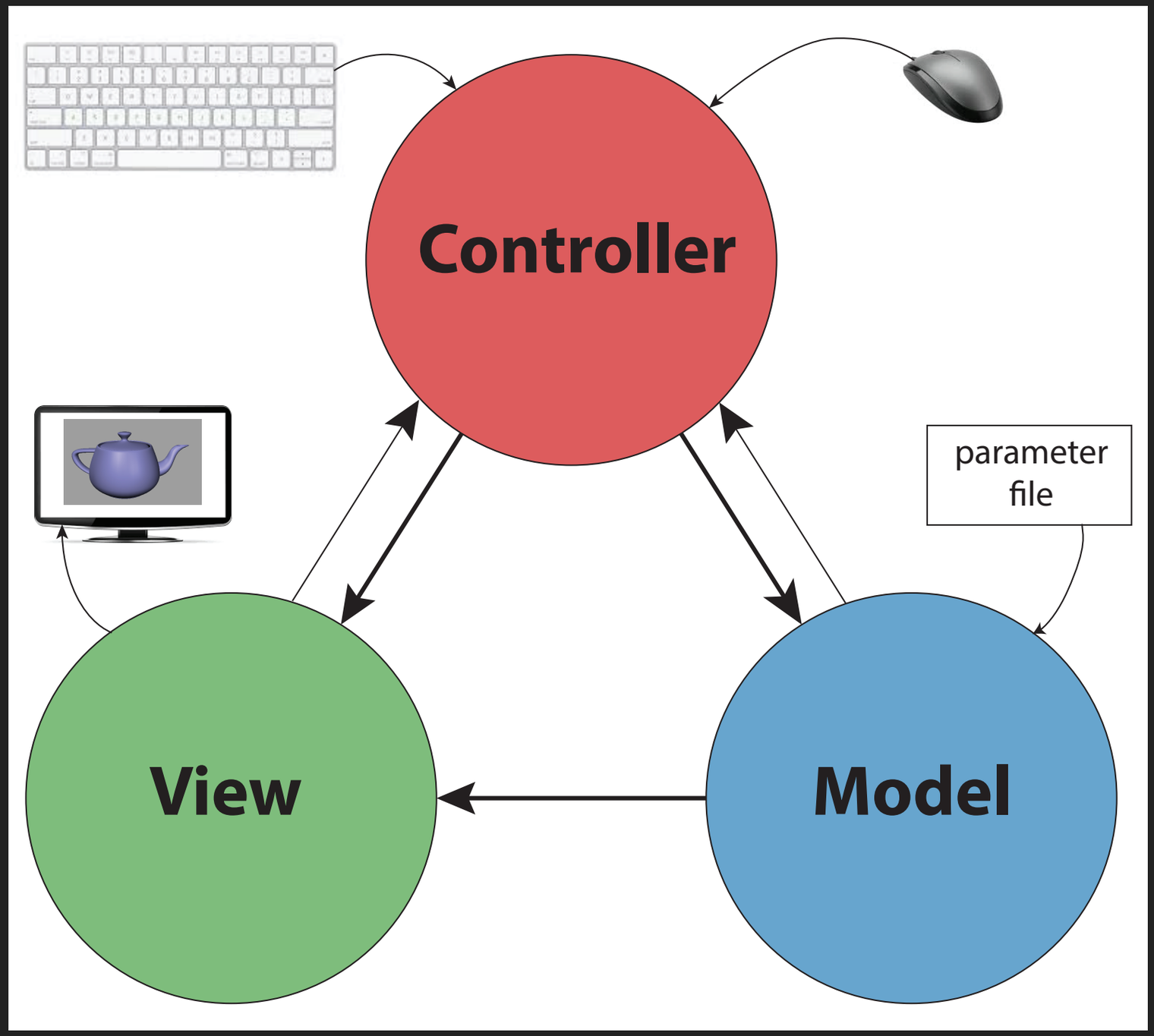
DONALD HOUSE: CPSC 8170, FALL 2018

DESIGN OF A REAL-TIME ANIMATION PROGRAM

DEMO

canonical.cpp

- The main program canonical.cpp acts as the controller, directing Model and View actions
- Keyboard events directed to Model and View
- Display and reshape events directed to View
- Idle events are directed to Model
- Controller gets display interval from Model



