FILE HIERARCHY

+ Solution directory

- · Solution file (*.sln)
- · Project file (*.vcxproj)
- **+ Assets** [static resources used in the game]
  - · GameConfig.xml
  - · TextureAssets.xml
- **+ Source** [.h and .cpp files]
  - · Object.h
  - · Object.cpp
    - · Object01.h + Object01.cpp
    - · Object02.h + Object02.cpp
    - · …
  - · ObjectFactory.cpp
    - · ObjectFactory01.h + ObjectFactory01.cpp
    - · ObjectFactory02.h + ObjectFactory02.cpp
    - · …
  - · ObjectFactory.h
  - · Library.h
  - · Library.cpp
  - · Engine.h
  - · Engine.cpp
  - · InputDevice.cpp
  - · InputDevice.h
  - · View.h
  - · View.cpp
  - · GraphicsDevice.h
  - · GraphicsDevice.cpp
  - · Texture.h
  - · Texture.cpp
  - · Timer.h
  - · Timer.cpp

  Use the ones in the examples
- **+ ThirdParty** [third-party libraries]
  - + tinyxml2

  No action required. Just include here the required sources

+ sdl2

+ Game

· Game.exe

+ Temp

· Intermediate_obj_files (*.obj)

· Intermediate_pdb_files (*.pdb)

No action required. Generated automatically

XML RESOURCE FILES

We use an xml to define the objects we might have in the game, their properties, and the values we will use to initialize every new instance created out of them.

We can also include in the xml file information regarding the global configuration of the games such as screen size, fps, font type used, input devices, etc.

What do we consider an object in the game?

Anything that will be drawn on screen or controlled by the user can be considered an object. For example: the player (main character of the game), enemies, weapons, elements in the scenario, etc.

Example

```xml
<Game>
  <Screen width ="1024" height ="768"></Screen>
  <FPS>100</FPS>
  <Devices>
    <Input create="true">
      <InputEvent inputEvent ="" gameEvent=""></InputEvent>
    </Input>
  </Devices>
  <Objects>
    <Player name="Player">
      <Position x="10.0f" y="-10.0f" angle="0.0f"/>
       <Asset name="texture" path="folder1/image.png"/>
    </Player>
    <SwordEnemy name="SwordEnemy">
      <Position x="1.0f" y="40.0f" angle="0.0f"/>
       <Asset name="texture" path="folder1/image.png"/>
    </SwordEnemy>
    <BowEnemy name="BowEnemy">
      <Position x="10.0f" y="2.0f" angle="0.0f"/>
       <Asset name="texture" path="folder1/image.png"/>
    </BowEnemy>
  </Objects>
</Game>
```

Object class

- Parent of any class you might need to create to represent a specific object type in your game.
- **IMPORTANT**. It is an abstract class. We just create to define the behavior and properties any object in the game might have.

Properties

- Position on screen. Two options
  - float x, y
  - Vector2D position (struct)
- Rotation angle. float angle
- Renderer. It will be used for the implementation of the draw method in any child class to draw the texture on screen
- Texture. It will be used for the implementation of the draw method in any child class

Methods

- Constructor. It initializes all the member with the values we pass in as arguments
- Deconstructor
- Update. `virtual std::unique_ptr<Object> update() = 0`
  - It is a pure virtual method. No implementation. Why? Because the update method is responsible for keep the properties of the object updated. Since every object might have its own properties (besides the ones in common defined in the object parent class), every child class needs to provide an implementation for the update method.
- Draw. `virtual void draw(View*) = 0`
  - It is a pure virtual method. No implementation. Why? Because the draw method is responsible for rendering the object on screen. Since every object might have its own sprite, every child class needs to provide an implementation for the draw method.

MyObject01 class

- Class that represents a specific object in the game.
- We need to provide here an implementation for the update and draw methods

Methods

- Constructor / Initialize. Here we need to construct the clipping sequence with the sprites I have in the texture

- HandleEvent. It translates the captured event into updates to be applied to some properties of the object
- Update. std::unique_ptr<Object> update()
    - Responsible for keep the properties of the object updated. Typically is here where we need to apply the changes we inferred on the HandleEvent method
- Draw. void draw(View*)
    - Responsible for rendering the object on screen. The sprite associated with the object is rendered here, adjusted by the view

ObjectFactory class

Since a game object will be fairly complex, we won't use the constructor to create a new instance. We will use an ObjectFactory instead.

- This ObjectFactory class is the parent of any class you might need to create to represent a factory for a specific object in the game.
- **IMPORTANT**. It is an abstract class. We just create to define the behavior and properties that are common to any factory class.

Properties

Methods

- Create. virtual std::unique_ptr<Object> create (tinyxml2::XMLElement* objectElement) =0;
    - It is a pure virtual method. No implementation. Why? Because the create method is responsible for creating a new object given a set of properties.
    - Since we will have different properties for every object type, every child factory class needs to implement the create method.
    - Properties used to initialize the created object are provided by means of an XML element created when reading the xml resource file.
    - The XML parsing for the object should be done within the create method of the corresponding Factory class

Library class

Class that will be used to keep information about the object factories available to create the different objects we might have in the game.

