Debugging

Motivation

- We need to have user interaction and take actions based on elapsed time
- Plus we want to examine some debugging techniques

Review

- Recall: Last time, we created an indexed render of a star

Debugging

- Sometimes, we need to debug our rendering
- We can do so in various ways
- One way: Draw as wireframe so we can see the triangles: glPolygonMode(GL_FRONT_AND_BACK, GL_LINES)
- Result:

- Hmm... Why don't we get the expected results?

Debugging

- GPU is fairly “unforgiving”
  - If we do anything wrong, GPU silently ignores erroneous operations
- Recent GL drivers include *debug facility* to help us out
- Makes program run a little slower, but we can disable it when we’ve finished debugging

Debugging

- Add these lines after the SDL_GL_CreateContext() call:

```
glDebugMessageCallback( debugCallback, None )
# Source, type, severity, count, ids, enabled
glDebugMessageControl(GL_DONT_CARE, GL_DONT_CARE, GL_DONT_CARE,
   0, None, True )
glEnable(GL_DEBUG_OUTPUT_SYNCHRONOUS)
glEnable(GL_DEBUG_OUTPUT)
```

- Create a top-level function:

```
def debugCallback( source, msgType, msgId, severity, length,
   message, param ):
    print(msgId,":",message)
```

Now...

- When I run the program, I get this output in the console window:

```
$ python3 main.py
131185 : Buffer detailed info: Buffer object 1 (bound to
   GL_ARRAY_BUFFER_ARB, usage hint is GL_STATIC_DRAW) will use
   VIDEO memory as the source for buffer object operations.
131185 : Buffer detailed info: Buffer object 2 (bound to
   GL_ARRAY_BUFFER_ARB, usage hint is GL_STATIC_DRAW) will use
   VIDEO memory as the source for buffer object operations.
1280 : GL_INVALID_ENUM error generated. <mode> is not a valid
   polygon mode.
```

Notice...

- Debug output includes some informational messages
  - These might be helpful if we were looking to optimize code
  - Or find bottlenecks
- It also contains an error message

Code

- We can make the debug output a little more useful
- Change the debug callback:

```
def debugCallback( source, msgType, msgId, severity, length,
   message, param ):
    print(msgId,":",message)
    if severity == GL_DEBUG_SEVERITY_HIGH:
        for x in traceback.format_stack():
            print(x,end="")
```

Result

- Now, we get this type of output:

```
$ python3 main.py
131185 : Buffer detailed info: Buffer object 2 (bound to GL_ARRAY_BUFFER_ARB,
    usage hint is GL_STATIC_DRAW) will use VIDEO memory as the source for buffer
    object operations.
1280 : GL_INVALID_ENUM error generated. <mode> is not a valid polygon mode.
  File "main.py", line 126, in <module>
    main()
  File "main.py", line 118, in main
    setup()
  File "main.py", line 64, in setup
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINES)
  File "gl.py", line 6678, in glPolygonMode
    return glPolygonMode(face, mode)
  File "gl.py", line 11473, in __pyglDebugMessageCallback
    __pyglDebugMessageCallbackFunc( src,typ,id_,sev,le,msg.decode(),
        __pyglDebugMessageCallbackArg)
  File "main.py", line 14, in debugCallback
    for x in traceback.format_stack():
```

Understanding It

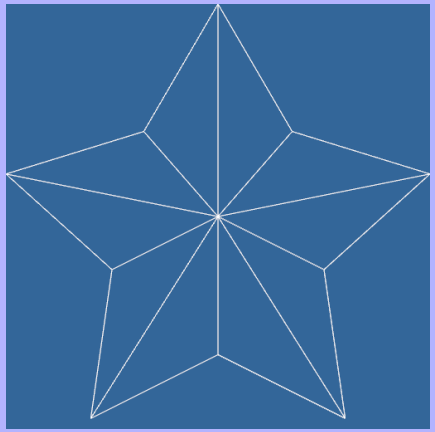
- Some of the traceback isn't useful for us
  - Anything in gl.py is just calls to GL
- But here's something more important:
  File "main.py", line 64, in setup
  glPolygonMode(GL_FRONT_AND_BACK, GL_LINES)

Docs

- Look up documentation (https://www.khronos.org/opengl/wiki/GLAPI/glPolygonMode)
- Aha! For mode, it says:
  Specifies how polygons will be rasterized. Accepted values are GL_POINT, GL_LINE, and GL_FILL. The initial value is GL_FILL for both front- and back-facing polygons.
- We have GL_LINE***S***!

Result

- After fixing the problem:



Updates

- Having a static display isn't very interesting
- How do we update virtual world state?
- We'll assume code is set up like so:

```
def update():
    ...
def draw():
    ...
def main():
    ...
    (**)
```

- The code we're going to look at fits where the (**) is

Option 1

- The simplest way: Go flat-out:

```
while True:
    update()
    draw()
    SDL_GL_SwapWindow(win)
```

- What's the problem of this approach?

