1 Programmaufbau der Übungsaufgabe Rastergraphik

Hinweis: Zur Lösung der Übungsaufgaben ist es nicht notwendig dieses Dokument zu lesen und zu verstehen. Es dient lediglich als Hilfestellung falls Sie ein globales Verständnis über das vorliegende Programm bekommen möchten. Der Inhalt dieses Textes ist in allen Übungsaufgaben denen er beiliegt ähnlich und jeweils nur um kleine Unterschiede im generellen Programmfluss und der Klassenstruktur geändert.

Zunächst wird die Programmstruktur und die Verantwortlichkeit der einzelnen Klassen kurz erläutert. Anschließend wird der Programmfluss beschrieben. Es wird dabei nicht auf konkrete Aufgabenstellungen oder die Funktionsweise von computergraphischen Themen eingegangen.

1.1 Programmstruktur

Im Projekt existieren diese Klassen:

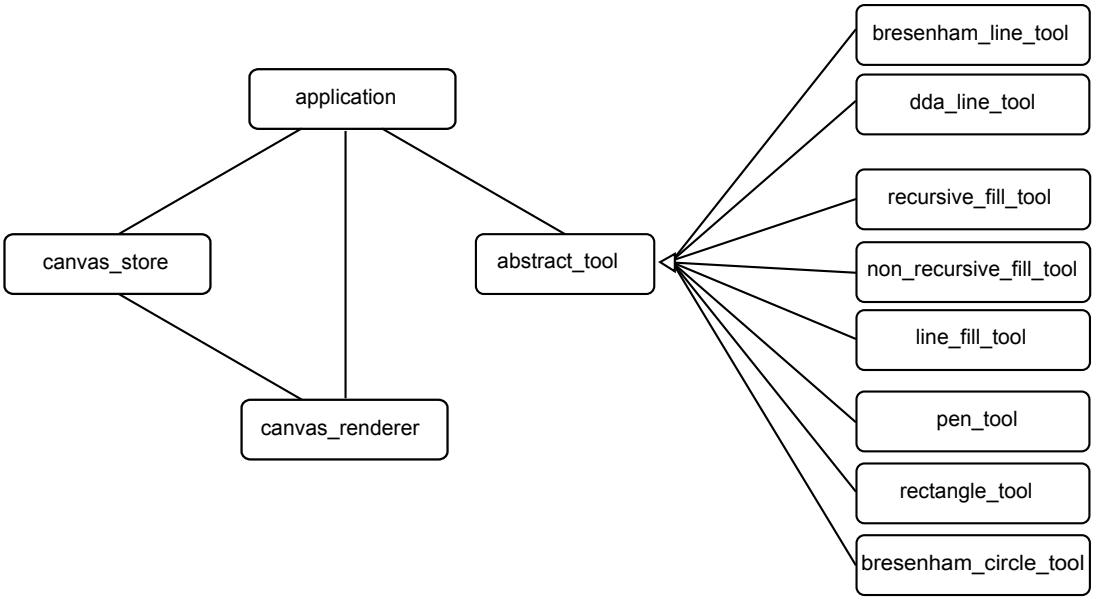

Abbildung 1: Grobe Klassenstruktur des Übungsprogramms

Zu erkennen ist, dass eine Klasse `application` existiert, die wiederum Objekte der Klasse `abtract_tool` verwendet. Von `abstract_tool` sind verschiedene neue Klassen abgeleitet, zum Beispiel `dda_line_tool`, `bresenham_line_tool` oder `recursive_fill_tool`. Außerdem verwendet `application` die Klassen `canvas_store` und `canvas_renderer`, die ihrerseits auch auf `canvas_store` zugreift.

Die Aufgaben dieser Klassen sind wie folgt:

application ist die Hauptklasse des Programms, von der aus der Programmfluss gesteuert und das Hauptfenster aufgebaut wird. Es existiert genau ein Objekt dieser Klasse mit diesen Methoden:

- `run` erstellt das Hauptfenster mittels des *GL-Utility-Toolkits (GLUT)*, bindet spezielle Methoden zur Ereignisbehandlung und startet die Hauptprogrammschleife.
- `set_tool` instanziiert die Variable `tool` mit einer von `abstract_tool` abgeleiteten Klasseninstanz.
- `key_down` enthält die Anweisungen zur Behandlung von Tastendrücken.
- `mouse_button` wird aufgerufen, wenn eine Maustaste gedrückt oder losgelassen wurde.
- `mouse_move` wird aufgerufen, wenn die Maus bewegt wird.
- `mouse_wheel` wird aufgerufen, wenn das Mausrad gedreht wurde.
- `context_menu_select` wird immer dann aufgerufen, wenn der Benutzer ein Element aus dem Kontextmenü aufgerufen hat.
- `display` wird immer aufgerufen, wenn der Inhalt des Fensters neugezeichnet werden soll.

