04 Hardware Rendering

Functional and Implementation Guidelines

Functional Guidelines

1. Your application should implement a functionality to for central config loading
   1. load configuration about the MonitorWindow (or alternative class)
   2. load configuration about the assets (their location)
2. Refactor your Engine (or alternative class)
   1. Create a new Game class (or an alternative one)
   2. Move all game specific logic inside a new Game class
   3. Keep only engine related logic inside the Engine
   4. Have a way to “draw” assets from the game into the engine draw method
3. Implement a Renderer class
   1. Support for hardware accelerated Textures (GPU primitive)
   2. Refactor the application to use the newly created renderer
   3. Switch all CPU primitives to GPU primitives

Implementation Guidelines

1. Implement EngineConfigLoader
   1. EngineConfig struct

| struct EngineConfig {  MonitorWindowConfig windowCfg;  //soon to populate other configs  }; |
| --- |

* 1. Implement loadEngineConfig() method

| EngineConfig loadConfig() {  EngineConfig cfg;  cfg.windowCfg.displayMode = DISPLAY\_MODE;  cfg.windowCfg.windowWidth = SCREEN\_WIDTH;  cfg.windowCfg.windowHeight = SCREEN\_HEIGHT;  cfg.windowCfg.windowName = WINDOW\_NAME;  return cfg;  } |
| --- |

1. Implement Game struct
   1. GameConfig

| enum GameImages {  UP, DOWN, LEFT, RIGHT, PRESS\_KEYS, LAYER\_2  };  struct GameConfig {  std::unordered\_map<GameImages, std::string> imgLoadPaths;  }; |
| --- |

* 1. Populate the GameConfig inside the EngineConfigLoader
  2. Move everything non-Engine related inside the game

| class Game {  SDL\_Surface \* \_currChosenKeyImg = nullptr;  std::unordered\_map<GameImages, SDL\_Surface \*> \_gameImgs;  }; |
| --- |

* 1. Public API

| int32\_t init(const GameConfig &cfg);  void deinit();  void draw();  void handleEvent(const InputEvent &e); |
| --- |

1. The game has no way to directly “draw” its widgets
2. The engine should not expose its internals (drawing primitives) to the game
3. Use a vector to store an arbitrary amount of widgets for the frame

| class Engine {  MonitorWindow \_window;  Renderer \_renderer;  InputEvent \_event;  Game \_game;  std::vector<SDL\_Surface\*> \_gameTextures;  }; |
| --- |

* 1. Change the game drawing API to accept and populate the new vector

| void draw(std::vector<SDL\_Surface\*> &outSurfaces); |
| --- |

* 1. Push assets into the vector

| void Game::draw(std::vector<SDL\_Surface\*> &outTextures) {  outTextures.push\_back(\_currChosenKeyImg);  } |
| --- |

1. Experiment with what happens when you draw more widgets
2. **Image layering - Lecture**
3. Check your CPU load
   1. So far we were using only the CPU - now we will use hardware acceleration (GPU)
4. **Double buffering - Lecture**
5. Implement the Renderer struct
   1. Structure

| //Forward declarations  struct SDL\_Renderer;  struct SDL\_Texture;  struct SDL\_Window;  class Renderer {  //forbid the copy/move constructors and assignment operators  SDL\_Renderer \*\_sdlRenderer = nullptr;  }; |
| --- |

* 1. SDL\_Texture - cross-platform GPU primitive
  2. Public API

| int32\_t init(SDL\_Window \*window);  void deinit();  void clearScreen();  void finishFrame();  void renderTexture(SDL\_Texture \*texture); |
| --- |

* 1. SDL internal methods

| SDL\_CreateRenderer() //pass SDL\_RENDERER\_ACCELERATED as flags  SDL\_SetRenderDrawColor()  SDL\_DestroyRenderer()  SDL\_RenderClear()  SDL\_RenderCopy()  SDL\_RenderPresent() |
| --- |

1. Expand the Texture set of functions
   1. createTextureFromSurface() //don’t forget to free the SDL\_Surface
   2. freeTexture()
   3. setRenderer() //helper method
   4. SDL Internal methods

| SDL\_CreateTextureFromSurface() SDL\_DestroyTexture() |
| --- |

1. Transform the Game’s SDL\_Surfaces to SDL\_Textures
2. Check your CPU usage again