AUSARBEITUNG SKRIPTSPRACHEN

Entwicklung des Spiels „BattleCastle“

Exposee

Ein Geschicklichkeitsspiel…
…in dem 2 Spieler…
…in einem vertikalen Labyrinth…
…gegeneinander kämpfen

Nils Liefländer, Alexander Schäfer
lieflaender.nils@fh-swf.de Schaefer.alexander2@fh-swf.de

Inhalt

1.0.0.0 Vorwort

Bei diesem Projekt handelt es sich um eine Ausarbeitung für das Fach Skriptsprachen (WS2019/2020) für die Fachhochschule Südwestfalen.

Die Aufgabe bestand darin in Teams zu zwei Personen ein Problem in der Skriptsprache Python zu lösen.

Auf Nachfrage ob die Ausarbeitung auch ein selbst entwickeltes Spiel sein darf, wurde dies bestätigt.

Deshalb wird in dieser Ausarbeitung verstärkt, wenn nicht sogar primär die Bibliothek „Pygame“ verwendet.

1.1.0.0 Vorstellung des Teams

Das Team für diese Ausarbeitung besteht aus:

Alexander Schäfer	Nils Liefländer
MatrikelNr: 10054274 FH-Email: schaefer.alexander2@fh-swf.de	MatrikelNr: 10054977 FH-Email: lieflaender.nils@fh-swf.de

2.0.0.0 Problem-Beschreibung/Idee:

Wir haben uns in dieser Ausarbeitung zur Aufgabe gemacht, ein grafisches Spiel mithilfe der Skriptsprache Python zu entwickeln.

Das Spiel beinhaltet eine Spiel-Oberfläche, aufgebaut als „vertikales Labyrinth“, in welcher 2 Charaktere von zwei Personen gesteuert werden können.

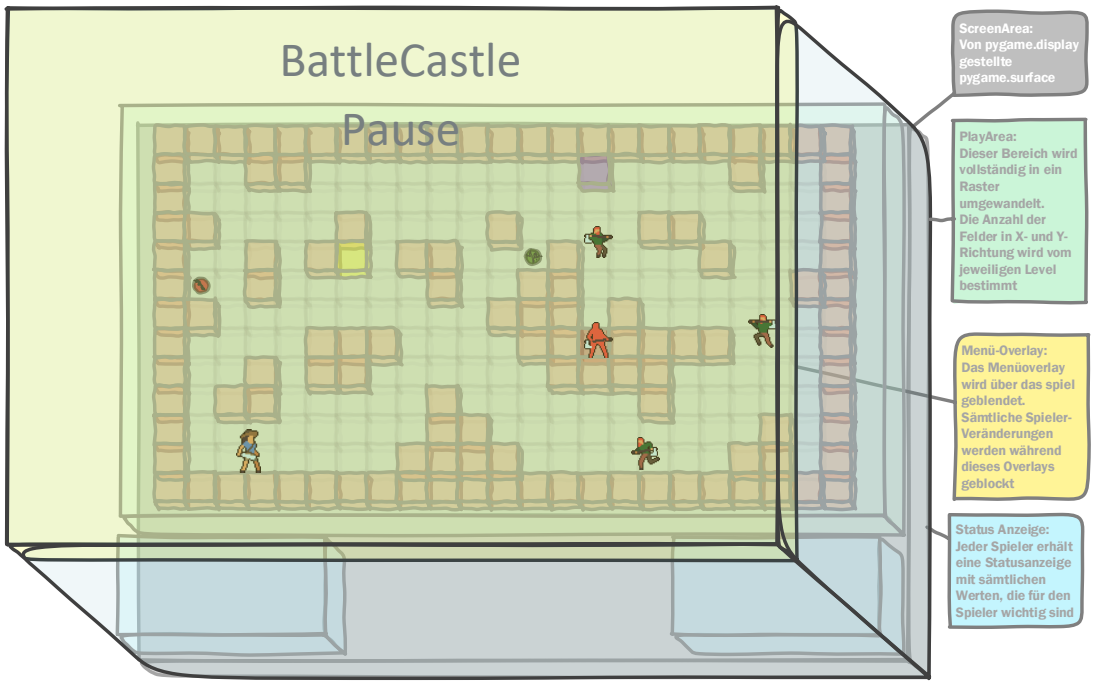

Abbildung 1:Spieloberfläche

2.1.0.0 Steuerung

Zur einfachen Benutzung des Spiels zu zweit, auch mit derselben Tastatur haben wir uns folgendes Tastenschema überlegt:

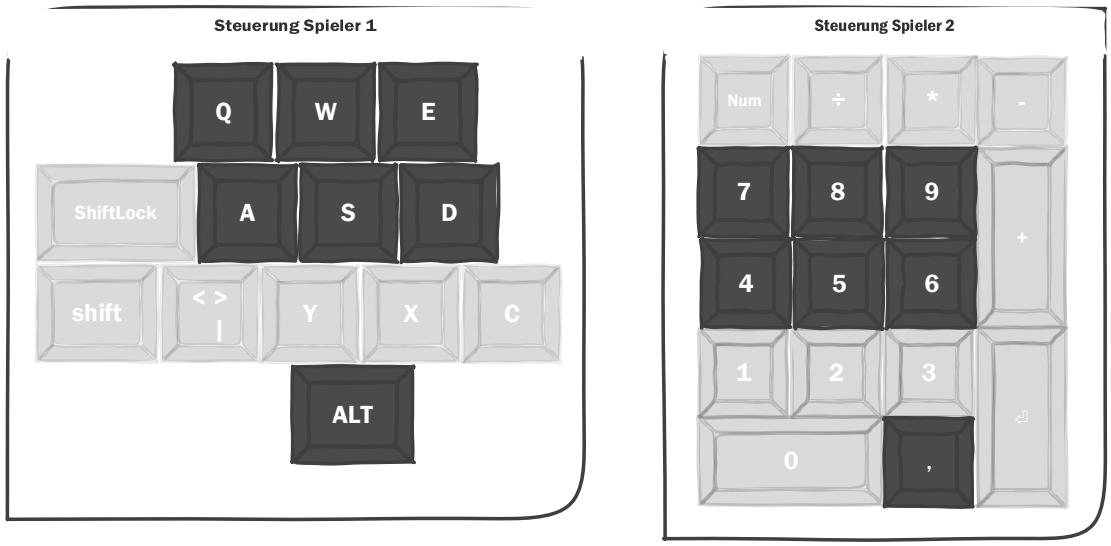

Abbildung 2: Steuerungs-Schema

3.0.0.0 Aufbau

Das Spiel ist in 4 Klassen implementiert:

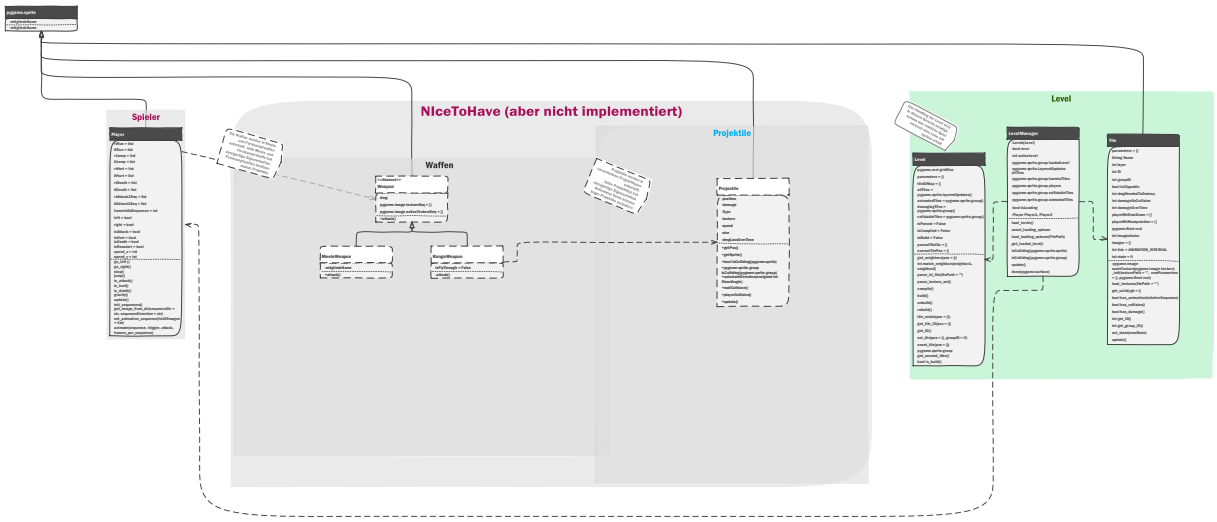

Abbildung 3: Klassendiagramm, zum Zoomen bitte separate PDF benutzen

3.1.0.0 Klasse Player (von Alexander Schäfer geplant und implementiert):

Wie schon oben beschrieben ist eine spielbare Figur im Groben eine Ansammlung an Bildern, die nach dem Auslösen eines Events dargestellt werden.
Also war es hier wichtig die Logik der Event zu beachten damit bei unterschiedlichen Events

