WarCraft II Internal Documentation

Linux

Game Logic

- Starting from main, a CApplicationData instance is created. It initializes saved settings for the game, the most of private member functions, and maps the controls for each action to a key.
- After an instance of CApplicationData is created, it calls its Run() function which invokes a chain of function calls to ultimately call g_application_run() in package gtk+-3.0
- After the gtk package sets up the game, it calls the function ActivateCallBack() which starts the activation process of the game.
- In CApplicationData, the function Activate() is called; this function creates the main window, creates the drawing area for gtk and initializes all the colors and stylistic attributes of the game including the sound. The documentation for the Activate() function is in the src code.
- Based on the map that is chosen, bin stores data for each different map. This is loaded into the game from the files in order for the maps to contain their own special attributes.
- After the map is loaded, Battle Mode is entered. This consists of the game logic that runs the actual battle of the game until the game is over. We have significant documentation of Battle Mode in the src code.
- An important part of the game logic is the function ChangeApplicationMode(). This function switches modes during the game; basically whenever a user goes to a new screen in the game this function was invoked to change the application mode. This is how the game changes state when going from the main menu to actually starting a game. Similarly this is how the game returns to the main menu when the game is over.

Classes

- **CApplicationData**
    - Class CApplicationData contains all the information we need to run the game. CApplicationData() initializes all the information we need to run the game.
    - The Activate() function invokes any events that require a different graphic. This includes loading different colors, fonts or graphics for attributes. Certain attributes can change states so some events will invoke a different graphic and this function activates these certain graphics.
    - The CopyGameStatus() function is used for save and load single player game.
    - The Instance() function invokes the constructor of CApplicationData, creates a shared pointer of CApplicationData, and returns that pointer.
    - RenderMenuTitle() sets up the menu display window with the given parameters, it sets the fonts, text color, shadows etc.
    - The Run() function is called from the main.cpp and begins the chain of different Run commands that start up the game. It invokes CGUIApplicationGTK3::Run(int argc, char

*argv[]) where it further invokes g_application_run() in package gtk+-3.0. Timeout() will be invoked every 50 ms. It will get all inputs from the player, such as: the location of the mouse, the number of left click and right click. It will calculate all the outputs of players' input. It then renders the right graph to players.

- **CApplicationMode**
    - Class CApplicationMode represents the "state" of our game. Most "Mode" classes only have functions, not data. All the necessary data is provided by "context", which is a shared_ptr of Class CApplicationData.
- **CAssetDecoratedMap**
    - Class CAssetDecoratedMap contains all information needed to render the map.
    - LoadMap() loads maps from the directory "map." Note that this function will only be called once during the whole game.
    - SAssetInitialization and SResourceInitialization structs are used to store information about each player's assets and resources, when the game begins, respectively.
- **CBattleMenuMode**
    - CBattleMenuMode() prepares all the information we need to exhibit during the battle.
- **CBattleMode**
    - InitializeChange() loads the game map that was selected in CApplication as well as the song to be played. This is done through the shared pointer 'context'.
    - Input() handles the key input and the behavior behind it. This includes basic arrow key input to signal movement. Implements CApplicationMode.
    - Instance() creates a shared pointer of the class BattleMode if one does not exist and returns it. If one does exist then it returns the existing one.
    - Render() is called to create the battlemode window and drawing area. Information about the state of the game is stored in context and Render() uses the context to render the main battlemode window. Implements CApplicationMode.
- **CButtonMenuMode**
    - Input() will call the corresponding callback function whenever players left click over a button.
    - DButtonTexts vector contains the texts on the button.
    - DButtonFunctions vector contains the callback functions of buttons.
    - DButtonLocations vector contains the locations and size of the button.
- **CEditOptionsMode**
    - String DTitle contains title of menu. Vector DButtonTexts contains the texts over the buttons. Vector DButton Functions contain the corresponding callback function of button. Vector DButtonLocations records each button’s location. Bool DButtonHovered checks if the mouse is over buttons.
    - Int DEditSelected indicates which locations we are gonna edit, -1 means no edit operation. Int DEditSelectedCharacter returns the selected edit text, which character is selected by users. Vector DEditLocations() contains the locations and sizes of input boxes.

    - Vector DEditTitles contains the titles of each input box. Vector DEditText contains the texts in the input boxes. Vector DEditValidationFunctions contains the validation functions.
- **CFileDataSource**
    - Class CCFileDataSource contains the absolute path to the src file and its corresponding file descriptor. Read() reads @length bytes into @data. It returns how many bytes are read.
- **CGameModel**
    - CGameModel() is a copy constructor. We need this to save the old status of the game model and load it in the future.
    - Given an asset id, Asset() returns a shared_ptr of that asset.
    - DActualMap is the map we want to store when storing the game.
    - DForestTime keeps track of the tree growth time.
    - DGameCycle stores all players' information.
- **CCGraphicSurface**
    - Class CCGraphicSurface contain all the information we need to exhibit a GUI
- **CGUIFactory**
    - Class CGUIFactory contains functions needed to create GUI
- **CGUIApplicationGTK3**
    - NewWindow() creates a new main window.
- **CMainMenuMode**
    - CMainMenuMode() prepares all the information we need to exhibit the main menu.
    - MultiPlayerGameButtonCallback() invokes function CApplicationData::ChangeApplicationMode.
    - SinglePlayerGameButtonCallback() invokes function CApplicationData::ChangeApplicationMode.
- **CMapRenderer**
    - Vector DPixelIndices stores infos about which color should be used on each tile type.
- **CMapSelectionMode**
    - Class CMapSelectionMode contains all the information we need to exhibit map-selection interface. CMapSelectionMode() prepares part of the information we need to exhibit map selection maps.
    - SelectMapButtonCallback() will be invoked when players press the "Select Map" button on the map-selection interface.
    - DButtonFunctions vector contains corresponding callback functions of buttons. DButtonLocations contain locations of the button.
- **CMapServerSelectionMode**
    - Class CMapServerSelectionMode contains all the information we need to exhibit server map-selection interface.
    - CMapSelectionMode() prepares part of the information we need to exhibit map selection maps.
    - SelectMapButtonCallback() will be invoked when players press the "Select Map" button on the map-selection interface.