MAIN PROGRAM TO SET UP CONTROLLER, MODEL, AND VIEW

canonical.cpp

```
int main(int argc, char* argv[]){

  // make sure we have exactly one parameter
  if(argc != 2){
    cerr << "usage: canoncial paramfilename" << endl;
    return 1;
  }
  paramfilename = argv[1];

  // start up the glut utilities
  glutInit(&argc, argv);

  // create the graphics window, giving width, height, and title text
  // and establish double buffering, RGBA color
  // Depth buffering must be available for drawing the shaded model
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
  glutInitWindowSize(TeapotView.getWidth(), TeapotView.getHeight());
  glutCreateWindow("Canonical 3D Animation Example");

  // register callback to handle events
  glutDisplayFunc(doDisplay);
  glutReshapeFunc(doReshape);
  glutKeyboardFunc(handleKey);
  glutMouseFunc(handleButtons);
  glutMotionFunc(handleMotion);

  // idle function is called whenever there are no other events to process
  glutIdleFunc(doSimulation);

  // set up the camera viewpoint, materials, and lights
  TeapotView.setInitialView();

  // load parameters and initialize the bubble model
  BubbleModel.loadParameters(paramfilename);
  BubbleModel.initSimulation();

  glutMainLoop();
}
```

canonical.cpp

```
//
// idle callback: let the Model handle simulation time step events
//
void doSimulation(){
  static int count = 0;

  BubbleModel.timeStep();

  // only update display after every displayInterval time steps
  if(count == 0)
    glutPostRedisplay();

  count = (count + 1) % BubbleModel.displayInterval();
}
```

CONTROLLER KEYBOARD EVENT HANDLER

canonical.cpp

```
//
// Keyboard callback routine.
// Send model and view commands based on key presses
//
void handleKey(unsigned char key, int x, int y){
  const int ESC = 27;

  switch(key){
    case 'b':             // start a bubble blowing
      BubbleModel.loadParameters(paramfilename); // reload parameters
      BubbleModel.initSimulation();     // reinitialize the simulation
      BubbleModel.startBubble();        // start the action
      break;

    case 'k':           // toggle key light on and off
      TeapotView.toggleKeyLight();
      break;

    case 'f':           // toggle fill light on and off
      TeapotView.toggleFillLight();
      break;

    case 'r':           // toggle back light on and off
      TeapotView.toggleBackLight();
      break;

    case 'i':     // I -- reinitialize view
    case 'I':
      TeapotView.setInitialView();
      break;

    case 'q':     // Q or Esc -- exit program
    case 'Q':
    case ESC:
      exit(0);
  }

  // always refresh the display after a key press
  glutPostRedisplay();
}
```

