Part 2 (now the fun really starts)

3D computer graphics with OpenGL

Karin Kosina (vka kyrah)
needful things for your toolbox

• fullscreen mode
• repeating key events
• animation using timers
• Mac OS X specific:
  • synchronizing SDL_GL_SwapBuffers() with the vertical refresh

See source code examples...
fullscreen mode

• simply add SDL_FULLSCREEN in SDL_SetVideoMode()
animation (basics)

```c
int main(int argc, char ** argv)
{
    // SDL and OpenGL setup code as usual

    // main application loop
    bool done = false;
    while (!done) {
        usleep(10000);
        mydisplay();
        SDL_Event event;
        while (SDL_PollEvent(&event)) {
            // ... as before
        }

    SDL_Quit();
    return 0;
}
```
animation using timers

• create and add a timer
  SDL_TimerID SDL_AddTimer(Uint32 interval,
                 SDL_NewTimerCallback callback,
                 void *param);

• define timer callback function
  typedef Uint32 (*SDL_NewTimerCallback)(Uint32 interval, void *param);

• in that callback function, create and send a user event
  SDL_Event event;
  event.type = SDL_USEREVENT;
  event.user.code = RUN_GAME_LOOP;
  SDL_PushEvent(&event);

• in your event processing loop, catch this event and call your display function
  if (event.type == SDL_USEREVENT)
      if (event.user.code == RUN_GAME_LOOP) {
          mydisplay();
      }
      
  }
animation using timers

int main(int argc, char ** argv)
{
    // SDL and OpenGL setup code as usual

    SDL_TimerID timer;
    timer = SDL_AddTimer(20, GameLoopTimer, NULL);

    bool done = false;
    while (!done) {
        SDL_Event event;
        while (SDL_PollEvent(&event)) {
            if (event.type == SDL_USEREVENT) {
                if (event.user.code == RUN_GAME_LOOP) {
                    mydisplay();
                }
            } else if (event.type == SDL_QUIT) {
                done = true;
            }
        }
    }

    SDL_RemoveTimer(timer);
    SDL_Quit();
    return 0;
}
int main(int argc, char ** argv)
{
    // SDL and OpenGL setup code as usual

    SDL_TimerID timer;
    timer = SDL_AddTimer(20, GameLoopTimer, NULL);

    bool done = false;
    while (!done) {
        SDL_Event event;
        while (SDL_PollEvent(&event)) {
            if (event.type == SDL_USEREVENT) {
                if (event.user.code == RUN_GAME_LOOP) {
                    mydisplay();
                }
            } else if (event.type == SDL_QUIT) {
                done = true;
            }
        }
    }

    SDL_RemoveTimer(timer);
    SDL_Quit();
    return 0;
}
animation using timers

```c
const int RUN_GAME_LOOP = 1;

Uint32 GameLoopTimer(Uint32 interval, void* param)
{
    // Create a user event to call the game loop.
    SDL_Event event;

    event.type = SDL_USEREVENT;
    event.user.code = RUN_GAME_LOOP;
    event.user.data1 = 0;
    event.user.data2 = 0;

    SDL_PushEvent(&event);
    return interval;
}
```
const int RUN_GAME_LOOP = 1;

Uint32 GameLoopTimer(Uint32 interval, void* param)
{
   // Create a user event to call the game loop.
   SDL_Event event;

   event.type = SDL_USEREVENT;
   event.user.code = RUN_GAME_LOOP;
   event.user.data1 = 0;
   event.user.data2 = 0;

   SDL_PushEvent(&event);
   return interval;
}
const int RUN_GAME_LOOP = 1;

Uint32 GameLoopTimer(Uint32 interval, void* param)
{
    // Create a user event to call the game loop.
    SDL_Event event;
    event.type = SDL_USEREVENT;
    event.user.code = RUN_GAME_LOOP;
    event.user.data1 = 0;
    event.user.data2 = 0;

    SDL_PushEvent(&event);
    return interval;
}
animation using timers

const int RUN_GAME_LOOP = 1;

Uint32 GameLoopTimer(Uint32 interval, void* param)
{
    // Create a user event to call the game loop.
    SDL_Event event;

    event.type = SDL_USEREVENT;
    event.user.code = RUN_GAME_LOOP;
    event.user.data1 = 0;
    event.user.data2 = 0;

    SDL_PushEvent(&event);
    return interval;
}
const int RUN_GAME_LOOP = 1;

Uint32 GameLoopTimer(Uint32 interval, void* param)
{
    // Create a user event to call the game loop.
    SDL_Event event;
    event.type = SDL_USEREVENT;
    event.user.code = RUN_GAME_LOOP;
    event.user.data1 = 0;
    event.user.data2 = 0;
    SDL_PushEvent(&event);
    return interval;
}
const int RUN_GAME_LOOP = 1;

