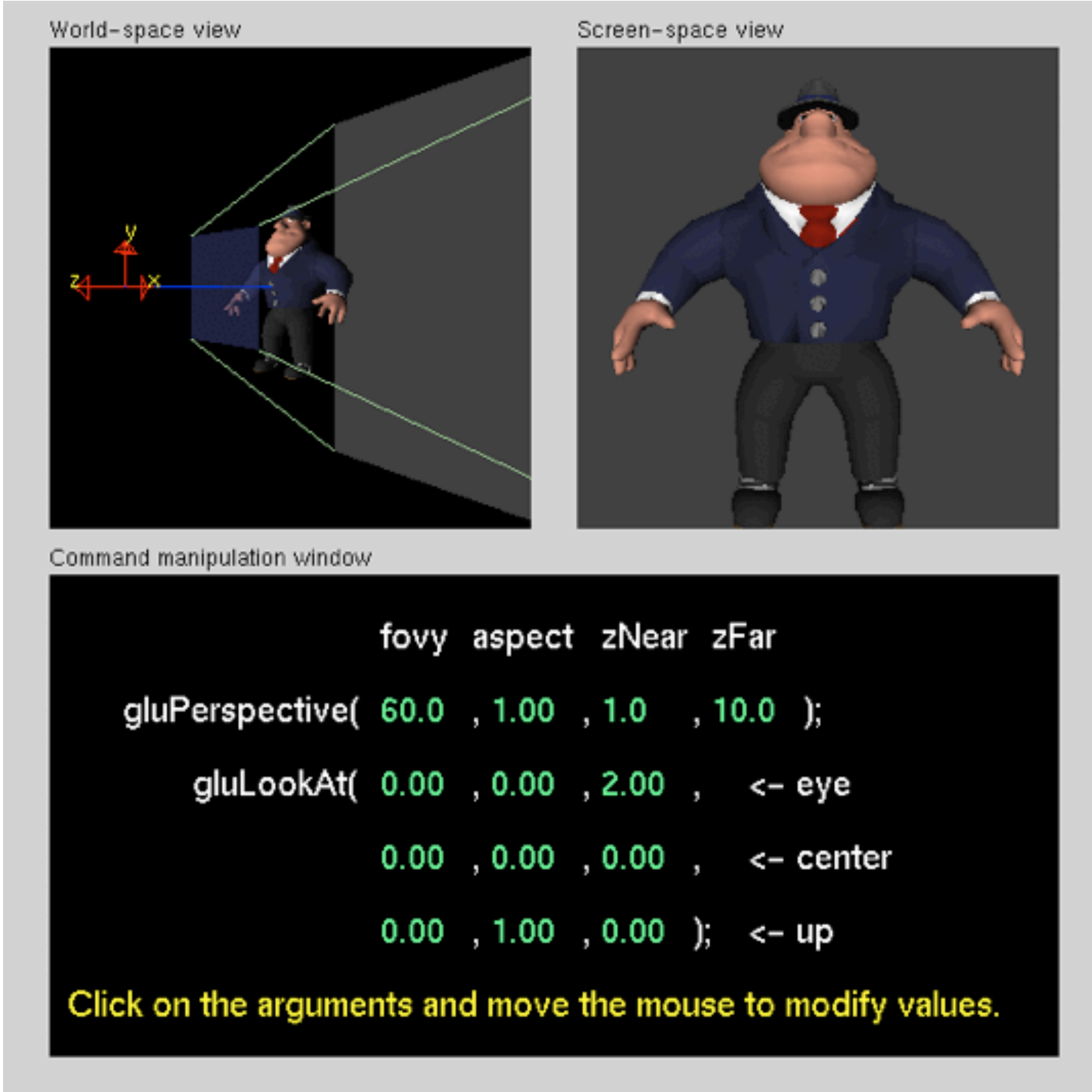
```
void myinit(int width, int height)
{
  glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
  glEnable(GL_DEPTH_TEST);
  glViewport(0, 0, width, height);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(45.0, (float)width/(float)height, 0.1, 100.0);

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt(0.0, 0.0, 4.0,  // eye
            0.0, 0.0, -1.0, // center
            0.0, 1.0, 0.0); // up
}
```

OpenGL initialisation

```
void myinit(int width, int height)
{
  glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
  glEnable(GL_DEPTH_TEST);
  glViewport(0, 0, width, height);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(-3, 3, -3, 3, 2, 10);

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt(0.0, 0.0, 4.0,  // eye
            0.0, 0.0, -1.0, // center
            0.0, 1.0, 0.0); // up
}
```


Perspective projection


Orthographic projection


World-space view
Screen-space view
y
z
x
Command manipulation window
fovy aspect zNear zFar
gluPerspective(60.0 , 1.00 , 1.0 , 10.0);
gluLookAt(0.00 , 0.00 , 2.00 , <- eye
0.00 , 0.00 , 0.00 , <- center
0.00 , 1.00 , 0.00); <- up
Click on the arguments and move the mouse to modify values.

OpenGL initialisation

```
void myinit(int width, int height)
{
  glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
  glEnable(GL_DEPTH_TEST);
  glViewport(0, 0, width, height);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(45.0, (float)width/(float)height, 0.1, 100.0);

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt(0.0, 0.0, 4.0,  // eye
            0.0, 0.0, -1.0, // center
            0.0, 1.0, 0.0); // up
}
```

OpenGL initialisation

```
void myinit(int width, int height)
{
  glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
  glEnable(GL_DEPTH_TEST);
  glViewport(0, 0, width, height);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(45.0, (float)width/(float)height, 0.1, 100.0);

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt(0.0, 0.0, 4.0,  // eye
            0.0, 0.0, -1.0, // center
            0.0, 1.0, 0.0); // up
}
```

drawing

```
void mydisplay()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glLoadIdentity();

  glBegin(GL_TRIANGLES);
  glVertex3f( 0.0f, 1.0f, 0.0f);
  glVertex3f( 1.0f,-1.0f, 0.0f);
  glVertex3f(-1.0f,-1.0f, 0.0f);
  glEnd();

  SDL_GL_SwapBuffers();
}
```

drawing

```
void mydisplay()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glBegin(GL_TRIANGLES);
  glVertex3f( 0.0f, 1.0f, 0.0f);
  glVertex3f( 1.0f,-1.0f, 0.0f);
  glVertex3f(-1.0f,-1.0f, 0.0f);
  glEnd();

  SDL_GL_SwapBuffers();
}
```

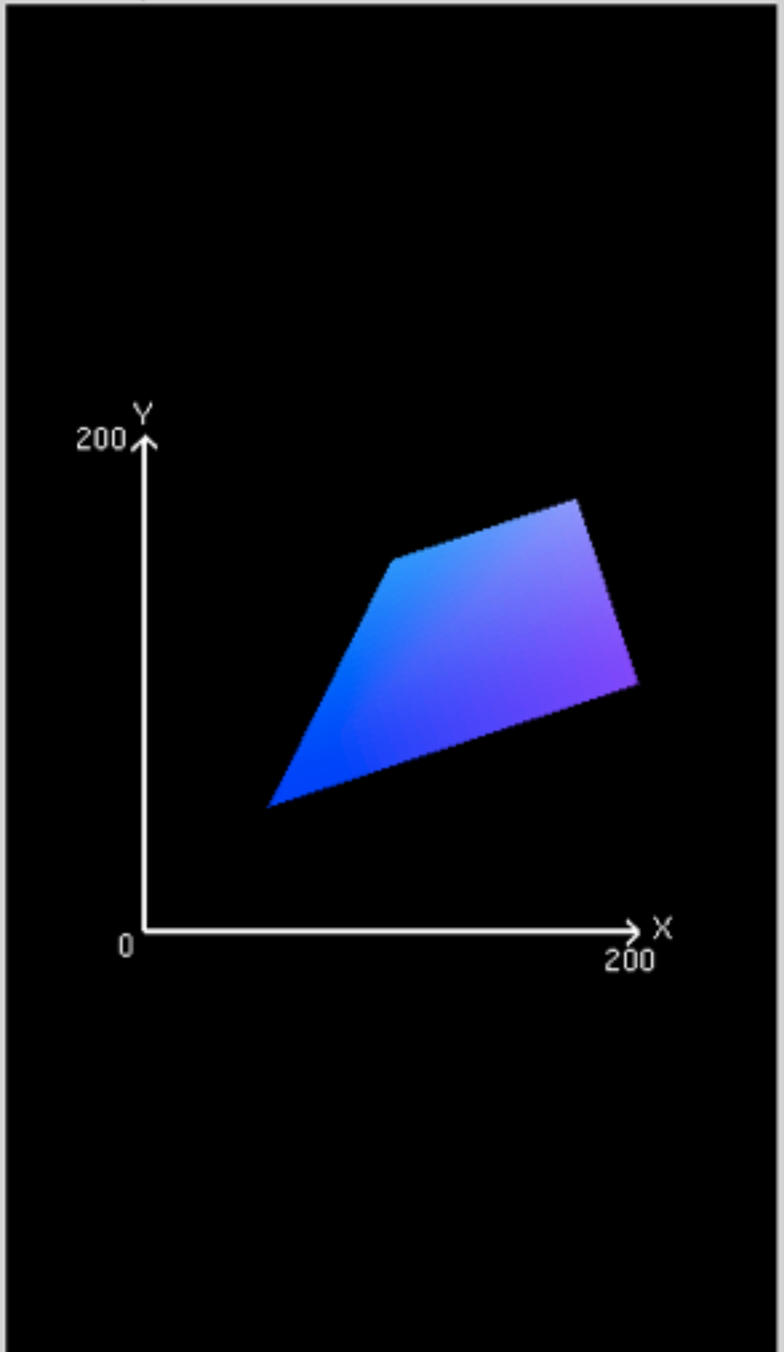
drawing

```
void mydisplay()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glBegin(GL_TRIANGLES);
  glVertex3f( 0.0f, 1.0f, 0.0f);
  glVertex3f( 1.0f,-1.0f, 0.0f);
  glVertex3f(-1.0f,-1.0f, 0.0f);
  glEnd();

  SDL_GL_SwapBuffers();
}
```

drawing

```
void mydisplay()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glBegin(GL_TRIANGLES);
  glVertex3f( 0.0f, 1.0f, 0.0f);
  glVertex3f( 1.0f,-1.0f, 0.0f);
  glVertex3f(-1.0f,-1.0f, 0.0f);
  glEnd();

  SDL_GL_SwapBuffers();
}
```

