OpenGL: Interaction, motion, and textures

Course web page:

http://goo.gl/EB3aA

Reminder: NO CLASS on Thursday, Feb. 16 and Tuesday, Feb. 21

Vincent will be giving a lecture on rasterization on Thursday, Feb. 23

Outline

- [Finish last lecture]
- Slightly more advanced OpenGL, GLUT
  - User interaction
  - Animation
  - Texture maps
- HW #1

Event-driven GLUT program structure

1. Configure and open window
2. Initialize OpenGL state, program variables
3. Register callback functions
   - Display (where rendering occurs)
   - Resize
   - User input: keyboard, mouse clicks, motion, etc.
4. Enter event processing loop

Portions of some slides adapted from "An Interactive Introduction to OpenGL Programming", D. Shreiner, E. Angel, V. Shreiner, SIGGRAPH 2001 course

More callbacks!!

```
#include <stdio.h>
#include <GL/glut.h>

void main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);
    glutInitWindowSize(100, 100);
    glutCreateWindow("hello");
    init();                          // set OpenGL states, variables

    glutDisplayFunc(display);        // register callback routines
    glutKeyboardFunc(keyboard);
    glutIdleFunc(idle);

    glutMainLoop();                  // enter event-driven loop
}
```

Simple OpenGL program

```
#include <stdio.h>
#include <GL/glut.h>

void main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);
    glutInitWindowSize(100, 100);
    glutCreateWindow("hello");
    init();                          // set OpenGL states, variables

    glutDisplayFunc(display);        // register callback routines
    glutKeyboardFunc(keyboard);
    glutIdleFunc(idle);

    glutMainLoop();                  // enter event-driven loop
}
```

adapted from E. Angel

Callback registration functions

- **Render** `glutDisplayFunc()`
- **Key down in window** `glutKeyboardFunc()`
- **Key up in window** `glutKeyboardUpFunc()`
- **Button press/release** `glutMouseFunc()`
- **Mouse motion (with button down)** `glutMotionFunc()`
- **Mouse motion (with button up)** `glutPassiveMotionFunc()`
- **Window reshaping** `glutResizeFunc()`
- **Nothing happening** `glutIdleFunc()`
- **X time has elapsed** `glutTimerFunc()`

- **Other callbacks (non-printing keys, other input devices):** See GLUT reference pages linked on course page

Example: Keyboard callback

- These arguments are **required** for your callback
- They say *what* key was pressed and *where* the cursor was when it was pressed:

```
glutKeyboardFunc(keyboard);
```

```
void keyboard(unsigned char key, int x, int y)
{
   switch(key) {
      case 'q' : case 'Q' :
         exit(1);
         break;

      case 'r' : case 'R' :
         rotate_mode = GL_TRUE;
         break;
   }
}
```

Example: Mouse button callback

- Which mouse button, where, and has it been pressed or released:

```
glutMouseFunc(mouse);
```

```
void mouse(int button, int state, int x, int y)
{
  if (button == GLUT_LEFT_BUTTON) {
    if (state == GLUT_DOWN) {
      left_down = TRUE;
      mouse_x = x;
      mouse_y = y;
    }
    else if (state == GLUT_UP)
      left_down = FALSE;
  }
}
```

Example: Idle callback

- Use for animation and continuous update

```
glutIdleFunc(idle);
```

```
void idle(void)
{
  // change position variables, etc.
  t += dt;

  // call glutDisplayFunc() callback ASAP
  glutPostRedisplay();
}
```

Example: Timer callback

- Can be used instead of glutIdleFunc() for control of **frames per second** (fps) on different systems
- Key argument is milliseconds until invocation
  - Also pass "context" argument to callback function
- Only is called once—re-register at end of function for repetition

```
        glutTimerFunc(1000, timer, 0)

void timer(int value)  // value = 0 here
{
  // change position variables, etc.

  glutPostRedisplay();
  glutTimerFunc(1000, timer, 0);
}
```

hello_motion example

- Add idle function to previous hello to move square
- Add some mouse interaction

hello_texture example

- Let's add some pizzazz to our boring colored squares…
- In their simplest form, **textures** are pictures we can "stick" to our polygons to make them more visually interesting
- I am providing a **Sprite** class that mostly takes care of the details for you for certain uses

What is Texture Mapping?