unterschiedliche Bilderabfolgen, also Animationen, dargestellt werden.
Um eine einfache Erweiterbarkeit um neue Spielfiguren und die Erweiterung von alten zu ermöglichen wird die Funktion "glob" verwendet. Diese ermöglicht ein einfaches Laden von Dateien mithilfe eines regulären Ausdrucks.
Dadurch wird es möglich neue Spielfiguren zu laden indem man einfach einen neuen Ordner mit einer definierten Ordnerstruktur in den "char" Ordner (/bin/gfx/char) einfügt und diesen dem Programm in der "playerCFG.py" bekannt macht. Das Programm übernimmt den Rest.

Für die einfache Implementierung von Player, erbt die Player Klasse von der pygame.sprite.Sprite. Diese Klasse beinhaltet alles was man zum Implementieren eines dargestellten Objektes in Pygame braucht. Als erstes wurden die wichtigen Variablen für die Logik des Spielers implementiert:

Das ist ein vertikales 2D Spiel also hat der Player als erstes eine self.left- und self.right-Variable bekommen, damit es klar ist in welche Richtung er sich grade bewegt und mit welcher Geschwindigkeit auf welcher Achse. Aus diesem Grund sind Variablen self.speed_x und self.speed_y auch dazu gekommen.

Beim Initialisieren des Players wurden die, dem Konstruktor übergebenen, Parameter verwendet, um die richtige Spielfigur zu laden. Alle Bilder, die diese Spielfigur für die Animationen benötigt, werden in Listen mit Surface-Objekten gespeichert. Diese Surface-Objekte haben bereits die entsprechenden Bilder geladen. Diese Aufgabe übernimmt die Funktion init_sequences(). Die pygame.sprite.Sprite Klasse von der wir erben beinhaltet eine Funktion zum updaten. Diese wird mit der Logik für das Verhalten bei Events überschrieben. Wenn ein Event ausgelöst wird, zum Beispiel die Spielfigur bewegt sich nach links, wird die Liste mit den Bildern für diese Animation durchrotiert.

Welches Bild aus der Animationsrehe dargestellt wird, wird durch eine Ganzzahlige Division bestimmt:

(Counter // (Anzahl der Frames die eine Animation zur Durchführung hat // Anzahl an Bildern in der Liste mit der Animation))

Das Ergebnis dieser Berechnung ist dann die Stelle in der Liste, von der wir das Bild laden. Derselbe Vorgang bestimmt die Bilder für alle Animationsabfolgen.

Das Reagieren auf Events ist einfach gemacht, wenn eine bestimmte Taste gedrückt wird, wird eine boolean Variable in der Player Klasse auf "True" gesetzt. Es wird also eine Flag gesetzt. Diese wird in einer gestapelten "if-Abfrage" abgefangen. Dadurch werden die richtigen Animationsabfolgen abgespielt. Manche von diesen Flags haben eine größere Priorität, dass wird dadurch gekennzeichnet, dass Diese Flag andere Flags außer Kraft setzen kann. Zum Beispiel wenn der Flag "Death" gesetzt ist, wird die komplette Steuerung für die tote Spielfigur deaktiviert bis die Animation einmal durchgespielt wurde. Am Ende wird nur das letzte Bild aus der Animationsfolge angezeigt und das Spiel wird beendet mit einer Gratulation für den gewonnenen Spieler.

3.2.0.0 Klasse Level (von Nils Liefländer geplant und implementiert)

Die Level-Klasse wird genutzt um ein Level, basierend auf einer übergebenen Level-Datei (.lvl) einzulesen, für die Nutzung vorzuhalten und bei Bedarf aufzubauen.

Um einen möglichst reibungslosen und geordneten Ablauf bei der Implementierung des Levels zu ermöglichen haben wir den gesamten Vorgang in 2 einzelne Unterbereiche aufgeteilt, dem Parsing- und dem Build-Bereich. Diese beiden Phasen muss das Level durchlaufen, um vollständig spielbereit zu sein.