- **CNetworkOptionsMode**
    - NetworkOptionsUpdateButtonCallback() is the callback function of "update" button. Only update network option if and only if all the edit texts are valid.
    - Bool ValidHostnameCallback() checks whether argument str is a valid host name. A valid hostname must be at least one characters long and shorter than 253 characters. It must be composed of alphanumerics. It can contain dots, but cannot begin with dot. Characters between dots must be less than 64.
    - ValidPortNumberCallback() checks whether port number is a valid one. A valid port number must be an integer between (1024,65535] without other characters.
- **CPixelPosition**
    - SetFromTile() transforms a tile position into its corresponding pixel position at its center.
- **CPlayerAIColorSelectMode**
    - PlayGameButtonCallback() will be invoked when players press "Play Game" button in the PlayerAiColorSelect interface.
- **CPlayerAsset**
    - Class CPlayerAsset contains all the extrinsic properties of a kind of asset.
    - CPlayerAsset() contains a unique Asset ID and a random number assigned each turn. operator< is overridden for shared_ptr<CPlayerAsset> so that we can call list::sort to sort all assets at once.
- **CPlayerAssetType**
    - Class CPlayerAssetType contains all the intrinsic properties of a kind of asset.
    - AddCapability() is used to add certain kinds of capability to an asset type. It could be invoked when an asset type is upgraded and gains new capabilities.
    - AssetRequirements() returns all the asset requirements needed before building this kind of asset.
    - Construct() constructs an asset of CPlayerAssetType type and returns a shared_ptr of this newly created asset.
    - HasCapability() is used to determine whether an asset type has a certain capability.
- **CPlayerCapability**
    - Register() checks if a certain capability has been registered. If not, register that capability and return true; If yes, return false. It takes in an argument ‘capability’ which is the capability that needed to be registered.
    - String DName contains the capability name. DAssetCapabilityType is the Corresponding type of that capability.
- **CPlayerData**
    - Class CPlayerData contains all the information about the player.
    - CPlayerData() is a copy constructor we need to save the current status of the game.
    - CreateAsset() increases asset ID by 1, whenever a player creates an asset.
    - CreateMarker() creates a target position on the map where assets can move to that position.
    - AddGroup(): One can assign a building or a group of up to 9 units to a single key. To do this, select want to assign, then hold down Control and select a number on the keyboard

between 0-9. Then, to select the assigned, simply press the number of the group wanted. Pressing a group number twice will center the screen on the group.
    - FindAsset() returns the weak_ptr of the corresponding asset, given as asset ID. If we fail to find the asset, we will return an empty weak_ptr. It can be checked using weak_ptr::expire() function.
- **CPosition**
    - DistanceSquared() is used to calculate the squared distance between two positions.
    - Distance() uses Newton's method to calculate an approximate integer square root.
- **CTerrainMap**
    - Enum ETerrainTileType() contains all information about the type of terrain tiles.
    - Given an index of an tile, return its tile type.
    - DTerrainMap is an ETerrainTileType two dimensional vector containing all the tiles of the map.
    - DMap is an ETileType two dimensional vector containing all the tiles of the map.
    - DMapIndices is a two dimensional vector containing the indices of the map.
    - String DMapName contains the name of the map.
- **CTilePosition**
    - Imagine decomposing the window into hundreds of blocks, each block is one of "tile", CTilePosition contains all the information about that tile position.
    - SetFromPixel() transforms a pixel position to its corresponding tile position.
- **SGUIKeyType**
    - Class SGUIKeyType contains all possible input from the keyboard.

Web Server

API

- **Login:**
  Post /api/login
  Our login endpoint is *warcraft2.tech/api/login*
  Make a POST request to this endpoint using HTTP Basic Auth headers (username, password) and a JSON will be sent back. The account must already exist, but they can sign up at our webpage (since creating an account via API isn't one of our goals yet)
  This Basic Auth header will be required in many other API calls (for different endpoints), so it will be best to store this info in the client locally so that the user isn't prompted every time.
  Example JSON format will be:

```
{
    "success": True,
    "message": "Successfully authenticated.",
    "userId": "xxxxxxxxxxxxxxxxxxx",
    "idToken": "xxxxxxxxxxxxxxxxxxxxxxxxxx"
}
```

- **Get Player Stats:**
  GET /api/player-stats/<userID>

  A GET request to this endpoint will return a JSON of the user's information (replace <userID> with the player's user id). Example returned JSON:

```
{
  "email": "this.is.an.email@gmail.com",
  "lastLogin": 1581039320.9944017,
  "loginMethod": "web",
  "losses": 0,
  "maps": [
    {
      "mapLink":
"https://firebasestorage.googleapis.com/v0/b/ecs160.appspot.com/o/maps%2F
S5viIuFALEN7W5CYFfXnCtIJWXq2%2Fbay.map?alt=media",
      "name": "bay",
      "numPlayers": 3,
      "pngLink":
"https://firebasestorage.googleapis.com/v0/b/ecs160.appspot.com/o/maps%2F
S5viIuFALEN7W5CYFfXnCtIJWXq2%2Fbay.png?alt=media"
    }
  ],
  "name": "John",
  "userId": "xxxxxxxxxxxxxxxxxx",
  "wins": 0
}
```

- **Edit Player Game Stats:**
  POST /api/game-stats

  A POST request to this endpoint using HTTP Basic Auth headers with a JSON in the request will increment users' wins and losses accordingly, and return a success/fail response with the corresponding message. JSON format to provide in this request (replace xxxxxxxxxxxxxxxxx with user id's):

```
{
  "winner": "xxxxxxxxxxxxxxxxx",
  "loser": "xxxxxxxxxxxxxxxxx"
}
```

- **Push a Lobby:**
  POST /api/push-lobby

  A POST request to this endpoint using HTTP Basic Auth headers with a JSON in the request will add a lobby to the server's list of lobbies, and return the same exact lobby record, but now also including the auto-generated lobbyId in JSON format. JSON format to provide in this request (replace xxxxxxxxxxxxxxxxx with user id's)

```
{
  "userId": "xxxxxxxxxxxxxxxxx",
  "ipAddress": "123.123.123.12",
  "portNumber": "4124"
}
```