So when we need to create a new game object, it is necessary to retrieve from the library the corresponding factory, and then use the latter to create the object

Properties

- Map in which every object type identified with a string is associated with the corresponding factory.
    - std::map<std::string, std::unique_ptr<ObjectFactory>> library
- Map in which we keep information about all the textures associated with the game objects [It is possible to create another xml file to define all the art resources to be used]
    - Std::map<string,std::shared_ptr<Texture>> artLibrary;

Methods

- Constructor
    - Here is where we populate the library map
- Deconstructor
    - We need to free up memory
- Method to get an art asset (Texture) given the key name
- Method to add a new art asset to the artLibrary map

Texture

The class that encapsulates the creation and rendering of a texture(image/sprite)

Properties

- The actual texture. SDL_Texture* sprite
- width
- height

Methods

- Constructor
- Destructor. Destroy the SDL texture and free memory
- Initialize.
    - Load the image given its path
    - Set the color key
    - Create the texture
    - Free the image (no longer needed)
    - Get and store the images's width and height as the textures' width and height
- Draw
    - Configure the render quad (the area where the texture will be rendered)
    - Render to screen

GraphicsDevice

The class that encapsulates the creation of the graphics window and the rendering of the game objects

Properties

- Screen. SDL_Window* screen
- Renderer. SDL_Renderer* renderer

Methods

- Constructor.
  - Initializes the SDL and the SDL_image subsystems
  - Constructs and initializes the graphic window (screen)
  - Initializes the renderer
- Destructor
  - Free the window
  - Free the renderer
  - Quit SDL and SDL_image subsystems
- Begin
  - Clears screen
- Present
  - The render presents ifself on the screen
- getRenderer

InputDevice

Class created to handle the user input

NOTE. Include an enum called gameEvent that represents all user input events for the game

Properties

- The current event. std::unique_ptr<SDL_Event> event
- Map to keep track of the keys' states. std::map<gameEvent, bool> keyStates;

Methods

- Constructor
  - Create the SDL event.
  - Initialize the keyStates map. Set "false" for every gameEvent
- GetEvent. Indicates whether the given gameEvent is enabled or not. It returns the Boolean value in keyStates for a given gameEvent
- Translate.
  - Maps the captured SDL event type to a gameEvent value
- Update
  - Polls the SDL event to get the latest event

    - Depending on the event type:
        - Translates. Maps the captured SDL event type to a gameEvent value
        - Updates the state for the gameEvent in the “keyStates” map

View

Class that represents the visible area of the game. It is used to adjust the object’s position for display

Properties

- Position. Can be expressed in two different ways
    - Vector2d position (struct)
    - float x, y
- Center. The same as position
- Input device. std::unique_ptr<InputDevice> inputDevice

Methods

- Constructor
- Update.
    - The current input event (gameEvent) is requested from the input device
    - The view shifts its position based on the event

Timer

Class that is used to keep the frame rate of the game stable

Engine class

- The brain of the game.
- It holds the vector of objects already included in the game, and an instance of the library that we can use to retrieve a factory object and then create a new object.
- Responsible for loading the game, meaning the creation of the several objects we might have. Those objects will be stored in a vector.
- Responsible for updating and drawing all the objects in the game on every game loop iteration. Every object is responsible to update and render itself. So the online thing that the engine does in this respect is to invoke for every object in the vector the update and draw methods.

Properties

- The library with both the objects and the art assets.
    - std::unique_ptr<Library> library;

- The several objects already present in the game.
    - std::vector<std::unique_ptr<Object>> objects;
- A graphics device. std::unique_ptr<GraphicsDevice> graphicsController
- An input device. std::unique_ptr<InputDevice> inputController
- A view. std::unique_ptr<View> view
- A timer. std::unique_ptr<Timer> frameRateController

Methods

- loadLevel(std:::string levelPath).
    - Here is where we load the objects that we will have in the game according to the xml resource file located in the given path. Here we need to load the art assets as well.
    - We initialize the Input Device, the View, and the Graphics Device members
    - **NOTE**. If the art assets are defined in a separate xml file it is necessary to provide a second argument with the assetPath
- Constructor
    - Here is where we initialize the engine
    - We initialize the library
    - Here we call the loadLevel method
- Destructor
    - Call reset
- Reset.
    - We reset all the members
    - We clear the vector of objects
- Update
    - Go through the objects in the vector and invoke its handle event method to translate the captured event into updates to be applied to some properties of the object.
    - Go through the objects in the vector and invoke its update method
- Draw. Go through the objects in the vector and invoke its draw method
- Run.
    - The main game loop.
    - Here is where we will ask the engine to update and then draw all the objects in the game
    - We use the frame rate here to keep the game's fps stable

Main program

```
int main(int argc, char* argv[]) {

	// Define the paths where the xml files can be found

	std::unique_ptr<Engine> engine{ std::make_unique<Engine>(levelConfig, 
libraryConfig) };
	while (engine->run()) {}
	engine = nullptr;
	return 0;
}
```