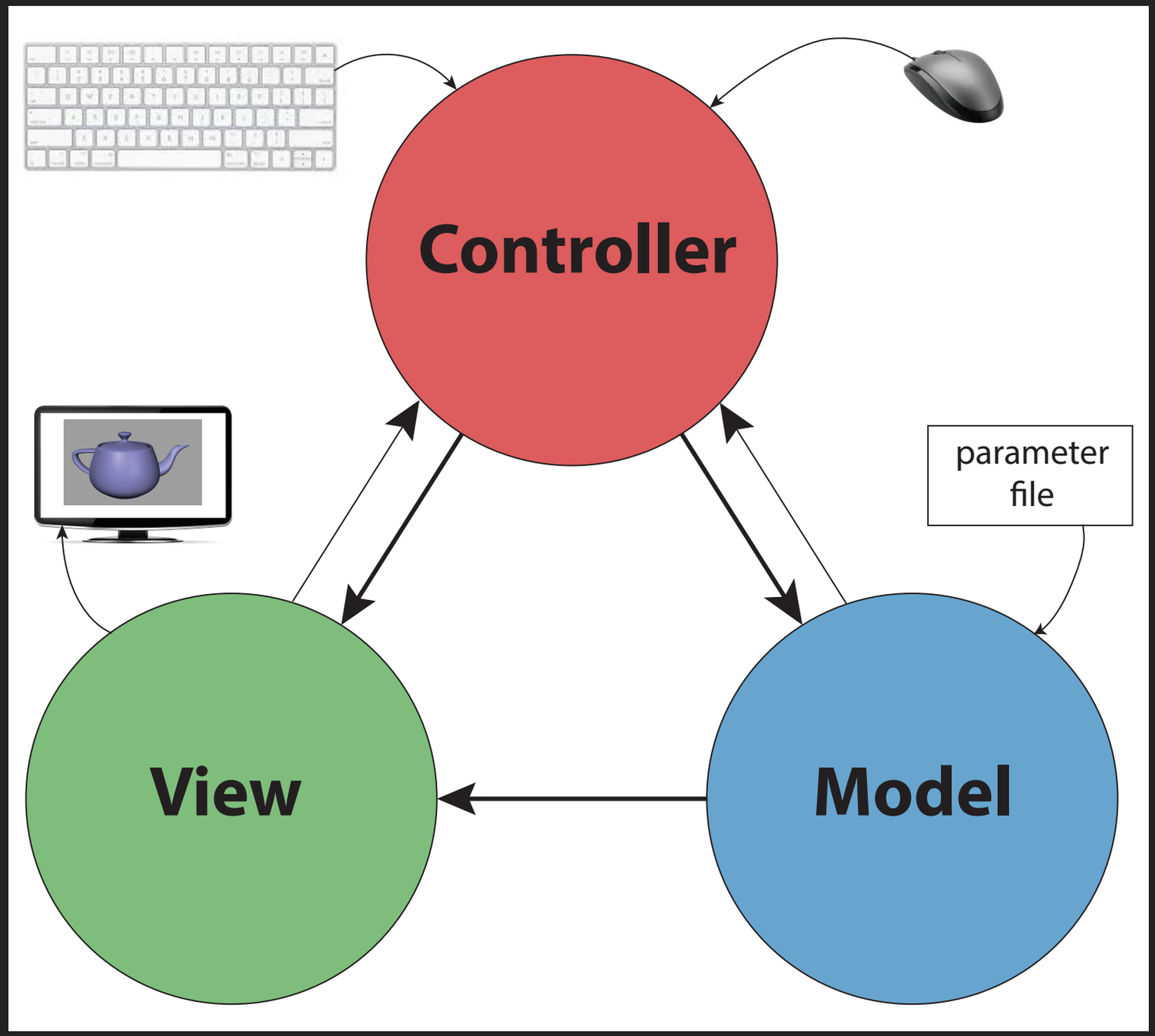
CONTROLLER MOUSE AND WINDOW EVENT HANDLERS

canonical.cpp

```
//
// let the View handle mouse button events
// but pass along the state of the shift key also
//
void handleButtons(int button, int state, int x, int y) {
  bool shiftkey = (glutGetModifiers() == GLUT_ACTIVE_SHIFT);

  TeapotView.handleButtons(button, state, x, y, shiftkey);
  glutPostRedisplay();
}

//
// let the View handle mouse motion events
//
void handleMotion(int x, int y) {
  TeapotView.handleMotion(x, y);
  glutPostRedisplay();
}

//
// let the View handle display events
//
void doDisplay(){
  TeapotView.updateDisplay();
}

//
// let the View handle reshape events
//
void doReshape(int width, int height){
  TeapotView.reshapeWindow(width, height);
}
```

Model

- Maintains model state
- Loads simulation parameters from file
- Sets simulation initial conditions
- Does numerical integration for each time step
- Passes model state to View



MODEL METHODS AND STATE

Model

```
private:
   // bubble simulation parameters
   float speed;
   float mass;
   float drag;
   float buoyancy;

   float h;
   int dispinterval;

   // State of bubble
   Vector3d BubblePos;
   Vector3d BubbleVel;
   bool Rising;
```