- **Get Lobbies:**
  GET /api/get-lobbies
  A GET request to this endpoint will return the server's list of lobbies in JSON format. Example returned JSON:

```
[
  {
    "ipAddress": "151.93.11.6",
    "lobbyId": "dZIulRshrxxzXVIGWTHa",
    "portNumber": 8765,
    "userId": "xxxxxxxxxxxxx"
  },
  {
    "ipAddress": "123.123.213.1",
    "lobbyId": "jqT6OUBB7UZN1bQCsrLo",
    "portNumber": 1923,
    "userId": "xxxxxxxxxxxxx"
  }
]
```

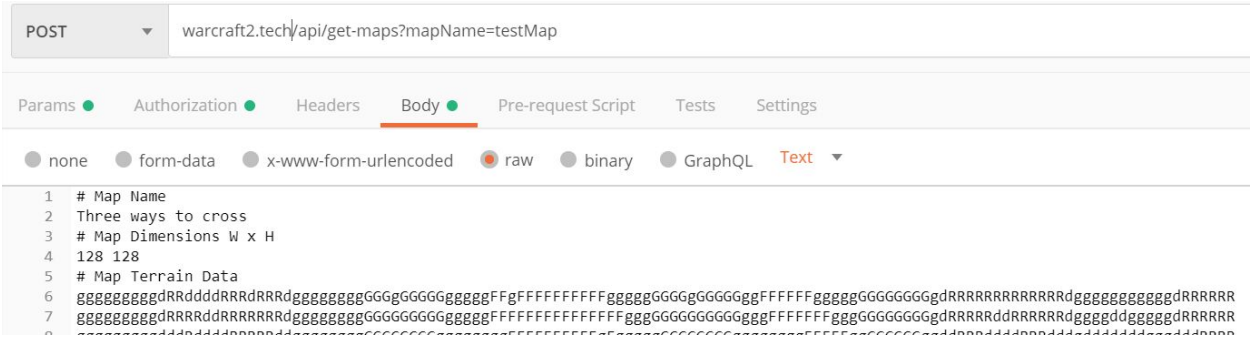
- **Pop Lobby:**
  POST /api/pop-lobby

A POST request to this endpoint using HTTP Basic Auth headers with a JSON in the request will pop a lobby from the server's list of lobbies. Will return a standard success/fail status JSON (same as auth and other api requests that require auth headers). JSON format to provide in this request (replace xxxxxxxxxxxxxx with lobby id):

```
{
  "lobbyId": "xxxxxxxxxxxxxx"
}
```

- **Push Map:**

  POST /api/push-map
  A POST request to this endpoint using HTTP Basic Auth headers with a .map file's text in the request body specified as raw text will upload the data as a map to the server. The text must be formatted in the same way a .map is formatted. The map name must be specified as a URL parameter, and using the same name as an existing map will override the old one:



- **Get Map:**

  POST /api/get-map
  A GET request to this endpoint using HTTP Basic Auth headers with a JSON in the request and will return a JSON with the user's maps containing the following fields:

```
[
 {
  "mapId": "xxxxxxxx",
  "mapLink": "url_to_download_the_map_on_firebase",
  "name": "xxxxx",
  "numPlayers": x,
  "pngLink": "url_to_the_map's_thumbnail",
  "public": false,
  "userId": "xxxxxx"
 },
 {
  "mapId": "",
  "mapLink": "",
  "name": "",
  "numPlayers": ,
```

```
    "pngLink": "",
    "public": false,
    "userId": ""
  }
 ...
]
```


File structure

/Ecs160web/
- /docker-compose.yml
- /nginx/
  - /nginx.conf
  - /Dockerfile
- /web/
  - /Dockerfile
  - /requirements.txt
  - /notifications/
    - /Dockerfile
    - /emailNotifications.py
    - /requirements.txt
    - /serviceAccountKey.json
  - /flaskapp/
    - /app.py
    - /app/
      - /__init__.py
      - /api.py
      - /auth.py
      - /database.py
      - /mapUtils.py
      - /routes.py
      - /static/
      - /templates/
      - /config/
        - /serviceAccountKey.json

/flaskapp/

- Contains the web servers main source code.

  /app.py

  - Imports the /app folder as a package and runs the Flask application

  /app/

  /__init__.py

  - Makes the /app folder a package for other Python files to import.

/routes.py

- Houses all the routes (web pages) for the Flask application. This is where the GET and POST requests are sent from the server and handled in Python.
- Each route is decorated with an "@app.route"

/api.py

- Handles all the GET and POST requests that are made to our database. These include both internal API calls and external calls for other teams to use. There is extensive user API documentation for each function available at http://warcraft2.tech/api
- This is written as a RESTful API built on top of Flask.

/auth.py

- A class that wraps Pyrebase authentication for application specific authentication calls.

/database.py

- Firebase database API wrapper for application specific database calls.

/mapUtils.py

- Functions related to parsing and extracting data from .map files

/static/

- A folder that contains all the static resources (images, stylesheets, scripts, etc.)

/templates/

- Contains all the HTML files for Flask to display. Uses Jinja 2 syntax to render dynamic content.

/config/

- Contains all the configuration files for the Firebase API.
- This is the same for both /flaskapp/ and /notifications/

  /serviceAccountKey.json
  - The service account key from Firebase. Required to enable the API

/notifications/

/emailNotifications.py

- Python script to check the database for unsent notifications and uses the SMTP protocol to send out forum and direct message notifications.

Docker/Docker-Compose

- We have 3 Docker services configured, each with its own unique Dockerfile:

  /web/notifications/Dockerfile
  - A Python container dedicated just to running our 24/7 script for sending regular email notifications

  /web/Dockerfile
  - A Ubuntu container that contains all application logic for the Flask app. Runs the Gunicorn WSGI application server.

/nginx/

/nginx.conf

- The custom configuration file for our Nginx server. Attaches the virtual Docker network created by our *web* service to the Nginx web server.

/Dockerfile

- An Nginx container that copies our custom configuration file into the proper Nginx directory. The configuration file is designed specifically for our Digital Ocean server.

/docker-compose.yml

- The 3 Docker services are orchestrated together with docker-compose, built in the above order

Deployment/Hosting

- Our current domain warcraft2.tech was purchased from domain.com and is set to expire on 01/20/2021
- We are currently using a Digital Ocean droplet for our web hosting. The specs are: 1GB Memory, 1vCPU, 25gb SSD
- Move the Ecs160Web files to the root directory of the web server (we use a Digital Ocean droplet)
- Build and run using the docker-compose file
  - `$docker-compose build`
  - `$docker-compose run -d`

/requirements.txt

- This is the same for both /flaskapp/ and /notifications/
- Contains a list of every Python library that this application depends on.