- `setup_context_menu` erstellt mit Hilfe von GLUT das Kontextmenü.
- statische Callbacks: Dienen als Vermittler zwischen den Ereignissen und den entsprechenden Behandlungsmethoden dieser Klasse. Für mehr Informationen, warum Ereignisbehandlungen nicht direkt durchgeführt werden schauen Sie sich bitte den Abschnitt *Ereignisbehandlungen und statische Funktionen* an.

abstract_tool stellt die Oberklasse für alle Zeichenwerkzeuge dar, die im Programm vorkommen. Die Klasse `application` hält einen Zeiger auf diesen Typ, der mit den jeweiligen konkreten Werkzeugen instanziiert wird. Vorgegeben sind diese Methoden:

- `abstract_tool` ist der Konstruktor der Klasse und erwartet als Parameter eine Referenz auf ein `canvas_store`-Objekt, in dem die Pixel gesetzt werden können. Der Wert des Parameters wird intern in der Variable `canvas` abgelegt.
- `draw(x,y)` stellt eine Methode zum Rendern von Primitiven dar, die mittels *eines* Punktes vollständig definiert sind. Hierzu zählt der einfache Zeichenstift und die Füllalgorithmen.
- `draw(x0,y0,x1,y1)` stellt eine Methode zum Rendern von Primitiven dar, die über *zwei* Punkte bestimmt sind. Das sind beispielsweise die Linienwerkzeuge, das Kreiswerkzeug oder das Rechteckswerkzeug.
- `get_shape` gibt ein Element aus der Aufzählung `ToolShape` zurück, das die Form des Werkzeugs definiert. Benötigt wird diese Information zum Rendern der grünen Vorschauen.
- `set_text` bietet die Möglichkeit in einen Datenstrom Text abzulegen, der immerhalb der `display`-Methode der Klasse `application` angezeigt wird.

Die von `abstract_tool` abgeleiteten Klassen implementieren die oben genannten Methoden und enthalten außerdem eigene Helfermethoden, auf die in den Kommentaren innerhalb der entsprechenden Quelldateien eingegangen wird.

canvas_store dient als Speicher für die Pixeldaten. Es handelt sich dabei um ein zweidimensionales Feld aus boolschen Variablen, deren Zustände geändert oder abgefragt werden können. Entsprechend werden folgende Methoden unterstützt:

- `canvas_store` ist der Konstruktor und erhält als Parameter die Anzahl der Elemente der Zeichenfläche in X- und Y-Richtung.
- `set_pixel` Setzt einen Wert an der angegebenen Position auf *true*.
- `get_pixel` liefert den Zustand des Elementes an der angegebenen Position.
- `clear_canvas` setzt alle Elemente auf *false*.
- `get_width` gibt die Anzahl der Elemente in X-Richtung der Zeichenfläche zurück.
- `get_height` gibt die Anzahl der Elemente in Y-Richtung der Zeichenfläche zurück.
- `draw_test_shape` setzt die Elemente für die Testfüllform.

Eine Referenz eines Objektes dieser Klasse wird jedem Werkzeug übergeben. Außerdem erhält der `canvas_renderer` eine Referenz. Im gesamten Projekt existiert eine Instanz des `canvas_store`, der von der `application`-Klasse verwaltet wird.

canvas_renderer dient der graphischen Repräsentation des `canvas_store`. Auch von dieser Klasse existiert genau ein Exemplar, dass von `application` verwaltet wird. Es unterstützt nach außen folgende Methoden:

- `canvas_renderer` ist der Konstruktor, der als Parameter eine Referenz auf einen `canvas_store` erhält, die er intern in der Variable `canvas` ablegt.
- `set_translation` setzt die Verschiebung des Rasterfeldes in Pixeln.
- `get_translation` liefert die aktuelle Verschiebung des Rasters in Pixeln.
- `set_zoom` setzt den Zoom-Faktor des Rasters.
- `get_zoom` liefert den aktuellen Zoom-Faktor.
- `screen_to_grid` wandelt Fensterkoordinaten in Rasterkoordinaten um.
- `snap_screen_coords` verändert Fensterkoordinaten so, dass sie dem nächstgelegenen Zentrum einer Rasterzelle entsprechen.

