

Visualization and Animation

Scientific visualization using computer graphics can be very useful:

- computer simulations can generate large amounts of data, which are easier to interpret visually;
- visualizing a simulation as it runs can be very useful in understanding the simulations;
- visualizing the state of program data as the program runs can be very helpful in finding programming errors and bugs;
- programming computer graphics and animations can be fun!

Visualization systems

There are numerous computer graphics and visualization systems. Many of them are special purpose programs designed for particular types of problems, and which make use of special graphics hardware.

Systems like Mathematica, Maple and Matlab have fancy 2-D and 3-D graphics built in, as well as facilities to perform animations.

If you use a lower level programming language and want your graphics programs to run on different platforms (Unix, Windows, MacOS at least!), then there are essentially two choices:

- Use Java, which is an excellent programming language. Java programs are probably the most portable programs you can write. *Write Once, Run Anywhere* is a Java slogan. Java graphics can be incorporated in *applets* which will run on most web browsers.
- Use OpenGL, which claims to be *The Industry's Foundation for High Performance Graphics*. The native language of OpenGL is C, so it is much easier to use with C++. OpenGL can be much more efficient than Java in exploiting graphics hardware. It is somewhat harder to learn than Java, and also not quite as platform independent.

The OpenGL library is useful for programming 3 dimensional graphics and animations. There are OpenGL functions to draw, color, and texturize points, lines, and surface, as well as bitmapped images. The OpenGL

Utility Toolkit (GLUT) has functions manipulate windows on computer displays: these functions are very convenient because they are portable across operating systems.

OpenGL: hello world program

The following simple program illustrates the use of OpenGL and GLUT library functions to open a window on the display and write a message on it.

Including `glut.h` automatically includes `gl.h` and `glu.h`.

```
// OpenGL hello program
```

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

`display()` is a *callback* function which is invoked by the program when the window to which it is attached needs to be repainted.

- `glClear` clears the window by painting it with the current background color, which defaults to black if it is not set.
- `glRasterPos2d` sets the origin for drawing a raster (bitmap) image.
- `glutBitmapCharacter` draws a character using the font specified in the first argument at the current raster position, and resets the raster position just beyond the bitmap.

```
void display() {  
    glClear(GL_COLOR_BUFFER_BIT);  
    char message[] = "Hello, world!";  
    glRasterPos2d(0, 0);  
    for (int i = 0; i < sizeof(message) / sizeof(message[0]); i++)  
        glutBitmapCharacter(GLUT_BITMAP_HELVETICA_12, message[i]);  
}
```

The `main` function must be defined with command line arguments which can be passed to

- `glutInit`, which initializes the graphics system.
- `glutInitWindowSize` sets the window width and height in pixels.
- `glutCreateWindow` creates and initializes a window data structure but does not actually place it on the screen.
- `glutDisplayFunc` registers its argument as the callback function which is invoked whenever the window needs repainting.
- `glutMainLoop` enters an infinite loop which waits for user input after painting the window on the screen.

```
int main(int argc, char *argv[]) {  
    glutInit(&argc, argv);  
    glutInitWindowSize(250, 100);  
    glutCreateWindow("OpenGL hello program");  
    glutDisplayFunc(display);  
    glutMainLoop();  
}
```