Screen-space view



Command manipulation window

```
glBegin (GL_TRIANGLE_FAN);
glColor3f  (0.00  , 0.00  , 1.00  );
glVertex2f (50.0  , 50.0  );
glColor3f  (0.00  , 0.50  , 1.00  );
glVertex2f (100.0 , 150.0 );
glColor3f  (0.50  , 0.50  , 1.00  );
glVertex2f (175.0 , 175.0 );
glColor3f  (0.50  , 0.00  , 1.00  );
glVertex2f (200.0 , 100.0 );
glEnd();
```

Click on the arguments and
move the mouse to modify values.

drawing

```
void mydisplay()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glBegin(GL_TRIANGLES);
  glVertex3f( 0.0f, 1.0f, 0.0f);
  glVertex3f( 1.0f,-1.0f, 0.0f);
  glVertex3f(-1.0f,-1.0f, 0.0f);
  glEnd();

  SDL_GL_SwapBuffers();
}
```

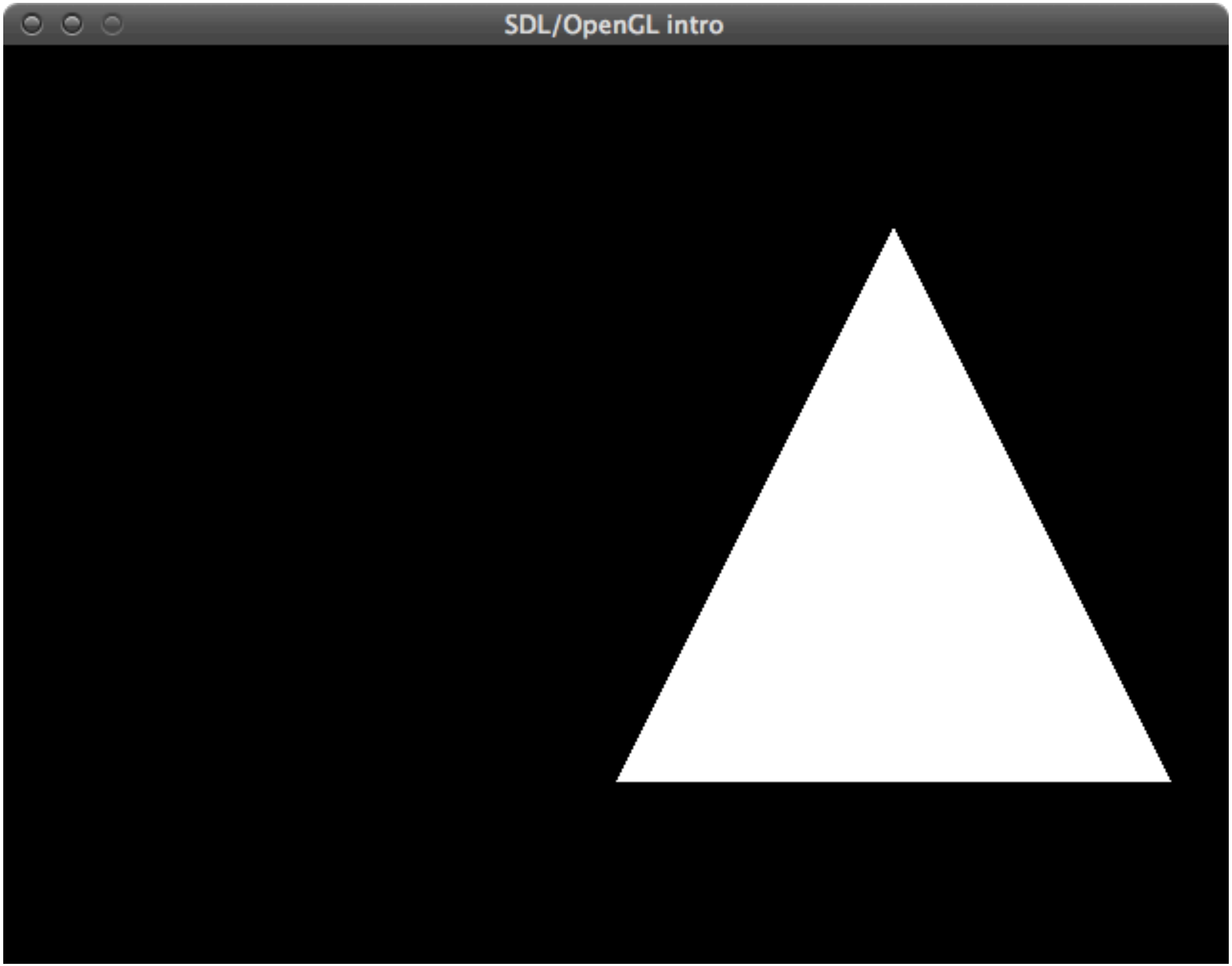
drawing

```
void mydisplay()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glBegin(GL_TRIANGLES);
  glVertex3f( 0.0f, 1.0f, 0.0f);
  glVertex3f( 1.0f,-1.0f, 0.0f);
  glVertex3f(-1.0f,-1.0f, 0.0f);
  glEnd();

  SDL_GL_SwapBuffers();
}
```

double-buffering

- To avoid visible flickering during the rasterization process, double buffering is used:
    - Rendering is done off-screen in the back buffer.
    - When the rendered scene is complete, front and back buffer are swapped.
    - The swapping is done during the vertical monitor sync, so that it is not visible.

let's move the triangle


SDL/OpenGL intro

modify drawing code

```
void mydisplay()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glBegin(GL_TRIANGLES);
  glVertex3f( 0.0f, 1.0f, 0.0f);
  glVertex3f( 1.0f,-1.0f, 0.0f);
  glVertex3f(-1.0f,-1.0f, 0.0f);
  glEnd();

  SDL_GL_SwapBuffers();
}
```

modified drawing code

```
void mydisplay()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glBegin(GL_TRIANGLES);
  glVertex3f( 1.0f, 1.0f, 0.0f);
  glVertex3f( 2.0f,-1.0f, 0.0f);
  glVertex3f( 0.0f,-1.0f, 0.0f);
  glEnd();

  SDL_GL_SwapBuffers();
}
```

transtri1.cpp

this works

but can get kinda tedious

there's a better way

original drawing code

```
void mydisplay()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glBegin(GL_TRIANGLES);
  glVertex3f( 0.0f, 1.0f, 0.0f);
  glVertex3f( 1.0f,-1.0f, 0.0f);
  glVertex3f(-1.0f,-1.0f, 0.0f);
  glEnd();

  SDL_GL_SwapBuffers();
}
```

add a translation

```
void mydisplay()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glTranslatef(1.0f, 0.0f, 0.0f);

  glBegin(GL_TRIANGLES);
  glVertex3f( 0.0f, 1.0f, 0.0f);
  glVertex3f( 1.0f,-1.0f, 0.0f);
  glVertex3f(-1.0f,-1.0f, 0.0f);
  glEnd();

  SDL_GL_SwapBuffers();
}
```

transtri2.cpp

and one more possibility

move the camera

```
void myinit(int width, int height)
{
  glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
  glEnable(GL_DEPTH_TEST);
  glViewport(0, 0, width, height);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(45.0, (float)width/(float)height, 0.1, 100.0);
  gluLookAt(0.0, 0.0, 4.0,  // eye
            0.0, 0.0, -1.0, // center
            0.0, 1.0, 0.0); // up
  glMatrixMode(GL_MODELVIEW);
}
```

move the camera

```
void myinit(int width, int height)
{
  glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
  glEnable(GL_DEPTH_TEST);
  glViewport(0, 0, width, height);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(45.0, (float)width/(float)height, 0.1, 100.0);
  gluLookAt(-1.0, 0.0, 4.0,  // eye
            -1.0, 0.0, -1.0, // center
            0.0, 1.0, 0.0); // up
  glMatrixMode(GL_MODELVIEW);
}
```

transtri3.cpp

a few words on coordinate systems

coordinate systems

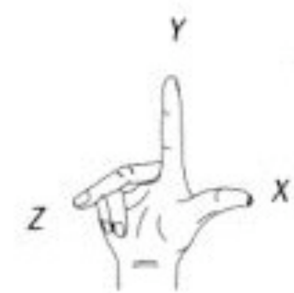
- On the way to the screen, a model is transformed into several different spaces or coordinate systems:
  - model space
  - world space [result of model transform]
  - camera space [result of view transform]
- Model transform and view transform are often concatenated for efficiency reasons.

coordinate systems

- <u>Model space (aka object space)</u>
  - Being in model space means that a model has not been transformed at all.
  - A model can be associated with a *model transform* to position and orient it.
  - Several model transforms associated with one model allow for multiple instances without geometry replication.

coordinate systems

- <u>World space</u>
  - After the model transform has been applied to the model, it is located in world space.
  - Model transform changes vertices and normals of the model.
  - World space is unique: After the models have been transformed by their respective model transforms, all models exist in this same space.