- `render` rendert zunächst ein regelmäßiges Gitter und anschließend die in `canvas` gesetzen Elemente als schwarze Quadrate.
- `get_cell_size` liefert die Höhe und Breite einer Rasterzelle in Pixeln, abhängig von der Fenstergröße und der aktuellen Zoomstufe.
- `reset_view` setzt Zoom und Verschiebung zurück auf die Startwerte.

Es teilen sich also das aktive Werkzeug und der `canvas_renderer` eine Instanz der Klasse `canvas_store`, über die sie mittels Referenzen zugreifen. Während die Werkzeuge den `canvas_store` verändern ist es Aufgabe des `canvas_renderer` ihn zu visualisieren.

1.2 Programmfluss

Wie in jedem C++-Programm beginnt die Programmausführung in der Methode `main`, welche in der Datei `main.cpp` implementiert ist. Hier wird lediglich ein Objekt der Klasse `application` erzeugt und dessen `run`-Methode aufgerufen.

Die `run`-Methode erfüllt nun folgende Aufgaben:

1. Initialisierung des GLUT-Systems. Dabei handelt es sich um eine Programmbibliothek zur plattform-übergreifenden Erstellung von Fenstern und zum Herstellen der Möglichkeit mit OpenGL zu rendern. Außerdem ermöglicht sie die Behandlung von typischen Fensterereignissen wie *Fenster zeichnen*, *Mausklicks behandeln*, *Mausbewegungen behandeln*, *Tastendrücke behandeln* oder *Timer erstellen und ausführen*. Für mehr Informationen zu den unterstützten Befehlen empfiehlt es sich auf der Internetseite von `freeglut` nachzuschauen. Die Initialisierung erfolgt mittels des Befehls

   ```
   glutInit(&argc, argv);
   ```

   Die Parameter `argc` und `argv` werden der `run`-Methode übergeben und stellen die Kommandozeilenparameter dar (also die Parameter, die z.B. in einer Konsole dem Programmaufruf mitgegeben werden können). `argc` enthält die Anzahl der Programmargumente und `argv` die eigentlichen Zeichenketten.

2. Erstellung des Hauptfensters. Das Programm läuft in einem Fenster, dessen Eigenschaften wie Titel, Größe und verwendeter Rendermodus festgelegt werden müssen. Dies geschieht mit den Befehlen

   ```
   glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
   glutInitWindowSize(620, 640);
   glutCreateWindow("ECG Rastergraphik");
   ```

   Die Anweisung `glutInitDisplayMode` stellt den Rendermodus mittels einer Liste von *Flags* ein. Dabei bedeutet `GLUT_DOUBLE` dass *Double-Buffering* verwendet werden soll. Notwendig ist das um flackerfrei rendern zu können. Das eigentliche Rendern erfolgt dabei nicht direkt auf dem Bildschirm, sondern in einem als *Backbuffer* bezeichneten Speicherbereich. Mittels eines Befehls (der in der `display`-Methode aufgerufen wird) wird dann der aktuell angezeigte Speicherbereich, der *Frontbuffer* mit dem Backbuffer vertauscht. Es wird also das gesamte fertige Resultat angezeigt. Das Flag `GLUT_RGB` bedeutet, dass im RGB-Farbraum gearbeitet wird. Ein Pixel der beschreibbaren Renderpuffer enthält also mindestens einen Rot-, Grün- und Blaukanal.

   Die Methode `glutInitWindowSize` legt die Größe der Zeichenfläche im Fenster fest. Schließlich wird das Fenster mit `glutCreateWindow`, das als Parameter den Titel erhält, erstellt.

3. Festlegung der Ereignisbehandlungsmethoden. Da das Programm nicht weiß, wie ein Fenster intern funktioniert, also wie das Betriebssystem solche erstellt und verwaltet, werden bestimmte Methoden übergeben, die immer dann aufgerufen werden, wenn ein spezielles Ereignis aufgetreten ist. Der Quellcode dafür lautet:

   ```
   glutDisplayFunc(display_callback);
   glutKeyboardFunc(key_down_callback);
   glutMouseFunc(mouse_button_callback);
   glutMotionFunc(mouse_move_callback);
   glutMouseWheelFunc(mouse_wheel_callback);
   ```

   Es werden hier 5 Methoden gebunden. Immer, wenn der Fensterinhalt erstellt werden soll, kommt es zum Aufruf der Methode `display_callback` und immer wenn der Benutzer eine Taste drückt wird die Methode `key_down_callback` aufgerufen. Analog erfolgt der Aufruf für die Methoden zur Behandlung von Mausereignissen. Die Argumente der oben gezeigten Methoden sind keine normalen Variablen, sondern selbst Methoden. Diese müssen eine vorgegebene Parameterliste besitzen um zur Ereignisbehandlung herangezogen werden zu können.

