Praktikum 3: Dungeon Crawler - Grafische Benutzeroberfläche

Lernziele

- Arbeiten mit Qt Widgets
- Anwenden des Observer-Patterns
- Selbstständiges Einarbeiten in ein großes Framework

Aufgabenstellung

In 3. Praktikum werden Sie den DungeonCrawler um eine grafische Benutzeroberfläche erweitern. Die neu zu entwickelnde Klasse `GraphicalUI` soll dabei die alte `TerminalUI` ersetzen, ohne dass die restliche Programmarchitektur weiter verändert werden muss.

Zwei eigene Fensterklassen dienen als Startbildschirm sowie als Darstellung des Spielfeldes.

Für alle UI-Komponenten finden Sie Beispieltexturen im Moodle – Ihnen steht es jedoch auch frei, sich selber künstlerisch zu betätigen. Sämtliche Beispielbilder sind nur als Vorschlag zu verstehen.

1 Die Klasse `GraphicalUI`

Unsere neue GUI-Klasse übernimmt vor allem 3 Aufgaben, die Sie nach und nach implementieren müssen:

1. Erzeugen und Verwalten der eigentlichen Fenster der Benutzeroberfläche: Im Konstruktor der `GraphicalUI` können Sie die gewünschten Fenster-Objekte erzeugen und anzeigen lassen.
2. Laden und speichern der benötigten Texturen: Texturen werden in Qt (u.A.) als `QPixmap` gespeichert. Überlegen Sie sich sinnvolle Speicherstrukturen, um alle Texturen im Konstruktor der `GraphicalUI` zu erzeugen.
   - Die Texturen nicht in der jeweiligen Fensterklasse zu speichern hat den großen Vorteil, dass sie nicht mehrfach geladen werden müssen, wenn sie in verschiedenen Fenstern oder Objekten benötigt werden.
3. Ähnlich der veralteten `TerminalUI` muss die `GraphicalUI` von `AbstractUI` und `AbstractController` erben und die Methoden `void draw(Level* level)` sowie `int move()` zur Verfügung stellen (doch dazu später mehr).

2 Der Startbildschirm

Zu Beginn des Programmes soll uns ein Startbildschirm begrüßen, welcher (neben einem Hintergrund) bis jetzt nur einen einzigen Button enthält: Neues Spiel. Erzeugen Sie sich hierfür eine StartScreen-Klasse, welche vom `QDialog` erbt (Qt bietet hierfür Voreinstellungen).

Damit Sie die Klasse jedoch im GUI-Konstruktor instanziieren und anzeigen können, müssen folgende Punkte erledigt werden:

1. Passen Sie Ihre Projekteinstellungen in der .pro-Datei von Qt an. Sie benötigen folgende Einträge (zur Not vergleichen Sie mit einem neu erstellen Qt Widget Projekt, plus `testlib`):

   ```
   QT += core gui
   QT += testlib
      greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
   CONFIG += c++11
   Source: += ...
   ```

   Und, falls es nicht automatisch hinzugefügt wurde, für die .ui-Dateien:

```
FORMS +=
   nameIhrerScreenKlasse.ui
```

2. Erzeugen Sie zu Beginn Ihrer `main.cpp` eine `QApplication` und starten Sie diese am Ende.
3. Springen Sie in Ihre DungeonCrawler-Klasse und ersetzen Sie dort die `TerminalUI` durch Ihre neue `GraphicalUI` (kommentieren Sie fürs Erste die play-Methode aus, damit das Programm compiliert).