- Spatially-varying modification of surface appearance at the pixel level
- Characteristics
  - Color
  - Shininess
  - Transparency
  - ***Bumpiness***
  - Etc.

from Hill

- Sometimes called a **"sprite"** when on a polygon with no 3-D

Texture mapping applications: Billboards

from Akenine-Moller & Haines

from www.massal.net/projects

Also called ”impostors”: Image-aligned polygons in 3-D

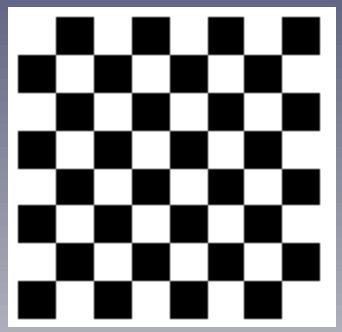
OpenGL texturing steps (Red book)

1. Create a texture object and specify a texture for that object
2. Indicate how the texture is to be applied to each pixel
3. Enable texture mapping with `glEnable(GL_TEXTURE_2D)`
4. Draw the scene, supplying both texture and geometric coordinates

Sprite does most of this – but what is it doing?

Create Texture Object

- From where?
  - Create programmatically (aka "procedurally" -- see Red Book Chap. 9 checker.c)
  - Load image from file (e.g., `load_ppm()` in Sprite.cpp)
- Name it
  - // Get unused "names" – not mandatory
    `glGenTextures(GLsizei n, GLuint *textures)`
  - // Create texture object w/ default params (or switch to existing one)
    `glBindTexture(GLenum target, GLuint texture)`

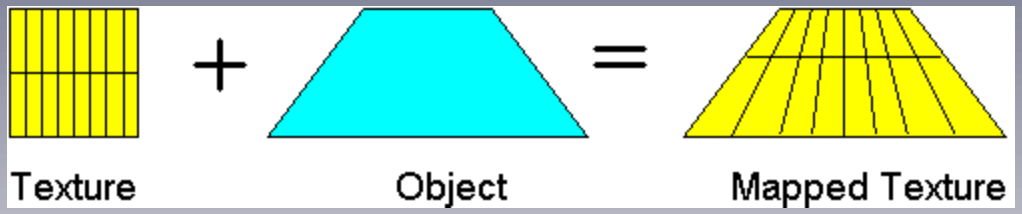


- // Store data in bound texture object (no ref because it's global)

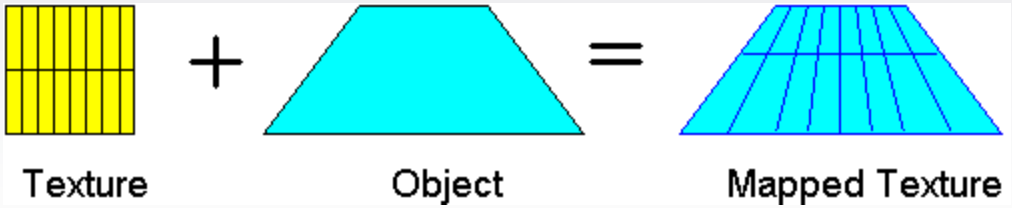
```
glTexImage2D(   GLenum target, GLint level,
                GLint internalFormat,
                GLsizei width, GLsizei height,
                GLint border, GLenum format,
                GLenum type,
                const GLvoid *pixels)
```