right-hand coordinate system

Model & View Transform → Lighting → Projection → Clipping → Screen Mapping

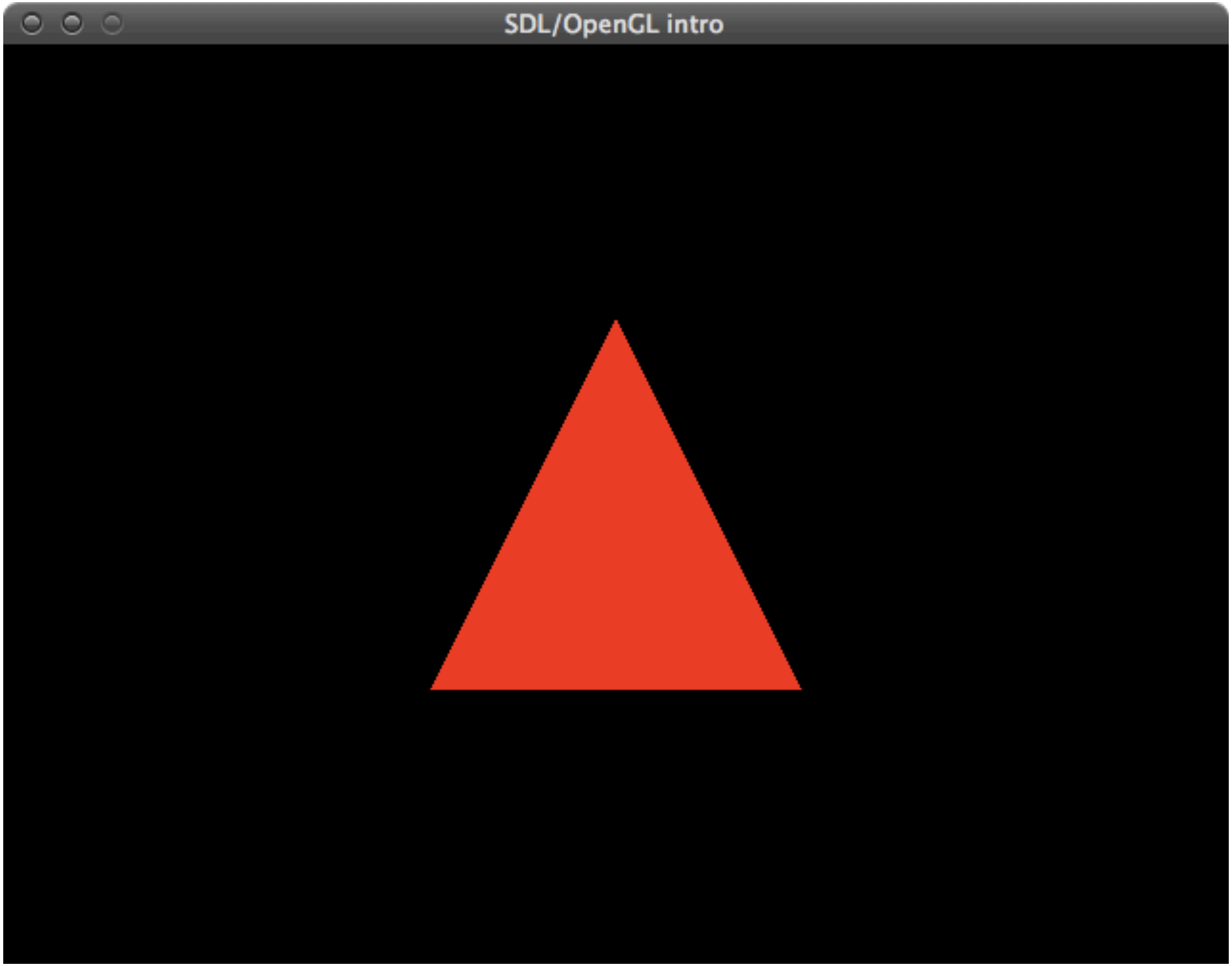
coordinate systems

- <u>Camera space</u>
  - Virtual camera has a location in world space and a direction.
  - The *view transform* places the camera at the origin and aims it to look in the direction of the negative z-axis, with the y-axis pointing upwards and the x-axis pointing right.
  - All models are transformed with the view transform to facilitate projection and clipping.

let's colour the triangle

drawing

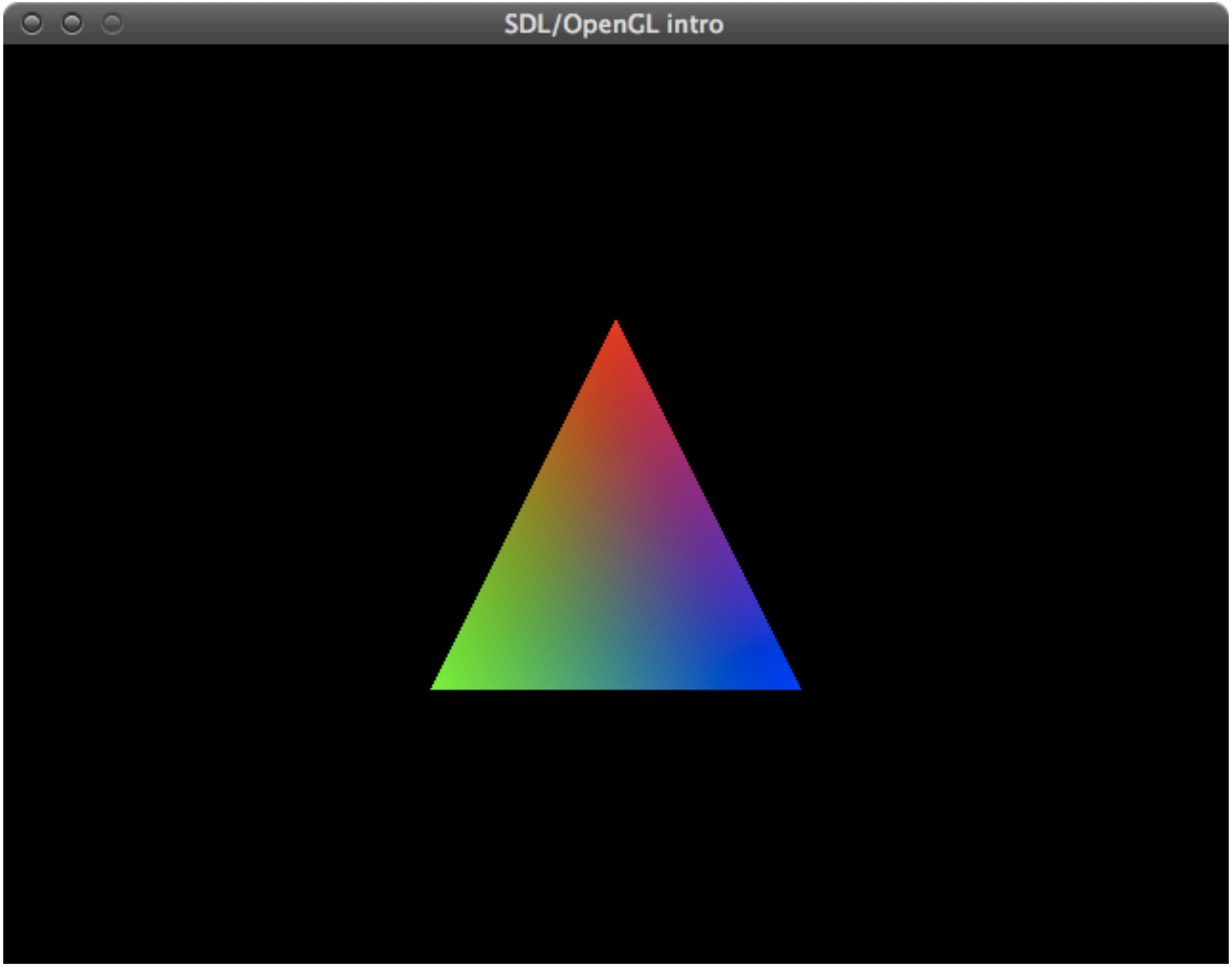
```
void mydisplay()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glBegin(GL_TRIANGLES);
  glColor3f(1.0f, 0.0f, 0.0f);
  glVertex3f( 0.0f, 1.0f, 0.0f);
  glVertex3f( 1.0f,-1.0f, 0.0f);
  glVertex3f(-1.0f,-1.0f, 0.0f);
  glEnd();

  SDL_GL_SwapBuffers();
}
```


SDL/OpenGL intro

drawing

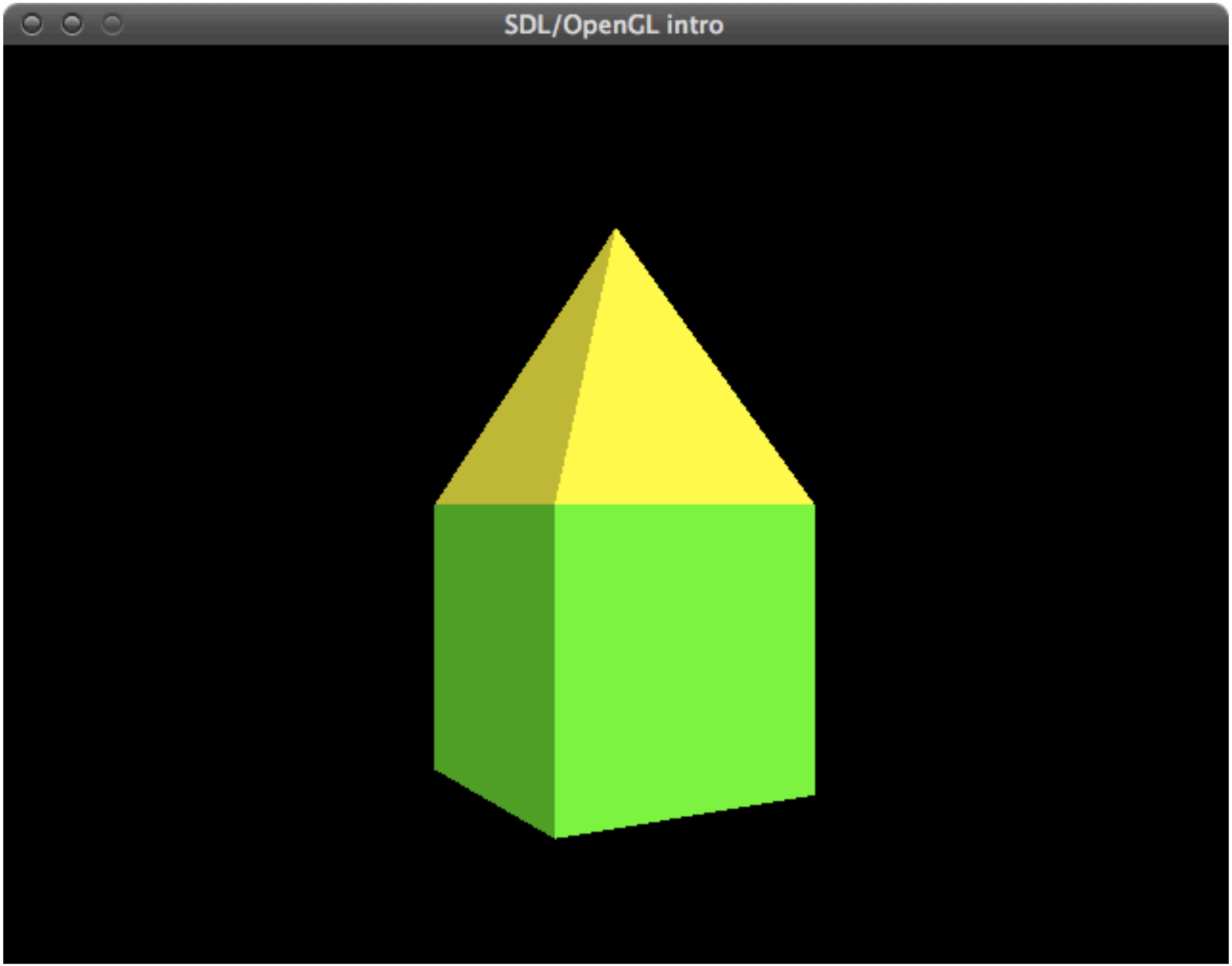
```
void mydisplay()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glBegin(GL_TRIANGLES);

  glColor3f(1.0f, 0.0f, 0.0f);
  glVertex3f( 0.0f, 1.0f, 0.0f);
  glColor3f(0.0f, 0.0f, 1.0f);
  glVertex3f( 1.0f,-1.0f, 0.0f);
  glColor3f(0.0f, 1.0f, 0.0f);
  glVertex3f(-1.0f,-1.0f, 0.0f);

  glEnd();

  SDL_GL_SwapBuffers();
}
```

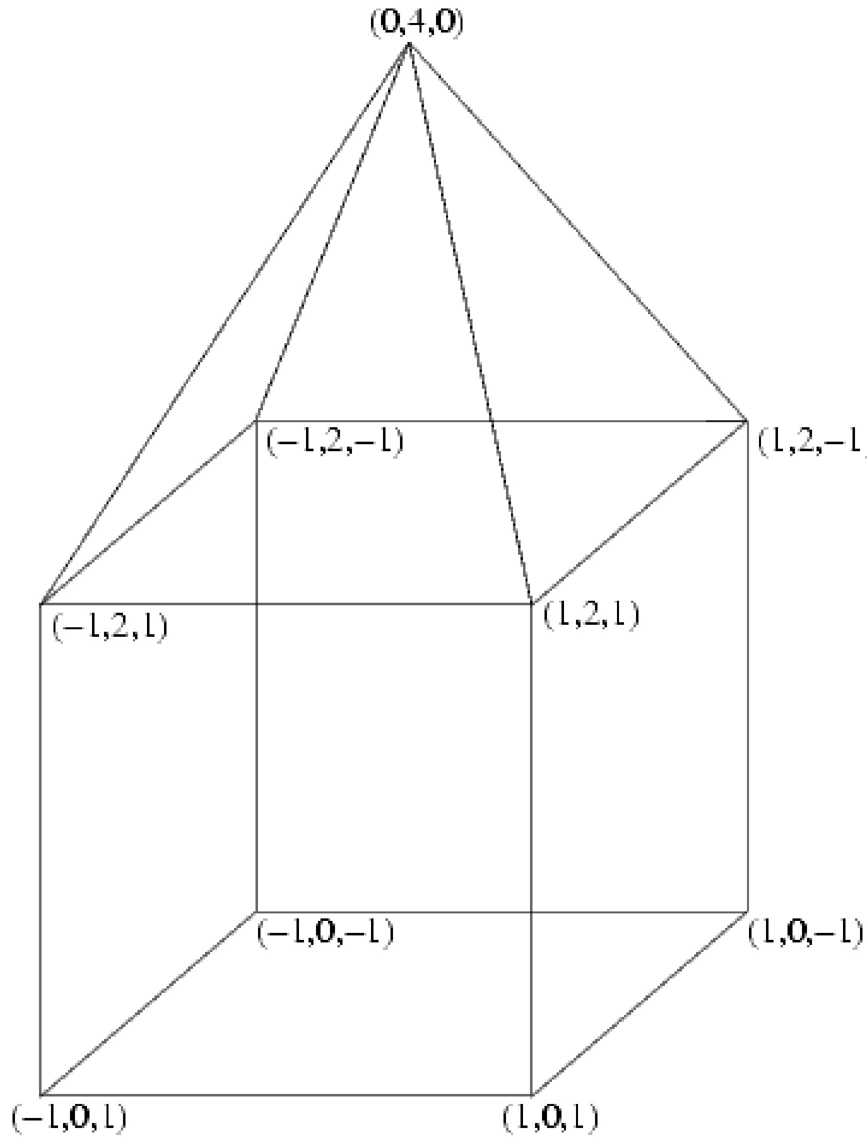
tricolor.cpp



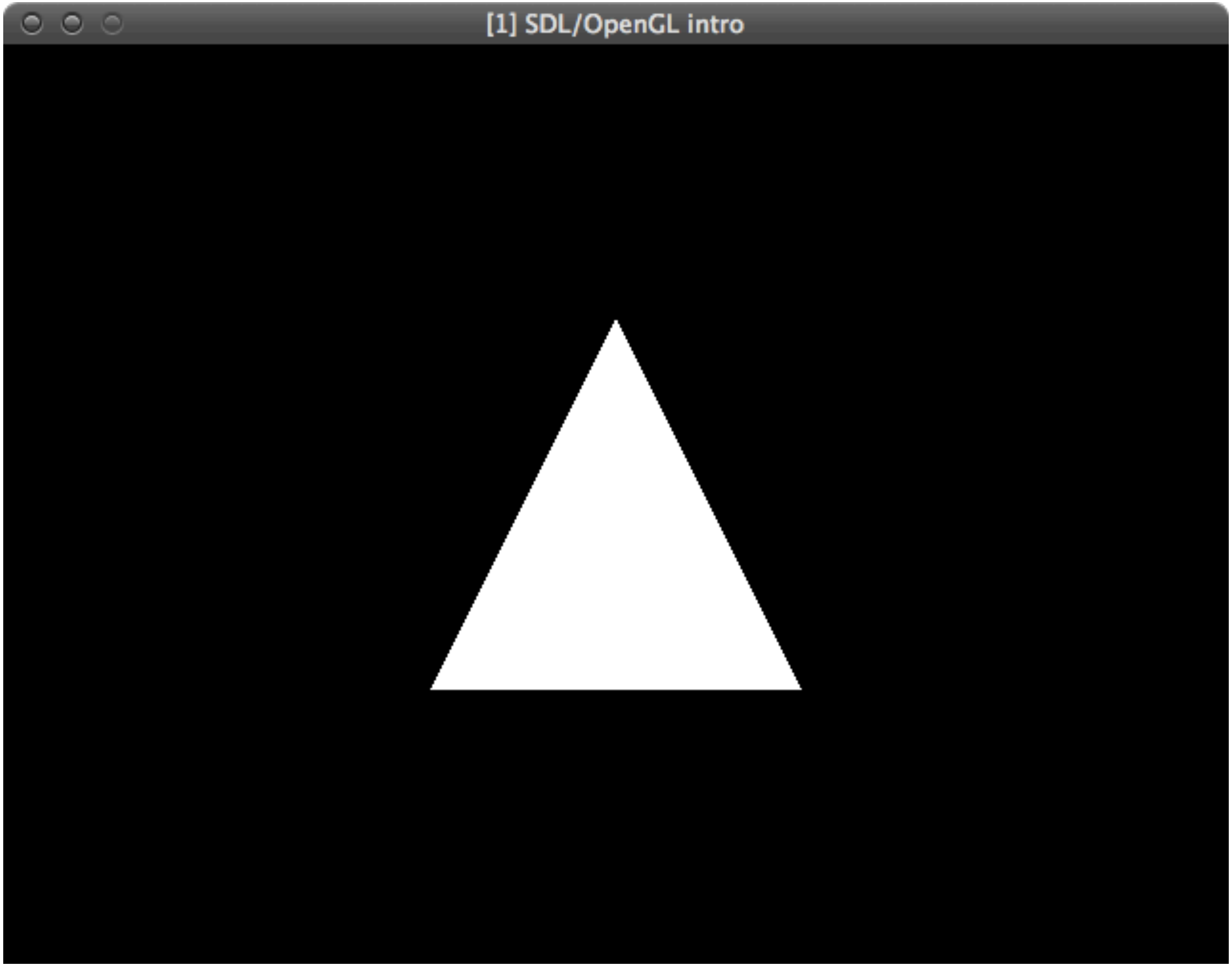
tricolor.cpp

so let's do some 3D drawing


SDL/OpenGL intro


(0,4,0)
(−1,2,−1)
(1,2,−1)
(−1,2,1)
(1,2,1)
(−1,0,−1)
(1,0,−1)
(−1,0,1)
(1,0,1)

start with framework
from last example


[1] SDL/OpenGL intro

drawing the first quad

```
void mydisplay()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glBegin(GL_TRIANGLES);
  glVertex3f( 0.0f, 1.0f, 0.0f);
  glVertex3f( 1.0f,-1.0f, 0.0f);
  glVertex3f(-1.0f,-1.0f, 0.0f);
  glEnd();

  SDL_GL_SwapBuffers();
}
```

drawing the first quad

```
void mydisplay()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);









  SDL_GL_SwapBuffers();
}
```