3.2.1.0 Parsing-Bereich:

Im Parsing-Bereich, wie der Name schon vermuten lässt, lesen wir die beiden Dateien in dem eigenen Level-Ordner ein. Dieser wird bei der Initialisierung der Level-Klasse als Argument übergeben.
Wir haben uns hier zunächst überlegt, wie man in Zukunft eigene Level einfach erstellen kann.
Folgende Punkte haben wir hierbei als wichtig erachtet:

- Es soll so einfach wie möglich sein ein neues Level zu erstellen oder ein bestehendes Level zu verändern.
- Es soll so einfach wie möglich sein die Texturen eines bestehenden Levels durch andere Texturen zu ersetzen
- Es sollen so viele Fehler wie möglich in diesen Leveln kompensiert werden können

Unsere Entscheidung fiel auf einen Datei-basierten Daten-Speicher.
Für das Finden der richtigen Daten und das Auslassen falscher Daten haben wir uns für die Verwendung von regulären Ausdrücken entschieden.
Da das Ermitteln der richtigen Ausdrücke nicht immer leicht ist, haben wir hierfür häufig das Online-Regex-Analysetool „https://regex101.com/“ verwendet.

Basierend auf der Entscheidung einen Datei-basierten Daten-Speicher zu verwenden haben wir für die Parameter der Level und dessen entsprechende Texturen einen eigenen Ordner erstellt (/bin/lvl/)
Die Implementierung des Parsing-Bereichs haben wir noch einmal in zwei Unter-Funktionen aufgeteilt, parseLvlFile und parseTextureSet. Diese Funktionen lesen die Daten der .lvl Dateien und der .conf Datei im "textur/tiles/"-Ordner im Levelverzeichnis aus.

3.2.1.1 Informationen zu .lvl-Dateien:

Hier wird das zum Level gehörende Grid* aufgebaut und zusätzliche Informationen wie Spieler-Start-Positionen* und Schwierigkeitsgrad gespeichert.
Der Datei-Name(ohne ".lvl") wird zudem als Level-Titel übernommen

3.2.1.1 Informationen zu .conf-Datein:

In der .conf-Datei werden für jede verfügbare Textur die Parameter gespeichert.
Jeder dort gespeicherte Parameter-Block* ist eindeutig mit einer ID durchnummeriert, der TileID.

Die TileID, wenn mit einem SequenzSuffix* und möglicher Bild-Endung angehängt, bildet den Dateinamen der zu tileID gehörenden Textur.

Die Parameterblöcke dürfen auf mehreren ".conf"-Dateien aufgeteilt sein, der Dateiname ist dabei egal.

Die beiden Funktionen "parse_lvl_file(filePath)" und "parse_texture_set()" dienen dem sicheren Auslesen und Speichern der in diesen Dateien enthaltenen Parameter.

3.2.2.0 Build-Bereich:

Wir haben bereits in der Planungs-Phase entschieden, dass das Laden des Levels in zwei Schritten stattfinden muss, um die Belastung der RAM-Ressourcen gering zu halten.

Im ersten Schritt, dem Parsing, (oben angegeben) wird das Level zunächst lediglich "vor-geladen", ohne tatsächlich eine Textur zu zeichnen. Hierdurch ist es in Zukunft möglich aus einem Menü eines der „vor-geladenen“ Level auszuwählen.
Zudem ist ein Wechsel des Levels einfacher, da das "alte" Level in den „vor-geladenen“ Zustand „zurückfallen“ kann und das "neue" Level nur noch die Texturen laden muss
Dieses Laden bzw. Entladen der Texturen wird in der "build()" bzw. "unbuild()" Funktion realisiert.

In der "build()" Funktion wird das gesamte Grid, welches wir zuvor mit "parse_lvl_file(filePath)" eingelesen haben durchlaufen.

An jeder Position im Grid wird die dort gespeicherte TileGroupID* ausgelesen.

