Eine kurze Einführung in die `basic_io` Bibliothek

Letzte Aktualisierung: 18.04.2023

1 Einleitung und Motivation

In der Vorlesung „Einführung in die Programmierung“ (EIP) benutzen wir die „basic_io“-Bibliothek für Python, um einfach Graphik ausgeben und mit dem Benutzer interagieren zu können. Es gibt für diesen Zweck sehr viel bessere und ausgereifte Bibliotheken, wie z.B. das Qt-Framework; diese Lösungen haben aber den Nachteil, dass sie sehr komplexe zu lernen und zu benutzen sind. Insbesondere erfordern die meisten Graphik & GUI (graphical user interface = graphische Benutzerschnittstellte) Bibliotheken eine „ereignisorientierte“ Programmarchitektur. Dies bedeutet, dass der Kontrollfluss des Programms in kleine Stücke aufgeteilt werden muss, die zu bestimmten Ereignissen ausgelöst / aufgerufen werden (z.B. wenn die Maus sich ein Stück bewegt hat). Das ist zwar durchaus eine gute Idee (da leicht erweiterbar), aber für Anfänger/innen sehr schwer zu beherrschen.

Aus diesem Grunde haben wir für die Vorlesung selbst ein kleines, minimales Tool für die graphische Programmierung zusammengestellt, dass sich eher an Graphikbibliotheken für Heimcomputer der 80er/90er Jahre orientiert (im Stile von Q-Basic, AMOS-Basic oder Simons-Basic, falls das noch jemand kennt). Hier werden alle Eingabe- und Ausgabeoperationen direkt, synchron ausgeführt, so dass man den Kontrollfluss des Programms nicht an die Graphikbibliothek anpassen muss. Dies hat zwar Nachteile für „echte“ Real-World Systeme (hohe CPU-Auslastung im Leerlauf, schlechtere Erweiterbarkeit), ist aber sehr viel leichter zu lernen. „Richtige“ GUI-Programmierung mit ereignisorientierter Architektur lernen wir selbstverständlich später auch noch (kurz in „EIP“ und ausführlicher in „EIS“).

Ein Teil der Bibliothek empfindet auch die Funktionalität der „image“-Bibliothek von PyRet nach; einem anderen Werkzeug, dass von anderen Dozenten gerne in der Vorlesung EIP eingesetzt wird. PyRet ist als Browser-basiertes System leichter zu installieren und zu nutzen, erfordert aber das Lernen einer weiteren Programmiersprache, die Ideen von Python und LISP/Scheme mischt. In unserer Veranstaltung arbeiten wir ausschließlich mit Python; dazu müssen wir aber unsere eigene Erweiterung „basic_io“ zusätzlich installieren. Dieses Dokument erklärt, wie das geht, und wie man die Bibliothek nutzt.

Noch eine abschließende Bemerkung zum didaktischen Konzept: Warum brauchen wir überhaupt eine Graphikbibliothek in EIP? Aus Sicht des Dozenten bietet dies mehrere Vorteile, die gewissen Nachteile aufwiegen:

Vorteile:

- Programmieren mit graphischer Ausgabe und Interaktion macht mehr Spaß. Wir können in den Übungen kleine Computerspiele programmieren, anstatt Zahlen zu sortieren und auf der Konsole auszugeben (Informatik lernt man bei beiden, aber letzteres ist etwas „trocken").
- Die Architektur von `basic_io` erlaubt es, beim Aufbau von Graphikausgaben live „zuzusehen"; dadurch kann man den Ablauf von Programmen intuitiver verstehen. Insbesondere für das Erlernen imperativer Programmierkonzepte ist dies ein Vorteil.
- Die „objekt-basierte" Schnittstelle der Bibliothek veranschaulicht auch strukturierte Datentypen und rekursiv konstruierte Daten (sowie Algorithmen, die darauf arbeiten). Damit können Ideen aus der funktionale Programmierung besser verstanden werden.

Nachteile:

Nichts ist umsonst im Leben; wir erkaufen uns diese Vorteile mit zusätzlichem Aufwand:

- Der Einarbeitungsaufwand ist etwas höher, da man zusätzlich zu Python die APIs (Application Programmer's Interfaces = Schnittstellen zur Benutzung einer Bibliothek) von `basic_io` lernen müssen. Dies ist zwar nicht direkt klausurrelevant (wir werden nie verlangen, Befehle aus dem Packet auswendig zu lernen), und das API ist sehr einfach, aber es kostet etwas Zeit, damit vertraut zu werden. In praktischen „real-World" Anwendungen jenseits von EIP+EIP-Praktikum wird man das nicht nutzen (wollen).
- Die Bibliothek wurde extra für die Vorlesung entwickelt; entsprechend hat sie nicht den Reifegrad kommerzieller Software oder großer open-Source Projekte.
- Es ist etwas extra Aufwand zu Installation erforderlich.
- Wie später in dem Dokument beschrieben ist die Nutzung der Bibliothek nicht völlig frei von Wechselwirkungen; so kann es z.B. unter Windows passieren, dass Fehlermeldung wie „`Application xyz has stopped working, Windows can check online for a solution to the problem…`" auftreten, wenn das eigene Programm nicht ordentlich beendet wird. Dies ist unproblematisch (Spoiler: Es gibt auch keine Lösungen online, die Windows finden könnte), nervt aber vielleicht etwas[1].

Nach dieser Vorrede, wie installieren wir basic_io?

[1] Man kann es (systemweit) abschalten; einfach mal Internetsuche nach „Disable Program Has Stopped Working Error Dialog in Windows" oder ähnlichem um die Registry-Keys zu finden, die man setzen muss.

2 Installation

2.1 Voraussetzungen

Um die Bilbiothek installieren zu können, brauchen wir eine moderne Python Installation. basic_io ist kompatibel mit Python Versionen 3.6 oder höher (getestet mit 3.6, 3.8, 3.9 und 3.10) und läuft auf allen gängigen Desktopbetriebssystemen (getestet mit Windows 8, Windows 10 und verschiedenen Ubuntu-Linux Versionen; MacOS sollte funktionieren; mangels verfügbarer Hardware wurde es aber noch nicht darauf getestet).

Schritt 1: Python 3 Installieren

Als erstes müssen Sie auf Ihrem System die übliche Pythonumgebung für die Version 3 der Programmiersprache installieren. Dies geschieht auf Debian-Linux Varianten (inkl. Ubuntu) mit apt. Für ein aktuelles Ubuntu wäre die Kommandozeile

```
sudo apt-get install python3-dev
```

Unter Windows kann man einen interaktiven "Installer" von der Webseite von python.org herunterladen und einfach in ein beliebiges Verzeichnis installieren. Bei neueren Versionen von Windows muss man dazu ggf. ausschalten, dass Python über den eingebauten „Store" installiert werden soll. Dazu geht kann man in der Systemkonfiguration unter „Aliase für App-Ausführung verwalten" die Python-Aliase ausschalten. Außerdem sollte man im Windows-Installer einstellen, dass die Pfade auf den Python-Interpreter richtig gesetzt werden.