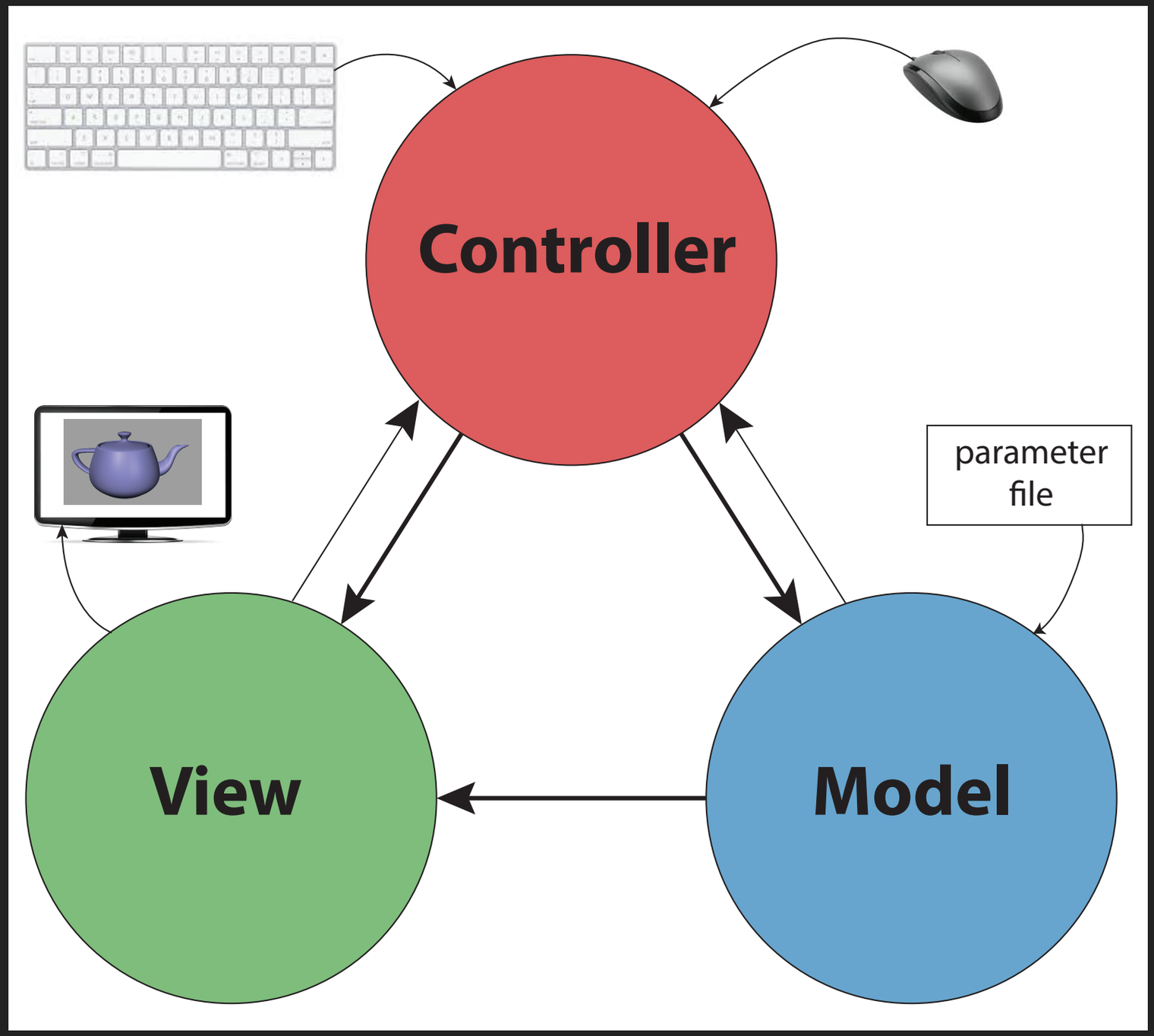
```
public:
  Model();

  bool loadParameters(const char *parmfilename); // get simulation parameters

  void initSimulation();    // initialize bubble to initial position and velocity
  void timeStep();          // take one time step
  void startBubble();       // make the bubble active

  // accessors to retrieve bubble state
  Vector3d bubblePosition(){return BubblePos;}
  Vector3d bubbleVelocity(){return BubbleVel;}
  bool isRising(){return Rising;}

  // accessor to retrieve display interval for controller
  int displayInterval(){return dispinterval;}
```