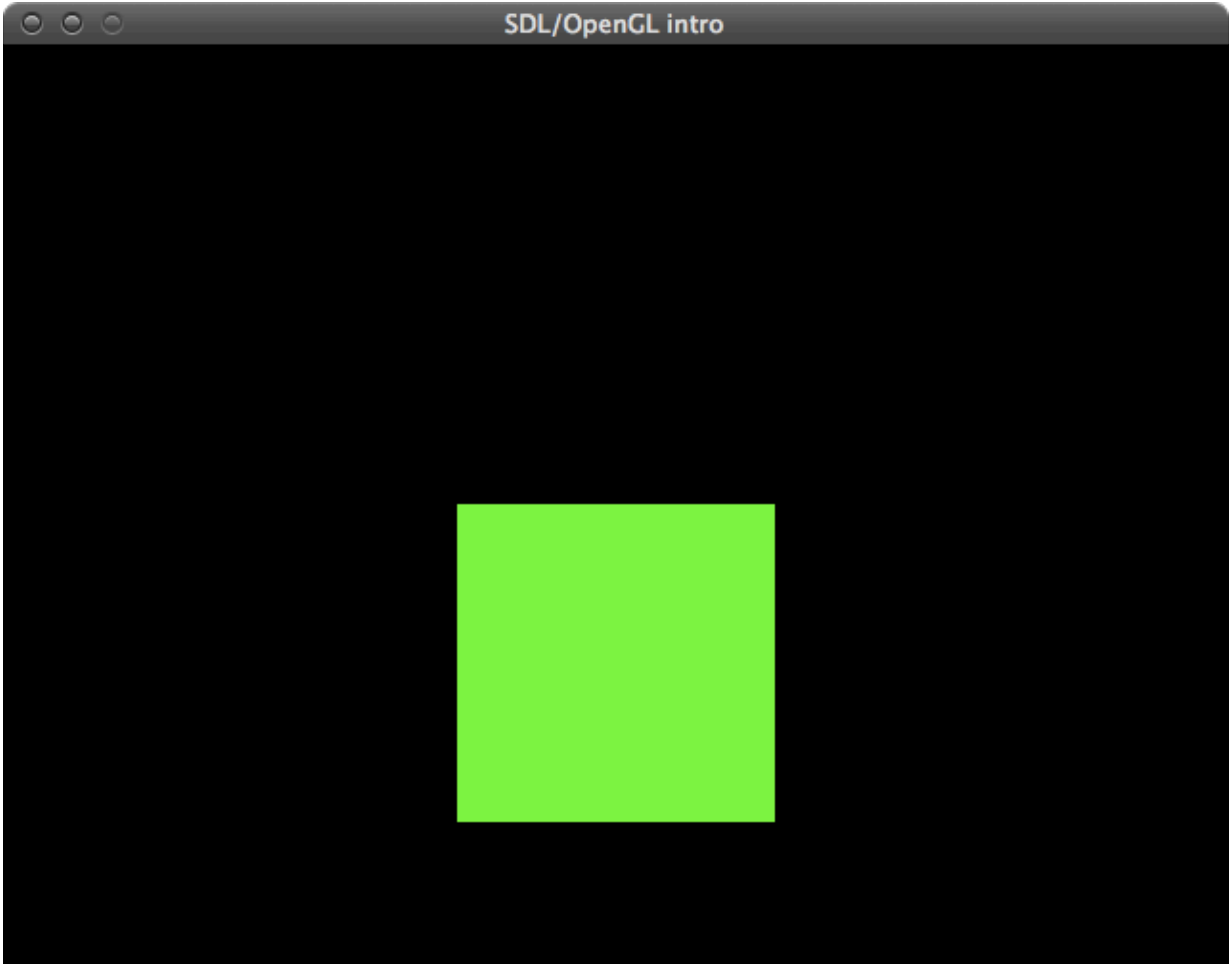
drawing the first quad

```
void mydisplay()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glBegin(GL_QUADS);
  // front
  glColor3f(0, 1, 0);
  glVertex3f(-1, 0, 1);
  glVertex3f(-1, 2, 1);
  glVertex3f(1, 2, 1);
  glVertex3f(1, 0, 1);
  glEnd();

  SDL_GL_SwapBuffers();
}
```

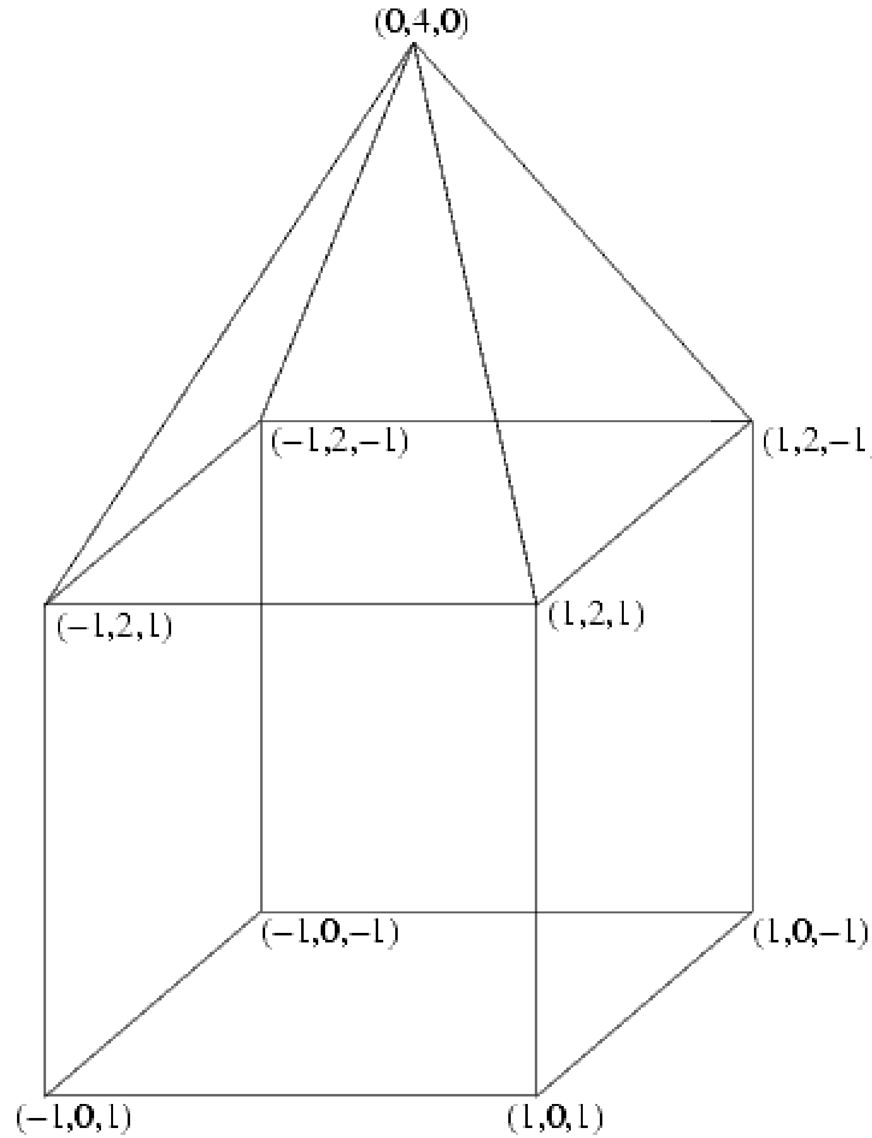
compile and run

drawing the first quad

```
void myinit(int width, int height)
{
  glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
  glViewport(0, 0, width, height);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(45.0, (float)width/(float)height, 0.1, 100.0);
  gluLookAt(0.0, 2.0, 8.0,  // eye
            0.0, 2.0, -1.0, // center
            0.0, 1.0, 0.0); // up
  glMatrixMode(GL_MODELVIEW);
}
```

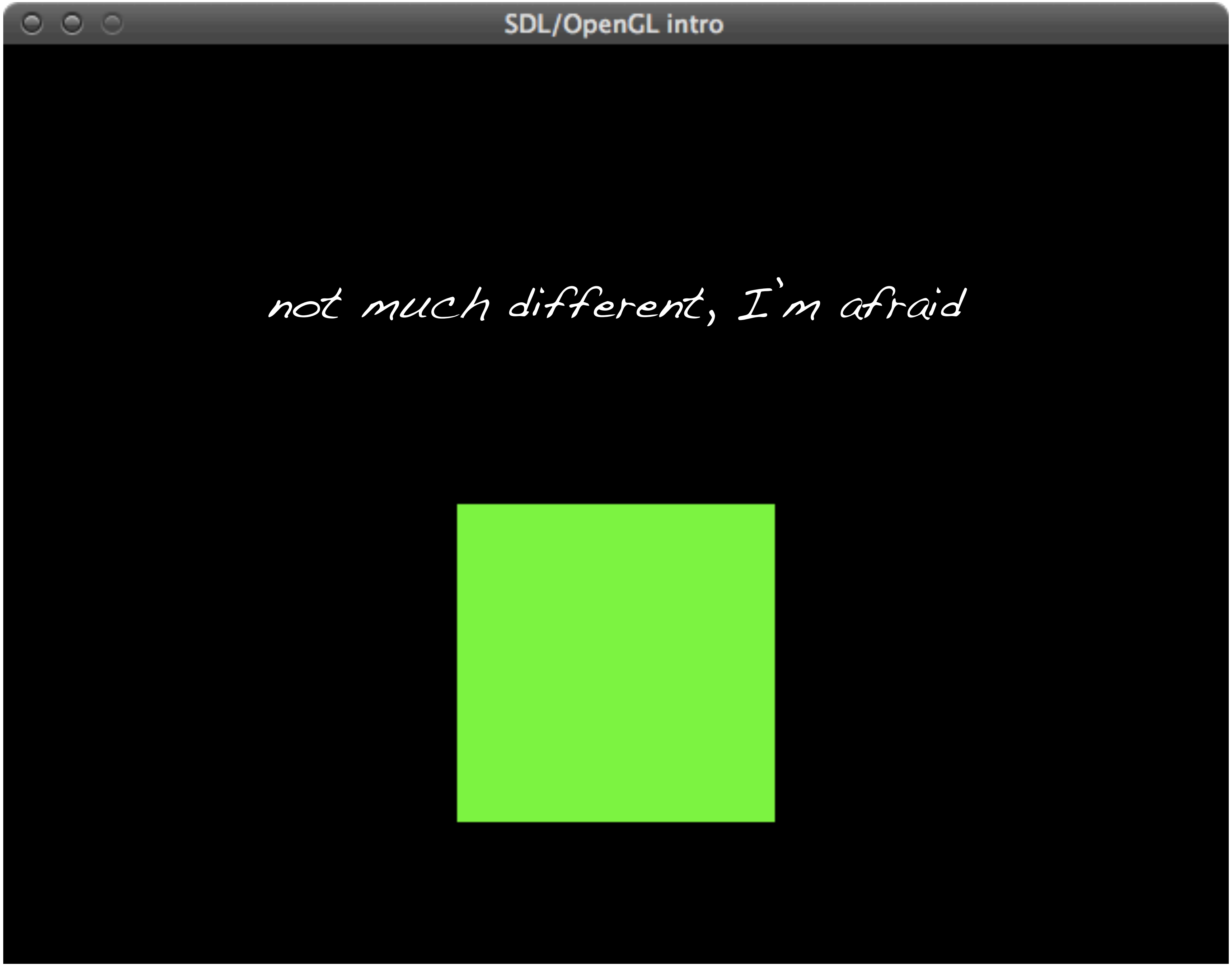

SDL/OpenGL intro

drawing the remaining quads
is trivial and left as an exercise to the
student


(0,4,0)
(−1,2,−1)
(1,2,−1)
(−1,2,1)
(1,2,1)
(−1,0,−1)
(1,0,−1)
(−1,0,1)
(1,0,1)

drawing the other quads

```
// back
glVertex3f(-1, 0, -1);
glVertex3f( 1, 0, -1);
glVertex3f( 1, 2, -1);
glVertex3f(-1, 2, -1);

// left
glVertex3f(-1, 0,  1);
glVertex3f(-1, 2,  1);
glVertex3f(-1, 2, -1);
glVertex3f(-1, 0, -1);

// right
glVertex3f(1, 0,  1);
glVertex3f(1, 0, -1);
glVertex3f(1, 2, -1);
glVertex3f(1, 2,  1);
```


SDL/OpenGL intro
not much different, I'm afraid

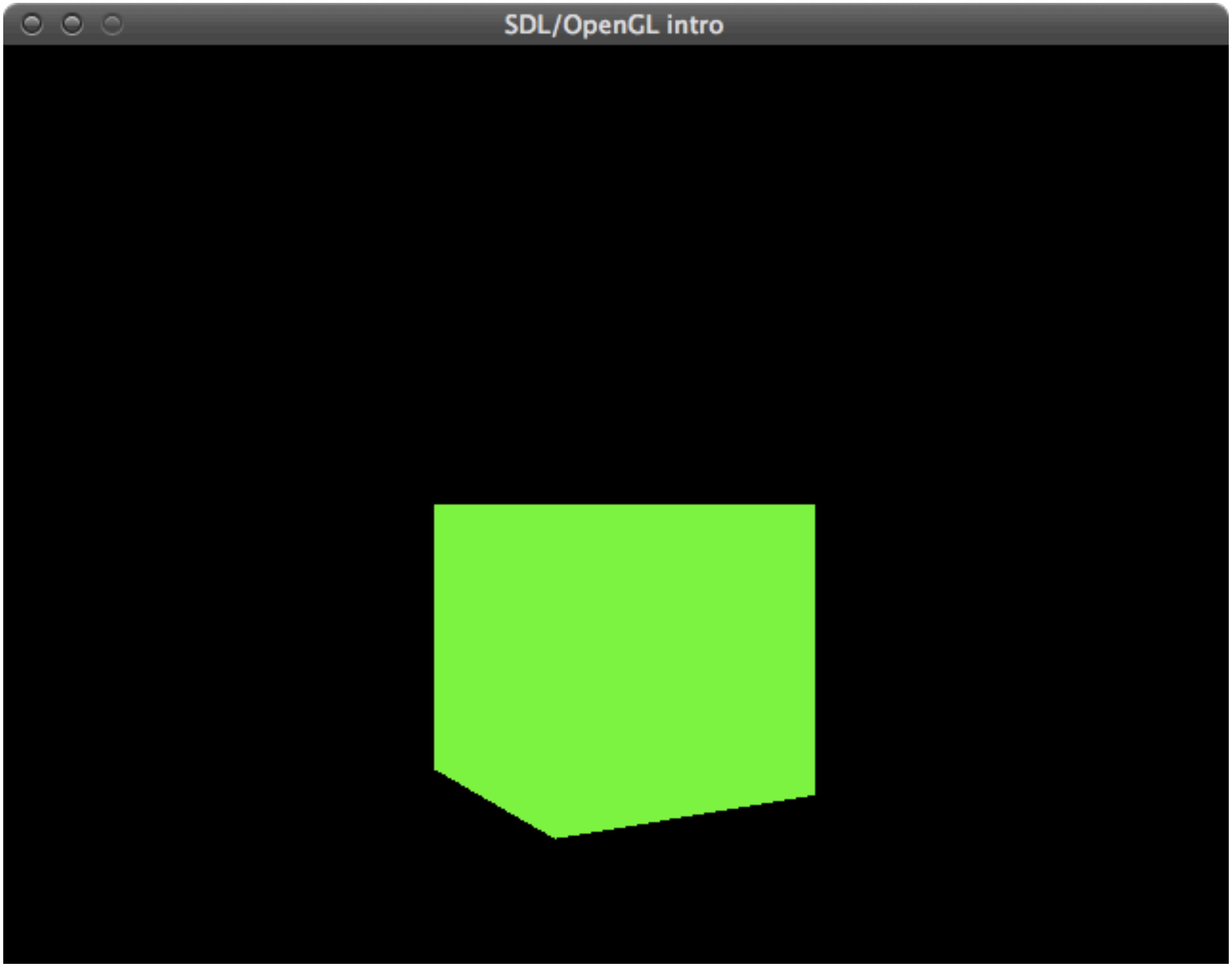
just a question of perspective

rotating the scene

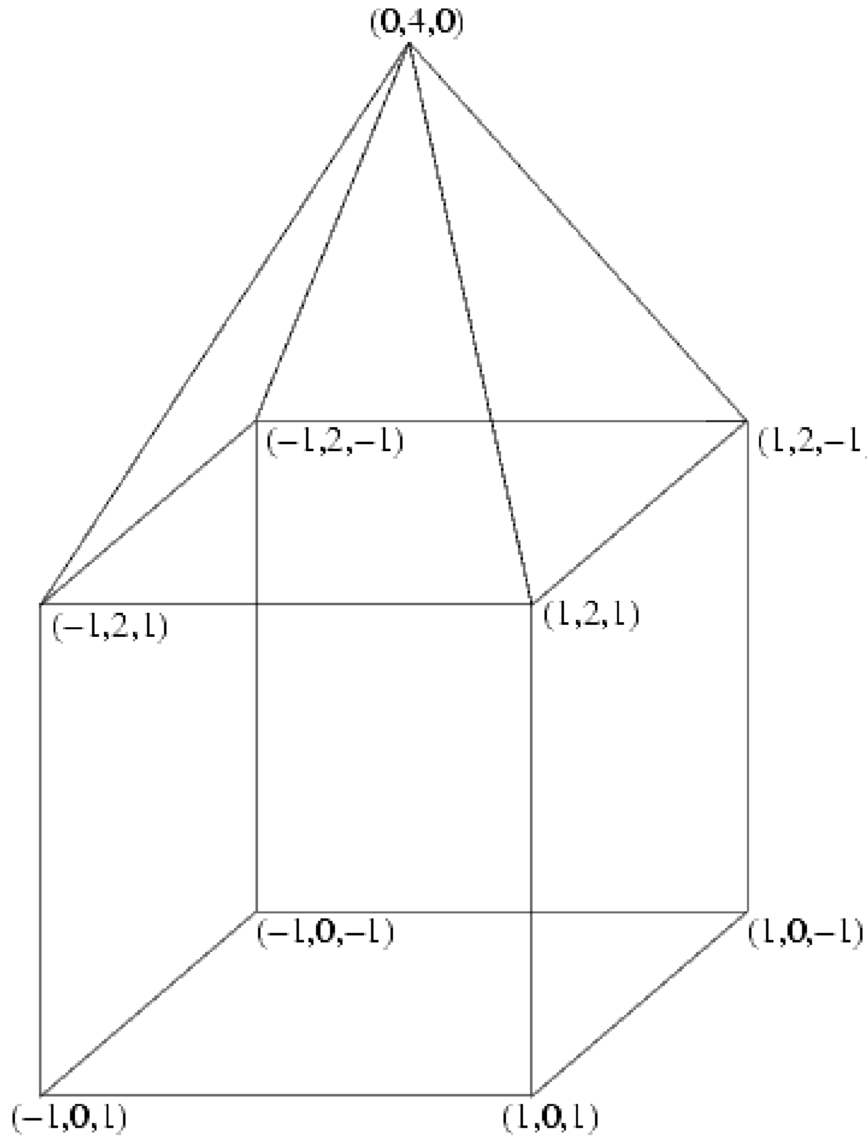
```
void mydisplay()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glRotatef(rotation, 0, 1, 0);

  glBegin(GL_QUADS);
  // front
  glColor3f(0, 1, 0);
  glVertex3f(-1, 0, 1);
  glVertex3f(-1, 2, 1);
  glVertex3f(1, 2, 1);
  glVertex3f(1, 0, 1);
  glEnd();

  SDL_GL_SwapBuffers();
}
```

rotating the scene

```
// in main()

while (!done) {
  mydisplay();
  SDL_Event event;
  while (SDL_PollEvent(&event)) {
    if (event.type == SDL_QUIT) done = true;
    if (event.type == SDL_KEYDOWN) {
      switch(event.key.keysym.sym) {
      case SDLK_ESCAPE:
        done = true;
        break;
      case SDLK_r:
        rotation = (rotation + 5) % 360;
        break;
      }
    }
  }
}
```

