Computer Graphics (CS 543)  
Lecture 3a: Mandelbrot set, Shader Setup & GLSL Introduction

Prof Emmanuel Agu

Computer Science Dept.  
Worcester Polytechnic Institute (WPI)
Mandelbrot Set

- Based on iteration theory
- Function of interest:

\[ f(z) = (s)^2 + c \]

- Sequence of values (or orbit):

\[
\begin{align*}
    d_1 &= (s)^2 + c \\
    d_2 &= (((s)^2 + c)^2 + c)^2 + c \\
    d_3 &= (((((s)^2 + c)^2 + c)^2 + c)^2 + c)^2 + c \\
    d_4 &= (((((((s)^2 + c)^2 + c)^2 + c)^2 + c)^2 + c)^2 + c)^2 + c \\
\end{align*}
\]
Mandelbrot Set

- Orbit depends on $s$ and $c$
- Basic question:
  - For given $s$ and $c$,
    - does function stay finite? (within Mandelbrot set)
    - explode to infinity? (outside Mandelbrot set)
- Definition: if $|d| < 1$, orbit is finite else infinite
- Examples orbits:
  - $s = 0, c = -1$, orbit $= 0,-1,0,-1,0,-1,0,-1,.....finite$
  - $s = 0, c = 1$, orbit $= 0,2,5,26,677.....explodes$
### Mandelbrot Set

- **Mandelbrot set:**
  - Set $s = 0$
  - Choose $c$ as a complex number

- **For example:**
  - $s = 0, \ c = 0.2 + 0.5i$

- **Hence, orbit:**
  - $0, \ c, \ c^2 + c, \ (c^2 + c)^2 + c, \ \ldots$

- **Definition:** Mandelbrot set includes all finite orbit $c$
Mandelbrot Set

- Some complex number math:
  \[ i \times i = -1 \]

- Example:
  \[ 2i \times 3i = -6 \]

- Modulus of a complex number, \( z = ai + b \):
  \[ |z| = \sqrt{a^2 + b^2} \]

- Squaring a complex number:
  \[ (x + yi)^2 = (x^2 - y^2) + (2xy)i \]
Mandelbrot Set

- Examples: Calculate first 3 terms
  - with $s=2$, $c=-1$, terms are
    
    \[
    2^2 - 1 = 3 \\
    3^2 - 1 = 8 \\
    8^2 - 1 = 63
    \]

  - with $s = 0$, $c = -2+i$
    
    \[
    (x + yi)^2 = (x^2 - y^2) + (2xy)i
    \]
    
    \[
    0 + (-2 + i) = -2 + i \\
    (-2 + i)^2 + (-2 + i) = 1 - 3i \\
    (1 - 3i)^2 + (-2 + i) = -10 - 5i
    \]
Mandelbrot Set

- **Fixed points:** Some complex numbers converge to certain values after $x$ iterations.

- **Example:**
  - $s = 0, c = -0.2 + 0.5i$ converges to $-0.249227 + 0.333677i$ after 80 iterations

- **Experiment:** square $-0.249227 + 0.333677i$ and add $-0.2 + 0.5i$

- Mandelbrot set depends on the fact the convergence of certain complex numbers
Mandelbrot Set Routine

- Math theory says calculate terms to **infinity**
- Cannot iterate forever: our program will hang!
- Instead iterate 100 times
- **Math theorem:**
  - if no term has exceeded 2 after 100 iterations, never will!
- Routine returns:
  - 100, if modulus doesn’t exceed 2 after 100 iterations
  - Number of times iterated before modulus exceeds 2, or

```
\[ S, C \rightarrow \text{Mandelbrot function} \rightarrow \begin{cases} \text{Number} = 100 \ (\text{did not explode}) \\ \text{Number} < 100 \ (\text{first term} > 2) \end{cases} \]
```
Mandelbrot dwell( ) function

\[(x + yi)^2 = (x^2 - y^2) + (2xy)i\]

\[(x + yi)^2 + (c_x + c_yi) = [(x^2 - y^2) + c_x] + (2xy + c_y)i\]

```c
int dwell(double cx, double cy)
{ // return true dwell or Num, whichever is smaller
    #define Num 100 // increase this for better pics
    double tmp, dx = cx, dy = cy, fsq = cx*cx + cy*cy;
    for(int count = 0; count <= Num && fsq <= 4; count++)
    {
        tmp = dx;     // save old real part
        dx = dx*dx - dy*dy + cx; // new real part
        dy = 2.0 * tmp * dy + cy; // new imag. Part
        fsq = dx*dx + dy*dy;
    }
    return count; // number of iterations used
}
```
Mandelbrot Set