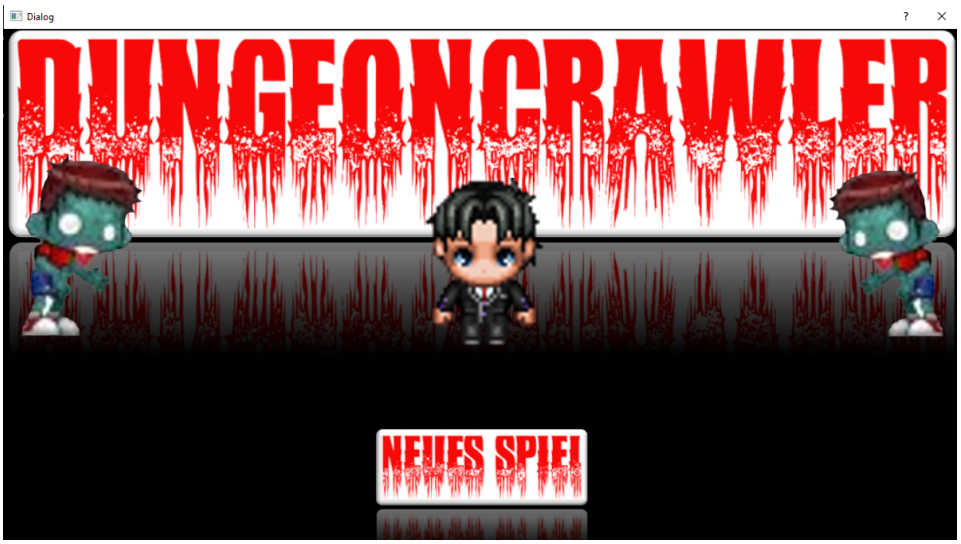
Erzeugen Sie in Ihrer StartScreen-UI ein Label zum Anzeigen des Hintergrundbildes. Es steht Ihnen dabei frei, den Qt-Designer zu verwenden oder alles im Quellcode des Fensters zu erstellen.

Hilfreiche Klassen / Methoden:
- `QWidget::setStyleSheet("background-color: black;");`
- `ui->myLabelName` [um auf Widgets des UIs zugreifen zu können]
- `Label::setPixmap`
- `QLabel:: setScaledContents(true)`

Anschließend platzieren Sie einen Button im Fenster.

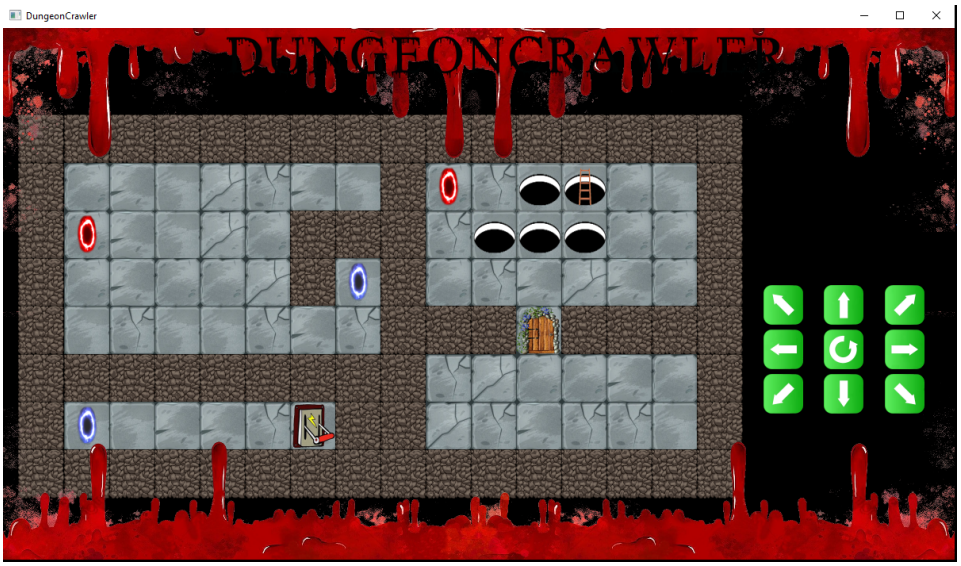
Hilfreiche Klassen / Methoden:
- `QPushButton`
- `QIcon`
- `QPushButton::setIcon`
- `QPushButton::setIconSize`

Erstellen Sie zum Schluss einen slot und verbinden ihn mit dem Button (`QObject::connect()`). In dieser Slot-Methode gilt es, eine `switchWindow()`-Methode in der GraphicalUI-Klasse aufzurufen, welche das Startfenster verschwinden (`QWidget::hide()`) und das MainWindow erscheinen lässt (`QWigdet::show()`). Wenn alles funktioniert, könnte das Ergebnis in etwa so aussehen:



3 Der Hauptbildschirm

Erzeugen Sie als nächstes ein eigenes `MainWindow` (vererbt von `QMainWindow`), wieder mit Hintergrundbild. Die geplante Struktur des Hauptfensters sieht vor, links unser eigentliches Spielfeld darzustellen, und rechts davon Buttons zur Steuerung zu platzieren (natürlich sind Sie frei, davon abzuweichen):



Die Reihenfolge, in denen überlappende Elemente übereinander dargestellt werden, hängt von der Definitionsreihenfolge ab, kann aber durch `QWidget::raise()` oder `QWidget::lower()` verändert werden. Um eine ähnliche Darstellung zu erhalten, bietet es sich an, für die einzelnen Tiles des Spielfeldes sowie für die Buttons jeweils ein `QGridLayout` zu nutzen (Vorschlag: Platzieren Sie das `QGridLayout` innerhalb einer `QGroupBox`, ist es einfacher, die maximalen Ausmaße des Layouts zu bestimmen).

3.1 Erstellung des Spielfeldes

Erzeugen Sie sich für jedes Tile Ihres Levels ein `QLabel`, versehen es mit einer Textur entsprechend der Art der Tile und fügen das `QLabel` dem GridLayout hinzu.

Hilfreiche Klassen / Methoden:
- `QGridLayout::addWidget`
- `QWidget::setMinimumSize`
- `QWidget::setMaximumSize`

Folgende Hinweise zu den einzelnen Tiles:

- Für die Floor-Tile gibt es mehrere Texturen, wählen Sie pro Floor-Tile zufällig eine davon aus.
- Um bei den Portalen unterschiedliche Einfärbungen zu unterstützen, erweitern Sie die Portal-Klasse um ein Attribut `int portalType` und setzen Sie dieses für zusammengehörige Portale auf den gleichen Wert. Wir haben 3 verschiedene Portaltexturen zur Auswahl.
- Die Door-Tiles sollen geöffnet sowie geschlossen dargestellt werden können. Erweitern Sie daher die Klasse Door (falls noch nicht geschehen) um ein Attribut `bool isOpen`, in dem Sie speichern, ob die Tür gerade geöffnet ist oder nicht.

3.2 Die Steuerungs-Buttons

Ziel: Implementieren Sie eine Variable `lastInput`, welche die Bewegungsrichtung (beispielsweise als int entsprechend des Ziffernblocks oder als enum) speichert.

Sie könnten 9 unterschiedliche Buttons erstellen, die jeweils eine eigene `onClick`-Methode befeuern. Schöner wäre jedoch folgende Alternative:

Erstellen Sie sich eine eigene Button-Klasse (vererbt von `QPushButton`), welche sich die geklickte Bewegungsrichtung speichert (z.B. wieder als int). Jetzt gehen Sie wie folgt vor: übernehmen Sie im Konstruktor als Parameter die Bewegungsrichtung des jeweiligen Buttons und verbinden (also connect) Sie das clicked-Signal des jeweiligen Buttons mit einer (selbstgeschriebenen) Slot-Methode. Diese macht nichts weiter, als ein neues Signal zu emittieren, dieses Mal jedoch mit der Klickrichtung als Parameter.

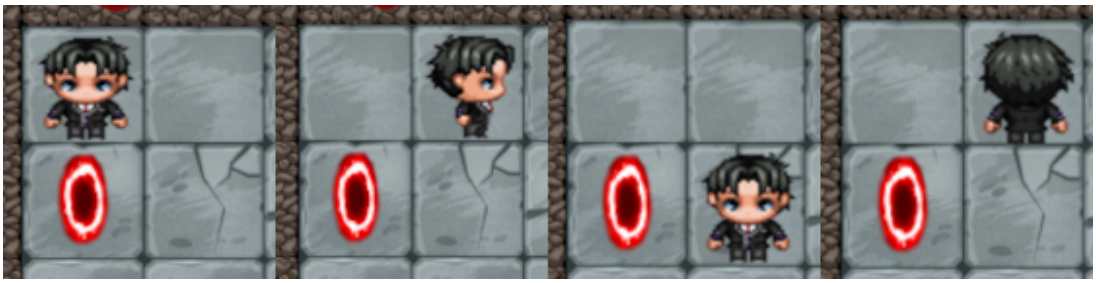
Implementieren Sie eine Slot-Methode `moveSlot` im MainWindow, welche den gleichen Parameter erhält, den Ihr Button-Signal aussendet. Beim Erzeugen der Buttons im Mainwindow verbinden Sie das jeweilige (selbstgeschriebene) Signal des Buttons mit der `moveSlot`-Methode.

Hilfreiche Klassen / Methoden:
- `QPushButton::setStyleSheet("border:none");`
- `QObject::connect`

Fazit: In jedem Fall muss jeder Button letztlich die Variable `lastInput` überschreiben. Sie können nun zudem Tastatureingaben abfangen und auf gleiche Weise die Variable `lastInput` ändern, um neben den Buttons auch eine Steuerung per Numpad zu ermöglichen.

4 Der Character

Der Character kann, ähnlich einer einfachen Tile, als Label mit entsprechender Textur dargestellt werden. Geben Sie für das Label als Parent die Tile an, auf der sich der Character gerade befindet, sollte er direkt an der passenden Position dargestellt werden. Wir wollen jedoch die Textur des Characters an dessen Bewegungsrichtung anpassen (im Moodle finden Sie viele Bewegungstexturen, falls Sie zusätzliche Animationen erzeugen wollen). Erweitern Sie daher die Character-Klasse um ein Attribut `moveDirection`, welche die letzte Bewegungsrichtung speichert, und nutzen Sie diese, um für das Character-Label die richtige Textur zu wählen:



5 Die Main Loop

Aktivieren Sie die play-Methode im Dungeon-Crawler wieder, damit jeden „Frame“ die draw-Methode der GUI und die move-Methode des Characters aufgerufen wird.

5.1 `draw()`

Der Draw-Befehl wird direkt weitergereicht von der GUI an das `MainWindow`, welches sich jeden Frame aktualisieren muss. Hier passiert Folgendes:

- Für den Character wird die Textur neu gesetzt (in Abhängigkeit von seiner gespeicherten Bewegungsrichtung).
- Für alle Türen wird die Textur neu gesetzt (in Abhängigkeit von dem bool `isOpen`).

- Die Pits nehmen eine Sonderrolle ein: Um den Effekt zu erzeugen, dass unser Character wirklich in das Loch gefallen ist, muss bei einem Pit die parent-child-Beziehung geändert werden: Der `Character` muss als Parent das `MainWindow` bekommen und manuell neu platziert werden. Zusätzlich muss über geschicktes `raise()` oder `lower()` die gewünschte Render-Reihenfolge erzeugt werden (Buttons>BackgroundLabel>Pit>Character).

Beispiel:

Hilfreiche Klassen / Methoden:
- `QPoint`
- `QObject::pos()`
- `QLabel::move(QPoint)`

5.2 `move()`

Wir haben in der DungeonCrawler-play-Methode bereits eine Loop erzeugt, benötigen daher die Main Loop von Qt für unser Spiel nicht mehr. Für das Abfangen der erzeugen Events von der Benutzersteuerung ist sie jedoch weiterhin notwendig (ein kleines Übel, welches wir in Kauf nehmen mussten, damit unsere GUI kompatibel zur `TerminalUI` bleibt).

Wir müssen daher in `move()` der GUI die eigene Loop etwas ausbremsen und nur dann die nächste Iteration starten, wenn ein Input gegeben wurde. ”Warten”Sie daher 50 ms mittels `QTest::qWait(50);`, überprüfen Sie einmal auf Events von außen durch `QCoreApplication::processEvents();` und prüfen Sie danach, ob im `MainWindow` ein neuer Input gespeichert wurde (und returnen diesen entsprechend an den DungeonCrawler).