Rasterization: Texture application modes

- **decal**: Overwrite object pixel with texel

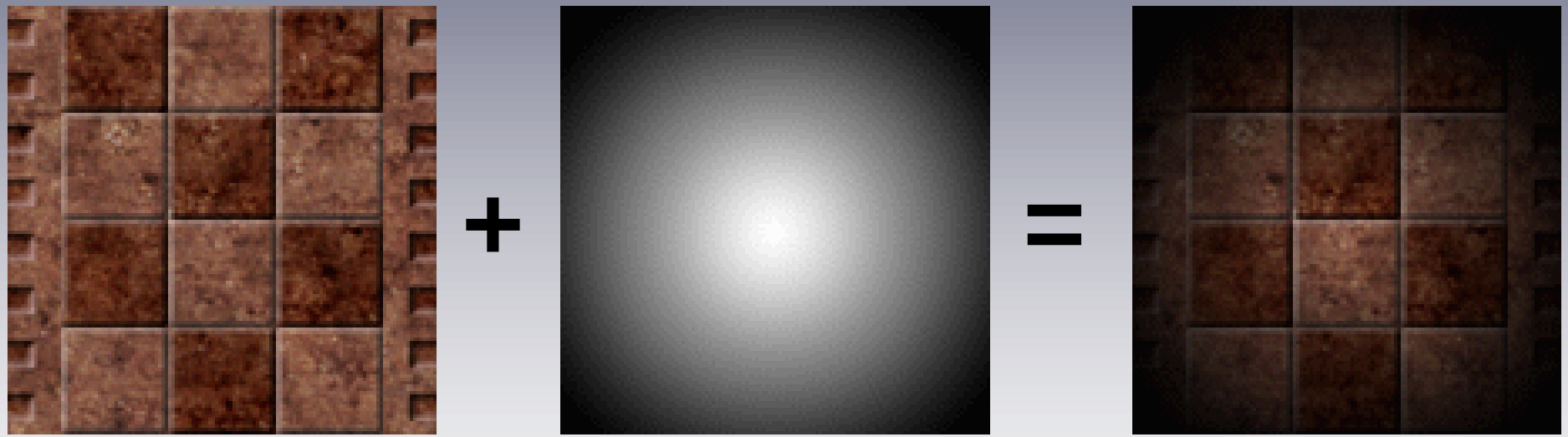


- **modulate**: Combine object pixel with texel via multiplication
  - Need this for multitexturing (i.e., lightmaps)



courtesy of Microsoft

Texture mapping applications: Lightmaps



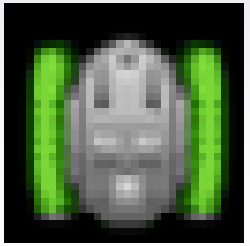
courtesy of K. Miller

Texture Application Modes

- **`glTexEnv(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, param)`**, where **`param`** is one of:
  - **`GL_REPLACE`**: Just overwrite surface pixel
  - **`GL_DECAL`**: Use alpha values of surface pixel and texel to blend in standard way
  - **`GL_MODULATE`**: Multiply surface pixel and texel colors
  - **`GL_BLEND`**: Blend surface and texel colors with **`GL_TEXTURE_ENV_COLOR`** (see **`glTexEnv()`** man page for details)
- One thing we're ignoring right now is **wrapping**—the idea of the texture being a repeating pattern

Transparency

- Remember `glColor4f(r,g,b,a)`?
- `a` is called the **alpha** channel and is used to specify transparency when blending textures
  - See Red Book, Chap. 6
- Used for overlapping sprites in Drop! game
  - `load_ppm()` sets full transparency when image pixel color is black, full opacity otherwise



Texturing: Enabling and Drawing

- To draw textured shape, texturing must first be enabled: **`glEnable(GL_TEXTURE_2D)`**
- Load current texture image with **`glTexImage2D()`**
  - Width, height must be powers of 2 (plus 2 if border is used)
  - Only one texture current; faster to change textures by preloading all and switching with **`glBindTexture()`** rather than reloading each time (this is what Sprite.cpp does)
- Assign texture coordinates (s, t) to vertices with **`glTexCoord()`**
  - Similar to **`glColor()`** command—sets a property for subsequent vertices that holds until it is changed

HW #1

- Your first programming assignment, due on **Tuesday, February 28**
- See course page for details