OpenGL: double-buffered rotating square

The following example is taken from the OpenGL Programming Guide (Red Book).

```
// double buffered rotating square  
  
#include <GL/glut.h>  
  
double spin = 0.0;                                // angle to rotate square
```

```
void init() {
    glClearColor(0.0, 0.0, 0.0, 0.0);    // background color black
    glShadeModel(GL_FLAT);
}

void display() {
    glClear(GL_COLOR_BUFFER_BIT);
    glPushMatrix();                      // save transformation matrix
    glRotated(spin, 0.0, 0.0, 1.0);      // rotate about z-axis by spin
    glColor3f(1.0, 1.0, 1.0);           // set color to white
    glRectd(-25.0, -25.0, 25.0, 25.0);   // draw a rectangle
    glPopMatrix();                       // restore transformation matrix
    glutSwapBuffers();                   // copy drawing buffer to screen
}

void spinDisplay() {
    spin += 2.0;
    if (spin > 360.0)
        spin -= 360.0;
    glutPostRedisplay();                 // request glut to repaint window
}

void reshape(int w, int h) {            // window reshape callback function
    glViewport(0, 0, w, h);              // use whole window for drawing
    glMatrixMode(GL_PROJECTION);          // select projection matrix
    glLoadIdentity();                    // make it the identity
    glOrtho(-50.0, 50.0, -50.0, 50.0, -1.0, 1.0); // orthographic projection
    glMatrixMode(GL_MODELVIEW);           // select modeling matrix
    glLoadIdentity();                    // make it the identity
}
```

```
}

void mouse(int button, int state, int x, int y) {    // mouse callback
    switch (button) {
        case GLUT_LEFT_BUTTON:
            if (state == GLUT_DOWN)
                glutIdleFunc(spinDisplay);    // set idle function to spinDisplay
            break;
        case GLUT_RIGHT_BUTTON:
            if (state == GLUT_DOWN)
                glutIdleFunc(NULL);            // set idle function to do nothing
            break;
        default:
            break;
    }
}

int main(int argc, char *argv[]) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);    // use double buffering
    glutInitWindowSize(250, 250);
    glutInitWindowPosition(100, 100);
    glutCreateWindow(argv[0]);
    init();
    glutDisplayFunc(display);                    // register display function
    glutReshapeFunc(reshape);                   // register reshape function
    glutMouseFunc(mouse);                      // register mouse function
    glutMainLoop();
}
```

One dimensional sandpile automaton

This is a simple one-dimensional cellular automaton. It's behavior is not very interesting, but it is relatively simple to understand, and to visualize using OpenGL graphics.

Imagine a one-dimensional array of L columns of sand grains with non-negative integer heights

$$h_i, \quad i = 0, 1, \dots, L-1.$$

At each column position we define the local *slope* of the pile

$$s_i \equiv h_i - h_{i+1}.$$

We also define a *critical slope* s_c : the pile is *unstable* if $s_i > s_{rmc}$ for any column i ; or equivalently, the pile is *stable* only if $s_i \leq s_c$ for all columns.

The local rule for updating the sand pile is as follows:

- If the pile is unstable, then for each column such that $s_i > s_c$, let $h_i \rightarrow h_i - 1$ and $h_{i+1} \rightarrow h_{i+1} + 1$, that is, move the top sand grain on column i to column $i+1$. This update is carried out *synchronously* on all columns.
- If the pile is stable, then add a sand grain to a randomly chosen column.

To make the model well defined, we need to say what happens at the two boundaries. Note that an unstable column in this model can only topple to the right (which is not very realistic). Thus the left boundary at $i = 0$ can be considered to be a rigid retaining wall. We can, for example apply open boundary conditions at the right boundary $i = L$ by fixing $s_L = 0$: thus if $h_{L-1} > s_c$ the unstable grains are removed from the system, and can be considered to fall off the edge of the table on which the sandpile is being built.

