Introduction to zenilib

The Zenipex Library

Mitchell Keith Bloch

University of Michigan
2260 Hayward Street
Ann Arbor, MI. 48109-2121
bazald@umich.edu

September 16, 2013

Outline

http://zenilib.com

- Full walkthrough to get you started
- Reference DOxygen-based documentation
- Tutorials / Examples

Design

- Portable Code
  - Runs on multiple operating systems, architectures
  - Previous students ported a game to iPhone
- Extensible
  - Objected Oriented Programming patterns
- Usable
  - Most API features of zenilib have clear use patterns
  - Efficiency is a high (but secondary) priority
  - Fairly powerful features are exposed

premake4-based Build System

- Modular design philosophy
  - Can use .DLL/.so files that I provide and build just your application
  - zenilib bugs can be fixed without game source code
  - Possible to use just zeni & zeni_audio, without using my full engine
  - OpenAL and Direct3D checked for at runtime
- A build system guide is available on my website
  - multi-premake_sh.bat regenerates project files for all platforms
    - Can generate them in a temp directory, such as “C:\temp”, saving AFS space and speeding your build times.
  - multi-build_sh.bat can build 32/64-bit in debug and release mode
  - multi-clean_sh.bat clears junk out of your zenilib directory

Licensing

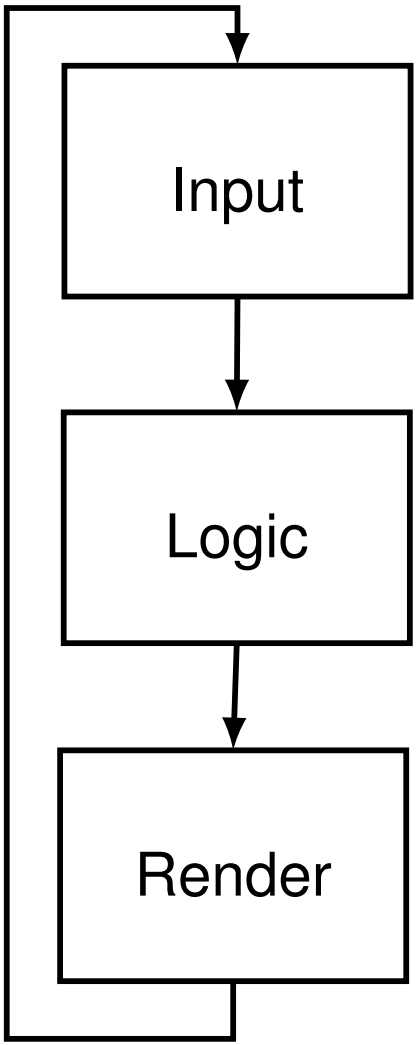
- zenilib is Free/Libre/Open Source Software
- Licensed to you under the LGPL v3
- Compiled executables must come with a copy of the GPL and either
  - A copy of zenilib source code with any changes you've made
  - A written ofter to provide a copy, free of charge
- Your code is your own (non-viral)
- Read the LGPL v3 sometime.

Outline

Organize Well

- Use classes for game objects
- Guarantee hardware independence
  - Resolution independence
  - Framerate independence

Game Loop

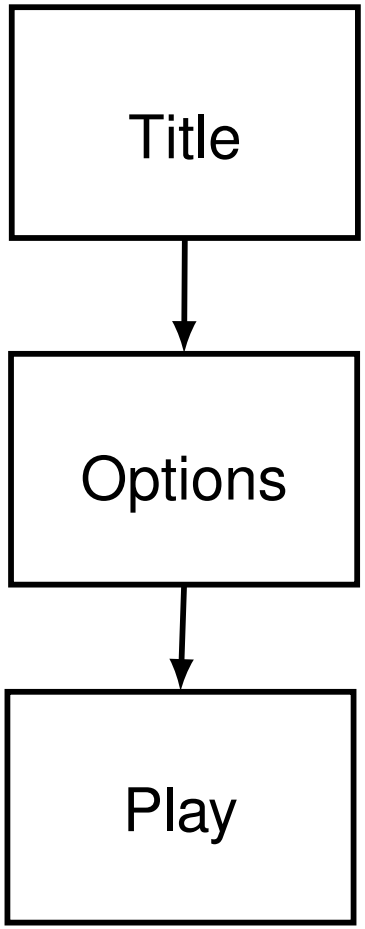


- Input
  - Process all new input events
- Logic – the “fun” part
  - Everything but input and rendering
  - Timing aware
- Render
  - Draw everything to the screen
  - ***NO*** game logic

Game Organization

- One class per state
- Each `Gamestate` offers
  - Input event callback functions
  - A `perform_logic` function
  - A `render` function (and a `prerender` function)
  - `on_push` and `on_pop` functions
- `Gamestate` stack is convenient

Example Game State Stack



- Typically stores states from the title screen to a state with actual gameplay
- Possible to discard unnecessary states rather than building up the stack