Problem

- Program will run at different speeds on different machines
  - Faster on more high-powered one
  - Slower on older one
- And it pegs the CPU at 100%
  - Not good for mobile platforms (battery life)
  - More heat
  - Noisy fan

Option 2

- Tie execution to “wall clock time”
- Suppose we want to target 60 fps
  - 60 fps = 1/60 = 16msec per frame
- SDL provides a couple of timing functions:
  - SDL_GetPerformanceCounter()
    - Returns integer: Number of ticks since some point in the past
  - SDL_GetPerformanceFrequency()
    - Returns integer: Number of ticks per second
- And: SDL_Delay(milliseconds)

Code

- Suppose we want 60 fps
- Define some code:

```
def main():
    ...SDL initialization stuff...
    DESIRED_FRAMES_PER_SEC = 60
    DESIRED_SEC_PER_FRAME = 1/DESIRED_FRAMES_PER_SEC
    DESIRED_MSEC_PER_FRAME = int(DESIRED_SEC_PER_FRAME * 1000)
    TICKS_PER_SECOND = SDL_GetPerformanceFrequency()
    setup()
    ...
```

Code

- Then, in main():

```
def main():
    ...
    lastTicks = SDL_GetPerformanceCounter()
    while True:
        nowTicks = SDL_GetPerformanceCounter()
        elapsedTicks = nowTicks-lastTicks
        lastTicks = nowTicks
        elapsedMsec = int(1000 * elapsedTicks / TICKS_PER_SECOND)
        update(elapsedMsec)
        draw()
        SDL_GL_SwapWindow(win)
        endTicks = SDL_GetPerformanceCounter()
```

Update

- update() would have code like so:

```
def update(elapsedMsec):
    ...
    spaceship.position = spaceship.position + elapsedMsec *
       spaceship.velocity
    ...
```

- Analyze: What's good/bad about this approach?

Analysis

- Speed of objects in game world is keyed to wall clock time
- Nothing keeps program from using 100% CPU however

Option 3

- Add a delay at end of function:

```
def main():
    ...
    lastTicks = SDL_GetPerformanceCounter()
    while True:
        nowTicks = SDL_GetPerformanceCounter()
        elapsedTicks = nowTicks-lastTicks
        lastTicks = nowTicks
        elapsedMsec = int(1000*elapedTicks / TICKS_PER_SECOND)
        update(elapsedMsec)
        draw()
        SDL_GL_SwapWindow(win)
        endTicks = SDL_GetPerformanceCounter()
        frameTicks = endTicks - nowTicks
        frameMsec = int(frameTicks / TICKS_PER_SECOND * 1000)
        leftoverMsec = DESIRED_MSEC_PER_FRAME - frameMsec
        SDL_Delay(leftoverMsec)
```

- There's still a problem. What is it?

Problem

- If frame takes too long to render, leftoverMsec becomes negative!
- We need a check:
  if leftoverMsec > 0:
      SDL_Delay(leftoverMsec)

Analysis

- This works even if system is slow or fast
  - Slow system: We call update less frequently but with a bigger time step on each call
- But: Still a problem: Nondeterministic

Explanation

- From frame to frame: Amount passed to update() will vary slightly
  - Many physics simulations rely on having a fixed timestep
  - And: If we get bugs, it's hard to diagnose/reproduce them
    - Times for update() hard to reproduce
  - Also: Human eye sensitive to varying refresh rate
    - Better to run at constant 40fps than to swing between 40fps and 60fps

Option 4

- We can think of having two “timelines”
  - Wall clock time
  - Game world time
- Suppose we decide game world time quantum is 5msec
  - This means every time we call render(), it renders world state as it existed at time t, where t = some multiple of 5ms
  - Likewise for update(): When it returns, game world state is how it existed at time t

Example

- Suppose game world time quantum is 5ms
- Suppose we begin an iteration of main game loop
  - 10ms have passed since the last time we started an iteration of the loop
  - Then we call update() twice, advancing game world state by 10ms
  - And then we call render() once

Example

- Again, game world time quantum is 5ms
- Suppose 12ms passed between the last time we started iteration of main loop and when we started the current iteration
  - Call update() once. Advances game world state by 5ms
  - Call update() again. Advances game world state by 5ms
  - Can't call update() a third time: Would push game world state too far ahead
  - So we just remember that we had 2ms left over
  - And we'll use that fact on the next iteration

Code

```
def main():
    UPDATE_QUANTUM_MSEC = 5
    ...
    lastTicks = SDL_GetPerformanceCounter()
    accumElapsedMsec = 0
    while True:
        nowTicks = SDL_GetPerformanceCounter()
        elapsedTicks = nowTicks-lastTicks
        lastTicks = nowTicks
        elapsedMsec = int(1000 * elapedTicks / TICKS_PER_SECOND)
        accumElapsedMsec += elapsedMsec
        while accumElapsedMsec >= UPDATE_QUANTUM_MSEC:
             update(UPDATE_QUANTUM_MSEC)
             accumElapsedMsec -= UPDATE_QUANTUM_MSEC
        draw()
        SDL_GL_SwapWindow(win)
        endTicks = SDL_GetPerformanceCounter()
        frameTicks = endTicks - nowTicks
        frameMsec = int(1000 * frameTicks / TICKS_PER_SECOND)
        leftoverMsec = DESIRED_MSEC_PER_FRAME - frameMsec
        if leftoverMsec > 0:
            SDL_Delay(leftoverMsec)
```

