From last time…

- Limited world view
  - A box on the domain $[-1,-1] \times [1,1]$
- Today’s issues:
  - Reality: pixels on the screen
  - Abstraction: the world as we wish to see it (on the screen)

More than you might suspect…

- A lot going on here!
- Julia Sets (fractal)
  - A visualization of a (seemingly) simple iterative function defined over the imaginary plane
  - Chaotic behavior. Small changes have dramatic effects

Julia set

- The Julia set $J_c$ for a number $c$ in the complex plane $P$ is given by:

$$J_c = \{ p \mid p \in P \text{ and } p_{i+1} = p_i^2 + c \text{ converges to a fixed limit} \}$$
Complex numbers

- 2 tuples – (Real, Imaginary)
- \( c = a + bi \)
- \( c = c_1 + c_2 \)
  \[ = (a_1 + a_2) + (b_1 + b_2)i \]
- \( c = c_1 \cdot c_2 \)
  \[ = (a_1a_2 - b_1b_2) + (a_1b_2 + a_2b_1)i \]
- \( c = (c_1)^2 \)
  \[ = ((a_1)^2 - (b_1)^2) + (2a_1b_1)i \]
- \(|c| = \sqrt{a^2 + b^2}\)

Convergence example

Real numbers are a subset of complex numbers:
- consider \( c = [0, 0] \), and \( p = [x, 0] \)
- for what values of \( x \) is \( x_{n+1} = x_n^2 \) convergent?

Convergence properties

- Suppose \( c = [0,0] \), for what complex values of \( p \) does the series converge?

  - For real numbers:
    - if \( |x| > 1 \) then the series diverges
  - For complex numbers
    - if \( |p| > 2 \) then the series diverges

\( x_{0.4} = 0.5, 0.25, 0.0625, 0.0039 \)

- BTW, the black points are the ones in c’s Julia set.
A peek at the fractal code

```python
class Complex:
    def __init__(self, r = 0, i = 0):
        self.re = r
        self.im = i

def Julia(p, c):
    maxIterations = 256
    for its in range(maxIterations):
        (p.re, p.im) = (p.re*p.re - p.im*p.im + c.re,
                        2*p.re*p.im + c.im)
        rsqr = p.re*p.re + p.im*p.im
        if rsqr > 4.0:
            break
    return (its, math.sqrt(rsqr))
```

The iterations “its” and “rsqr” are used to assign a color

Some graphics code

```python
def display():
    initialize()
    delta = (world.r - world.l)/float(width)
    p = Complex()
    for y in xrange(height):
        y = world.b + y*delta
        for x in xrange(width):
            x = world.l + x*delta
            (p.re, p.im) = (x, y)
            (its, R) = Julia(p, c)
            if its == 255:
                glColor3d(0,0,0)
            else:
                r = R/float(3)
                g = its/float(128)
                b = R/float(its+1)
                glColor3d(r,g,b)
                glBegin(GL_POLYGON)
                glVertex2d(x, y)
                glVertex2d(x, y+delta)
                glVertex2d(x+delta, y+delta)
                glVertex2d(x+delta, y)
                glEnd()
    glFlush()
```

set the color
draw pixel

How can we see more?

- Our world view only allows us to see so much
- We need to define a mapping from our desired world view to our screen.

Screen space

- Graphics generally presented by setting colors for a set of discrete samples called “pixels”
- Pixels displayed on screen in windows
- Pixels addressed as 2D arrays "indices are "screen-space" coordinates"
**Pixel independence**
- Often easier to structure graphics independent of screen or window sizes.
- Define graphics in “world-space”
  - Often had physical origins

![Diagram showing pixel independence](image)

**Normalized device coordinates**
- Intermediate “rendering-space”
  - compose world and screen space
- Sometimes called “canonical screen space”

![Diagram showing normalized device coordinates](image)

**Why introduce NDC?**
- Easy to convert NDC fixed-point. Why?
  - Simplifies many rendering operations
    - Clipping, computing coefficients for interpolation
  - Separates the bulk of geometric processing from the specifics of rasterization (sampling)

**World space to NDC**
\[
\begin{align*}
\frac{x_n - (-1)}{1 - (-1)} &= \frac{x_w - w.l}{w.r - w.l} \\
x_n &= \frac{2x_w - w.l}{w.r - w.l} - 1 \\
x_n &= Ax_w + B \\
A &= \frac{2}{w.r - w.l} \quad B = \frac{w.r + w.l}{w.r - w.l}
\end{align*}
\]
World space to NDC

- Since our world space to rendering space mappings are linear, they can be accomplished via a matrix multiplication

\[
\begin{bmatrix}
  x_n \\
  y_n \\
  1
\end{bmatrix} =
\begin{bmatrix}
  \frac{2}{w.r-w.l} & 0 & -\frac{w.r+w.l}{w.r-w.l} \\
  0 & \frac{2}{w.l-w.b} & -\frac{w.l+w.b}{w.l-w.b} \\
  0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
  x_w \\
  y_w \\
  1
\end{bmatrix}
\]

NDC to screen space

- Same approach

\[
\begin{align*}
  \frac{x_s - \text{origin}.x}{\text{width}} &= \frac{x_n - (-1)}{1 - (-1)} \\
  \text{Solve for } x_s \\
  x_s &= \text{width}\frac{x_n + 1}{2} + \text{origin}.x \\
  x_s &= Ax_n + B \\
  A &= \frac{\text{width}}{2}, \quad B = \frac{\text{width}}{2} + \text{origin}.x
\end{align*}
\]

Mapping from world to screen

- NDC-to-screen mapping
  - does not could use matrix transform
  - uses dedicated hardware
  - World-to-NDC: floating point
  - NDC-to-screen: fixed-point
- Provides opportunity to compensate for different coordinate frames
- OpenGL does not provide screen-to-world mapping

OpenGL Tools Available

Typical OpenGL code to establish a window:

- \texttt{glutInitWindowSize(400, 400)}
- \texttt{glutInitWindowPosition(100, 100)}

Code to set up a viewport:

- \texttt{glViewport(0, 0, w, h)}

To establish a world space coordinate system:

- \texttt{glOrtho2D(world.l, world.r, world.b, world.t)}
Screen-to-world in practice

```python
def initialize():
    glMatrixMode(GL_PROJECTION)  # Set up world to screen
    glLoadIdentity()  # mapping
    gluOrtho2D(world.l, world.r, world.b, world.t)
    glMatrixMode(GL_MODELVIEW)
    glLoadIdentity()

    glColor3d(0, 0, 1)  # Clear the screen
    glBegin(GL_POLYGON)
    glVertex2d(world.l, world.b)
    glVertex2d(world.r, world.b)
    glVertex2d(world.r, world.t)
    glVertex2d(world.l, world.t)
    glEnd()
```

Initializing GLUT

```python
def main():
    glutInit(sys.argv)
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)
    glutInitWindowSize(width, height)
    glutInitWindowPosition(100, 100)
    glutCreateWindow("Julia Set (a.k.a. Whoville)")
    glutDisplayFunc(display)
    glutMouseFunc(onMouseButton)
    glutKeyboardFunc(onKeyPress)
    glutReshapeFunc(onResize)
    glutMainLoop()
```

What is GLUT?

- A "O/S agnostic" OpenGL environment
- Advantages:
  - Portable: Windows, Cygwin, Linux, Mac-OS
  - minimal-overhead
    (Hides away details of opening windows, etc.)
  - Appeals to C-hackers (console for printf()’s, etc)
- Disadvantages
  - Ugly (lacks look-and-feel of real app, outdated call-back-based event-handling model)
  - Limited Interaction
  - Global variables galore

Getting GLUT

- Web site:
  Windows: [www.xmission.com/~nate/glut.html](http://www.xmission.com/~nate/glut.html)
  Others: [www.opengl.org/developers/documentation/glut.html](http://www.opengl.org/developers/documentation/glut.html)
  [www.sourceforge.net/projects/uncpython tools](http://www.sourceforge.net/projects/uncpython tools)
    (under uncOpenGL link)
- Overview:
  Appendix D of OpenGL Programming Guide
Lots of global variables

```python
import sys
import math
from GL import *
from GLU import *
from GLUT import *

class Complex:
    def __init__(self, r = 0, i = 0):
        self.re = r
        self.im = i

class Extent:
    def __init__(self, l = 0, r = 0, b = 0, t = 0):
        self.l = l
        self.r = r
        self.b = b
        self.t = t

world = Extent(-1, 1, -1, 1)
c = Complex(0.109, 0.603)
width = 200
height = 200

def Julia(p, c):
    maxIterations = 256

    (p.re, p.im) = (p.re*p.re - p.im*p.im + c.re,
                   2*p.re*p.im + c.im)
    rsqr = p.re*p.re + p.im*p.im
    if rsqr > 4.0:
        break
    return (its, math.sqrt(rsqr))
```

Initialize

- Executed at the beginning of display():

```python
def initialize():
    global world
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    gluOrtho2D(world.l, world.r, world.b, world.t)
    glMatrixMode(GL_MODELVIEW)
    glLoadIdentity()
```

- Note: by default GLUT sets up a viewport, we only need to mess with it if the window size changes.

Speaking of Resize

- Resize gets called when the window size changes

```python
def onResize(w, h):
    world.l = cx - 0.5*dy * w/h
    world.r = cx + 0.5*dy * w/h
```

Main display loop

```python
def display():
    initialize()
    delta = (world.r - world.l)/float(width)
    p = Complex()
    for j in xrange(height): # Loop over pixels
        y = world.b + j*delta
        for i in xrange(width):
            x = world.l + i*delta
            (p.re, p.im) = (x, y)
            (its, R) = Julia(p, c) # Test for convergence
            if its == 255:
                glColor3d(0,0,0)
            else:
                r = R/float(3)
                g = its/float(128)
                b = R/float(its+1)
                glColor3d(r,g,b)
            glBegin(GL_POLYGON) # Draw pixel
            glVertex2d(x, y)
            glVertex2d(x, y+delta)
            glVertex2d(x+delta, y+delta)
            glVertex2d(x+delta, y)
            glEnd()
    glFlush()
```
Recap all the GLUT stuff

```python
def main():
    glutInit(sys.argv)
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)
    glutInitWindowSize(width, height)
    glutInitWindowPosition(100, 100)
    glutCreateWindow("Julia Set (a.k.a. Whoville)")
    glutDisplayFunc(display)
    glutMouseFunc(onMouseButton)
    glutKeyboardFunc(onKeyPress)
    glutReshapeFunc(onResize)
    initialize()
    glutMainLoop()

if __name__ == "__main__":
    main()
```

Now the GUI stuff

```python
def onMouseButton(button, state, mx, my):
    x = xScreenToWorld(mx)
y = yScreenToWorld(my)
    dx = world.r - world.l
dy = world.t - world.b
    if (button == GLUT_LEFT_BUTTON) & (state == GLUT_DOWN):
        world.l = x - dx/4.0  # zoom in
        world.r = x + dx/4.0
        world.b = y - dy/4.0
        world.t = y + dy/4.0
        display()
    elif (button == GLUT_RIGHT_BUTTON) & (state == GLUT_DOWN):
        world.l = x – dx  # zoom out
        world.r = x + dx
        world.b = y - dy
        world.t = y + dy
        display()```

Screen-to-world mapping

```python
def xScreenToWorld(scrX):
    return (world.r - world.l) * scrX / float(width) + world.l
```

```
def yScreenToWorld(scrY):
    return (world.t - world.b) * (1 - scrY / float(height)) + world.b
```

That sure was a lot of lecture for so little code!

Keyboard handling

```python
def onKeyPress(key, x, y):
    global c
    print key
    if (key == 'r') | (key == 'R'):
        (c.re, c.im) = (0.109, 0.603)
        world.l = -1
        world.r = 1
        world.b = -1
        world.t = 1
        display()
    elif (key == 'c') | (key == 'C'):
        (c.re, c.im) = (0, 0)
        world.l = -1
        world.r = 1
        world.b = -1
        world.t = 1
        display()```

```
```
A Horse Race

Python - Interpreted

C++ - Compiled

Mandelbrot set

- Vary c over the plane but start iteration at p= (0,0)

Next Time

- 2D imaging and transformations in OpenGL