Model

```
// restore the bubble simulation to its initial state
void Model::initSimulation(){
  // return the bubble to its starting position and velocity
  BubblePos.set(0.0, 0.0, 0.0);
  // start bubble moving 30 degrees to right
  BubbleVel.set(0.866 * speed, 0.5 * speed, 0.0);
  Rising = false;
}

// this does one time step in the simulation
void Model::timeStep(){
  const Vector3d Buoyancy(0, buoyancy, 0);  // buoyancy force up

  // If bubble is rising, do modified Euler update of state
  if(Rising){
    Vector3d newBubbleVel =
         BubbleVel + h * (-drag / mass * BubbleVel + Buoyancy);
    BubblePos = BubblePos + h * (newBubbleVel + BubbleVel) / 2;
    BubbleVel = newBubbleVel;
  }
}

// start the bubble
void Model::startBubble(){
  Rising = true;
}
```

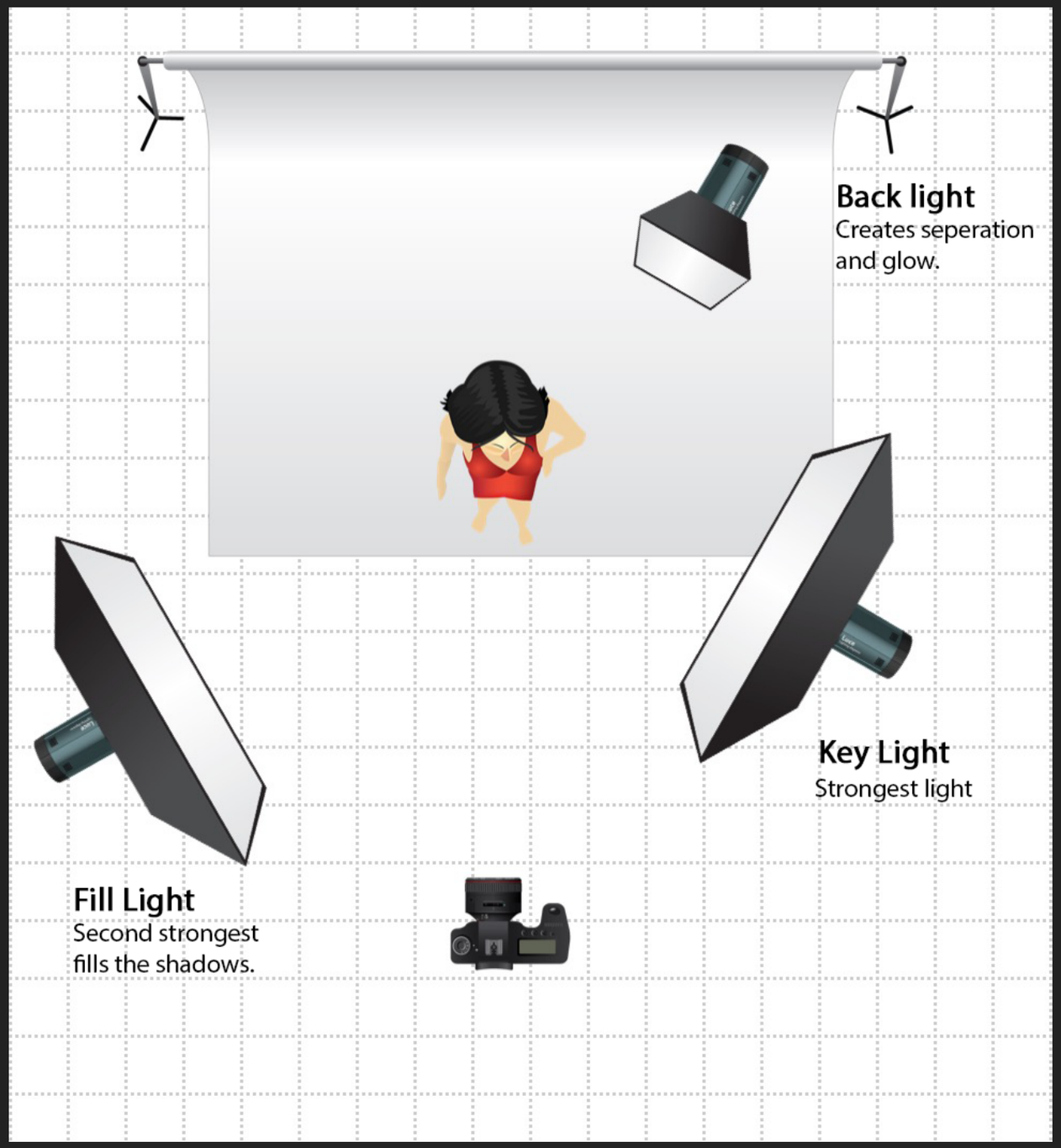
View

- Specifies materials and shading model
- Specifies camera viewing parameters
- Specifies geometry and positions geometric models and lights
- Handles camera interaction
- Handles window reshape
- Renders the scene



View

- Key Light is main light source
- Fill Light provides some illumination in Key Light shadows
- Back Light fills from behind, and provides rim glow



lights in code are mirror image of this diagram

DEMO

VIEW METHODS AND STATE

View

```
 // Switches to turn lights on and off
 bool KeyOn;
 bool FillOn;
 bool BackOn;

 // Current window dimensions
 int Width;
 int Height;

 // position the lights, never called outside of this class
 void setLights();

// draw the model, never called outside of this class
 void drawModel();


public:
 View(Model *model = NULL);

// initialize the state of the viewer to start-up defaults
void setInitialView();

// Toggle lights on/off
void toggleKeyLight();
void toggleFillLight();
void toggleBackLight();

// Handlers for mouse events
void handleButtons(int button, int state, int x, int y, bool shiftkey);
void handleMotion(int x, int y);

// redraw the display
void updateDisplay();

// handle window resizing, to keep consistent viewscreen proportions
void reshapeWindow(int width, int height);

// accessors to determine current screen width and height
int getWidth(){return Width;}
int getHeight(){return Height;}
```

View

```
View::View(Model *model):
  camera(NULL), themodel(model),
  width(WIDTH), height(HEIGHT),
  near(NEAR), far(FAR), fov(FOV),
  modelsize(MODELSIZE), modeldepth(MODELDEPTH),
  diffuse_fraction(DIFFUSE_FRACTION), specular_fraction(SPECULAR_FRACTION),
  shininess(SHININESS),
  white{WHITE}, dim_white{DIM_WHITE}, grey_background{GREY_BACKGROUND},
  base_color{BASE_COLOR}, highlight_color{HIGHLIGHT_COLOR}{

  // Set up camera: parameters are eye point, aim point, up vector,
  // near and far clip plane distances, and camera vertical FOV in degrees
  camera = new Camera(Vector3d(0, 0, modeldepth), Vector3d(0, 0, 0),
                      Vector3d(0, 1, 0),near, far, fov);

  // point to the model
  themodel = model;

  // initialize current window dimensions to match default
  Width = width;
  Height = height;
}
```

VIEW INITIALIZATION METHOD: SHADING, DEPTH TESTING, AND LIGHTS

View

```
void View::setInitialView(){
  // return camera to its default settings
  camera->Reset();

  // opaque grey background for window
  glClearColor(grey_background[0], grey_background[1],
               grey_background[2], grey_background[3]);

  // smooth shade across triangles if vertex normals are present
  glShadeModel(GL_SMOOTH);

  // make sure that all surface normal vectors are unit vectors
  glEnable(GL_NORMALIZE);

  // enable depth testing for hidden surfaces
  glEnable(GL_DEPTH_TEST);
  glDepthRange(0.0, 1.0);
```

```
  // set the colors of the key, fill, and back lights
  glLightfv(GL_LIGHT0, GL_DIFFUSE, white);
  glLightfv(GL_LIGHT0, GL_SPECULAR, white);

  glLightfv(GL_LIGHT1, GL_DIFFUSE, dim_white);
  glLightfv(GL_LIGHT1, GL_SPECULAR, dim_white);

  glLightfv(GL_LIGHT2, GL_DIFFUSE, dim_white);
  glLightfv(GL_LIGHT2, GL_SPECULAR, dim_white);

  // turn on lighting
  glEnable(GL_LIGHT0);      // key light
  KeyOn = true;
  glEnable(GL_LIGHT1);      // fill light
  FillOn = true;
  glEnable(GL_LIGHT2);      // back light
  BackOn = true;

  // turn on shading
  glEnable(GL_LIGHTING);
  // consider light position for specular
  glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);
```

View

```
// define the diffuse and specular colors of the teapot's
// material, and set its specular exponent
float diffuse_color[4], specular_color[4];
for(int i = 0; i < 3; i++){
  diffuse_color[i] = diffuse_fraction * base_color[i];
  specular_color[i] = specular_fraction * highlight_color[i];
}
diffuse_color[3] = specular_color[3] = 1;
```

```
  glMaterialfv(GL_FRONT, GL_DIFFUSE, diffuse_color);
  glMaterialfv(GL_FRONT, GL_SPECULAR, specular_color);
  glMaterialf(GL_FRONT, GL_SHININESS, shininess);
}
```

View

```
//
// Redraw the display, including the lights and teapot-bubble model
//
void View::updateDisplay(){
  // clear the window to the background color
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  // initialize modelview matrix
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();

  // lights are positioned in camera space so they move with camera
  setLights();

  // position and aim the camera in model view space
  camera->AimCamera();

  // draw the model
  drawModel();

  glutSwapBuffers();
}
```

View

```
//
// Position the 3 lights
//
void View::setLights(){
  // key is point light above and behind camera to the left
  const float key_light_position[4] = {-modeldepth / 2, modeldepth / 2,
                                       modeldepth / 2, 1};
  glLightfv(GL_LIGHT0, GL_POSITION, key_light_position);

  // fill is point light at eye level to right
  const float fill_light_position[4] = {modeldepth / 2, 0, 0, 1};
  glLightfv(GL_LIGHT1, GL_POSITION, fill_light_position);

  // back is parallel light coming from behind object, and above and to left
  const float back_light_direction[4] = {-2 * modeldepth, 2 * modeldepth,
                                         -2 * modeldepth, 0};
  glLightfv(GL_LIGHT2, GL_POSITION, back_light_direction);
}
```

View

```
// draw the teapot, and also the bubble if a bubble is rising
void View::drawModel(){
  // position of end of spout.
  // Note: these magic numbers were determined experimentally
  Vector3d BubbleOrigin(49 * modelsize / 32, modelsize / 2, 0);

  // Draw the teapot
  glutSolidTeapot(modelsize);

  // nothing to do if bubble not rising
  if(themodel->isRising()){
    // at its starting position, the bubble is in the teapot spout
    glTranslatef(BubbleOrigin.x, BubbleOrigin.y, BubbleOrigin.z);

    // use the bubble's current position to place it relative to spout
    Vector3d bubble = themodel->bubblePosition();
    glTranslatef(bubble.x, bubble.y, bubble.z);

    // draw the bubble
    glutSolidSphere(modelsize / 10, 36, 13);
  }
}
```

METHOD FOR UPDATING VIEWPORT AND CAMERA PARAMETERS ON A WINDOW RESHAPE EVENT

View

```
//
// When window resized, keep viewport proportions the same as the camera's
// viewscreen proportions to avoid distortion of scene
//
void View::reshapeWindow(int w, int h){
  float camaspect = float(width) / float(height);   // camera's aspect ratio
  float newaspect = float(w) / float(h);            // current window aspect
ratio
  float x0, y0;

  // tentatively set viewport dimensions to current window dimensions
  Width = w;
  Height = h;

  // correct Width or Height to match camera's aspect ratio
  if(newaspect > camaspect)
    Width = int(h * camaspect);
  else
    Height = int(w / camaspect);

  // offset viewport to keep it centered in the window
  x0 = (w - Width) / 2;
  y0 = (h - Height) / 2;

  // update the viewport
  glViewport(x0, y0, Width, Height);

  // set up camera projection matrix
  camera->PerspectiveDisplay(Width, Height);
}
```