Problem

- One problem remains: Render is not synchronized with current game state
- Suppose we render at wall clock time 1004
- Game will render state as it existed at time 1000
  - 5 msec quantum
- To get even more accurate display:
  - Pass accumElapsedMsec to draw()
  - draw() will render scene as it will exist that many milliseconds in the future

Keys

- Need to keep track of which keys are down
- We can do this with a global
- To keep things organized, put all globals in a file “globs.py”

Example

- Our initial globs.py:

```
#globs.py

keyset = set()
```

Main

- Then in main.py, we import:
  import globs
- When a key is pressed: Inside update():
  if ev.type == SDL_KEYDOWN:
    globs.keyset.add(ev.key.keysym.sym)
- When key is released:
  if ev.type == SDL_KEYUP:
    globs.keyset.discard(ev.key.keysym.sym)

Keys

- Example of how we might use the key state:

```
if SDLK_a in globs.keyset:
    hero.moveLeft()
if SDLK_d in globs.keyset:
    hero.moveRight()
if SDLK_SPACE in globs.keyset:
    hero.jump()
```

Audio

- Audio is an important part of an immersive environment
- SDL provides easy way to play music (if you have the SDL_Mixer library)
- Need to do: from sdl2.sdlmixer import *
- When calling SDL_Init:
  SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO)
  Mix_Init(MIX_INIT_OGG | MIX_INIT_MP3)
  Mix_OpenAudio(22050, MIX_DEFAULT_FORMAT, 1, 4096)
- At setup time, load sound files:
  globs.organSound = Mix_LoadWAV(os.path.join(“assets”, “organ.ogg”).encode())
  - Don’t forget to import os.path!

Play

- To play the sound:
  rv = Mix_FadeInChannelTimed(chanNum, globs.organSound, numLoops, fadeTime, playTime)
  - chanNum tells which channel to use (-1 means “choose a free one”)
  - numLoops says how many times to repeat (0=play once, 1=loop once, -1=loop forever, etc.)
  - fadeTime tells how many milliseconds to take to fade the sound in
  - playTime gives a maximum duration for playing/looping the sound, in milliseconds (-1=unlimited)
  - The return value is the channel number or -1 for error (use Mix_GetError() to get an error string)

Play

- To stop playing the sound:
  Mix_FadeOutChannel(chanNum, fadeTime)
  - chanNum says which channel to use (-1 means “all channels”)
  - fadeTime is in milliseconds

Assignment

- When the user presses the spacebar, begin fading the background color (i.e., from black to red)
- When the user releases the spacebar, change the background back to black and draw a hexagon (or circle if that's what you did in the previous lab) to the screen
- If the user presses the spacebar while the hexagon/circle is on the screen, again begin fading the background color. When the spacebar is released, draw another hexagon and re-set the background color.
  - (It will be on top of the first hexagon, so it won't look like much...for now.)
- More info follows...

Assignment

- Each hexagon has a lifetime of 750 milliseconds, after which it disappears.
- Your code must be efficient: **Don't re-create buffers every frame**: This leaks memory
  - [-30%] for memory leaks
- More follows...

Assignment

- Bonus [+10%]: Add sound when the spacebar is released
- Don't play the sound on key auto-repeat!
  - **No points** if you submit sound files that infringe copyright: Either make your own or find ones that are licensed under Creative Commons or some other suitable license
  - Cite your source: **You get a zero on the lab if you have any uncited resources**
    - Put the citation in assets/sources.txt
  - Keep the size of the audio file(s) under 100KB total; use either Ogg Vorbis or wave (Vorbis is usually much smaller than wave; MP3 has a brief delay that makes it unsuitable – this is a side effect of the codec's compression scheme)

Sources

- Robert Nystrom. Game Programming Patterns. Genever Benning.
- Various authors. Vertex Specification. https://www.khronos.org/opengl/wiki/Vertex_Specification#Vertex_Array_Object

Created using LaTeX.

Main font: Gentium Book Basic, by Victor Gaultney. See http://software.sil.org/gentium/
Monospace font: Source Code Pro, by Paul D. Hunt. See https://fonts.google.com/specimen/Source+Code+Pro and http://sourceforge.net/adobe
Icons by Ulisse Perusin, Steven Garrity, Lapo Calamandrei, Ryan Collier, Rodney Dawes, Andreas Nilsson, Tuomas Kuosmanen, Garrett LeSage, and Jakub Steiner. See http://tango-project.org