Outline

Gamestates

- `Game.h`
  - Stack of `Gamestate` objects
  - `replace_Popup_Menu_State_Factory(...)` allows you to replace the popup menu
  - `replace_Popup_Pause_State_Factory(...)` allows you to replace the pause screen
- `Gamestate.h`
  - `Gamestate_Base` is base of all states
- `bootstrap.cpp`
  - Modify `Bootstrap::Gamestate_One_Initializer::operator()` to return an instance of your title state
  - You are free to modify `main`, but you probably don't have much reason for doing so

Singleton Interfaces

- `Game.h` ← `Gamestate` stack
  - Get approximate FPS
  - Access debugging console
- `Timer.h` ← Get Timing information directly
  - Prefer `Chronometer<Time>` for game logic
- `Video.h` ← Rendering device interface
- `Sound.h` ← Sound device interface
- `Net.h` ← Provides a socket interface

Singleton Interfaces

- `Game.h` ← `Gamestate` stack
  - Get approximate FPS
  - Access debugging console
- `Timer.h` ← Get Timing information directly
  - Prefer `Chronometer<Time>` for game logic
- `Video.h` ← Rendering device interface
- `Sound.h` ← Sound device interface
- ~~`Net.h` ← Provides a socket interface~~

Singleton Databases

- `Colors.h` ← `Color` objects
- `Fonts.h` ← `Font` objects
- `Sounds.h` ← `Sound_Buffer` objects
- `Textures.h` ← `Texture` objects

XML configuration files in `zenilib/assets/config/`
and accessible through the IDEs

Game Loop (Again)

- Input handlers (callback functions)
  - Keep code lightweight
  - Called *many* times per frame
- `perform_logic()`
  - Do all game logic
  - Should handle variable time steps correctly
- `prerender()`
  - Logic strictly required to happen before rendering
- `render()`
  - ***NO*** rendering calls outside this function
  - ***NO*** game logic inside this function
    - Commenting this out shouldn't break *anything*

Polling for Input

- Input handlers listed in `Gamestate.h`
- Override ones of interest in derived `Gamestate`
- See SDL (or zenilib) Documentation if autocomplete is failing you
  - zenilib documentation for SDL_K*
  - See SDL documentation for SDL_K*

2D Rendering

- `get_Video().set_2d(...)`
  - If never called, will get 800x600
  - Default (no args) is resolution dependent
- `Triangle` and `Quadrilateral` are primitives
  - Composed of `Vertex2f_Color` or `Vertex2f_Texture` objects
- Helpers in `EZ2D.h`
  - `render_image(...)`
  - `play_sound(...)`
  - `Sprite` manipulation

3D Rendering

- `Camera.h` ← 3D `Camera`
- `Fog.h` ← distance-based `Fog`
- `Light.h` ← `Light`
- `Model.h` ← 3D `Model`
  - .3ds is the only supported format
  - Getting animations to work can be tricky
    - Test *iteratively* and ***often***
    - Don't use mesh morphs

Math and Coordinates

- `Coordinate.h` ← Point2i, Point2f, Point3i, Point3f
- `Vector2f.h` ← Vector2f
- `Vector3f.h` ← Vector3f
  - Useful for representing directions, linear velocity, linear forces
  - Can be helpful to use 3D coordinates in 2D
- `Quaternion.h` ← `Quaternion`
  - Useful for representing rotations, rotational velocity, rotational forces
- `Matrix4f.h` ← `Matrix4f`

Odds and Ends

- `String` ← Debug/Release CRT independant string class
  - Can construct from C-style string and `std::string`
  - Provides `c_str()` and `std_str()`
- `Chronometer<Time>` ← timing information
  - `Chronometer<Time_HQ>` ← sub-millisecond precision
  - Timer and Timer_HQ bypass the pausing system
- `get_Game().get_fps()` ← framerate estimate
- `Collision.h` ← 3D collision detection “primitives”
- `Widget.h, Widget_State.h` ← help with Widget creation and management

More Odds and Ends

- `get_Game().get_console()` ← access debug console
  - Requires `#ifndef NDEBUG`
  - `get_Game().write_to_console(...)` is simpler
  - Pull up the console in game with `ALT-`` `
- `Projector.h` ← helpers for transforming coordinates between screen and world coordinates
- `Random.h` ← simple pseudo-random number generator

Outline

Modernization (In Progress)—For OpenGL Wizards

- The OpenGL Shader rendering system is new, supporting OpenGL 3.2-4.4, compatibility profile only (for now)
- OpenGL ES shaders are implemented using the Angle project
  - But not for Direct3D9 (implementation requires an extra step I haven't figured out yet—see my question in the Google group
  - I do not yet provide an API for argument passing
- Faster batch render calls that I haven't abstracted away yet are demonstrated in `zenilib/jni/application/modern.h`, including example shader use
  - Ignore that the example works for Direct3D9—it's a hack
  - `Vertex_Buffer` should still be faster
  - I recommend you stick to functionality provided by my API unless you know what you're doing—you can get plenty of speed and great graphics without any of this

Help

- Skim header files
- Check out online documentation
- Ask on the phorum

Next Time

I will do the ZeniTank tutorial