OpenGL: 1-D sandpile animation

```
// One dimensional sandpile cellular automaton model
#include <cstdlib>
```

```
using namespace std;

#include <GL/glut.h>

int L = 20;           // length of sandpile (number of columns)
int criticalSlope = 1; // sand topples if slope exceeds this value

int *height;          // height of pile of grains
int *slope;           // slope of sandpile
bool *stable;         // true if slope is <= critical slope

void allocate() {      // allocates storage
    static int oldL = 0;
    if (oldL != L) {
        if (height) {
            delete [] height;
            delete [] slope;
            delete [] stable;
        }
        height = new int [L];
        slope = new int [L];
        stable = new bool [L];
        oldL = L;
    }
}

void initialize() {    // initialize to empty sandpile
    allocate();
}
```

```
    for (int i = 0; i < L; i++) {
        slope[i] = height[i] = 0;
        stable[i] = true;
    }
}

bool pileIsStable() {          // check pile stability and reset slopes
    bool pileStable = true;
    for (int i = 0; i < L; i++) {
        slope[i] = height[i];
        if (i < L - 1)
            slope[i] -= height[i + 1];
        stable[i] = true;
        if (slope[i] > criticalSlope)
            stable[i] = pileStable = false;
    }
    return pileStable;
}

void takeStep() {              // local update rule
    if (pileIsStable()) {      // add a grain to a randomly chosen column
        int i = int(rand() / double(RAND_MAX) * L);
        ++height[i];
    } else {                    // topple one grain from each unstable column
        for (int i = 0; i < L; i++)
            if (!stable[i]) {
                --height[i];      // decrease unstable column height by 1
                if (i < L - 1)
                    ++height[i + 1]; // unstable grain topples to right
            }
    }
}
```

```
        }
    }

    glutPostRedisplay();    // request glut to repaint the window
}

int xPixels = 400;          // width of window in pixels
int yPixels = 400;          // height of window in pixels
bool running;              // is the animation running?

void display() {            // display callback function - repaint window
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 0.4, 0.0);    // this is a shade of orange in RGB

    for (int i = 0; i < L; i++) {
        for (int j = 0; j < height[i]; j++) {
            glRectd(i, j, i + 0.9, j + 0.9);
        }
    }

    glutSwapBuffers();        // draw the buffer in memory to the screen
}

void reshape(int w, int h) { // invoked when shape of window changes
    glViewport(0, 0, w, h);   // use the full window for drawing
    glMatrixMode(GL_PROJECTION); // select projection matrix
    glLoadIdentity();         // set it to the identity
    glOrtho(0, L, 0, L * h / double(w), -1.0, 1.0); // orthographic projection
    // columns fill window, and grains are drawn square
}
```

```
}

void reset(int menuItem) { // callback function for popup menu
    switch (menuItem) {
        case 1: // user selects "Reset" from popup
            initialize();
            break;
        case 2: // user selects "++ critical slope" from popup
            criticalSlope += 1;
            break;
        case 3: // user selects "-- critical slope" from popup
            criticalSlope -= 1;
            break;
        default:
            break;
    }
}

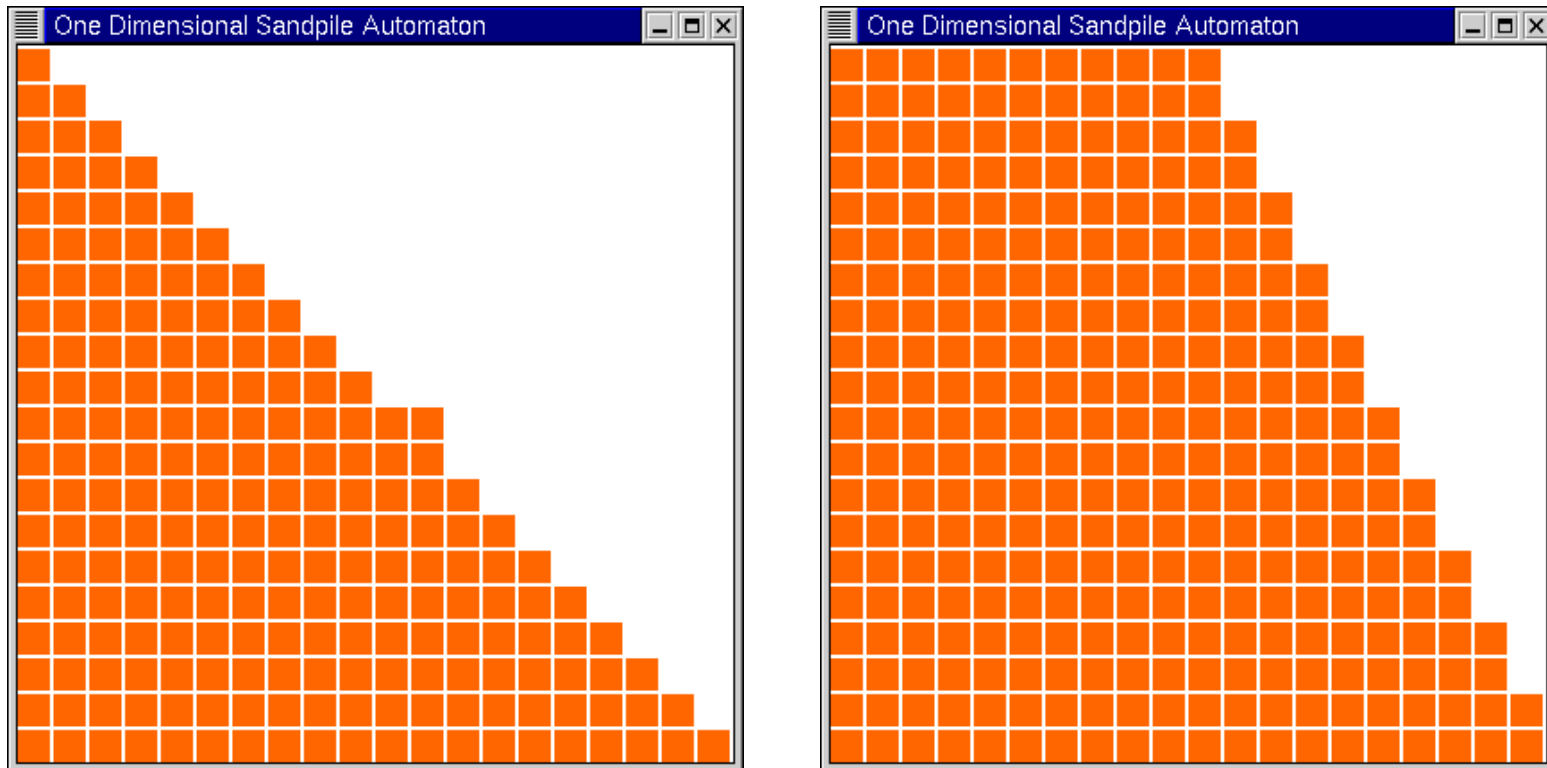
void mouse(int button, int state, int x, int y) {
    switch (button) {
        case GLUT_LEFT_BUTTON: // use left button to toggle animation on/off
            if (state == GLUT_DOWN) {
                if (running) { // animation is running
                    glutIdleFunc(NULL); // set idle callback to do nothing
                    running = false; // now not running
                } else {
                    glutIdleFunc(takeStep); // set idle callback to take step
                    running = true; // now running
                }
            }
    }
}
```

```
        }
        break;
    default:
        break;
    }
}

int main(int argc, char *argv[]) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(xPixels, yPixels);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("One Dimensional Sandpile Automaton");
    if (argc > 1)                // first argument of command line
        L = atoi(argv[1]);       // is the number of columns
    initialize();
    glClearColor(1.0, 1.0, 1.0, 0.0); // background color white
    glShadeModel(GL_FLAT);
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutCreateMenu(reset);        // create a popup menu
    glutAddMenuEntry("Reset", 1);
    glutAddMenuEntry("++ critical slope", 2);
    glutAddMenuEntry("-- critical slope", 3);
    glutAttachMenu(GLUT_RIGHT_BUTTON); // attach it to right mouse button
    glutMouseFunc(mouse);
    glutMainLoop();
}
```

Results from running the program

This sandpile automaton has a simple asymptotic behavior: the slope at each column is equal to the critical slope, and any grains of sand added to the pile cascade down the slope without changing it.



The image on the left shows the steady state of the pile with $s_c = 1$: note the single grain of sand on its way down the stable slope. The image on the right shows the steady state with $s_c = 2$.