Schritt 2: Zusätzliche Pakete („Dependencies") installieren

Um `basic_io` benutzen zu können, sind zwei zusätzliche Python-Module notwendig, die nicht zur Standardinstallation gehören:

- `PyQt5` – eine Schnittstelle zur plattformübergreifenden GUI-Bibilithek Qt (in der Version 5), die unsere Bibliothek intern benutzt, um das Ein-/Ausgabefenster zu realisieren.
- `typeguard` – eine Bibliothek, die dynamische Typprüfungen vereinfacht. Dies wird benutzt, um automatisch Parameter zu validieren, die Sie basic_io beim Aufruf von Funktionen übergeben. Achtung: Derzeit ist basic_io nur mit Version 2.x.x kompatibel! *(Ergänzung SoSem 23)*

Die Installation ist auf allen Plattformen einfach (auch Windows), wenn Sie PIP in der Kommandozeile benutzen (Achtung, auf einigen älteren Systemen, wie Ubuntu vor Version 20.x, müssen Sie pip3 aufrufen, um sicherzustellen, dass es um Python 3.x und nicht Python 2.x geht):

```
pip install pyqt5
pip install typeguard==2.13.3
```

Nun ist alles bereit zur Nutzung der Bibliothek.

2.2 Manuelle Nutzung (ohne Installation)

Laden Sie als nächstes das Packet **basic_io.zip** von der Webseite des Kurses oder aus dem LMS herunter. Entpacken Sie den Inhalt in ein leeres Verzeichnis. Sie sollten eine Verzeichnisstruktur erhalten, die ungefähr so aussieht:

```
.
├── jguvc_eip
│   ├── basic_io_error.py
│   ├── basic_io.py
│   ├── colors.py
│   ├── image_objects.py
│   ├── _BasicIOWindowUI.py
│   ├── _BasicIOWindowUI.ui
│   ├── __init__.py
│   ├── _io_messages.py
│   ├── _io_window.py
│   └── _mylabel.py
├── license.txt
└── test_basic_io.py
```

Wichtig sind dabei nur die fettgedruckten Bestandteile:

- Das Verzeichnis „**jguvc_eip**“ ist ein Python-Modul, welches unsere Bibliothek enthält. „JGUVC“ steht dabei für die JGU AG Visual Computing, die das Modul entwickelt hat.
- **basic_io.py**, **basic_io_error.py**, **colors.py** und **image_objects.py** sind Teilmodule, verschiedene Funktionen der Bibliothek beinhalten.
- **test_basic_io.py** ist ein kleines Demoprogramm für die Funktionen der Bibliothek, mit dem man auch testen kann, ob alles funktioniert.
- Die Dateien, die mit einem Unterstrich beginnen, sind interne Module, die Sie nicht direkt benutzen sollten.

Die ganze Bibliothek steht unter der MIT-Open-Source Lizenz (siehe **license.txt**).

Sie können nun die Bibliothek ausprobieren, in dem Sie das Demo starten. Rufen Sie dazu auf der Konsole den folgenden Befehl auf

```
python test_basic_io.py
```

nachdem Sie in das Verzeichnis gewechselt sind, in das Sie den Inhalt der ZIP-Datei ausgepackt hatten. Es sollte sich ein Fenster öffnen, in dem einfache Graphiken und Animationen angezeigt werden. Mit den Tasten ‚**n**' und ‚**p**' kann man durch die Demos blättern, und mit ‚**q**' kann man das Programm beenden. Was genau hier zu sehen ist, ist nicht so interessant; wichtig ist nur, dass es überhaupt funktioniert.

Sie können nun eigene Programme schreiben, die basic_io nutzen, indem Sie diese im gleichen Verzeichnis wie „**test_basic_io.py**" anlegen (das Testscript selbst ist dafür natürlich nicht mehr nötig und kann gelöscht werden, falls gewünscht).

2.3 Installation von `basic_io` als Paket (in Vorbereitung)

Eleganter ist es, basic_io als Packet via PIP zu installieren. Zum Zeitpunkt, wo diese Dokumentation geschrieben wurde, war dies noch in Vorbereitung. Sie werden in der Vorlesung über die üblichen Kanäle Bescheid bekommen, sobald ein Packet zur Verfügung steht. Voraussichtlich wird die Installation dann unspektakulär via

```
pip install jguvc_eip
```

möglich sein (daher auch der komplizierte Name, damit es keine Konflikte im weltweiten PIP-Repository gibt).

3 Programmieren mit `basic_io`

Um die Bibliothek nutzen zu können, müssen Sie diese zunächst in Ihr Python-Script importieren. Sofern sie als PIP-Paket installiert wurde, oder Ihr eigenes Skript wie oben beschrieben so abgespeichert wurde, dass das Verzeichnis „**jguvc_eip**" ein direktes Unterverzeichnis des Verzeichnisses ist, in dem Ihr eigenes Skript liegt, dann können Sie wie folgt vorgehen:

`Datei: mein_script.py`

```
from jguvc_eip import basic_io as bio   # Hauptbibliothek
from jguvc_eip.colors import *          # [optional] Farbnamen
from jguvc_eip import image_objects     # [optional] Bildobjekte
```

Verzeichnisstruktur (falls nicht als PIP-Packet installiert):

```
.
├── jguvc_eip
│   └── ...
├── ...
└── mein_script.py
```

Mit den `import`-Anweisungen wird das „immediate-mode“ API von `basic_io` unter dem Namen `bio` bereitgestellt und außerdem noch einige Namen für Farben in den globalen Namensraum aufgenommen (was das Programmieren einfacher Programme erleichtert).

Sie können nun einfache Graphik wie folgt ausgeben:

`Datei: mein_script.py` (Fortsetzung, nach den “imports”)

```
# Initialisieren der Biblithek, öffnet das Fenster
bio.start()

# Ein roten Kreis zeichnen,
# Mittelpunkt x=100, y=100, Radius 50 Pixel
# Das Koordinatensystem ist links oben in der Ecke
# Standardmäßig haben Fenster 640x480 Pixel Auflösung
bio.draw_circle(100, 100, 50, RED)

# Warten, bis der Benutzer das Fenster wieder schließt
# (Damit wir die Ausgabe auch sehen können.)
bio.wait_close()
```

Die erste Anweisung sollte immer die erste in Ihrem Programm sein (zumindest bei einfachen Skripten); sie öffnet das Fenster und initialisiert die notwendige Hilfsstrukturen (Prozesse, Threads, GUI-Elemente) der Bibliothek.

Die zweite Anweisung zeichnet einen roten Kreis oben links in die Ecke des Fensters. Man übergibt ihr die x- und y-Koordinate, gefolgt vom Radius des Kreises und, optional, eine Füllfarbe. Weitere Parameter sind möglich (Randfarbe, Randstärke in Pixeln). Die Koordinaten nutzen ein Koordinatensystem, bei dem die linke obere Ecke des Ausgabebereiches (initial weiße Fläche im Fenster) die Koordinaten $(0,0)$ hat. Das Fenster ist am Anfang 640x480 Pixel groß – die untere rechte Ecke hat also die Koordinate $(639,479)$. Die Vergrößerung der Pixel ist in 4 Stufe im GUI von 1-fach bis 4-fach einstellbar (für High-DPI Displays); Standardwert ist zweifache Vergrößerung.

Die letzte Anweisung wartet darauf, dass das Fenster geschlossen wird. Dies sollte bei einfachen Programmen immer die letzte Anweisung in Ihrem Script sein, damit das Fenster noch sichtbar bleibt, wenn das Script schon durchgelaufen ist. Das Script endet dann automatisch, wenn der Benutzer das Fenster schließt.

Zum Zeichnen von Graphiken stehen eine Reihe von Befehlen zur Verfügung – siehe Kapitel 4.1 für eine komplette Liste.

Die Zeichenbefehle teilen sich dabei in ein „immediate-mode-API“ und ein „objektbasiertes API“ ein. Im ersten Fall geben Sie `basic_io` Befehle, die direkt Ausgaben erzeugen. Im zweiten Fall erzeugen Sie zunächst ein Datenobjekt, das eine ggf. komplexere Graphik repräsentiert, und zeichnen dieses dann in einem Rutsch (mit einem Zeichenbefehl) im Bild ein.