Mit dieser TileGroupID* wird eine Unter-Gruppe aus den mit "parse_texture_set()" zuvor gefundenen Parameterblöcken gebildet

Mit dieser Untergruppe werden Tiles definiert, die zueinander "ähnlich" sind, jedoch eine andere Kombination an "freien Nachbarn*" haben. Zum Beispiel sind die texturen 001 - 036 teil der Gruppe 1 (sozusagen Gruppe "Fels")

Aus dieser Untergruppe muss nun zunächst das am besten passende Tile ermittelt werden.
Hierzu haben wir die Funktionen "get_neighbors(position={"X": 0, "Y": 0})" und "match_neighbors(neighbors1=[], neighbors2=[])" implementiert

Zum Hintergrund: jeder ParameterBlock, einer TexturID* beinhaltet einen Parameter genannt "preferred_neighbors", der eine 3x3-Liste aller Nachbarn und dem eigenen tile enthält.*

Mit get_neighbors(position) werden zunächst die Nachbarn des betrachteten Tiles betrachtet. Für den Fall, dass die hier betrachteten Tiles Rand-Tiles sind, haben wir uns hier dazu enschieden, eine Art "Endlos-Effekt*" einzubauen. (näheres zum „Endlos-Effekt“ im Glossar)

Die zurückgegebene 3x3-Liste wird mithilfe von match_neighbors(neighbors1, neighbors2) mit allen Elementen der im Grid jeweils betrachteten TileGruppe verglichen. Der ParameterBlock mit dem besten Ergebnis wird genutzt um ein Objekt der Klasse Tile zu erstellen.

Abschließend wird dieses frisch erstellte Tile den entsprechenden Spritegruppen* animatedTiles, damagingTiles, collidableTiles und allTiles hinzugefügt.

unbuild() ruft alle enthaltenen pygame.Sprites aus der Spritegruppe* allTiles ab und ruft dessen kill()-Funktion auf, welche die betroffenen sprites aus sämtlichen Gruppen entfernt. Danach wird das entsprechende Sprite "vergessen" bzw. überschrieben.

Eine verstärkte Nutzung von regulären Ausdrücken in Verbindung mit der Funktion re.findall(pattern, string) aus der Python-Bibliothek re erleichterte uns beim Einlesen der Daten die Arbeit sehr.

*Da weiterhin die Möglichkeit besteht, das einzelne Ausdrücke oder sogar ganze Dateien nicht gefunden werden, haben wir in ausgegliederten Python-skripts (/bin/config/) Default-Daten und Parameter definier. Durch diese von Uns sogenannten "*CFG.py"-Dateien ist im Falle eines Fehlers beim Einlesen der externen Dateien ein Rückfall auf Standard-Parameter möglich.*

Neben den Standard-Parametern werden hier auch für den jeweiligen Programm Abschnitt andere Parameter und Voreinstellungen gehalten.

3.3.0.0 Klasse Tile (von Nils Liefländer geplant und implementiert)

Die Klasse Tile beschreibt ein "Tile*" im Grid*. Es handelt sich hierbei um ein pygame.sprite erbendes Objekt.

Ein Tile-Objekt lädt bei intitialisierung die zur TileID zugeordneten Bilder in mehreren Animations-Sequenzen*. Es gibt drei Animations-Sequenzen*: „passive“, „active“ und „dying“ innerhalb der Dateien werden diese Sequenzen mit den suffixes "_", "-" und "#" auseinandergehalten. Eine zusätzlich angehängte ID gibt die Position in der Animations-Sequenz* an.

Die Bilder in den Animations-Sequenzen, werden mit tile.update() in einem in "generalCFG.py" definierten Zeit-Intervall rotiert.

3.4.0.0 Klasse BattleCastle (von Nils Liefländer und Alexander Schäfer geplant und implementiert)

In dieser Klasse werden alle anderen Klassen eingebunden und das Spielfeld wird aufgebaut, inklusive des aktuellen Levels, mit den dazugehörigen Tiles und Spielfiguren.

Diese Klasse wird wie ein .pygame.sprite behandelt und ist vorbereitet, um in einem externen pygame-Programm einfach als Sprite genutzt zu werden. Sämtliche Internen Aufrufe wie Player.update() werden über die BattleCastle-Klasse orchestriert

4.0.0.0 Durchführung

Angefangen haben wir mit einer gründliche Planungsphase, in der wir Ideen gesammelt haben. In dieser Zeit haben wir versucht die selbst gestellten Fragen zu beantworten:

- was soll das für ein Spiel werden,
- wie soll es aussehen,
- was ist das Ziel,
- welche Spielmechaniken,
- wie Waffen implementieren,
- wie Power-Ups implementieren

In dieser Zeit ist ein Klassendiagramm (Abbildung 2), ein Zeitplan nach dem wir vorgegangen sind und der Name des Spiels entstanden.