Apple (MacOS/iOS porting)

- **MAC**:
  GameScene.swift:
  This is the file that contains the code for loading in the different sets of tiles and which calls the functions necessary to display the map and play the music.
  It also handles mouse clicks and user I/O
  It also handles panning of the map view

  Mainly done in:
  //loads tiles and displays map
  didMove(to view: SKView) {
  Loadscterrain tiles
  Loads asset tiles
  Plays music
  }

- **iOS**
  LaunchViewController.swift
  This file loads the splash screen, initial sound, and a label for the user's instruction. The file works only on launchScreen.storyboard.

  LoadMapViewController.swift
  In this class, we load the map and the mini map. We also load the peasant. The code handle touches by the user are handled here as well, in the touchesBegan and touchesMoved functions. Additionally playing music is handled here when the map is loaded and when there are touches. All of this is done in the viewDidLoad method

- **Both Mac and IOS:**

  Displaying of Map:

  The display of map is done in several steps. The first one is the loading of terrain tiles and displaying them.

- **TerrainMap.swift:**
  The main function of this file is to get and store the data required to display the terrainmap properly. It also updates that data as required to display the proper terrainmap

  //reads int .map data and stores it
  LoadMap() {
  This function read a .map file with filename and it will get all lines without comment and removes "\n"

Get terrainmap string matrix and convert terrainmap string matrix to ETerrainTIletype
Also reads in and stores Dpartials data
Get partial strings and convert partial strings to partial UInt8
}

//Calculates tile type and index to be displayed at location (x,y)
CalculateTileTypeAndIndex(x:Int , y:Int, type:InOut ETileType, index: InOut Int) {
Uses data from DTerrainmap and DPartials data stored in Terrain map class to decide the proper Index and ETileType of the Tile to display at location (x,y). Allows the proper tile image to be displayed
}

//Stores DMap data which contains the proper matrix size(WxH) for the map to be Displayed
RenderTerrain() {
It iterates through the DMap matrix and DMapIndices matrix to store the appropriate ETileType in DMap and indexes in DMapIndices
It also uses CalculateTileTypeAndIndex() to help calculate the ETileType and Index in (x,y) location that are not the edges of the map
}

- **MapRenderer.swift:**
This file is responsible for mainly reading "MapRendering.dat" file and loading in the data and storing it for later purposes of retrieving the right images at the right location

//initializes MapRenderer with the necessary DTileSet and Pixel Index information
init(config: [String], tileset: CGraphicTileSet, map: CTerrainMap) {
Reads in "MapRendering.dat" data and stores it into PixelIndices
Used data from DTileSet to store in the correct index of each terrain image data in DTileIndices
For later retrieval in DrawMap
}

DrawMap(surface: CGraphicSurface, typesurface: CGraphicSurface, rect: inout SRectangle)){
Convert each tile index into a tile and store it in the map node
}

- **GraphicTileSet.swift**:

LoadTileSet() {
Reads in the .dat file and uses the PNG path to create a mapping between Tilenames and its index.

And to also load the SKTexture indexes
}

- **GraphicFactory.swift:**
  Reads in .dat file and gets the necessary data of the corresponding png file. Uses this data to slice up the PNG file appropriately.

  LoadTiles() {
  Uses tilenames from .dat files and the corresponding images to create a mapping between tilenames and tile images
  }

  makeTile() {
  Slices up the image and returns the NSImage with the appropriate height and width
  }

Windows

- Our Classes are mostly the same as Linux team with the same functions
- **CApplicationData**
    - Class CApplicationData contains all the information we need to run the game. CApplicationData() initializes all the information we need to run the game.
    - The Activate() function invokes any events that require a different graphic. This includes loading different colors, fonts or graphics for attributes. Certain attributes can change states so some events will invoke a different graphic and this function activates these certain graphics.
    - The Instance() function invokes the constructor of CApplicationData, creates a new pointer of CApplicationData, and returns that pointer.
    - RenderMenuTitle() sets up the menu display window with the given parameters, it sets the fonts, text color, shadows etc.
    - RenderSplashStep() sets up the splash screen
- **CApplicationMode**
    - Class CApplicationMode represents the "state" of our game. Most "Mode" classes only have functions, not data. All the necessary data is provided by context.
- **CAssetDecoratedMap**
    - Class CAssetDecoratedMap contains all information needed to render the map.
    - LoadMap() loads maps from the directory "map." Note that this function will only be called once during the whole game.
    - SAssetInitialization and SResourceInitialization lists are used to store information about each player's assets and resources, when the game begins, respectively.
- **CBattleMenuMode**
    - CBattleMenuMode() prepares all the information we need to exhibit during the battle.
- **CBattleMode**
    - InitializeChange() loads the game map that was selected in CApplication as well as the song to be played. This is done through the shared pointer 'context'.
    - Input() handles the key input and the behavior behind it. This includes basic arrow key input to signal movement. Implements CApplicationMode.
    - Instance() creates a new pointer of the class CBattleMode if one does not exist and returns it. If one does exist then it returns the existing one.
    - Render() is called to create the battlemode window and drawing area. Information about the state of the game is stored in context and Render() uses the context to render the main battlemode window. Implements CApplicationMode.
- **CButtonMenuMode**
    - ReturnGameButtonCallback() restores the game state to what it was and sends back to the game
    - ReturnMainMenuButtonCallback uses ChangeApplicationMode() to send back to main menu

- **CEditOptionsMode**
    - String DTitle contains title of menu. List DButtonTexts contains the texts over the buttons. List DButtonFunctions contain the corresponding callback function of button. List DButtonLocations records each button's location. Bool DButtonHovered checks if the mouse is over buttons.
    - Int DEditSelected indicates which locations we are gonna edit, -1 means no edit operation. Int DEditSelectedCharacter returns the selected edit text, which character is selected by users. List DEditLocations() contains the locations and sizes of input boxes.
    - List DEditTitles contains the titles of each input box. List DEditText contains the texts in the input boxes. List DEditValidationFunctions contains the validation functions.
- **CFileDataSource**
    - Class CCFileDataSource contains the absolute path to the src file and its corresponding file descriptor. Read() reads @length bytes into @data. It returns how many bytes are read.
- **CGameModel**
    - CGameModel() is a copy constructor. We need this to save the old status of the game model and load it in the future.
    - DActualMap is the map we want to store when storing the game.
    - DGameCycle stores all players' information.
- **CCGraphicSurface**
    - Class CCGraphicSurface contain all the information we need to exhibit a GUI
- **CGUIFactory**
    - Class CGUIFactory contains functions needed to create GUI
- **CGUIApplicationGTK3**
    - NewWindow() creates a new main window.
- **CGUIApplicationGTK3**
    - NewWindow() creates a new main window.
- **CMainMenuMode**
    - CMainMenuMode() prepares all the information we need to exhibit the main menu.
    - MultiPlayerGameButtonCallback() invokes function CApplicationData::ChangeApplicationMode.
    - SinglePlayerGameButtonCallback() invokes function CApplicationData::ChangeApplicationMode.
- **CMapRenderer**
    - Int array DPixelIndices stores infos about which color should be used on each tile type.
- **CMapSelectionMode**
    - Class CMapSelectionMode contains all the information we need to exhibit map-selection interface. CMapSelectionMode() prepares part of the information we need to exhibit map selection maps.
    - SelectMapButtonCallback() will be invoked when players press the "Select Map" button on the map-selection interface.
- **CMapServerSelectionMode**

    - Class CMapServerSelectionMode contains all the information we need to exhibit server map-selection interface.
    - CMapSelectionMode() prepares part of the information we need to exhibit map selection maps.
    - SelectMapButtonCallback() will be invoked when players press the "Select Map" button on the map-selection interface.
- **CPixelPosition**
    - SetFromTile() transforms a tile position into its corresponding pixel position at its center.
- **CPlayerAIColorSelectMode**
    - PlayGameButtonCallback() will be invoked when players press "Play Game" button in the PlayerAiColorSelect interface.
- **CPlayerAsset**
    - Class CPlayerAsset contains all the extrinsic properties of a kind of asset.
- **CPlayerAssetType**
    - Class CPlayerAssetType contains all the intrinsic properties of a kind of asset.
    - AddCapability() is used to add certain kinds of capabilities of an asset type. It could be invoked when an asset type is upgraded and gains new capabilities.
    - RemoveCapability() is used to remove certain kinds of capabilities of an asset type.
    - AssetRequirements() returns all the asset requirements needed before building this kind of asset.
    - HasCapability() is used to determine whether an asset type has a certain capability.
- **CPlayerCapability**
    - Register() checks if a certain capability has been registered. If not, register that capability and return true; If yes, return false. It takes in an argument ‘capability’ which is the capability that needed to be registered.
    - String DName contains the capability name. DAssetCapabilityType is the corresponding type of that capability.
- **CPlayerData**
    - Class CPlayerData contains all the information about the player.
    - CPlayerData() is a copy constructor we need to save the current status of the game.
    - CreateAsset() increases asset ID by 1, whenever a player creates an asset.
    - CreateMarker() creates a target position on the map where assets can move to that position.
    - AddUpgrade() adds upgrades to the assets
- **CPosition**
    - DistanceSquared() is used to calculate the squared distance between two positions.
    - Distance() uses Newton's method to calculate an approximate integer square root.
- **CTerrainMap**
    - Enum ETerrainTileType() contains all information about the type of terrain tiles.
    - Given an index of a tile, return its tile type.
    - DTerrainMap is an ETerrainTileType two dimensional array containing all the tiles of the map.
    - DMap is an ETileType two dimensional array containing all the tiles of the map.

    - DMapIndices is a two dimensional array containing the indices of the map.
    - String DMapName contains the name of the map.
- **CTilePosition**
    - Imagine decomposing the window into hundreds of blocks, each block is one of "tile", CTilePosition contains all the information about that tile position.
    - SetFromPixel() transforms a pixel position to its corresponding tile position.
- **SGUIKeyType**
    - Class SGUIKeyType contains all possible input from the keyboard.

Multiplayer

The Warcraft 2 Multiplayer mode allows users to play against and with each other using both LAN and over a server connect. This document describes the code that the Multiplayer Team has developed to describe the protocol and functions used to communicate between game clients.

Classes:

- TCPConnect
- RemoteServer
- WebServerRequest

TCPConnect

This class is the base of the game and provides functions for clients to connect, read and write through sockets.

Data Variables:

- int SocketFileDescriptor
  - Contains the socket's file descriptor which allows the socket to read and write to the other side.
- RemoteServer DRemoteServer
  - Contains an instance of a remote server that will allow the game to communicate over direct connect.
- std::thread DServerThread
  - Contains a thread which runs the remote server so that the game can run concurrently with the remote server.

Methods:

- void TCPConnectClient(std::string remotehost, int portnumber)
  - std::string remotehost: the hostname (typically IP address)
  - int portnumber: the port number being used by both machines
    - This function starts a socket connection between the current computer and the server side. It saves the socket file descriptor into the data variable SocketFileDescriptor.
- void TCPConnectServer(int portnumber)
  - int portnumber: the port number to be used in connection
    - This function creates a remote server that can be used to communicate between machines over direct connect
- std::string TCPRead()

        - This function reads over the socket connection and then returns the string that it reads from the other side. It first parses the package prepend for the length of the string and then returns the proper sized substring of the packet.
- void TCPWrite(std::string serialized)
    - std::string serialized: this string is the serialized packet data which the client is trying to send
        - This function writes the serialized data over the socket connection to the server. It first prepends a 4 bytes to the packet where the first 3 bytes are the length of the rest of the packet and the 4th byte is the '&' symbol.
- std::string SerializeCommand(SPlayerCommandRequest command, int playernumber)
    - SPlayerCommandRequest command: this object is of the type which is used internally in the game to represent a player's command to a player entity.
    - int playernumber: the number of the player in game
        - This function serializes the arguments into one singular string with set format and returns it. The format is as follows:
          `PlayerNumber|DPacketType|DAction|CPlayerAsset1,CPlayerAsset2...|DTargetNumber|DTargetType|DTargetLocationX,DTargetLocationY`
- SPlayerCommandRequest DeserializeCommand(std::string Serialized, const std::shared_ptr<CPlayerData>& playerdata)
    - std::string Serialized: this is the serialized command
    - playerdata: this object is a reference to the in game state for a single player
        - The string is deserialized using a reference to the current player's data. This command request is then returned as an object from the type of the internal game's type.

RemoteServer

This class is used during server connect and direct connect. It provides a middleman connection so that all the clients are able to send to the server first, then receive one singular packet back with a response from all the other clients.

Data Variables:

- int n
    - The number of connections the server should expect. This number should be changed based of off the type of map / number of people connected and should be decided before the server starts.
- int SocketFileDescriptors[6]
    - An array containing socket file descriptors to each of the clients.

Methods:

- void ServerSetup(int portnumber)
    - int portnumber: the port number which the server should use to listen to sockets
        - This function sets up the server by opening all of the sockets and then connecting to each of the clients. Then it stores their socket file descriptors into the array noted above.
- std::string TCPRead(int fd)
    - int fd: the file descriptor which the server should use to read from
        - This function simply reads from the given file descriptor in the arguments. Like the read function from TCPConnect, it also accounts for the package prepend and will return the correct substring.
- void TCPWrite(std::string tosend, int fd)
    - std::string tosend: the package which the server wants to send
    - int fd: the file descriptor which the server is trying to send to
        - This function writes the string in tosend to the file descriptor fd. Like the write function from TCPConnect, it also adds the package prepend so that strings will be sent properly.
- void RunServer(int portnumber)
    - int portnumber: the port number which the server should listen to sockets
        - This function runs the entire server starting from ServerSetup. It reads from each of the clients and then puts each packet together into one large packet which contains each of the clients' packages separated by the character '%'

WebServerRequest

This class is used to call HTTP requests to the Web Team's online API. This class also uses sockets for the purpose of calling HTTP requests, but instead the formatting of the strings is much more strict.

<u>Data Variables:</u>

- int DSocketFileDescriptor
    - This file descriptor is the socket connection to the web server. Reading / writing to this file descriptor is the same as reading / writing to the web server.
- std::string DUserID
    - This ID gets retrieved from the web server and is used in several of their API calls.

Methods:

- void WebServerConnect()
    - This function opens the socket connection to the Web team's server. It connects to warcraft2.tech over the port number 80 because it uses HTTP.
- int ServerAuthenticate(std::string username, std::string password)
    - std::string username: the player's username on the web server's database (email)
    - std::string password: the player's password on the web server's database
        - This function calls the HTTP POST request on warcraft2.tech/api/login. It allows the user to authenticate with the server using basic authentication where the information is sent using base64 encoding. The server authentication will save the DUserID field upon completion. Outside of getting player stats, this function is pretty useless because all of the HTTP requests require a basic authentication.
    - int return: the return value of this function is an error code. If it returns a non-zero value, there is an error.
- int GetPlayerStats()
    - This function calls the HTTP GET request on warcraft2.tech/api/player-stats/<userID>. This function makes use of the DUserID field which is saved after calling ServerAuthenticate(). It gets the JSON format that is returned from the web server and then prints out the JSON format return.
    - int return: the return value of this function is an error code. If it returns a non-zero value, there is an error.
- int PostGameResults(std::string username, std::string password)
    - std::string username: the player's username on the web server's database (email)
    - std::string password: the player's password on the web server's database
        - This function calls the HTTP POST request on warcraft2.tech/api/game-stats. It takes in a json which uses the DUserID field that gets saved from ServerAuthenticate and sends it back to the server. It then prints out the resulting json from the server.
    - int return: the return value of this function is an error code. If it returns a non-zero value, there is an error.

Tools

About WCMapEditor

WCMapEditor is a QT created Warcraft 2 level editor that helps you create custom maps for your game. Its primary purpose is to create or edit maps for use in Warcraft 2. Its goal is to provide core features for map editing as well as strong integration services for easy access to the Warcraft 2 base game.

WCMapEditor works mainly with 2D rectangular tiles and contains a set grid of the underlying tiles for precise editing.

As built, it contains a auto formatting for the correct type of terrain imported as well as

Terminology

- Widget
    - Defines the UI objects that provide window features for connection to the backend
- Dialog
    - Defines the pop up windows that occur when instantiating a new map, importing a tileset, etc
- Slot
    - Defines the button or function triggers when interacting with buttons within the main window UI and calls its related handler
- View
    - Defines the widget to images are rendered upon
- Cursor
    - Defines the actions and properties of the user's cursor within the application for interaction with its features
- Pixmap
    - Defines the image type of all images imported and created within WCMapEditor

Class Hierarchy

- WCMapEditor acts as the application control center
- Mainwindow acts as the notifier to all backend classes, providing notifications to other classes in order to call functions
- Cache acts as the middleware for all importing, exporting, saving, etc functions pertaining to files
- Map acts as the primary backend and container for map data. Functions within the class relate to the manipulation of its data and all corresponding subgroups within the backend
- MapView acts as the frontend including all renderings of the map and tiles within its grid

Classes

- **Main**
    - Main instantiates the application window and application
    - Main should not contain any additional functions or code as it serves as it should only execute the program

- **Mainwindow**
    - Mainwindow provides the UI interface for user interaction including all dialogs and widgets presented
    - Mainwindow contains all functionalities regarding the application UI and handling of input
    - Functions
        - getCharFromTileId
        - set_AssetTab
            - Set the current index of the asset tabs
        - set_AssetList
        - set_PlayerList
        - set_TerrainList
        - get_terrainModel
        - get_assetModel
        - get_AssetTab
        - set_TerrainTiles
        - addAssetPack
            - Add asset by directory
        - addAsset
            - Add asset by file
        - addSongPack
        - addSoundPack
        - on_actionNew_triggered
            - Handle new map instantiation
        - on_actionSave_triggered
            - Handle project save
        - on_actionImport_triggered
            - Handle map import
        - on_actionBucket_toggled
            - Toggle bucket fill tool
        - on_actionQuit_triggered
            - Handle application quit
        - showPlayersContextMenu
            - Show context menu of player's list view (right click)
        - showAssetsContextMenu
            - Show context menu of assets's list view (right click)
        - showTerrainsContextMenu
            - Show context menu of terrain's list view (right click)
        - on_actionAdd_Tileset_triggered
            - Handle add tileset
        - on_actionRemove_Tileset_triggered
            - Handle remove tileset

        - on_tabWidgetMap_currentChanged
            - Handle map tab change
        - on_actionProperty_triggered
            - Handle map property dialog
        - terrainClicked
            - Handle click on terrain
        - tilesetsClicked
            - Handle click on tileset
        - assetsClicked
            - Handle click on asset
        - pAssetsClicked
            - Handle click on player assets
        - on_actionResourceOpen_triggered
            - Handle resource dialog
        - on_actionDownload_triggered
            - Handle download from web server dialog
        - on_actionUpload_triggered
            - Handle upload to web server dialog
        - on_actionAdd_Assets_triggered
            - Adds all the assets based on a directory
        - on_actionRemove_Assets_triggered
        - on_importedTilesetTab_tabCloseRequested
            - Remove tileset tab on close request
        - onTileClicked
            - Set cursor to selected tile
        - on_tabWidgetAssets_currentChanged
            - Change the tab to either the assets to place or the assets per player
            - * index 0 is the asset tab, 1 is the players tab
        - on_playerNumber_valueChanged
            - Specifies the ownership of an asset
- **WCMapEditor**
    - WCMapEditor should
    - WCMapEditor contains all information regarding the application and serves as the connection between all classes
    - Functions
        - name
        - setName
        - tilesetPaths
            - Get tileset paths
        - findMap
            - Find map by mapId
        - addMap
            - Add map to wcMaps

        - removeMap
            - Remove map from wcMaps by id
        - Maps
            - Get maps
        - mapSize
            - Get wcMap size
        - findTileset
            - Find tileset by tilesetId
        - addTileset
            - Add tileset path to wcTilesetPaths
        - removeTileset
            - Remove tileset path from wcTilesetPaths by index
        - tilesetCount
            - Get number of tilesets
        - findTerrain
            - Find terrain by terrainId
        - addTerrain
            - Add terrain path to wcTerrainPaths
        - terrainCount
            - Get number of terrains
        - addAssetPack
            - Add asset by directory
        - addAsset
        - addSong
        - addSound
        - setBrushTool
        - getPack
            - Returns pointer to current resource pack
- **Cache**
    - Cache should be used for saving or loading from external files, including map files, asset dat files and settings files.
    - Caches contains functions for loading/saving maps, loading/saving a settings file and exporting a data director
        - saveMap
            - Saves the passed map object as a map file
        - importMap
            - Loads map file based on path to the file passed in by argument. If no path is provided, function will generate a file explorer window to retrieve file path. Returns a map object if loaded successfully, or nullptr if not loaded successfully
        - importResourcePack
            - Loads resources into the main window from a file referenced by the path passed to function. This file is a “settings” file that holds the names and

paths to all assets, songs, and tilesets that were used the last time the program was executed.

- saveResourcePack
  - Saves a “settings” file in launch directory based on all loaded assets, songs, and tilesets
- saveDataDirectory
  - Creates a new Data directory with img, res, and snd subdirectories. Each subdir is populated with the relevant files that correspond to the program’s loaded resources, such as dat and png files for img dir.

- **MapPropertyDialog**
  - MapSettingDialog provides a dialog for editing the current map properties such as its name, number of players, and player starting resources.
  - Functions
    - editMapProperty
      - When a field that corresponds to a map property (name or max number of players) is edited, a signal is sent to MapPropertyDialog, and this function is called as a result. It updates the corresponding map property.
    - editPlayerProperty
      - When a field that corresponds to a map player property (starting resources) is edited, a signal is sent to MapPropertyDialog, and this function is called as a result. It updates the corresponding map player property.
    - changeCurPlayer
      - When the spinbox that designates the current player number is changes, this function is called. This function resets the spin box values for the player starting resources to match what is stored in the corresponding player object.
- **NewMapDialog**
  - NewMapDialog provides a dialog for instantiating a new map project and initializing its default values for the constructor
  - Functions
    - addMap()
      - This function takes the values from the ui fields, such as map name, map dimensions, player count and tile dimensions, to create a new map object, then adds this object to wcMapEditor.
- **NewTilesetDialog**
  - NewTilesetDialog provides a dialog for instantiating a new tileset and initializes its default values for the constructor
  - Functions
    - addTileset
      - This function takes values from the ui fields, such as name and path, and creates a new tileset that is added to wcMapEditor
    - Browse

                - This function is used to verify that files selected from the file browser are valid formats for creating a tileset.
- **MapView**
    - MapView provides all map rendering and interaction functionality. This involves all graphic manipulation within the map editor window
    - Functions
        - makeMapImage
            - This function creates an displays a pixmap of the current map based on the resource packs terrain tile set.
        - setCurrentMap
            - This map sets the current map
        - initAssets
            - This function is called to draw all assets in the current map
        - drawAsset
            - This function draws the specified asset to the scene if a matching asset exists in the resource pack
        - setAssetColor
            - This function returns an asset pixmap that has changed color based on the current player.
        - removeAsset
            - This function removes an asset from the player that owns it as well as from the scene based on the point specified
        - mousePressEvent
            - This function edits the map based off of the current cursor type and signals to the map to update itself as well
        - mouseMoveEvent
            - This function edits the map based off of the current cursor type and signals to the map to update itself as well
- **Players**
    - Players provide an organized hierarchy for all player related features and functions
    - Functions
        - assets()
            - Get assets for player
        - gold()
            - Get gold for player
        - setGold()
        - lumber()
            - Return lumber value for player
        - setLumber()
        - color()
            - Return enum Color Type for player
        - setColor()
        - qColor()

        - Return qColor for player
    - setQColor()
    - setRGB()
        - Sets RGB alpha value
    - colorIndex()
        - Gets pColorIndex value for player
    - setColorIndex()
    - curColors()
        - Gets all current colors for each player
    - removeAsset()
        - Removes selected asset and display of the map
- **Asset**
    - Assets provide a generic class containing all asset types from Warcraft 2 buildings to units
    - Functions
        - type()
            - Returns enum for Asset type
        - setType()
            - Sets the enum Asset type
        - x()
            - Returns the x position of the asset
        - setX()
        - y()
            - Returns the y position of the asset
        - setY()
        - tiles()
            - Get the frames of each asset
        - setTiles()
        - name()
            - Gets the name of the asset
        - setName()
        - path()
            - Gets the path to the asset
        - setPath()
        - display()
            - Gets the object to display the asset on the map
        - setDisplay()
        - animations()
            - Gets the animations for the asset
        - setAnimations()
        - properties()
            - Gets the mapping of each asset and properties
        - properties(QString key)

            - Gets the value of the key from the Map properties
        - setProperties()
            - Sets the properties of each asset with the mapping
        - loadAnimations()
            - Sets the animation vector of an asset based off frame data held in the passed data file
        - loadProperties()
            - Sets properties of asset based off property values held in passed data file
        - getNumberOfFrames()
            - Return the total number of frames of all combined animations
        - propertyKeys()
            - Gets the keys from the map for the properties (aProperties)
        - getAllProperties()
            - Gets values of all the property keys
- **Cursor**
    - Cursor provides input functionalities from the user's mouse upon the application and focuses on providing map editing capabilities
    - Values
        - currentTile
            - Holds the current terrain tile type for single and special tool
        - curAsset
            - Holds the current asset for asset tool
        - currentTool
            - Specifies which tool is being used for map editing
        - specialIndex
            - Specifies bit offset of current loaded tile for special tool
- **Map**
    - Map serves as a container for all map objects functionalities for the backend of map manipulation
    - Contains a grid and a terrain grid
        - The grid contains the actual tile whereas the terrain grid contains the terrain tiles necessary for the blending of tiles
        - Each tile has 4 corners of terrain tiles that determine the correct version of the tile
    - Functions
        - clearMap
            - Resets the map grid and the terrain grid
        - bucketTool
            - Set enclosed area with current tile selected
        - mousePressEvent
            - Draw tile, place asset, remove asset based on current cursor tool on mouse press
        - mouseMoveEvent
            - Draw tile on mouse movement

        - getTileFromCorners
            - Returns the tile image index from the specified terrain types of the corners of the tile
        - getCornersFromTile
            - Returns the terrain types of the corners for a specified tile
        - loadAsset
            - Interprets a String into an asset, then assigns created asset to a player
- **Tileset**
    - Tileset provides the storage of all tileset properties and serves as a container for all tileset manipulations
    - Functions
        - loadFromImage()
            - An image is passed to a QImage and the image is segmented into individual tiles for storage in a QVector
        - loadFromFile()
            - The file path of the image is passed to a QImage and calls loadFromImage() to segment into individual tiles
- **Tile**
    - Tile provides the storage of tile properties and serves as a container for all tile manipulations
    - Functions
        - setImage()
            - Sets the tile image as a pointer
- **TerrainTileset**
    - TerrainTileset provides the storage of all terrain tileset properties and serves as a container for all terrain tileset manipulations
    - Functions
        - TerrainTileset
            - Initializes terrainTiles
        - loadTerrainFile
            - Load terrain tileset by reading from .dat file
        - getTileAtIndex
            - Returns the index at which the proper tile is stored in terrainTiles tileset determined by the parameters
        - getTile
            - Get terrain tile by index
        - stringToType
            - Convert terrain string to terrain type
        - getPath
            - Get terrain tileset path
- **TerrainTile**
    - TerrainTile is a subclass of the Tile class and holds the properties of the terrain type selected

- **Sound**
    - Sound is a class that handles the importing of songs (mp3) and sounds (wav) as well as holding references and metadata for the songs and sounds
    - Functions
        - importSong()
            - Called when importing song from dialog
            - Returns pointer to QStringListModel containing song
        - importSound()
            - Called when importing sound from dialog
            - Returns pointer to QStringListModel containing sound
        - soundList()
            - Returns reference to list of sounds
        - soundMap()
            - Returns key-value map where key is sound name and value is sound path
        - songList()
            - Returns reference to list of songs
        - songMap()
            - Returns key-value map where key is song name and value is song path
- **ResourceDialog**
    - ResourceDialog is a class that contains the logic and functionalities behind the dialog box that is opened when the user clicks Resources > Open from the toolbar at the top of WCMapEditor
    - Functions
        - open()
            - Opens resource dialog
        - tabSelected()
        - on_actionMusic_triggered()
            - Action listener for opening Music tab
        - on_actionSounds_triggered()
            - Action listener for opening Sounds tab
        - displayMusic()
            - Function that handles displaying all imported music mp3 files from QStringListModel
        - on_actionImport_triggered()
            - Import depending on tab selected
        - on_actionReplace_triggered()
            - Replace selected song or sound with one chosen from a file open dialog
        - on_actionExpport_triggered()
        - on_actionPlay_triggered()
            - Preview selected item by playing sound
        - on_actionPause_triggered()
            - Pause song or sound currently playing
        - on_actionDelete_triggered()

                - Delete selected item from resource dialog
            - importTerrainTileset()
                - Import terrain tileset
            - importAsset()
                - Import asset
            - importSong()
                - Import song file
            - importSound()
                - Import sound file
            - on_asset_clicked()
                - Load current asset clicked
            - on_animation_clicked()
                - Set asset animation
            - on_actionProperties_triggered()
                - Create message box displaying current asset properties
            - on_step_clicked()
                - Step through animation for selected asset frame-by-frame
            - on_speed_changed()
                - Change speed with which animation for selected asset plays
- **WebManager**
    - Webmanager is a class that contains the logic and functionality for handling requests to the warcraft 2 web server. It handles the requests for both the download of .map files and upload of .map files. Each request is filtered through the authentication service handled by the web server and links each user account with their request.
    - Functions
        - downloadFile()
            - Creates a request from the given url and passes data to downloadFinished()
        - cancelDownload()
            - Clear request
        - downloadFinished()
            - Downloads the file into the designated location
        - downloadReadyRead()
            - When all packages are ready to be read, read from file
        - uploadFile()
            - Creates a request from the given url and sets the headers for the sending package. The .map file is then encapsulated and sent to the web server
        - void cancelUpload()
            - Clear request
        - void uploadFinished()
            - Clear request
        - slotAuthenticationRequired()

        - The slot is called whenever an authentication is required for the user and creates a window to prompt the user to sign in
- sslErrors()
    - Print error messages