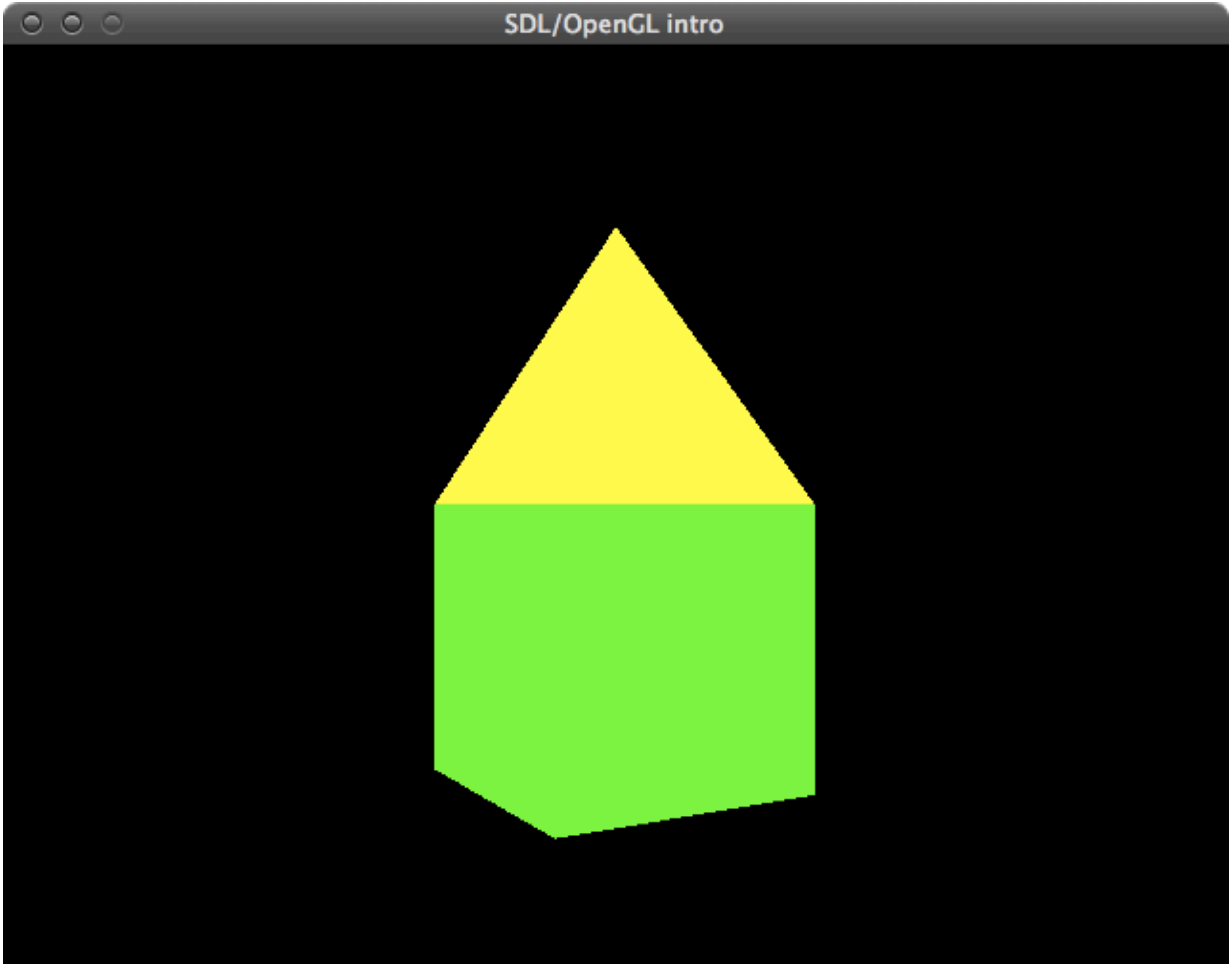

SDL/OpenGL intro

now for the pyramid...


(0,4,0)
(−1,2,−1)
(1,2,−1)
(−1,2,1)
(1,2,1)
(−1,0,−1)
(1,0,−1)
(−1,0,1)
(1,0,1)

drawing the pyramid

```
glBegin(GL_TRIANGLES);

  // front
  glColor3f(1, 1, 0);
  glVertex3f(-1, 2, 1);
  glVertex3f( 0, 4, 0);
  glVertex3f( 1, 2, 1);

  // right
  glVertex3f(1, 2,  1);
  glVertex3f(1, 2, -1);
  glVertex3f(0, 4,  0);

  // back
  glVertex3f( 1, 2, -1);
  glVertex3f(-1, 2, -1);
  glVertex3f( 0, 4,  0);

  // left
  glVertex3f(-1, 2,  1);
  glVertex3f( 0, 4,  0);
  glVertex3f(-1, 2, -1);

glEnd();
```



3ddrawing.cpp

a few words on 3D transformations

transformations overview

- OpenGL uses 4x4 matrices for modeling transformations.
  - Why not 3x3?
  - You don't want to know... (But I will tell you anyway.)
- Convenience functions for many operations:
  - **`glRotate*(), glTranslate*(), glScale*()`**
- Effects of transformations can be localized
  - **`glPushMatrix(), glPopMatrix()`**

manipulating the matrix stack

- **`glPushMatrix()`**
  - push all matrices in the current stack (determined by glMatrixMode()) down one level (the topmost matrix is duplicated)
- **`glPopMatrix()`**
  - pop the top matrix off the stack. The second matrix from the top of the stack becomes top, the contents of the popped matrix are destroyed.

OpenGL modelview matrix

- 4x4 matrix
- OpenGL uses column vectors instead of row vectors
- Matrices in OpenGL are defined like this:

$$M = \begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix}$$

model transformations in OpenGL

- 3 modeling transformations
  - **`glTranslate*()`**
  - **`glRotate*()`**
  - **`glScale*()`**
- Multiply a proper matrix for transform/rotate/scale to the current matrix and load the resulting matrix as current matrix.

maths alert

`glScalef(a,b,c)`

- $x_1 = ax_0$; $y_1 = by_0$; $z_1 = cz_0$
- How can we write this in matrix form?

$$\begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} = \begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix} = \begin{bmatrix} ax_0 \\ by_0 \\ cz_0 \end{bmatrix}$$

- Thus the scaling matrix is

$$S = \begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix}$$

`glRotatef(a, x, y, z)`

- Similarly for rotation we have:
  - glRotatef(a, 1, 0, 0): $\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos a & -\sin a \\ 0 & \sin a & \cos a \end{bmatrix}$
  - glRotatef(a, 0, 1, 0): $\begin{bmatrix} \cos a & 0 & \sin a \\ 0 & 1 & 0 \\ -\sin a & 0 & \cos a \end{bmatrix}$
  - glRotatef(a, 0, 0, 1): $\begin{bmatrix} \cos a & -\sin a & 0 \\ \sin a & \cos a & 0 \\ 0 & 0 & 1 \end{bmatrix}$

- **`glTranslatef(x,y,z)`**

- How is a translation defined?
- $x_1 = x_0 + x$
  $y_1 = y_0 + y$
  $z_1 = z_0 + z$

!! This is a problem !!

There is no way to represent this
as a multiplication of 3x3 matrices

- `glTranslatef(x,y,z)`

- Where there's a will, there's a workaround.
- Use 4x4 matrices!

$$T = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

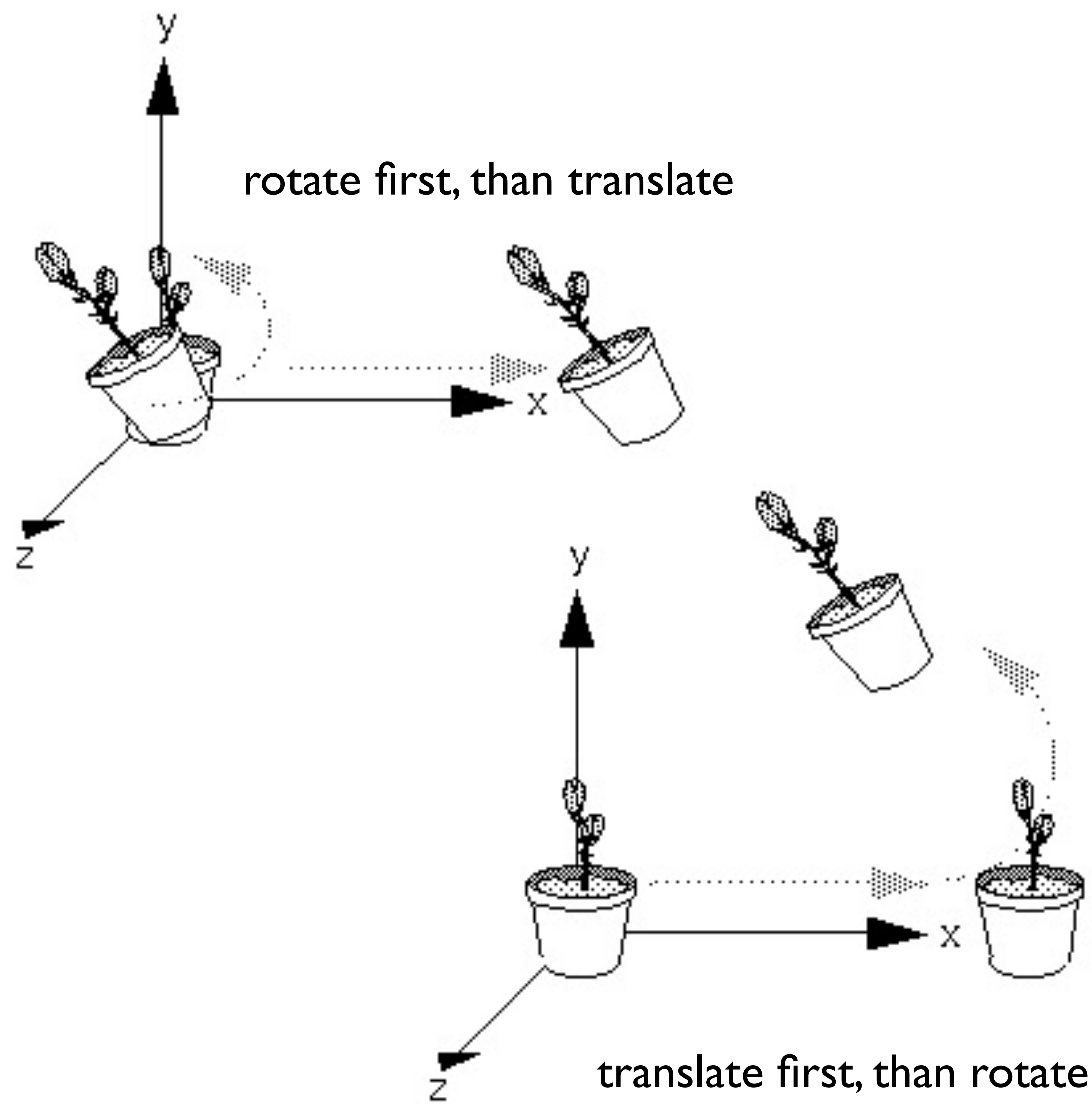
- This actually gives us the correct results:

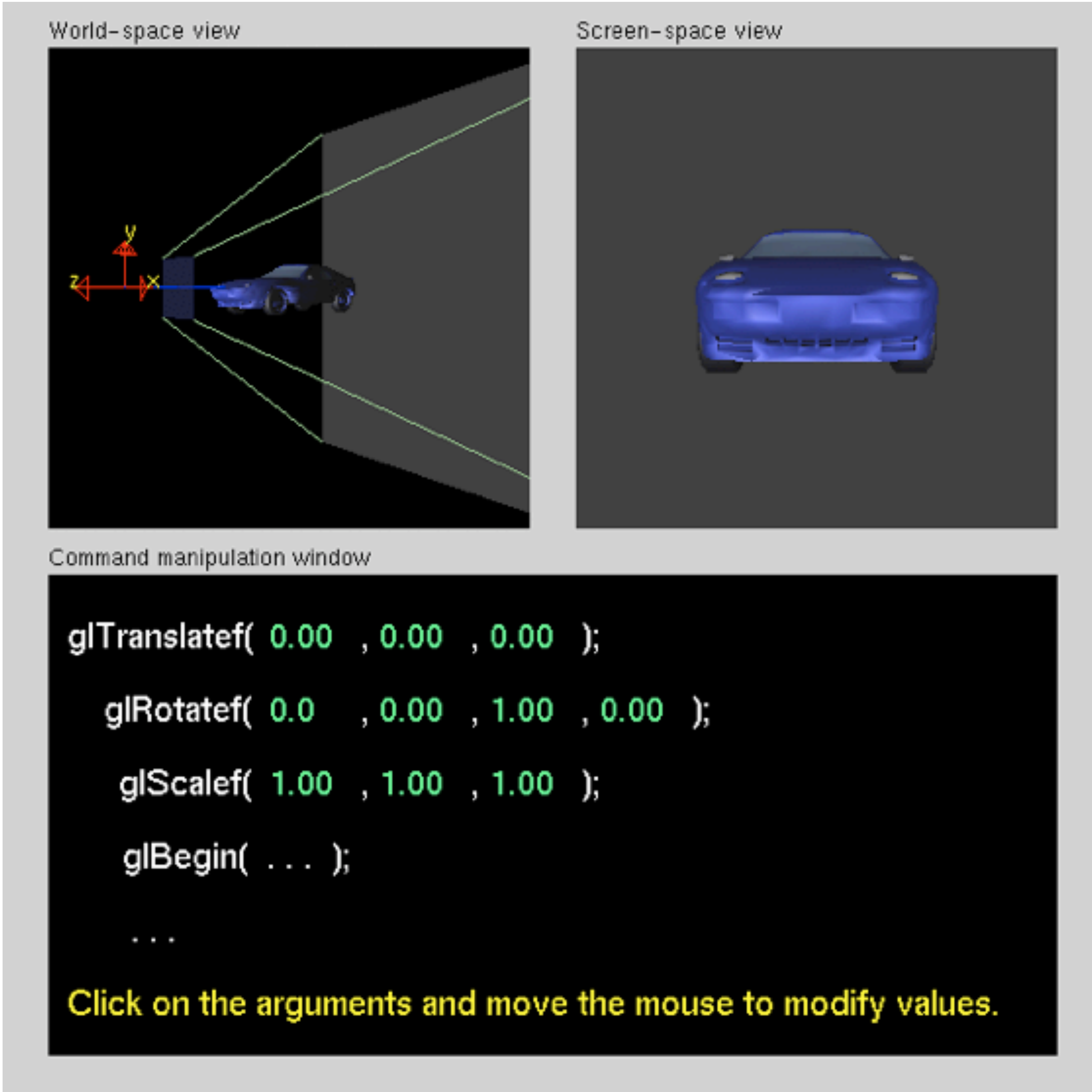
$$\begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{bmatrix} = \begin{bmatrix} x_0+x \\ y_0+y \\ z_0+z \\ 1 \end{bmatrix}$$

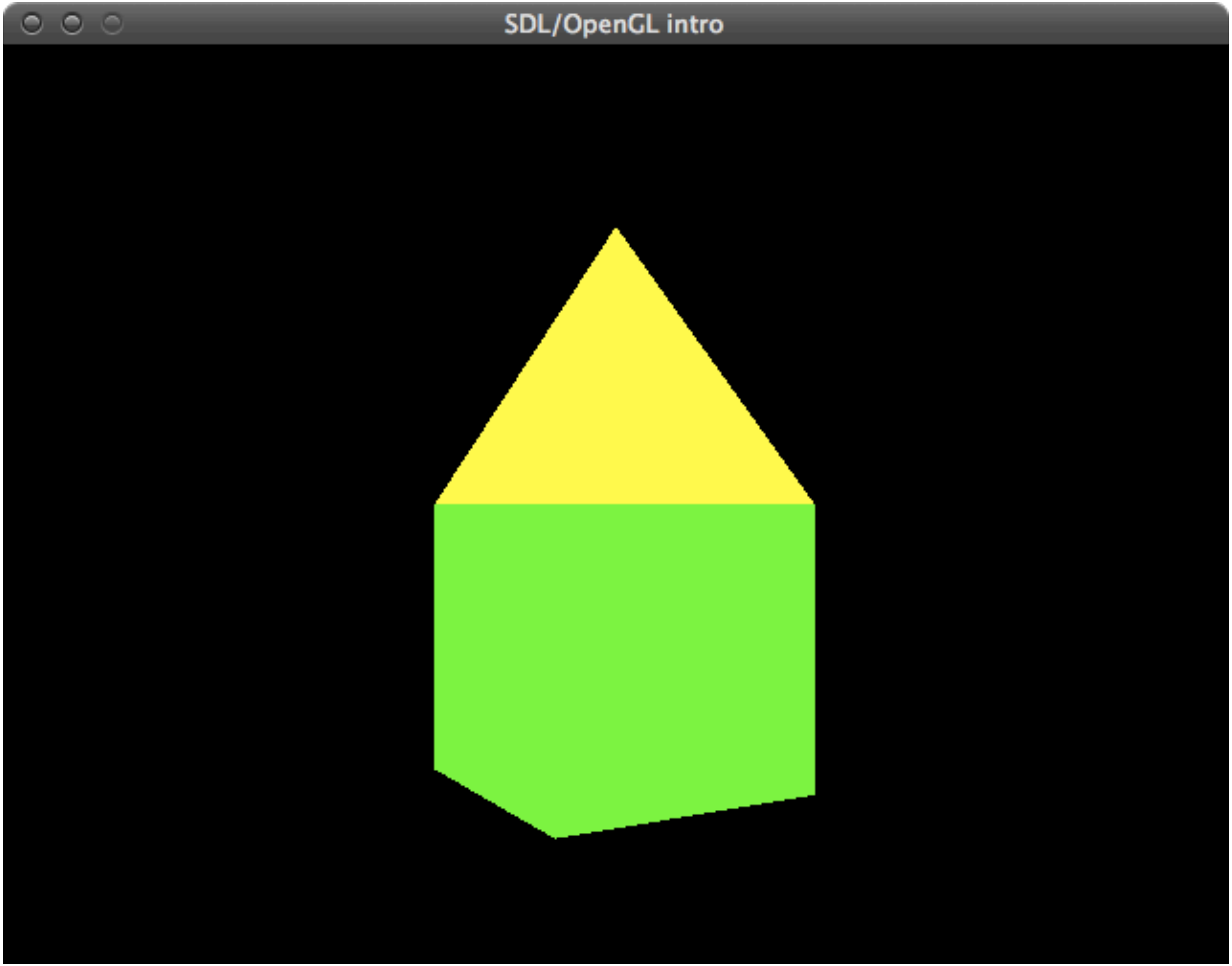
you can open your eyes again

order of transformations

- Matrix multiplication is not commutative.
  - The order of operations is important!
  - Example: Rotation and translation




World-space view
y
z
x
Screen-space view
Command manipulation window
glTranslatef(0.00 , 0.00 , 0.00);
glRotatef(0.0 , 0.00 , 1.00 , 0.00);
glScalef(1.00 , 1.00 , 1.00);
glBegin(. . .);
. . .
Click on the arguments and move the mouse to modify values.


SDL/OpenGL intro