- Map real part to x-axis
- Map imaginary part to y-axis
- Decide range of complex numbers to investigate. E.g:
  - X in range [-2.25: 0.75], Y in range [-1.5: 1.5]

E.g. -1.5 + i

Range of complex Numbers (c)
X in range [-2.25: 0.75], Y in range [-1.5: 1.5]

Call ortho2D to set range of values to explore
Mandelbrot Set

- Set world window (ortho2D) (range of complex numbers to investigate)
  - X in range [-2.25: 0.75], Y in range [-1.5: 1.5]
- Set viewport (glViewport). E.g:
  - Viewport = [V.L, V.R, W, H] = [60,80,380,240]
So, for each pixel:
- For each point \((c)\) in world window call your \(dwell()\) function
- Assign color \(<\text{Red,Green,Blue}>\) based on \(dwell()\) return value

Choice of color determines how pretty

Color assignment:
- Basic: In set (i.e. \(dwell() = 100\)), color = black, else color = white
- Discrete: Ranges of return values map to same color
  - E.g 0 – 20 iterations = color 1
  - 20 – 40 iterations = color 2, etc.
- Continuous: Use a function

\[
\text{Mandelbrot function:} \quad \text{Number} < 100 \quad (\text{first term} > 2) \\
\text{Number} = 100 \quad (\text{did not explode})
\]
Free Fractal Generating Software

- Fractint
- FracZoom
- 3DFrac
OpenGL function format

**glUniform3f(x, y, z)**

- Function name: `glUniform3f`
- Arguments:
  - `x`, `y`, `z` are floats
- Belongs to GL library
- Argument is array of values
- Argument is array of values
- `p` is a pointer to array

**glUniform3fv(p)**

- Function name: `glUniform3fv`
- Argument is array of values
- `p` is a pointer to array
Lack of Object Orientation

- OpenGL is not object oriented
- Multiple versions for each command
  - `glUniform3f`
  - `glUniform2i`
  - `glUniform3dv`
# OpenGL Data Types

<table>
<thead>
<tr>
<th>C++</th>
<th>OpenGL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signed char</td>
<td>GLByte</td>
</tr>
<tr>
<td>Short</td>
<td>GLShort</td>
</tr>
<tr>
<td>Int</td>
<td>GLint</td>
</tr>
<tr>
<td>Float</td>
<td>GLFloat</td>
</tr>
<tr>
<td>Double</td>
<td>GLDouble</td>
</tr>
<tr>
<td>Unsigned char</td>
<td>GLubyte</td>
</tr>
<tr>
<td>Unsigned short</td>
<td>GLushort</td>
</tr>
<tr>
<td>Unsigned int</td>
<td>GLuint</td>
</tr>
</tbody>
</table>

**Example:** Integer is 32-bits on 32-bit machine but 64-bits on a 64-bit machine. Good to define OpenGL data type: same number of bits on all machines.
Recall: Single Buffering

- If display mode set to single framebuffers
- Any drawing into framebuffer is seen by user. How?
  - `glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);`
    - Single buffering with RGB colors

- Drawing may not be drawn to screen until call to `glFlush( );`

```c
void mydisplay(void) {
    glClearColor(GL_COLOR_BUFFER_BIT); // clear screen
    glDrawArrays(GL_POINTS, 0, N);
    glFlush( ); ← Drawing sent to screen
}
```
Double Buffering

- Set display mode to double buffering (create front and back framebuffers)
  - `glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);`
    - Double buffering with RGB colors
    - Double buffering is good for animations, avoids tearing artifacts

- Front buffer displayed on screen, back buffers not displayed
- Drawing into back buffers (not displayed) until swapped in using `glutSwapBuffers()`

```c
void mydisplay(void) {
    glClear(GL_COLOR_BUFFER_BIT); // clear screen
    glDrawArrays(GL_POINTS, 0, N);
    glutSwapBuffers();
}
```

Back buffer drawing swapped in, becomes visible here
Recall: OpenGL Skeleton

```c
void main(int argc, char** argv){
    glutInit(&argc, argv); // initialize toolkit
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(640, 480);
    glutInitWindowPosition(100, 150);
    glutCreateWindow("my first attempt");
    glewInit();

    // … now register callback functions
    glutDisplayFunc(myDisplay);
    glutReshapeFunc(myReshape);
    glutMouseFunc(myMouse);
    glutKeyboardFunc(myKeyboard);
    glewInit();
    generateGeometry();
    initGPUBuffers();
    void shaderSetup();
    glutMainLoop();
}
```

```c
void shaderSetup( void )
{
    // Load shaders and use the resulting shader program
    program = InitShader( "vshader1.glsl", "fshader1.glsl" );
    glUseProgram( program );

    // Initialize vertex position attribute from vertex shader
    GLuint loc = glGetUniformLocation( program, "vPosition" );
    glEnableVertexAttribArray( loc );
    glVertexAttribPointer( loc, 2, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0) );

    // sets white as color used to clear screen
    glClearColor( 1.0, 1.0, 1.0, 1.0 );
}
```
Recall: OpenGL Program: Shader Setup

- **initShader()**: our homegrown shader initialization
  - Used in main program, connects and link vertex, fragment shaders
  - Shader sources read in, compiled and linked

```cpp
GLuint program = InitShader( "vshader1.glsl", "fshader1.glsl" );
glUseProgram(program);
```

What’s inside `initShader`??

Next!
Coupling Shaders to Application (initShader function)

1. Create a program object
2. Read shaders
3. Add + Compile shaders
4. Link program (everything together)
5. Link variables in application with variables in shaders
   - Vertex attributes
   - Uniform variables
Step 1. Create Program Object

- Container for shaders
- Can contain multiple shaders, other GLSL functions

```c
GLuint myProgObj;
myProgObj = glCreateProgram();
```

Create container called **Program Object**

Main Program
Step 2: Read a Shader

- Shaders compiled and added to program object

- Shader file code passed in as null-terminated string using the function glShaderSource

- Shaders in files (vshader.glsl, fshader.glsl), write function readShaderSource to convert shader file to string
Shader Reader Code?

```c
#include <stdio.h>

static char* readShaderSource(const char* shaderFile)
{
    FILE* fp = fopen(shaderFile, "r");
    if ( fp == NULL ) { return NULL; }

    fseek(fp, 0L, SEEK_END);
    long size = ftell(fp);
    fseek(fp, 0L, SEEK_SET);
    char* buf = new char[size + 1];
    fread(buf, 1, size, fp);
    buf[size] = '\0';
    fclose(fp);

    return buf;
}
```

Shader file name (e.g. vshader.glsl) \(\rightarrow\) readShaderSource \(\rightarrow\) String of entire shader code
Step 3: Adding + Compiling Shaders

```c
GLuint myVertexObj;
GLuint myFragmentObj;

GLchar* vSource = readShaderSource("vshader1.glsl");
GLchar* fSource = readShaderSource("fshader1.glsl");

myVertexObj = glCreateShader(GL_VERTEX_SHADER);
myFragmentObj = glCreateShader(GL_FRAGMENT_SHADER);
```

- Declare shader object (container for shader)
- Read shader files
- Convert code to string
- Create empty Shader objects

![Diagram](attachment:diagram.png)

Main Program

- `example.cpp`
- `vshader1.glsl`
- `fshader1.glsl`
Step 3: Adding + Compiling Shaders
Step 4: Link Program

Read shader code *strings* into shader objects

```c
glShaderSource(myVertexObj, 1, vSource, NULL);
glShaderSource(myFragmentObj, 1, fSource, NULL);

glCompileShader(myVertexObj);
glCompileShader(myFragmentObj);

glAttachShader(myProgObj, myVertexObj);
glAttachShader(myProgObj, myFragmentObj);

glLinkProgram(myProgObj);
```

Main Program

Fragment Shader

**vshader1.glsl**

**fshader1.glsl**

Attach shader objects to program object
Uniform Variables

- Variables that are **constant** for an entire primitive
- Can be changed in application and sent to shaders
- Cannot be changed in shader
- Used to pass information to shader
  - **Example:** bounding box of a primitive
Uniform variables

- Sometimes want to connect uniform variable in OpenGL application to uniform variable in shader

Example?
- Check “elapsed time” variable (`etime`) in OpenGL application
- Use elapsed time variable (`time`) in shader for calculations
Uniform variables

- First declare **etime** variable in OpenGL application, get time

```cpp
float etime;
etime = 0.001*glutGet(GLUT_ELAPSED_TIME);
```

- Use corresponding variable **time** in shader

```cpp
uniform float time;
attribute vec4 vPosition;
main()
{
    vPosition.x += (1+sin(time));
    gl_Position = vPosition;
}
```

- Need to connect **etime** in application and **time** in shader!!
Connecting **etime and time**

- Linker forms table of shader variables, each with an address
- Application can get address from table, tie it to application variable
- In application, find location of shader variable **time** variable in linker table

Glint timeLoc;

timeLoc = glGetUniformLocation(program, "time");

- Connect: **location** of shader variable **time** to **etime**!

`glUniform1(timeLoc, etime);`

Location of shader variable **time**  
Application variable, **etime**
GL Shading Language (GLSL)

- GLSL: high level C-like language
- Main program (e.g. example1.cpp) program written in C/C++
- Vertex and Fragment shaders written in GLSL
- From OpenGL 3.1, application must use shaders

**Example code of vertex shader**

```cpp
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
out vec3 color_out;

void main(void){
    gl_Position = vPosition;
    color_out = red;
}
```

What does keyword `out` mean?
Passing values

- Variable declared **out** in vertex shader can be declared as **in** in fragment shader and used

- Why? To pass result of vertex shader calculation to fragment shader

```cpp
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
out vec3 color_out;

void main(void){
  gl_Position = vPosition;
  color_out = red;
}
```

```cpp
in vec3 color_out;

void main(void){
  // can use color_out here.
}
```
Data Types

- **C types:** int, float, bool
- **GLSL types:**
  - float vec2: e.g. (x,y) // vector of 2 floats
  - float vec3: e.g. (x,y,z) or (R,G,B) // vector of 3 floats
  - float vec4: e.g. (x,y,z,w) // vector of 4 floats

```
const float vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
out float vec3 color_out;

void main(void){
    gl_Position = vPosition;
    color_out = red;
}
```

- Also:
  - int (ivec2, ivec3, ivec4) and
  - boolean (bvec2, bvec3, bvec4)
Data Types

- Matrices: mat2, mat3, mat4
  - Stored by columns
  - Standard referencing m[rown][column]
- Matrices and vectors are basic types
  - can be passed in and out from GLSL functions
- E.g
  mat3 func(mat3 a)
- **No pointers** in GLSL
- Can use C structs that are copied back from functions
Operators and Functions

- Standard C functions
  - **Trigonometric**: cos, sin, tan, etc
  - **Arithmetic**: log, min, max, abs, etc
  - Normalize, reflect, length

- Overloading of vector and matrix types

```c
mat4 a;
vec4 b, c, d;
c = b*a;     // a column vector stored as a 1d array
d = a*b;    // a row vector stored as a 1d array
```
Swizzling and Selection

- **Selection**: Can refer to array elements by element using [] or selection (.) operator with
  - x, y, z, w
  - r, g, b, a
  - s, t, p, q
  - `vec4 a;`
  - `a[2], a.b, a.z, a.p` are the same

- **Swizzling** operator lets us manipulate components
  - `a.yz = vec2(1.0, 2.0);`
References