Zusätzlich gibt es auch Befehle für Eingaben: Tastatur- und Mauseingaben können abgefragt werden, z.B. um eine interaktive Anwendung oder ein Computerspiel zu programmieren.

Schlussendlich gibt es auch noch organisatorische Befehle, die z.B. mehrere (optionale) Bildpuffer verwalten, oder darauf warten, das der Benutzer das Programmfenster schließt (wie oben gerade gesehen).

Im folgenden Kapitel schauen wir uns alle Möglichkeiten genauer an. Das Kapitel ist als Referenz gestaltet, kann also auch zum Nachschlagen genutzt werden.

4 API Referenz

4.1 Modul `jgu_vc.basic_io`

Import des Moduls z.B. via

```
from jguvc_eip import basic_io  # Hauptbibliothek
```

(Man kann auch noch das "basic_io" abkürzen mit einem angehängten "`from jguvc_eip import basic_io as bio`" oder ähnlichem, wie oben im Text gemacht.)

4.1.1 Session Management

Diese Anweisungen steuern, ob und wann das Hauptfenster selbst erscheint und/oder wieder geschlossen wird.

`basic_io.start()`

Diese Funktion öffnet das Hauptfenster und initialisiert alle notwendigen Strukturen der Bibliothek (Hintergrundprozesse, Qt-Objekte, etc.).

Dies sollte die erste Anweisung in jedem einfachen Programm sein.

Falls man `start()` vergisst, wird es bei der ersten Eingabe- bzw. Ausgabeoperation automatisch nachgeholt. Dies ist aber etwas unsauber – man sollte es explizit aufrufen.

Rückgabetyp

None

`basic_io.wait_close()`

Diese Anweisung wartet schlicht darauf, dass das Hauptfenster durch den Benutzer geschlossen wird.

Dies ist typischerweise die letzte Anweisung in jedem einfachen Programm.

Rückgabetyp

None

`basic_io.close_and_exit()`

Diese Anweisung schließt das Hauptfenster und beendet das Programm sofort (egal, wo und wann sie aufgerufen wird).

Rückgabetyp

None

4.1.2 Zeichenbefehle

Diese Anweisungen geben graphische Elemente aus.

basic_io.**draw_line**(*start_x, start_y, end_x, end_y, color=(0, 0, 0), thickness=1*)

Zeichnet eine Line auf das aktive Bild.

Parameter

- **start_x** (int) – Startpunkt der Line (x-Koordiante) [Pixel]
- **start_y** (int) – Startpunkt der Line (y-Koordinate) [Pixel]
- **end_x** (int) – Endpunkt der Linie (x-Koordinate) [Pixel]
- **end_y** (int) – Endpunkt der Linie (y-Koordinate) [Pixel]
- **color** (Tuple[int, int, int]) – Farbe der Linie (Standardwert: BLACK (Schwarz)) [0..255]³
- **thickness** (int) – Linienbreite, Standardwert 1 [Pixel]

Rückgabetyp

None

basic_io.**draw_circle**(*x, y, radius, fill_color=(0, 0, 0), border_color=(0, 0, 0), border_thickness=1*)

Zeichnet einen Kreis auf das aktive Bild.

Parameter

- **x** (int) – Mittelpunkt, x-Koordinate [Pixel]
- **y** (int) – Mittelpunkt, y-Koordinate [Pixel]
- **radius** (int) – Radius [Pixel]
- **fill_color** (Optional[Tuple[int, int, int]]) – Farbe für die Füllung ([0…255]³). Wird dieser Wert auf **None** gesetzt, dann wird nicht gefüllt.
- **border_color** (Optional[Tuple[int, int, int]]) – Farbe für die Umrandung ([0…255]³). Wird dieser Wert auf **None** gesetzt, dann wird der Rand nicht gezeichnet.
- **border_thickness** (int) – Breite/Dicke der Umrandung (ein Wert von 0 bedeutet, dass ebenfalls keine Umrandung gezeichnet wird) [Pixel]

Rückgabetyp

None

basic_io.**draw_ellipse**(*x*, *y*, *radius_x*, *radius_y*, *fill_color=(0, 0, 0)*, *border_color=(0, 0, 0)*, *border_thickness=1*)

Zeichnet eine Ellipse auf das aktive Bild (mit Hauptachsen entlang der x- und y-Richtung).

Parameter

- **x** (int) – Mittelpunkt, x-Koordinate
- **y** (int) – Mittelpunkt, y-Koordinate
- **radius_x** (int) – Halbmesser in x-Richtung
- **radius_y** (int) – Halbmesser in y-Richtung
- **fill_color** (Optional[Tuple[int, int, int]]) – Farbe für die Füllung ([0…255]³). Wird dieser Wert auf **None** gesetzt, dann wird nicht gefüllt.
- **border_color** (Optional[Tuple[int, int, int]]) – Farbe für die Umrandung ([0…255]³). Wird dieser Wert auf **None** gesetzt, dann wird der Rand nicht gezeichnet.
- **border_thickness** (int) – Breite/Dicke der Umrandung (ein Wert von 0 bedeutet, dass ebenfalls keine Umrandung gezeichnet wird) [Pixel]

Rückgabetyp

None

basic_io.**draw_rectangle**(*x*, *y*, *width*, *height*, *fill_color=(0, 0, 0)*, *border_color=(0, 0, 0)*, *border_thickness=1*)

Zeichnet ein Rechteck auf das aktive Bild.

Parameter

- **x** (int) – x-Koordinate der linken oberen Ecke [Pixel]
- **y** (int) – y-Koordinate der linken oberen Ecke [Pixel]
- **width** (int) – Breite [Pixel]
- **height** (int) – Höhe [Pixel]
- **fill_color** (Optional[Tuple[int, int, int]]) – Farbe für die Füllung ([0…255]³). Wird dieser Wert auf **None** gesetzt, dann wird nicht gefüllt.
- **border_color** (Optional[Tuple[int, int, int]]) – Farbe für die Umrandung ([0…255]³). Wird dieser Wert auf **None** gesetzt, dann wird der Rand nicht gezeichnet.
- **border_thickness** (int) – Breite/Dicke der Umrandung (ein Wert von 0 bedeutet, dass ebenfalls keine Umrandung gezeichnet wird) [Pixel]

Rückgabetyp

None

basic_io.**draw_polygon**(*points*, *fill_color=(0, 0, 0)*, *border_color=(0, 0, 0)*, *border_thickness=1*)

Zeichnet ein Polygon auf das aktive Bild.

Parameter

- **points** (List[Tuple[int, int]]) – Eine Liste von Tupeln (x:int, y:int) die jeweils die x- und y-Koordinaten der Punkte des Polygonsenhalten ([Pixel] x [Pixel])#Punkte
- **fill_color** (Optional[Tuple[int, int, int]]) – Farbe für die Füllung ([0…255]³). Wird dieser Wert auf None gesetzt, dann wird nicht gefüllt.
- **border_color** (Optional[Tuple[int, int, int]]) – Farbe für die Umrandung ([0…255]³). Wird dieser Wert auf None gesetzt, dann wird der Rand nicht gezeichnet.
- **border_thickness** (int) – Breite/Dicke der Umrandung (ein Wert von 0 bedeutet, dass ebenfalls keine Umrandung gezeichnet wird) [Pixel]

Rückgabetyp

None

basic_io.**draw_text**(*x*, *y*, *text*, *color=(0, 0, 0)*, *background_color=None*, *font_height=16*)

Zeichnet eine Beschriftung (einzeiliger Text) auf das aktuelle Bild. Der Text wird in einem Font mit fester Breite dargestellt (alle Zeichen sind gleich breit).