Es wurde beschlossen die Entwicklung wie folgt aufzuteilen:

- Nils Liefländer beschäftigt sich mit dem Labyrinth (Level)
- Alexander Schäfer beschäftigt sich mit den Spielfiguren.

Aufgrund der Komplexität der Level-Klasse mussten Wir eine mehrtägige Debugging-Phase ansetzen, welche bis auf das Verbleiben weniger, jedoch substantieller Probleme abgeschlossen werden konnte.

5.0.0.0 Probleme, ToDo für die Zukunft

- während build() iteriert der self.currentPos Zeiger im negativen Zahlenbereich
- -preferredNeighbors wird in parse_texture_set nicht korrekt ausgelesen und kann deshalb nicht verglichen werden
- -beim Auslesen der. lvl Datei werden bei den Grid-Zeilen unterschiedliche Längen erlaubt, kürzere Grid-Zeilen werden jedoch nicht mit leeren Tiles aufgefüllt.
- -Des Weiteren ist die .config Datei -für die Texturen noch unvollständig.

Diese Probleme verhindern leider die Lauffähigkeit des Spiels. In der zukünftig folgenden Version 0.1.2 (Außerhalb der Bewertung) werden diese Fehler noch behoben.

6.0.0.0 Ergebnis

Als Ergebnis der Ausarbeitung ist eine solide Basis, für die einfache Entwicklung und Erweiterung des Spiels mit weiteren Levels, Spielfiguren und Spiel-Mechaniken, entstanden.

7.0.0.0 Glossar:

- Grid
  Ist ein Kachelförmiger Aufbau der Spieloberfläche. Jede Kachel enthält einen ganzzahligen Wert der einer TileGroupID* entspricht.
- TileGroupID
  Einen Zusammensetzung von TileIDs die dasselbe Verhalten und ein ähnliche Textur haben z.B.: haben alle Fels-Teile TileGroupID = 1
- TileID
  Ist ein Eindeutiger Identifier eines Tiles innerhalb einer Gruppe, die Tiles unterscheiden sich in ihren Öffnungs-Kombinationen
- Tile
  Eine quadratische Textur mit unterschiedlichen offenen Enden aus denen das Level gebaut wird
- Spieler-Start-Position
  Ein X und Y Koordinate für die Position an der, eine Spielfigur zu Anfang des Spiels gezeichnet wird
- Sequenz_Suffix
  Ist in dem Namen jeden Teils enthalten und gestimmt dessen Animation ("_" => passive Animation "-" =>aktive Animation "#" => Zerstörungsanimation)
- Parameter Block
  Ist ein Block der von „{“ und „ }" umgeben ist und die Parameter einer TileID beschreiben. Beinhaltet auch die GroupID zu der die TileID gehört
- freie Nachbarn
  Nachbarfelder mit GroupID = 0
  Nachbarn wo keine Tiles Existieren
  =leeres Tile*
- „endlos-Effect“
  wenn man aus dem Level auf einer Seite rausgeht kommt man auf der gegenüberliegenden Seite wieder raus
- Spritegruppen
  Von pygame verwendete pygame.sprite.Group zum Gruppieren von pygame.sprite.Sprite Objekten (siehe: pygame Doku)
- leere Tiles
  Tiles ohne Textur
  = freier Nachbar*
- Animations Sequencen
  eine Reihe an geladenen Bildobjekten in einer Liste

8.0.0.0 Quellen

- https://www.instructables.com/id/Interface-Python-and-Arduino-with-pySerial/
  serielle Kommunikation mit Arduino
- https://www.pygame.org/docs/genindex.html
  Befehls-Referenz von Pygame
- https://craftpix.net/
  kostenfreie Texturen
- https://loading.io/
  kostenfreie Lade-Animationen
- https://regex101.com/
  Regex testen
- https://www.studieren-im-netz.org/im-studium/studieren/seminararbeit
  Ausarbeitung verfassen
-

8.1.0.0 Programm-Beispiele:

- http://programarcadegames.com/python_examples
  Verzeichnis mit Demos
- http://programarcadegames.com/python_examples/show_file.php?file=platform_jumper.py
  ProgrammBeispiel Kollision und Gravitation
- https://techwithtim.net/tutorials/game-development-with-python/pygame-tutorial/pygame-animation/
  ProgrammBeispiel Animationen
- https://www.bensound.com/royalty-free-music/rock Zugrifsdatum:04.03.2020
  Music: