CS 432 – Interactive Computer Graphics

Lecture 3 – Part 2

Animation

Reading

- Angel: Chapter 2
- Red Book: Chapter 2

Animation

- Obviously in most real graphic scenes things move
    - Animation
    - Interaction
- Or things may change in the scene
    - Lighting changes
    - Objects appear/disappear
- Therefore we may need to do things like
    - Update vertex locations
    - Change uniform variable values
- Often we can do this in our client application and/or the shader program
    - We have to decide which, why, and when!

Animation

- So how do we do these updates?
- We could just wait for the next time the display callback is called
  - Terrible idea
  - Who knows when that will happen
- Recall from the list of callback functions we have
  - `glutIdleFunc`
  - `glutTimerFunc`

Animation

- The idle callback
  - Occurs whenever the program is idle
- The timer callback
  - Gets called after some number of microseconds
- Idle vs Timer
  - Generally speaking we’ll use the timer callback so our animation is smooth.
  - Idle callback won’t usually occur at even intervals. So we’d have to augment our animation step size based on system timestamps

In ms

An ID so we can differentiate multiple callbacks

Animation

- Bind a timer callback in the main function
  - `glutTimerFunc(50,timerCallback,0);`
- In the timer callback function
  - Update stuff (in our case vertex locations)
  - Move updated stuff to GPU
    - `glBindArray`, `glBindBuffer` and `glBufferData`
      - Now we may want our data to be `GL_DYNAMIC_DRAW` instead of `GL_STATIC_DRAW` since it’s being updated often.
      - Also if the amount of data changed, we may need to update the shader attribute information
  - Request a redisplay
    - `glutPostRedisplay();`
  - Schedule another timer call
    - `glutTimerFunc(50, timerCallback, value);`

Frame Buffers for Animation

- The **frame buffer** stores information for drawing to the screen. This includes at least
  - Color at each pixel
  - Depth at each pixel
- If we’re doing animation we don’t want to get caught in a situation that we’re writing to the frame buffer when it has to be used to display
- So let’s have a 2nd frame buffer!
  - Display the front one
  - Write to the back one
  - When we’re done writing to the back one, **swap** them.

Frame Buffers for Animation

- To use two buffers all we need to do is:
  - In the main, initialize the display mode to use double buffering

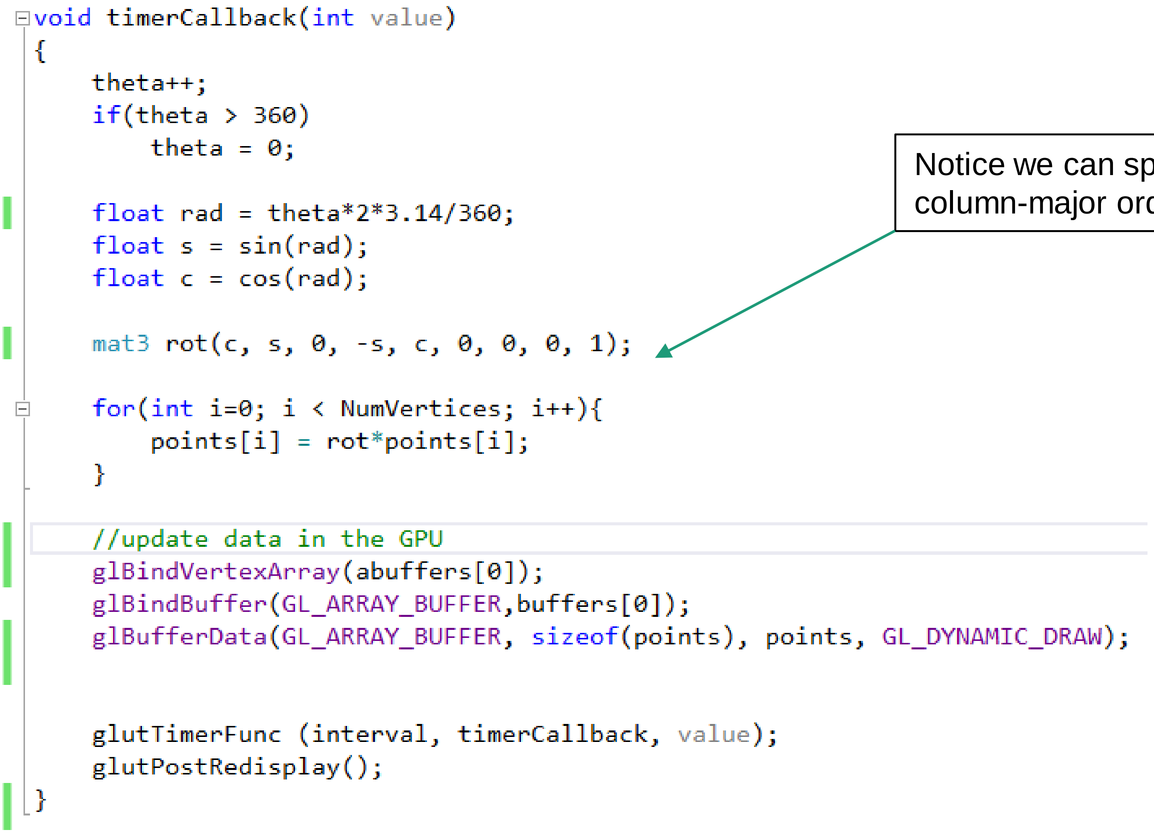
    `glutInitDisplayMode(GLUT_RGBA|`**`GLUT_DOUBLE`**`)`
  - In the display callback instead of calling `glFlush()` at the end, call `glutSwapBuffers()`;

Animation Example

- Rotate a rectangle by 1 degree every 1/10 of a second
  - `glutTimerFunc(100,timerCallback,0);`
- Alternatively you could use the idle callback and use timestamps to determine how much to rotate
  - `present_time = glutGet(GLUT_ELAPSED_TIME);`
  - `float dt = 0.001*(present_time-last_time);`
  - `last_time = present_time;`
- Recall our 2D homogenous rotation matrix:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

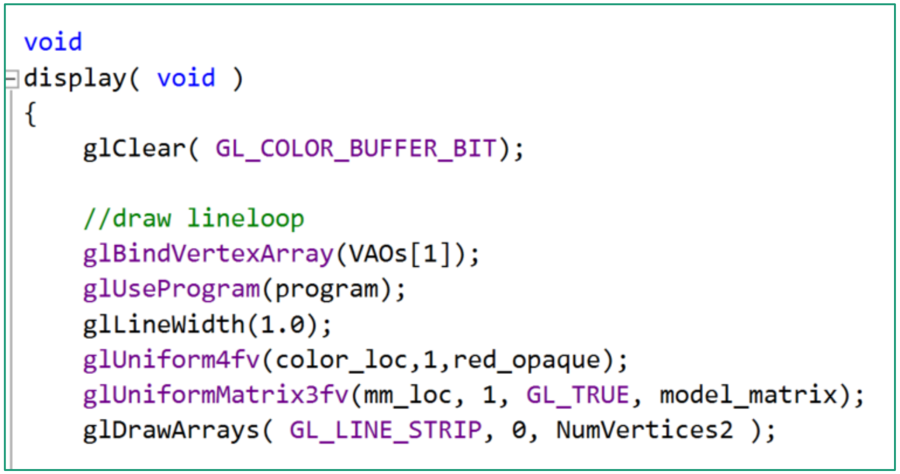
Client-Side Animation Example



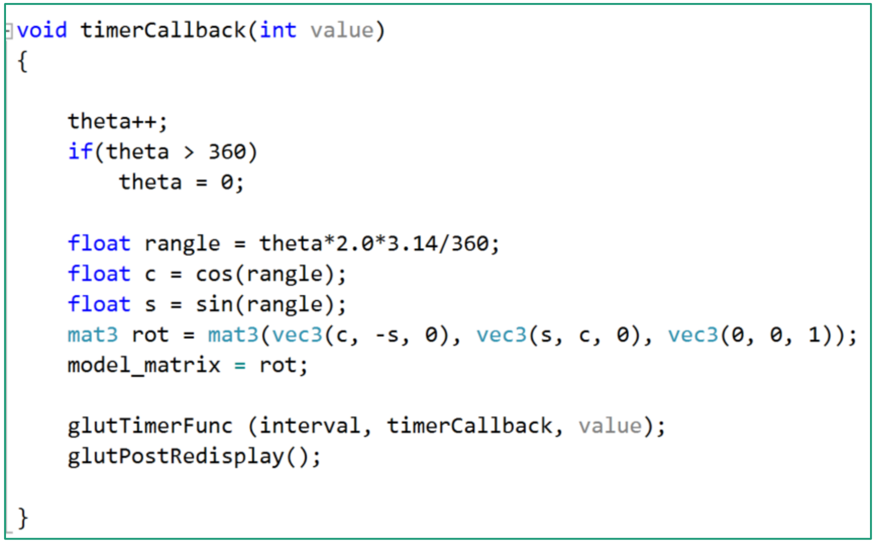
```
void timerCallback(int value)
{
    theta++;
    if(theta > 360)
        theta = 0;

    float rad = theta*2*3.14/360;
    float s = sin(rad);
    float c = cos(rad);

    mat3 rot(c, s, 0, -s, c, 0, 0, 0, 1);

    for(int i=0; i < NumVertices; i++){
        points[i] = rot*points[i];
    }

    //update data in the GPU
    glBindVertexArray(abuffers[0]);
    glBindBuffer(GL_ARRAY_BUFFER,buffers[0]);
    glBufferData(GL_ARRAY_BUFFER, sizeof(points), points, GL_DYNAMIC_DRAW);


    glutTimerFunc (interval, timerCallback, value);
    glutPostRedisplay();
}
```

Notice we can specify the matrix in column-major order

Shader-side animation example



```
void
display( void )
{
    glClear( GL_COLOR_BUFFER_BIT);

    //draw lineloop
    glBindVertexArray(VAOs[1]);
    glUseProgram(program);
    glLineWidth(1.0);
    glUniform4fv(color_loc,1,red_opaque);
    glUniformMatrix3fv(mm_loc, 1, GL_TRUE, model_matrix);
    glDrawArrays( GL_LINE_STRIP, 0, NumVertices2 );
```



```
void timerCallback(int value)
{

    theta++;
    if(theta > 360)
        theta = 0;

    float rangle = theta*2.0*3.14/360;
    float c = cos(rangle);
    float s = sin(rangle);
    mat3 rot = mat3(vec3(c, -s, 0), vec3(s, c, 0), vec3(0, 0, 1));
    model_matrix = rot;

    glutTimerFunc (interval, timerCallback, value);
    glutPostRedisplay();

}
```

Vertex Shader

```
#version 150

in  vec3 vPosition;
uniform mat3 model_matrix;

void main()
{
     gl_Position = vec4((model_matrix*vPosition).xy,0,1);

}
```

When and why?

- Generally speaking you want the GPU to do as much as possible
  - Parallelizable!
- However sometimes you still need to update things on the client-size
  - In order to do global computations
    - Like collision detection
    - Graphics pipeline treats each vertex independently ☹
- Still may be better to do computations in both places
  - Can avoid the bottleneck of moving data from CPU to GPU