Uint32 GameLoopTimer(Uint32 interval, void* param)
{
  // Create a user event to call the game loop.
  SDL_Event event;

  event.type = SDL_USEREVENT;
  event.user.code = RUN_GAME_LOOP;
  event.user.data1 = 0;
  event.user.data2 = 0;

  SDL_PushEvent(&event);
  return interval;
}
animation using timers

int main(int argc, char ** argv)
{
    // SDL and OpenGL setup code as usual

    SDL_TimerID timer;
    timer = SDL_AddTimer(20, GameLoopTimer, NULL);

    bool done = false;
    while (!done) {
        SDL_Event event;
        while (SDL_PollEvent(&event)) {
            if (event.type == SDL_USEREVENT) {
                if (event.user.code == RUN_GAME_LOOP) {
                    mydisplay();
                }
            } else if (event.type == SDL_QUIT) {
                done = true;
            }
        }
    }

    SDL_RemoveTimer(timer);
    SDL_Quit();
    return 0;
}
animation using timers

```c
int main(int argc, char ** argv)
{
    // SDL and OpenGL setup code as usual

    SDL_TimerID timer;
    timer = SDL_AddTimer(20, GameLoopTimer, NULL);

    bool done = false;
    while (!done) {
        SDL_Event event;
        while (SDL_PollEvent(&event)) {
            if (event.type == SDL_USEREVENT) {
                if (event.user.code == RUN_GAME_LOOP) {
                    mydisplay();
                }
            } else if (event.type == SDL_QUIT) {
                done = true;
            }
        }
    }

    SDL_RemoveTimer(timer);
    SDL_Quit();
    return 0;
}
```

"toolbox/animation_timer.cpp"
repeating key events

- interaction through key events so far:
  - increase translation/rotation value on key-down

- new and improved interaction through key events:
  - set movement flag on key-down
  - clear movement flag on key-up
  - update animation if movement flag is set
int rotation = 0;

void mydisplay()
{
    glPushMatrix();
    glRotatef(rotation, 0.0, 1.0, 0.0);
    // draw scene here
    glPopMatrix();
}

// in event processing loop
if (event.type == SDL_KEYDOWN) {
    switch(event.key.keysym.sym) {
    case SDLK_r:
        rotation = (rotation + 5) % 360;
        break;
    }
}

don't do this:

int rotation = 0;

void mydisplay()
{
    glPushMatrix();
    glRotatef(rotation, 0.0, 1.0, 0.0);
    // draw scene here
    glPopMatrix();
    }

// in event processing loop
if (event.type == SDL_KEYDOWN) {
    switch(event.key.keysym.sym) {
    case SDLK_r:
        rotation = (rotation + 5) % 360;
        break;
    }
}
int rotation = 0;
bool spinning = false;

void mydisplay()
{
    if (spinning) rotation = (rotation + 1) % 360;
    glPushMatrix();
    glRotatef ((float) rotation, 0.0, 1.0, 0.0);
    // draw scene here
    glPopMatrix();
}

// in event processing loop

if (event.type == SDL_KEYDOWN) {
    switch(event.key.keysym.sym) {
    case SDLK_RIGHT:
        spinning = true;
        break;
    }
}
else if (event.type == SDL_KEYUP) {
    switch(event.key.keysym.sym)
    case SDLK_RIGHT:
        spinning = false;
        break;
}
solar system example
lighting
what is light?

• particles or waves or neither or both...
• it’s complicated...
it's all a fake
light in OpenGL consists of

- ambient light
- scattered light (seemingly coming from all directions)
- diffuse light
  - light coming from one direction
  - scattered evenly when bouncing off a surface
- specular light ("shininess")
  - light coming from one direction
  - bounces off the surface in a preferred direction
- emitted light
  - originates from object – unaffected by light sources
Lambert's cosine law:

The brightness of a diffusely radiating plane surface is proportional to the cosine of the angle formed by the line of sight and the normal to the surface.

- Same intensity regardless if the viewers position.
- Used for diffuse lighting component in OpenGL.

\[
I_o = L_d * M_d * \cos(\theta)
\]

- \(I_o\) ... reflected intensity
- \(L_d\) ... the light's diffuse intensity
- \(M_d\) ... the material's diffuse coefficient
The Phong model:

The specular component is proportional to the cosine between the light reflection vector and the eye vector.

If the eye vector coincides with the reflection vector then we get the maximum specular intensity.

OpenGL uses a simplification of the Phong model: the Blinn-Phong model

$L$ is the vector from the light to the vertex being shaded.