Parameter

- **x** (int) – x Koordinate an der der Text beginnt (obere linke Ecke) [Pixel]
- **y** (int) – y Koordinate an der der Text beginnt (obere linke Ecke) [Pixel]
- **text** (str) – Der auszugebende Text [string]
- **color** (Tuple[int, int, int]) – Farbe für den Text (optional, Standardwert=BLACK) [0…255]³
- **background_color** (Optional[Tuple[int, int, int]]) – Farbe für den Hintergrund, der als Rechteck hinter den Text gelegt wird. Standardwert ist None (was bedeutet, dass kein Hintergrund gezeichnet wird) Wertebereich sonst: [0…255]³
- **font_height** (int) – Die Höhe der zu zeichnenden Zeichen in Pixel (optional, Standardwert=16 Pixel) [Pixel]

Rückgabetyp

None

`basic_io.draw_Pixel(x, y, color=(0, 0, 0))`

Färbt einen einzelnen Pixel in der angegebenen Farbe ein.

Parameter

- **x** (`int`) – x Koordinate des Pixels
- **y** (`int`) – y Koordinate des Pixels
- **color** (`Tuple[int, int, int]`) – Farbe (optional, Standardwert=BLACK (Schwarz)) $[0…255]^3$

Rückgabetyp

`None`

4.1.3 Auslesen des aktuellen Bildes

Man kann auch Pixel des aktuellen Bildes auslesen (z.B. für Kollisionserkennung in Spielen).

`basic_io.read_Pixel(x, y)`

Liest die Farbe eines Pixels aus dem aktiven Bild aus.

Parameter

- **x** (`int`) – x Koordinate des Pixels
- **y** (`int`) – y Koordinate des Pixels

Resultat

Eine Farbe (RGB Tuple aus {0,1,2,…,255}^3 oder (-1, -1, -1) falls etwas schiefgegangen ist, insbesondere, falls die Pixelkoordinate außerhalb des Bereiches des aktiven Bildes lag).

Rückgabetyp

`Tuple[int, int, int]`

4.1.4 Unterstützung für Sprites/Icons/Bobs

Die nächsten zwei Befehle dienen dazu, (typischerweise kleine) Bildobjekte anzulegen, die man auf das aktive Bild einzeichnen kann. Als Quelle sind PNG und JPG-Dateien erlaubt, wobei die Transparanz von PNG-Dateien beim Zeichnen berücksichtigt / genutzt wird (z.B., um eine Spielfigur für ein Computerspiel zu bauen).

Die Bilder werden dabei immer zunächst aus einer Datei eingeladen, und mit einer Nummer versehen, die den Bildspeicher angibt. Unter Angabe dieser Nummer können die Bilder später auf das aktive Bild des Ausgabefensters gezeichnet werden.

`basic_io.load_image(`*`filename`*`)`

Lädt eine Bilddatei aus einer Datei ein.

Parameter

`filename` (`str`) – Dateiname. Die Funktion akzeptiert sowohl PNG wie auch JPG Dateien. Bei PNG-Dateien wird ein möglicher Alphakanal (Transparenz) mit eingelesen und korrekt dargestellt (die gezeichneten Graphiken sind entsprechend teilweise durchsichtig). Dies ist nützlich um Sprites und Bobs[2] oder Icons für ein GUI darzustellen.

Rückgabewert

Ein index (ganze Zahl / **`int`**), unter der das Bild intern abgelegt wurde. Sollte irgend eine Art von Fehler auftreten (insbesondere "file not found", also die Datei mit dem angegebenen Namen wurde nicht gefunden, oder enthält keine Daten in einem unterstützten Format), so wird der Index -1 zurückgegeben. Gültige Bilder haben immer positive Indices 0,1,2,3,... Entsprechend bezeichnet der Index -1 immer ein ungültiges Bild (welches auch nicht auf dem Bildschirm ausgegeben werden kann)

Rückgabetyp

`int`

[2] Sprites waren kleine bewegliche Graphikobjekte auf alten Heimcomputern, die über einen Hintergrund eingeblendet werden konnten. Für unsere Library müssen wir erst den Hintergrund zeichnen und dann die geladenen Graphiken darübermalen, um diesen Effekt zu erzielen. Dies war auf leistungsfähigerer Retro-Hardware wie dem Commodore AMIGA als „BOBs" (Blitter-Objects; blitting steht für das Kopieren von Biddaten) bekannt.

`basic_io.draw_image(x, y, index)`

Zeichnet ein zuvor eingeladenes Bild auf das Fenster (genauer: den aktiven Bildpuffer, wie bei allen Zeichenbefehlen).

Parameter

- **x** (int) – x-Koordinate des linken oberen Pixels des zu zeichnenden Bildes
- **y** (int) – y-Koordinate des linken oberen Pixels des zu zeichnenden Bildes
- **index** (int) – Der Index eines zuvor eingeladenen Bildes. Sollte dieser ungültig (zu groß oder zu klein) sein, so passiert gar nichts.

Rückgabetyp

None

4.1.5 Objekt-Basierte Graphikausgabe

Mit dem folgenden Befehl können komplexere Graphikobjekte auf dem Bildschirm ausgegeben werden. Der Ausgabebefehl arbeitet dabei wie ein normaler Zeichenbefehl, stellt aber (auch) komplexere, zusammengesetzte Objekte dar.

`basic_io.draw_object(obj, x=0, y=0)`

Zeichnet ein Objekt auf dem Bild ein. Wie man Objekte erstellen kann, ist im nächsten Abschnitt 4.2 (Module image_objects.py) erklärt.

Parameter

- **obj** (ImageObject) – Das Bildobjekt, das gezeichnet werden soll.
- **x** (int) – (optional) Eine Verschiebung / Offset in x-Richtung (Standardwert: 0)
- **y** (int) – (optional) Eine Verschiebung / Offset in y-Richtung (Standardwert: 0)

Rückgabetyp

None

4.1.6 Verwaltung der Bildpuffer

Die Bibliothek unterstützt bis zu zehn verschiedene Bildpuffer. Einer davon ist dabei immer als aktuell *sichtbarer* und einer als aktuell *aktiver* Puffer ausgewählt. Der sichtbare wird angezeigt, und der aktive ist der, auf den gezeichnet wird. Standardmäßig ist Puffer 0 aktiv und sichtbar.

Man kann z.B. mehrere Puffer nutzen, um „Double-Buffering“ zu implementieren, d.h., dass ein Bild in einem unsichtbaren Puffer aufgebaut wird, und erst nachdem es fertig erstellt ist sichtbar gemacht wird. Damit kann man Flackern verhindern.

Eine weitere mögliche Anwendung ist das Erstellen von Hintergrundbildern, die z.B. bei einem Spiel oder einem Anwendungsprogramm mit dem Kopierbefehl in das aktive Bild hereinkopiert werden können.

`basic_io.clear_image(color=(255, 255, 255))`

Dieser Befehl löscht das aktive Bild, d.h. überschreibt alle Pixel mit der angegebenen Farbe.

Parameter

color (`Tuple`[`int`, `int`, `int`]) – die Farbe, mit der das Bild übermalt werden soll. Standardwert ist WHITE (weiß).

Rückgabetyp

`None`

`basic_io.set_active_image(buffer)`

Dieser Befehl wählt ein Bild als aktiv aus, d.h. alle Zeichenbefehle sowie auch `read_pixel` wirken auf diese Bild.

`basic_io` stellt 10 "Bildpuffer" bereit, indiziert mit den Zahlen 0 bis 9. Zahlen außerhalb dieses Wertebereiches lösen eine Ausnahme (Exception) aus. Falls ein Bildpuffer noch nicht existiert, wird dieser automatisch neu angelegt, mit einer Standardauflösung von 640x480 Pixeln. Standardmäßig existiert nur Puffer 0, und dieser ist sowohl aktiv wie auch sichtbar.

Parameter

buffer (`int`) – Index des Puffers auf den ab jetzt alle Zeichenbefehle wirken sollen

Rückgabetyp

`None`

`basic_io.set_visible_image(buffer)`

Dieser Befehl wählt ein Bild als sichtbar aus, d.h. die Bibliothek überträgt den Inhalt des Bildpuffers laufend in das Hauptfenster (bis zu ca. 60 mal pro Sekunde, aber nur bei Änderungen).

`basic_io` stellt 10 "Bildpuffer" bereit, indiziert mit den Zahlen 0 bis 9. Zahlen außerhalb dieses Wertebereiches lösen eine Ausnahme (Exception) aus. Falls ein Bildpuffer noch nicht existiert, wird dieser automatisch neu angelegt, mit einer Standardauflösung von 640x480 Pixeln. Standardmäßig existiert nur Puffer 0, und dieser ist sowohl aktiv wie auch sichtbar.

Parameter

buffer (`int`) – Index des Puffers, der ab sofort sichtbar ist (angezeigt wird).

Rückgabetyp

`None`

basic_io.**copy_image**(*from_buffer, to_buffer*)

Dieses Kommando kopiert den Inhalt eines Bildpuffers in einen anderen. Dabei werden die Abmessungen (Breite x Höhe) des Quellpuffers übernommen.

Parameter

- **from_buffer** (int) – Index der Quelle [0,1,…,9]
- **to_buffer** (int) – Index des Ziels [0,1,…,9]

Rückgabetyp

None

basic_io.**resize_image**(*width, height, color=(255, 255, 255)*)

Dieser Befehl setzt eine neue Größe für den aktiven Bildpuffer. Dabei wird die angegebene Farbe benutzt, um das Bild initial damit zu füllen (alles wird gelöscht).

Parameter

- **width** (int) – neue Breite des Bildes in Pixeln, muss im Bereich [1,2,…,10000] liegen
- **height** (int) – neue Höhe des Bildes in Pixeln, muss im Bereich [1,2,…,10000] liegen
- **color** (Tuple[int, int, int]) – die Farbe, mit der das Bild initialisiert wird. Standardwert ist weiß (WHITE).

Rückgabetyp

None

4.1.7 Live-Eingabe (nicht-blockierend)

Mit den Befehlen in diesem Abschnitt kann man „live“ abfragen, ob der Benutzer über die Tastatur oder Maus dem Programm Eingaben oder Befehle hat zukommen lassen. Live bedeutet genauer, dass die Befehle nicht blockieren, also immer den aktuellen Zustand abfragen und nicht warten, bis ein bestimmtes Ereignis eingetreten ist.

Es gibt dennoch zwei Varianten: Für die Tastatureingabe ist eine Eingabe mit und ohne „Puffer“ möglich. (Achtung: Der Begriff Puffer bezieht sich in diesem Abschnitt *nicht* auf die Bildpuffer; es ist nur das gleiche Wort.)

Mit Puffer werden gedrückte Tasten auch dann aufgezeichnet, wenn sie während des Aufrufes des Befehls nicht gedrückt waren, aber seit dem letzten Aufruf gedrückt wurden.

Ohne Puffer wird nur der aktuellen Zustand (Taste gedrückt oder nicht) berichtet; die Vergangenheit spielt keine Rolle.

Beide Varianten sind nicht-blockierend. Es gibt auch eine blockierende Eingabeoption – siehe weiter unten (Abschnitt 4.1.8).

Wann immer eine ungepufferte Eingabe abgefragt wird, löscht dies automatisch den vorhandenen Puffer (um logische Probleme zu vermeiden).

Ungepufferte Eingaben sind besonders nützlich zur „live"-Steuerung wie z.B. der Steuerung einer Spielfigur. Gepufferte Eingaben sind nützlich, um ein Programm zu mit Tastaturkommandos zu steuern oder getippten Text entgegenzunehmen, da keine Tastendrücke verloren gehen können, auch nicht wenn das Programm gerade noch rechnet.

Noch eine Anmerkung zur Effizienz: Wenn man eine Schleife programmiert, die ohne Unterbrechung Eingaben abfragt, dann führt dies zu 100% Auslastung einer CPU. Wenn man dies verhindern möchte, kann man das Programm immer wieder kurz Schlafen legen:

```
from time import sleep

while True:                         # Hauptschleife z.B. eines Spiels
    key = basic_io.get_current_keys_down()
    # ...Spielsteuerung...
    # ...Bild neuzeichnen...
    sleep(0.016)
```

Eine Wartezeit von 16msec entspricht einer Bildwiederholrate von 60Hz[3]; wenn die Rechnungen nicht sehr aufwendig sind, kann man auch eine kürzere Wartezeit (1…5msec) nutzen.

[3] Für Experten/innen: Eine Sychronisation mit dem Bildaufbau des Monitors („WaitVBL") ist leider nicht vorgesehen, da schwer platformabhängig zu realisieren.

Eingabe mit Pufferung

`basic_io.get_last_key_pressed_event()`

Diese Funktion behandelt gepufferte Tastatureingaben.

Sie funktioniert wie folgt: Immer wenn eine Taste gedrückt wird, dann speichert basic_io diese in einem internen Puffer (auch, wenn Ihr Programm gerade noch läuft und etwas anderes macht). Wird die Funktion dann aufgerufen, so liefert sie die Ereignisse, die seit dem letzten Aufruf aufgetreten sind. Jeder Aufruf gibt den ältesten Tastendruck aus dem Puffer aus und löscht ihn daraus. Das heißt, die Ereignisse erscheinen in der Reihenfolge, in der Sie ursprünglich aufgetreten. Sollte der Puffer leer sein (z.B. weil gar nichts passiert ist), dann wird ein leerer String `''` zurückgegeben.

Unterstützte Tasten und Einschränkungen:

- Derzeit werden nur die englischen Buchstaben 'a'-'z' und 'A'-'Z' aus dem ASCII-Zeichensatz, sowie die Zahlen and numbers '0'-'9' erkannt; alle anderen Zeichen und Buchstaben werden ignoriert.
- Spezielle Steuertasten, die erkannt werden sind:
  - Die Leertaste (space bar) als `' '`
  - Die Eingabetaste (return) als `'\n'`
  - Die Cursortasten als `'cursor_left'`, `'cursor_right'`, `'cursor_up'`, `'cursor_down'`
  - Klicks mit der Maus werden ebenfalls erkannt und als `'left_mouse_button'` bzw. `'right_mouse_button'` weitergegeben.

Rückgabetyp

`str`

Rückgabewert

Ein String der entweder ein einzelnes Zeichen enthält (`' '`, `'a'`, …, `'z'`, `'A'`, …, `'Z'`, `'0'`, …, `'9'` bzw. `'\n'`, was auch ein einzelnes Zeichen ist) oder ein Mehrzeichenstring für eine Spezialtaste (`'cursor_left'`, `'cursor_right'`, `'cursor_up'`, `'cursor_down'`, oder `'left_mouse_button'`, `'right_mouse_button'` für Mausklicks). Falls keine Eingabe vorliegt, wird ein leerer String (`''`) zurückgegeben.

`basic_io.clear_key_pressed_event_buffer()`

Löscht alle noch im Eingabepuffer vorhandenen Tastenereignisse.

Rückgabetyp

`None`

Eingabe ohne Pufferung

`basic_io.get_current_keys_down()`

Diese Funktion verarbeitet Tastatureingaben ohne Puffer.

Sie gibt eine Liste von Tasten zurück, die gerade zum aktuellen Zeitpunkt niedergedrückt sind (genauer: zu dem Zeitpunkt, zu dem der Fensterprozess die Nachricht zur Abfrage der aktuell gedrückten Tasten erhält).

Jeder Aufruf dieser Funktion löscht auch den Puffer für die gepufferte Tastatureingabe, um Vermischungen und Verwirrungen zu verhindern.

Unterstützte Tasten und Einschränkungen:

- Derzeit werden nur die englischen Buchstaben ‘a’-‘z’ und ‘A’-‘Z’ aus dem ASCII-Zeichensatz, sowie die Zahlen and numbers ‘0’-‘9’ erkannt; alle anderen Zeichen und Buchstaben werden ignoriert.
- Spezielle Steuertasten, die erkannt werden sind:
  - Die Leertaste (space bar) als `' '`
  - Die Eingabetaste (return) als `'\n'`
  - Die Cursortasten als `'cursor_left'`, `'cursor_right'`, `'cursor_up'`, `'cursor_down'`
  - Klicks mit der Maus werden ebenfalls erkannt und als `'left_mouse_button'` bzw. `'right_mouse_button'` weitergegeben.

Rückgabetyp

`List[str]`

Rückgabewert

Eine Liste von gedrückten Taste. Jede einzelne Taste ist dabei als String kodiert, der entweder ein einzelnes Zeichen enthält (`' '`, `'a'`, …, `'z'`, `'A'`, …, `'Z'`, `'0'`, …, `'9'` bzw. `'\n'`, was auch ein einzelnes Zeichen ist) oder ein Mehrzeichenstring für eine Spezialtaste (`'cursor_left'`, `'cursor_right'`, `'cursor_up'`, `'cursor_down'`, oder `'left_mouse_button'`, `'right_mouse_button'` für Mausklicks). Falls keine Eingabe vorliegt, ist die Liste leer, die zurückgegeben wird..

`basic_io.`**`get_current_mouse_position`**`()`

Fragt die aktuelle Position der Maus ab (sofern diese sich über dem Ausgabefenster befindet; sonst wird (-1,-1) zurückgegeben. Ein Klicken und gedrückthalten der Maus über dem Bild „fängt“ den Mouse-cursor ein, so dass auch Koordinten außerhalb des Programmfensters zurückgegeben werden können, solange die Maustaste gedrückt bleibt.

Die Mauskoordinaten beziehen sich auf den aktuell *sichtbaren* Bildpuffer, können also auch dann negativ sein, wenn sich der Mauszeiger noch über dem Fenster befindet. Da es möglich ist, nur Auschnitte aus dem sichtbaren Bildpuffer darzustellen (mittels Zoom und Scrollbalken im GUI), ist es auch möglich, dass nur ein Ausschnitt des Bildes von der Maus erreicht wird (genau der sichtbare Ausschnitt).

Rückgabetyp

`Tuple`[`int`, `int`]

Rückabewert

Die aktuellen Mauskoordinaten als Tuple (x:int, y: int) in Pixeln; falls keine Mauskoordinaten zur Verfügung stehen, wird (der Einfachheit halber) (-1,-1) zurückgegeben (auch wenn dieser Wert auch als „gültige“ Koordinate möglich ist, wenn der Mauszeiger zwar das sichtbare Bild, aber nicht das Programmfenster verläßt.

4.1.8 Eingebaute Konsole

Das Hauptfenster hat eine einfache Konsole eingebaut, die man zusätzlich zu / unabhängig von der Python Standardausgabe, die mit **`print()`** angesprochen würde, benutzen kann. Grund hierfür ist, dass es manchmal praktischer ist, die Ausgaben in der Nähe des Graphikfensters zu sehen, und man nicht erst das zugehörige Konsolenfenster suchen möchte. Dies ist z.B. nützlich für Debugging oder diagnostische Zwecke; auch kurze Erklärungen und Anleitungen zum eigenen Programm können hier bereitgestellt werden.

`basic_io.`**`print_message`**`(`*`text`*`)`

Gibt einen Text auf dem eingebauten Konsolenbereich (Standardmäßig unten am Hauptfenster angeheftet) aus.

Parameter

`text` (`str`) – der auszugebende Text als (einzelnes) Stringobjekt. Listen mit mehreren Ausgaben (wie bei Python **`print(x, 'text', y)`** sind nicht möglich).

Rückgabetyp

`None`

basic_io.**print_html**(*text*)

Gibt den Text im HTML-Format aus (der Inhalt des Strings wird als HTML interpretiert). Dies nutzt direkt die von Qt bereitgestellten Funktionen zu diesem Zwecke. Die Funktion ist allerdings nicht sehr stabil und unterstützt nicht viele Tags/Features; von der Benutzung wird daher eher abgeraten.

Parameter

text (str) – der auszugebende String im HTML-Format

Rückgabetyp

None

basic_io.**input_string**(*question*)

Dieses Kommando öffnet eine kleine Eingabebox am unteren Rande des Konsolenbereiche und fragt einen Text ab (mit der Frage ***question***) und gibt das Resultat zurück. Dieses Kommando ist blockierend, d.h., die Funktion wartet, bis der Benutzer die Eingabe abgeschlossen hat, was durch Drücken der Eingabetaste im Editierfeld oder durch Drücken eines ebenfalls auftauchenden OK-Buttons geschehen kann. Die Ergebnisse sind (daher) immer einzeilige Strings (da Return die Eingabe beendet).

Parameter

question (str) – Die Frage, die vor der Eingabebox erschienen soll. Dies sollte ein kurzer String sein.

Rückgabewert

Der eigegebene Text als String (kann leer sein)

Rückgabetyp

str

4.2 Modul `jgu_vc.image_objects`

Import des Moduls z.B. via

```
from jguvc_eip import image_objects
```

(Man kann auch hier noch das "image_objects" abkürzen mit einem angehängten "`as im_objs`" oder ähnlichem, um Tipparbeit zu sparen.)

4.2.1 Übersicht der Klassen/Typen von Bildobjekten

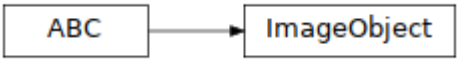
- `ImageObject`: Basisklasse – nicht direkt benutzbar.
- `Rectangle`: Ein Rechteck
- `Circle`: Ein Kreis
- `Ellipse`: Eine Ellipse
- `Polygon`: Ein Polygon
- `Text`: Ein Text / eine Beschriftung (mehr Optionen als die entsprechende "immediate-mode" Funktion aus Abschnitt 4.1).
- `VerticalStack`: Hiermit kann man mehrere Objekte übereinander stapeln.
- `HorizontalStack`: Hiermit kann man mehrere Objekte nebeneinander anordnen.
- `Overlay`: Hiermit kann man Objekte übereinander legen (nacheinander übereinander gezeichnet).
- `Translate`: Hiermit kann man Objekte verschieben.
- `Scale`: Hiermit kann man Objekte vergrößern und verkleinern ("Skalieren").
- `Rotate`: Diese Klasse rotiert Objekte gegen den Uhrzeigersinn um ihren Mittelpunkt.

4.2.2 Basisklasse

Die Basisklasse definiert, wie abstrakte Bildobjekte sich allgemein verhalten. Die Basisklasse ist für den Endanwender uninteressant. Wenn Sie nur Graphikobjekte erstellen möchten (und keinen eigenen Klassen von Bildobjekten definieren möchten), dann können Sie diesen Abschnitt einfach überblättern.

class image_objects.ImageObject

Vererbungshierarchie



Methoden

abstract draw(*painter: QPainter*)

Zeichnet das Objekt auf einem QPainter-Objekt ein. Diese Methode muss man überschreiben, wenn man neue, eigene Typen an Bildobjekten definieren möchte. Für die Anwendung der Bibliothek ist die Funktion ohne Belang.

Rückgabetyp

None

Parameters

painter (*PyQt5.QtGui.QPainter*) – das QPainter objekt (Qt-interne Klasse)

abstract get_height()

Gibt die Höhe des Bildobjektes in Pixeln zurück. Auch dies wird für die reine Anwendung der Bibliothek in der Regel nicht gebraucht.

Rückgabetyp

int

abstract get_width()

Gibt die Breite des Bildobjektes in Pixeln zurück. Auch dies wird für die reine Anwendung der Bibliothek in der Regel nicht gebraucht.

Rückgabetyp

int

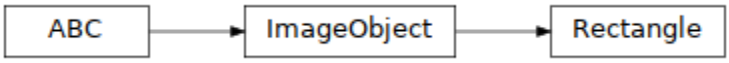
4.2.3 Konkrete Formen

Die folgenden Klassen stellen konkrete geometrische Objekte dar.

class image_objects.Rectangle(*width*, *height*, *fill_color=(0, 0, 0)*, *border_color=(0, 0, 0)*, *border_thickness=1*)

Diese Klasse repräsentiert Rechtecke. Das Objekt hat eine Gesamtgröße von [0, width] x [0, height].

Vererbungshierarchie



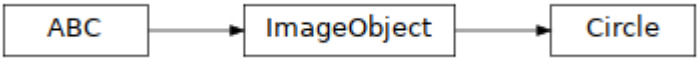
Parameter des Konstruktors

- **width** (*int*) – Breite des Rechteckes
- **height** (*int*) – Höhe des Rechtecks
- **fill_color** (*Optional[Tuple[int, int, int]]*) – Füllfarbe des Rechtecks. None schaltet die Füllung ab.
- **border_color** (*Optional[Tuple[int, int, int]]*) – Rahmenfarbe des Rechtecks. None schaltet den Rahmen ab.
- **border_thickness** (*int*) – Dicke des Rahmens

class image_objects.Circle(*radius*, *fill_color=(0, 0, 0)*, *border_color=(0, 0, 0)*, *border_thickness=1*)

Diese Klasse repräsentiert einen Kreis. Die Gesamtgröße ist [0,2*radius] x [0,2*radius].

Vererbungshierarchie



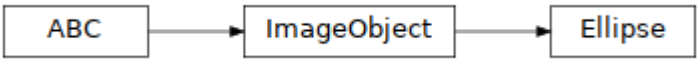
Parameter des Konstruktors

- **radius** (*int*) – Radius des Kreises
- **fill_color** (*Optional[Tuple[int, int, int]]*) – Füllfarbe des Rechtecks. None schaltet die Füllung ab.
- **border_color** (*Optional[Tuple[int, int, int]]*) – Rahmenfarbe des Rechtecks. None schaltet den Rahmen ab.
- **border_thickness** (*int*) – Dicke des Rahmens

class image_objects.Ellipse(*width*, *height*, *fill_color=(0, 0, 0)*, *border_color=(0, 0, 0)*, *border_thickness=1*)

Diese Klasse repräsentiert eine Ellipse mit Hauptachsen in x- und y-Richtung (andere Orientierungen sind mit einem zusätzlichen Rotate-Objekt möglich). Die Gesamtgröße ist [0, width] x [0, height].

Vererbungshierarchie



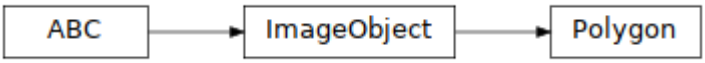
Parameter des Konstruktors

- **width** (*int*) – Breite der Ellipse
- **height** (*int*) – Höhe der Ellipse
- **fill_color** (*Optional[Tuple[int, int, int]]*) – Füllfarbe des Rechtecks. None schaltet die Füllung ab.
- **border_color** (*Optional[Tuple[int, int, int]]*) – Rahmenfarbe des Rechtecks. None schaltet den Rahmen ab.
- **border_thickness** (*int*) – Dicke des Rahmens

class image_objects.Polygon(*points*, *fill_color=(0, 0, 0)*, *border_color=(0, 0, 0)*, *border_thickness=1*)

Objekte dieser Klasse repräsentiertieren Polygonzüge. Als Breite und Höhe des Objektes werden die Maximalwerte der x- und y- Koordinaten genommen. Die minimalen x- und y-Koordinaten sollten genau 0 sein (und nicht negativ), damit die Ausrichtung in Containern wie „VerticalStack“ richtig funktioniert. Dies wird aber nicht überprüft / erzwungen.

Vererbungshierarchie



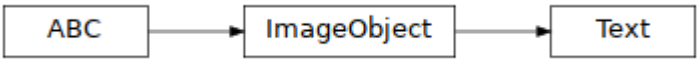
Parameter des Konstruktors

- **points** (*List[Tuple[int, int]]*) – Eine Liste von Punkten (2-Tupel von x- und y-Koordinaten als ganze Zahlen (int)).
- **fill_color** (*Optional[Tuple[int, int, int]]*) – Füllfarbe des Rechtecks. None schaltet die Füllung ab.
- **border_color** (*Optional[Tuple[int, int, int]]*) – Rahmenfarbe des Rechtecks. None schaltet den Rahmen ab.
- **border_thickness** (*int*) – Dicke des Rahmens

class image_objects.Text(*text*, *color=(0, 0, 0)*, *background_color=None*, *font_height=16*, *bold=False*, *italic=False*, *fixed_width=True*)

Objekte dieser Klasse repräsentiert ein Stück Text (eine Beschriftung), die durch einen String angegeben wird. Der Text kann in einer frei wählbaren Farbe und optional auf einen Hintergrund mit frei wählbarer Farbe gezeichnet werden. Es gibt zwei verschiedene Font-Stiele (feste Zeichenbreite [Standard] und den Standard-System-Font, der in der Regel proportional, ist, also so, dass „l" schmaler als „W" ausfällt). Schrift kann auch in Fettgedruckt und Kursiv gesetzt werden.

Vererbungshierarchie



Parameter des Konstruktors

- **text** (*str*) – der auszugebende Text
- **color** (*Tuple[int, int, int]*) – die Farbe der Schrift (Standard: Schwarz)
- **background_color** (*Optional[Tuple[int, int, int]]*) – die Farbe des Hintergrundes; bei None (Standard) wird kein Hintergrund gezeichnet.
- **font_height** (*int*) – die Höhe der Zeichen in Pixeln
- **bold** (*bool*) – True schaltet fette Schrift ein. Standard: False.
- **italic** (*bool*) – True schaltet kursive Schrift ein. Standard: False.
- **fixed_width** (*bool*) – True wählt den Font mit fester Zeichenweite, false den Proportionalfont. Standard: True.

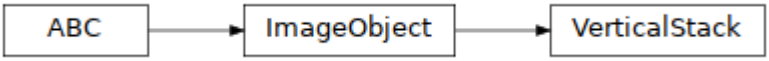
4.2.4 Zusammengesetzte Objekte

Um komplexere Graphikobjekte zu konstruieren, können mehrere Objekte zusammengefasst werden. Es ist möglich, zusammengefasste Objekte nochmals zusammenzufassen; dabei entsteht dann eine mehrstufige Schachtelung, formal: eine sogenannte Baumstruktur[4].

class image_objects.VerticalStack(*objects*, *margin=0*)

Diese Klasse fasst mehrere Objekte zu einem zusammengesetzten Objekt zusammen, indem die einzelnen Objekte in der angegebenen Reihenfolge von oben nach unten vertikal aufeinandergesetzt werden. Optional kann ein innerer Rand („margin“) angegeben werden, also ein Abstand, der um jedes Objekt herum freigehalten wird.

Vererbungshierarchie



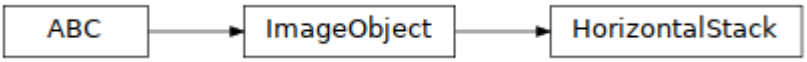
Parameter des Konstruktors

- **objects** *(List[image_objects.ImageObject])* – Eine Liste von Bildobjekten. Alle in Abschnitt 4.2 gelisteten Klassen sind zuslässig.
- **margin** (*int*) – Abstand um jedes Objekt herum, der zusätzlich freigehalten werden soll. (Optional – Standard ist 0).

class image_objects.HorizontalStack(*objects*, *margin=0*)

Diese Klasse fasst mehrere Objekte zu einem zusammengesetzten Objekt zusammen, indem die einzelnen Objekte in der angegebenen Reihenfolge von links nach rechts horizontal nebeneinandergesetzt werden. Optional kann ein innerer Rand („margin“) angegeben werden, also ein Abstand, der um jedes Objekt herum freigehalten wird.

Vererbungshierarchie



Parameter des Konstruktors

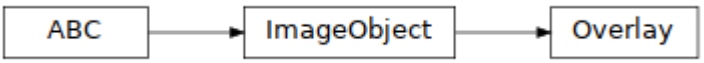
- **objects** *(List[image_objects.ImageObject])* – Eine Liste von Bildobjekten. Alle in Abschnitt 4.2 gelisteten Klassen sind zuslässig.
- **margin** (*int*) – Abstand um jedes Objekt herum, der zusätzlich freigehalten werden soll. (Optional – Standard ist 0).

[4] Für Experten/innen: Es sind nicht nur Bäume möglich, da ein Knoten prinzipiell auch mehrmals in andere eingesetzt werden kann. Eine zyklische Zusammenfassung muss aber vermieden werden (dies wird nicht überprüft, aber die Zeichenroutine fällt bei Zyklen in eine endlose Rekursion und das Programm stürzt ab). Allgemeine azyklische Graphen sind möglich; in der Praxis nutzt man aber sicher meist einfache Baumstrukturen (Schachtelungen in denen ein Objekt nicht mehrmals eingesetzt wird).

class image_objects.Overlay(*objects*)

Diese Klasse fasst mehrere Objekte zu einem zusammengesetzten Objekt zusammen, indem die einzelnen Objekte in der angegebenen Reihenfolge übereinander gezeichnet werden (ohne geometrische Verschiebung, lediglich malen in fester Reihenfolge). Zur Positionierung können Transformationsobjekte eingefügt werden (siehe nächster Unterabschnitt).

Vererbungshierarchie



Parameter des Konstruktors

- **objects** (*List[image_objects.ImageObject]*) – Eine Liste von Bildobjekten. Alle in Abschnitt 4.2 gelisteten Klassen sind zuslässig.

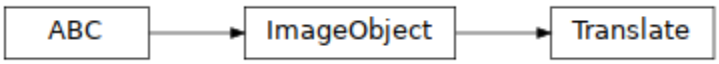
4.2.5 Transformationsobjekte

Die Aufgabe eines Transformationsobjektes ist es, die Lage des darin enthaltenen Objektes zu verändern, dieses also z.B. zu verschieben, drehen oder zu vergrößern/verkleinern („Skalierung“). Skalierung und Drehung arbeiten immer um den Mittelpunkt der Objekte (width()/2, height()/2).

class image_objects.Translate(*obj, offset_x, offset_y*)

Diese Klasse erzeugt Objekte, die ein anderes Bildobjekt enthalten und dieses um einen angegebenen Vektor verschieben („Translation“).

Vererbungshierarchie



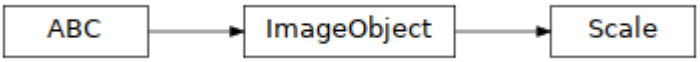
Parameter des Konstruktors

- **obj** *(image_objects.ImageObject)* – Ein Bildobjekt. Alle in Abschnitt 4.2 gelisteten Klassen sind zulässig.
- **offset_x** (*int*) – Verschiebung in x-Richtung.
- **offset_y** (*int*) – Verschiebung in y-Richtung.

class image_objects.Scale(*obj, scale*)

Diese Klasse erzeugt Objekte, die ein anderes Bildobjekt enthalten und dieses in der Größe verändern („Skalierung“).

Vererbungshierarchie



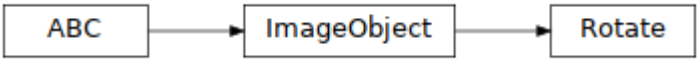
Parameter des Konstruktors

- **obj** (*image_objects.ImageObject*) – Ein Bildobjekt. Alle in Abschnitt 4.2 gelisteten Klassen sind zulässig.
- **scale** (*float*) – Skalierungsfaktor (reelle Zahl/Fließkommazahl; 2.0 entspricht z.B. einer Verdoppelung der Größe, 1.0 ändert gar nichts).

class image_objects.Rotate(*obj, rotation_angle*)

Diese Klasse erzeugt Objekte, die ein anderes Bildobjekt enthalten und dieses um einen angegebenen Winkel gegen den Uhrzeigersinn um den Mittelpunkt des Objektes drehen. Der Winkel wird in Grad (0..360°) angegeben.

Vererbungshierarchie



Parameter des Konstruktors

- **obj** (*image_objects.ImageObject*) – Ein Bildobjekt. Alle in Abschnitt 4.2 gelisteten Klassen sind zulässig.
- **rotation_angle** (*float*) – Der Rotationswinkel gegen den Uhrzeigersinn in Grad (360.0 = eine volle Drehung).

4.3 Das Modul `jgu_vc.colors`

Das `colors` Modul stellt einige symbolische Namen für Farben im RGB-Format (3-Tupel von ints im Bereich 0…255) bereit. Folgende Farben werden benannt:

`BLACK`, `WHITE`, `RED`, `ORANGE`. `YELLOW`, `GREEN`, `TEAL`, `SKY` (etwas helleres Blau), `BLUE`, `VIOLET`, `PINK`

Die Namen sollten selbsterklärend sein. Die Namen können mit den Prefixes „LIGHT“, „`SLIGHTLY_LIGHTER`“, „`DARK`“ und „`SLIGHTLY_DARKER`“ versehen werden, um die Helligkeit anzupassen. Z.B. ergibt „`DARK_BLUE`“ ein sehr dunkeles Blau, und „`SLIGHTLY_LIGHTER_PINK`“ einen helleren Rosaton.

Die Helligkeitsanpassung kann auch rechnerisch erfolgen: Hierzu gibt es zu jeder Farbe ein „`SHADES_OF`“ Objekt, also z.B. „`SHADES_OF_GREEN`“. Dabei handelt es sich um ein Array mit 7 einträgen; der Mittlere (Index 3) ist die Originalfarbe (das wäre im Beispiel „`SHADES_OF_GREEN[3]`“). Die höheren Indizes (4,5,6) werden immer heller und die unteren (2,1,0) immer dunkler. Bei Weiß und Schwarz sind jeweils nur die unteren bzw. oberen drei Einträge sinnvoll (es gibt kein dunkleres Schwarz; das bleibt einfach schwarz).

4.4 Das Modul `jgu_vc.basic_io_error`

Das Modul stellt die Ausnahmeklasse (Exception-Typ) BasicIOError bereit. Diese gehört zum öffentlichen Interface; Sie müssen diese aber in der Regel nicht explizit importieren.