4. Erstellung des Kontextmenüs. Dies erfolgt in der Methode `setup_context_menu` der `application`-Klasse.

5. Einstellen des Startwerkzeugs. Die Zeile

   ```
   context_menu_select(MA_TOOL_PEN);
   ```

   mag zunächst ein wenig ungewöhnlich scheinen, da der Befehl `context_menu_select` ja eigentlich nur aufgerufen wird, wenn etwas aus dem Kontextmenü ausgewählt wird. In dieser Methode erfolgt jedoch die komplette Initialisierung eines konkreten Werkzeuges, weshalb sie an dieser Stelle missbraucht wurde. Der Parameter `MA_TOOL_PEN` ist ein Element der Aufzählung `MenuActions`, die in `application.h` festgelegt sind.

6. Starten der Hauptschleife. Mittels des Befehls

   ```
   glutMainLoop();
   ```

   endet der vom Programm aus steuerbare lineare Programmfluss. Die Methode `glutMainLoop` dauert solange, wie das Fenster existiert. Von nun an werden lediglich die oben festgelegten Ereignisbehandlungen ausgeführt.

Gehen wir nochmal einen Schritt zurück und schauen uns an, wie das Kontextmenü in der Methode `setup_context_menu` erstellt wird. Der relevante Teil lässt sich so verallgemeinern:

```
glutCreateMenu(context_menu_callback);
glutAddMenuEntry("␣Pen␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(p)␣", MA_TOOL_PEN);

// Hier erfolgt der Aufruf von glutAddMenuEntry fuer andere Menueeintraege
...

glutAttachMenu(GLUT_RIGHT_BUTTON);
```

Der erste Befehl erstellt das Kontextmenü. Als Parameter wird eine Methode festgelegt, die immer dann ausgeführt werden soll, wenn ein Menüpunkt vom Benutzer angewählt wurde. Sie trägt den Namen `context_menu_callback`, die ihrerseits `context_menu_select` aufruft. Mit dem Befehl `glutAddMenuEntry` wird dem Kontextmenü ein Menüeintrag hinzugefügt. Der erste Parameter stellt die Zeichenkette dar, die für diesen Menüpunkt angezeigt werden soll und die zweite eine Identifikationsnummer, die `context_menu_callback` beim Aufruf als Parameter übergeben wird - ansonsten könnte diese Methode die verschiedenen Menüpunkte in der Behandlung nicht unterscheiden. Das Element `MA_TOOL_PEN` gehört zu einer Aufzählung (*Enumeration*, `enum`), die in `application.h` festgelegt ist. Für jeden Befehl des Kontextmenüs existiert ein Eintrag in dieser Aufzählung. Mit dem letzten Befehl wird das Kontextmenü an die rechte Maustaste gebunden.

Bisher existiert also ein Fenster, es wurden Methoden zum Zeichnen und zur Behandlung von Tastaturereignissen festgelegt, das Kontextmenü wurde erstellt und der Programmfluss wurde an die Hauptschleife abgegeben. Wenn jetzt der Benutzer also ein Element aus dem Menü auswählt, dann wird die bereits erwähnte Methode `context_menu_select` aufgerufen. Dort wird unterschieden, welcher Menüpunkt genau ausgewählt wurde. Exemplarisch hier der Quellcode für den Menüpunkt `MA_TOOL_PEN`:

```
switch(item)
{
        // Set the pen as tool
        case MA_TOOL_PEN:
                set_tool(new pen_tool(*canvas));
                break;
        ...
}
```

Die Variable `item` ist der Parameter der Methode dessen Inhalt innerhalb der `switch`-Anweisung abgefragt wird. Ist er belegt mit `MA_TOOL_PEN`, so wird die Methode `set_tool` aufgerufen. Für den Parameter dieser Methode wird eine neue Instanz der Klasse `pen_tool` erstellt. Innerhalb von `set_tool` wird der alte Inhalt der Variable `tool`, eine Membervariable der `application`-Klasse, gelöscht und mit dem Parameterwert belegt. In unserem Beispiel eine neue Instanz der Klasse `pen_tool`. Das ist möglich, da die Variable selbst vom Typ `abstract_tool` ist, von der die spezielle Stiftwerkzeug abgeleitet ist. Damit ist die Abarbeitung der Ereignisbehandlung für Kontextmenüauswahlen (bis auf Kleinigkeiten) beendet.

Innerhalb des Fensters muss etwas gezeichnet werden, daher wird für das Programm nicht sichtbar die festgelegte Zeichenmethode aufgerufen, also `display_callback` die ihrerseits `display` aufruft. Hier passieren folgende Dinge:

1. Aktivieren einer orthographischen Projektion, bei der eine OpenGL-Einheit einem Pixel entspricht.

2. Löschen des Inhalts des aktuellen Zeichenpuffers.
3. Rendern des Pixelfeldes
4. Rendern einer Vorschau, falls notwendig
5. Ausgabe des Debug-Textes
6. Vertauschen des Front- und Backbuffers.

Während damit die Visualisierung implementiert ist wurde noch keine Interaktion ermöglicht. Diese erfolgt in den Ereignisbehandlungen für Mausbewegungen oder bei Mausknopfereignissen. Dort wird unter bestimmten Voraussetzungen eine der `draw`-Methoden der `tool`-Variable aufgerufen, die ja mit einer Instanz eines bestimmten Zeichenwerkzeugs belegt ist.

Für die Funktionsweise der Tastaturbehandlung ist die (vorher in `run` festgelegte) Methode `key_down` verantwortlich. Als Parameter erhält sie die gedrückte Taste. Je nach Taste wird die Methode `context_menu_select` mit der entsprechenden ID einer bestimmten Aktion aufgerufen. Es wird also so getan, als wenn ein Menüpunkt aus dem Kontextmenü aufgerufen wurde.

1.3 Ereignisbehandlungen und statische Funktionen

Beim Verstehen des Quellcodes fällt auf, dass Ereignisbehandlungen nicht direkt, sondern über Umwege aufgerufen werden. Beispielsweise erfolgt das Zeichnen des Fensterinhalts in der Methode `display` der `application`-Klasse. Der Befehl `glutDisplayFunc` erhält jedoch nicht `display` als Parameter, sondern `display_callback`. In dieser Methode geschieht folgendes:

```
instance->display();
```

Es wird also lediglich `display` aufgerufen und zwar als Methode einer Variable namens `instance`. Um dem Grund für diesen Umweg nachgehen zu können muss man verstehen wie (z.B. mit `new`) erzeugte Instanzen von Klassen intern funktionieren. Im Prinzip handelt es sich dabei um einen Datenbereich, der alle veränderbaren Eigenschaften des Objektes, also die Membervariablen, enthält. Wenn nun eine Methode der Klasse aufgerufen wird, eben zum Beispiel `display`, dann wird dieser Methode intern als erster Parameter (der den Namen `this` trägt) ein Zeiger auf diesen Speicherbereich übergeben. Wird nun jedoch eine Methode als Ereignisbehandlung gebunden, so ist dem System lediglich die Methode bekannt, aber nicht der `this`-Zeiger. Es kann also nicht auf Membervariablen zugegriffen werden. Nun ist es in C++ möglich Methoden zu definieren, die unabhängig von `this` sind. Diese werden mit dem Kennwort `static` gekennzeichnet. So ist z.B. `display_callback` eine statische Methode. Damit wird das Problem jedoch nicht gelöst, sondern erstmal nur in den Bereich der Klasse `application` verschoben. Die eigentliche Lösung bietet die Variable `instance`. Dabei handelt es sich um eine globale Variable, die im Konstruktor der `application`-Klasse gesetzt wird. Dort findet sich die Zeile

```
instance = this;
```

Die Variable `instance` bezeichnet also ein instanziiertes Exemplar der Klasse `application`. Das hat den Vorteil, dass eben wieder ein Zugriff auf Membervariablen möglich ist, aber den Nachteil, dass die `application`-Klasse nur einmal erstellt werden kann. Würde eine zweite Instanz existieren, von der natürlich auch der Konstruktor aufgerufen wird, so überschreibt dieser die Variable `instance` mit seinem Datenzeiger, was das erste Exemplar von `application` für die Ereignisbehandlung nutzlos machen würde.