$R$ is the vector $L$ mirror reflected on the surface.

$N$ is the normal vector, and $\text{Eye}$ is the vector from the vertex to the eye, or camera.

The specular component is proportional to the cosine of alpha.
The Blinn-Phong model:

The specular component is based on the cosine of the angle between the half vector and the normal.

Used for specular lighting component in OpenGL (the “bright spot”).

\[ Spec = (N \cdot H)^s \times L_s \times M_s \]

- \(s\) ... shininess value
- \(L_s\) ... the light's specular intensity
- \(M_s\) ... the material's specular coefficient

L is the vector from the light to the vertex being shaded.

H is the half vector, a vector with a direction half-way between the eye vector and the light vector.

N is the normal vector, and Eye is the vector from the vertex to the eye, or camera.

The specular component is proportional to the cosine of alpha.
shading models

in OpenGL:

• flat shading
• face normals (one color per polygon)

• gouraud shading
• vertex normals (one colour per vertex, interpolated over the polygon along edges and scanlines)

• phong shading
• interpolate vertex normals at each pixels (not just the colour values)

not implemented!
Flat shading vs. Gouraud shading

glShadeModel(GL_FLAT);

glShadeModel(GL_SMOOTH);
lighting step by step

• Define normal vectors for each vertex

• Normals determine the orientation of the object relative to the light source

• Create, select, and position one or more light sources.

• Select a lighting model.

• Define material properties for the objects in the scene.
void myinit(int width, int height)
{
    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };  
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glShadeModel(GL_SMOOTH);

    // continue with initialisation code as before
    // ....
void mydisplay()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glLoadIdentity();

  GLUquadricObj* q = gluNewQuadric();
  gluQuadricDrawStyle (q, GLU_FILL);
  gluQuadricNormals   (q, GLU_SMOOTH);
  gluSphere (q, 1, 200, 200);
  gluDeleteQuadric (q);

  SDL_GL_SwapBuffers();
}
The color of a material depends on the percentage of incoming red, green, and blue light it reflects.

Like lights, materials have different ambient, diffuse, and specular colors.

Material colors determine reflectance of the light component

Ambient and diffuse reflectances define the color of the material (typically similar or identical)

Specular reflectance is usually white or gray
void myinit(int width, int height)
{
    GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 }
    GLfloat mat_shininess[] = { 10.0 }
    GLfloat mat_ambient_and_diffuse[] = { 0.0, 1.0, 0.0, 1.0 }

    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient_and_diffuse);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_ambient_and_diffuse);

    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 }
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glShadeModel(GL_SMOOTH);

    // continue with initialisation code as before
    // ....
A green sphere illuminated by a white light
WTF are these strange light components?
components.cpp (ambient light only)
components.cpp (diffuse light only)
components.cpp (ambient and diffuse light)
light source properties

• Properties of light sources can be changed using `glLight*()` calls

• Available properties:

  • GL_AMBIENT (r, g, b, a – default: 0 0 0 1)

  • GL_DIFFUSE (r, g, b, a – default: 1 1 1 1)

  • GL_SPECULAR (r, g, b, a – default: 1 1 1 1)

  • GL_POSITION (x, y, z, w position – default: 0 0 1 0)
void myinit(int width, int height)
{
    GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat mat_shininess[] = { 10.0 };
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);

    GLfloat light_ambient[] = { 0.0, 1.0, 0.0, 1.0 };
    GLfloat light_diffuse[] = { 0.0, 1.0, 0.0, 1.0 };
    GLfloat light_specular[] = { 1.0, 1.0, 1.0, 1.0 };
    glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
    glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);

    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    // ...
}
A white sphere illuminated by a green light
A green sphere illuminated by a white light
A green sphere illuminated by a white light
A white sphere illuminated by a green light

SAME RESULT

ALMOST same RESULT
moving the light

- Lights are influenced by the modelview matrix like any other object
- Translating the light relative to a stationary object?
  - Change model transform to specify the light position
  - Set light position after this
- Something like this:

```c
glPushMatrix();
  glRotatef ((float) spin, 0.0, 1.0, 0.0);
  glLightfv (GL_LIGHT0, GL_POSITION, light_position);
glPopMatrix();
drawScene();
```
now add lighting to our 3D example
what you need

• a light source

• `glMaterial` instead of `glColor`

• normal vectors
  
  • faces must be defined in counter-clockwise order

  • to test:  
    
    ```
    glEnable(GL_CULL_FACE);
    glFrontFace(GL_CCW);
    ```

  • normals should be unit length
    
    • either do normalisation yourself (recommended)
      
      ```
      glEnable(GL_NORMALIZE);
      ```
  
  • or let OpenGL do it for you: