**Programmdokumentation – Individualteil**

**Pacman in C++ und Raylib:**

1. **Constants**

Constants ist eine Header-Datei, welche feste Konstanten für den Spiel-, bzw. Programmablauf benötigt werden. So werden hier zum einen die Fenster-, als auch Spielfeld- und Kachelgröße als konstante integer festgelegt. Zum anderen wird hier ein Enum für mögliche Zellentypen erstellt; aktuell implementiert sind die Typen „WALL, EMPTY, COIN, FRUIT“. In unserem Fall ermöglicht diese zentralisierte Definition eine bessere Lesbarkeit und Verständlichkeit des Codes. Besonders im Rahmen von Funktionen wie der Kollisionsabfrage oder dem Sammeln von Punkten durch Münzen und Früchte, kann hier unkompliziert auf die Zellentypen zurückgegriffen werden.

**Verbesserungsvorschläge:**

Die Verwendung von festen Konstanten für Spielfeld- und Fenstergröße ist grundsätzlich schon eine sinnvolle Sache, da so unbeabsichtigte Verzerrungen oder Einschnitte vermieden werden. Jedoch würde eine Möglichkeit für automatische Skalierung an verschiedene Bildschirmgrößen und -formate sowie ein Vollbildmodus einen erweiterten Komfortaspekt darstellen; dies ist jedoch in der aktuellen Implementation aufgrund der Konstanten nicht möglich.

Die Verwendung eines Enums für die Zellentypen ist ebenfalls verbesserungswürdig, besonders wenn man den Programmcode um neue Typen und zugehörige Funktionen erweitern möchte. Eine erweiterte Klassenstruktur würde hier deutlich effektiver arbeiten:

So wäre das Programm objektorientierter aufgebaut und besser wartbar. Beispielsweise wäre eine abstrakte Basisklasse Cell, welche bereits Attribute wie x- und y-Position und einen Basiskonstruktor besitzt, denkbar. Sie könnte an zwei abgeleitete Klassen Statics und Collectibles vererben, welche wiederum je zwei Klassen ableiten: Wall und Empty könnten von Statics erben, während Coin und Fruit von Collectibles erben. Eine solche Klassenstruktur würde sowohl bei Erweiterung des Codes als auch in der aktuellen Version vorteilhaft sein.

Aktuell existiert beispielsweise eine Funktion spawnFruit(), welche sich im Grunde direkt auf Fruit bezieht, durch die Verwendung des Enums aber in Game.cpp und nicht in einer eigenen Fruit.cpp implementiert ist.

1. **Entity**

Entity ist die Basisklasse in unserem Programm; die beiden Klassen Player und Ghost, welche die Spielfiguren implementieren, werden public von ihr abgeleitet. Als Basisklasse ergibt sich die Möglichkeit, alle Spielfiguren in Datenstrukturen als Entities zu speichern, was im Spielverlauf einheitliche Behandlung in Bezug auf gemeinsame Attribute mit sich bringt.

In ihr definiert, sind die Attribute Position (x, y) und Bewegungsänderung (dx, dy). Diese werden von Ghost und Player verwendet und sind notwendig, um im Laufe des Spiels die Bewegungen auf dem Spielfeld zu berechnen. Diese Attribute ermöglichen eine einfache Wiederverwendbarkeit und Erweiterbarkeit des Programms: Neue Klassen mit denselben Attributen Position und Bewegungsänderung können somit durch Vererbung mit eben diesen ausgestattet werden, wie es bei Ghost und Player bereits der Fall ist.

Ebenfalls vorhanden sind ein Basiskonstruktor, welche den Standardkonstruktor ersetzt und ein weiterer Konstruktor, welcher ein Objekt mit Startposition und einer Startrichtung initiieren kann. Das ermöglicht eine flexible Initialisierung von verschiedenen Objekten dieser oder abgeleiteter Klassen.

Zusätzlich sorgt der virtuelle Destruktor virtual ~Entity() = default; für eine korrekte polymorphe Verwaltung von Objekten abgeleiteter Klassen.

**Verbesserungsvorschläge:**

Die Klasse Entity besitzt aktuell zwar einen virtuellen Destruktor, ermöglicht aber immer noch die direkte Initialisierung eines Objektes von Entity. Vorteilhafter und sinnvoller wäre hier eine Implementierung als abstrakte Klasse.

Außerdem hätte man auch noch eine virtuelle Funktion update() oder move() definieren können, welche von den abgeleiteten Klassen überschrieben wird. Im aktuellen Code besitzen sowohl Ghost als auch Player eine vergleichbare Klasse, werden jedoch erst dort definiert und nicht bereits von Entity geerbt.

Die Attribute von Entity sind zurzeit vollständig als public vorhanden. Eine Kapselung in private Attribute und public Getter-Funktionen zu ihrer Manipulation wäre ein weiterer Punkt, der hätte verbessert werden können.

1. **Player**

Player ist eine public von Entity abgeleitete Klasse. Sie erbt die Attribute Position und Bewegungsänderung, um im Programmverlauf von Funktionen auf dem Spielfeld lokalisiert zu werden.

Dazu kommen das private Attribut score, welches eine Integer-Variable für den Punktestand des Spielers darstellt, und eine Referenz board auf das Gameboard. Damit kann in der Player.cpp-Datei einfacher auf Attribute und Funktionen des Spielfeldes zugegriffen werden.

Public in der Header-Datei definiert sind zum einen der erweiterte Konstruktor, welcher ein Objekt der Klasse Player mitsamt eines Punktestandes (zu Beginn = 0) initialisiert, eine Getter-Funktion getScore() für den Zugriff auf den Punktestand von außerhalb der Player-Klasse sowie die zwei Funktionen movePacman() und checkCollision(). Sie übernehmen die Steuerung des Pacmans und überprüfen auf Kollision zwischen Geist & Pacman.

Zur Implementierung der Steuerung in movePacman() wurde eine Kombination aus Funktionen von Raylib und der Standardbibliothek von C++ verwendet. Mithilfe der Raylib-Funktion IsKeyDown() können konkrete Tastenanschläge einfach verarbeitet und in unserem Fall als Richtungseingabe der Pacmanbewegung genutzt werden.

Zur Bewegung der Figur wird die neue Position durch Veränderung des aktuellen x-, bzw. y-Wertes mit der Richtungseingabe des Spielers berechnet. Bevor Pacman jedoch auf die neue Position gesetzt wird, prüft die Funktion, ob die Funktion eine Wand ist oder gar nicht erst auf dem Spielfeld liegt. Trifft mindestens einer dieser Fälle zu, kehrt die Funktion zum Anfang zurück und wartet auf eine neue Eingabe; ist das Feld jedoch begehbar, so wird Pacman im Rahmen seiner eingestellten Geschwindigkeit, in diesem Fall über die Bildwiederholrate zwischen den Bildern der Bewegung, auf das neue Feld bewegt und kann dort gegebenenfalls Münzen oder Früchte einsammeln.

Die Abfrage, ob das neue Feld eine Münze oder eine Frucht beinhaltet, wird ebenfalls in movePacman() getätigt. Dabei wird verglichen, ob der Zellentyp des neuen Feldes einem der zwei Enum-Möglichkeiten (COIN, FRUIT) entspricht; ist dies der Fall, wird im Rahmen der Münze der Punktestand score um 10 und im Rahmen der Frucht sogar um 100 erhöht. Daraufhin verschwindet das jeweilige Item. War das Feld bereits leer, passiert nichts weiter.

In der Funktion checkCollision() wird unkompliziert geprüft, ob sich Pacman und ein Geist in dem aktuellen Feld treffen. Ist das der Fall, wird die Boolean-Variable als true zurückgegeben und führt im Rahmen der Spielelogik in der Datei Game.cpp zum Spielende. Treffen sie sich nicht, wird false zurückgegeben und die Spielschleife läuft weiter durch.

**Verbesserungsvorschläge:**

Während die Header-Datei in ihrem aktuellen Umfang schon sinnvoll gekapselt und strukturiert ist, besteht in der .cpp-Datei noch eine gewisse Unübersichtlichkeit. Die bereits in 2. Entity aufgeführte Verbesserung einer abstrakten Klasse mit einheitlicher Funktion move() hätte hier angewendet werden können.

MovePacman() besteht zurzeit aus mehreren kleinen Funktionen, die zur besseren Wartbarkeit und Lesbarkeit, hätten ausgelagert werden können. Beispielsweise das Input-Handling der Tastatursteuerung oder auch das Einsammeln von Münzen oder Früchten trägt nicht direkt zur Bewegung von Pacman bei und hätte entsprechend außerhalb einer solchen move()-Funktion implementiert werden können.

Ebenfalls aufgefallen ist mir, dass ich Player zwar dx und dy für die Bewegungsänderung vererbe, sie aber gar nicht nutze. Stattdessen habe ich intern neue Variablen inputDx und inputDy eingeführt, welche genau die gleiche Funktionalität erfüllen. Eine solche Doppelung sollte in Zukunft vermieden werden.

1. **Ghost**

Die Klasse Ghost erbt ebenfalls von Entity und kann genauso wie Player durch ihre Positionsdaten von anderen Funktionen auf dem Spielfeld angesteuert werden. Die private Attribute von Ghost umfassen weitere Positionen startX und startY, mit denen die Startposition gespeichert werden kann. Dies ist besonders im Hinblick auf etwaige Erweiterungen des Programms von Bedeutung. Beispielsweise eine Rückkehr zum Startpunkt nach Auslöschung durch Pacmans Power-Ups, wie sie in resetToStart() bereits implementiert, aber aufgrund fehlender Power-Ups auskommentiert ist, würde Gebrauch von ihnen machen können. Die Geister werden durch einfache Kreise in Raylib visualisiert und besitzen zur besseren Unterscheidung von Münze, Frucht, Wand und Pacman ein eigenes Attribut color, welches bei der Initialisierung mit dem Wert Purple (lila) gefüllt wird. Dazu kommen noch zwei weitere Attribute speed und moveCounter, welche sich um die Bewegungsgeschwindigkeit kümmern. Durch die private Definition bleiben die Daten der Member Variablen vor äußeren Manipulationen geschützt, was besonders bei der Bewegung von Vorteil ist.

Public definiert sind in Ghost nur der Konstruktor, welcher die Geister mit Farbe initialisiert, und die Funktion update(), welche den gesamten Bewegungsalgorithmus der Geister steuert.

Damit die Geister nicht ziellos durch das Feld geistern, habe ich mich hier besonders für einen Greedy-Algorithmus entschieden: Der Algorithmus sucht also bei jeder Bewegung aufs Neue den aktuell schnellsten Weg der Geister zu Pacman und wählt den nächsten Schritt auf Basis der aktuellen Distanz. Vorausberechnungen möglicher nächster Bewegungen finden hier somit nicht statt. Eine Rückwärtsbewegung ist grundsätzlich ausgeschlossen, es sei denn es gäbe keine andere Möglichkeit. So wird das Risiko für „gefangene“ Geister, die auf einer Strecke hin- und herpendeln vermieden. Ist nur eine Rückwärtsbewegung möglich, weil auf allen anderen drei Seiten eine Mauer ist, so wird diese ausgeführt; so bleibt ein Geist auch in einer solchen Situation nicht stecken.

**Verbesserungsvorschläge:**

Auch für die Ghost-Klasse gelten ähnliche Verbesserungsvorschläge, wie für Player: Anstelle der einzelnen update()-Funktion wären hier mehrere Hilfsfunktionen, welche aktuell in dem gesamten Funktionskörper implementiert sind, angebracht. Beispielsweise könnten die Berechnung der nächsten Richtung und die Überprüfung dieser auf ihre Zulässigkeit ausgelagert werden, sodass jede Funktion ihre einzelne konkrete Aufgabe besitzt. Das würde zu einer verbesserten Lesbarkeit und Wartbarkeit führen und zusätzlich dem Single-Responsibility-Principle entsprechen.

Ebenfalls verbessern, könnte man die Daten Kapselung. Die Daten sind zwar schon in private und public gekapselt, jedoch fehlt es an Getter-Funktionen für die private Attribute. Zum jetzigen Stand stellt das kein weiteres Problem dar, könnte aber bei Code-Erweiterungen zu Zugriffsproblemen führen.

1. **Game**

Game ist von zentraler Bedeutung des Programms. Hier werden alle wichtigen Spielzustände verwaltet und die grundlegende Spielelogik festgelegt. Beginnend in der Header-Datei sind alle Spielzustände, bzw. Menüs in Form eines GameState-Enums festgelegt. Die Zustände werden zur klaren Benutzung der switch-case Anweisung verwendet und ermöglichen dort eine verständliche Ordnung der Programmabläufe. Zentrale Inhalte der Klasse Game sind in private und public gekapselt. Unter den private Attributen lassen sich Referenzen auf alle verwendeten Klassen und Objekte finden. Dadurch wird der Zugriff auf bestimmte Funktionen und Attribute der entsprechenden Objekte deutlich vereinfacht. Ebenfalls werden hier drei Game-interne Funktionen getPlayerName(), spawnFruit(), waitForKeyRelease(), definiert. Sie werden außerhalb von Game nicht verwendet und sollen auch nicht von außen manipuliert werden können. Zudem sind noch drei wesentliche Variablen definiert, welche im Laufe des Programms gebraucht werden; playerName speichert den Namen, den ein Spieler zu Beginn eingibt. Er wird beispielsweise für das Leaderboard wieder benötigt. GameState speichert den aktuellen Zustand des Spiels und ist für eine reibungslose Funktionsweise des Programmablaufs notwendig, wohingegen gameOver den Spieldurchgang stoppt, sofern sie den Wert true annimmt.

Public definiert sind die Funktionen, die von anderen Programmteilen aufgerufen werden. Zudem ist der Game()-Konstruktor public: Er initialisiert alle benötigten Objekte. Dazu gehören unter anderem das Spielfeld (board), den Spieler (pacman), die Geister (ghosts), und auch die Bestenliste mit den Punkteständen der besten Spieler (leaderboard). GameOver wird bei der Initialisierung zudem auf false gesetzt, damit der Spieldurchlauf starten kann. Als Start-Zustand initialisiert der Konstruktor das Start-Menü. Die Spielfeldgeneration wird aufgerufen und die Geister werden an ihre Startpositionen gesetzt. Die Münze auf Pacman’s Startpunkt wird vor Beginn entfernt. Für die drei private Variablen gibt es in public wieder entsprechende Getter-Funktionen. Run() ist als zentrale Funktion für den Spielablauf zuständig und bearbeitet die Menüführung und das Spielgeschehen.

Beim Programmstart öffnet Game das Spielfenster und ruft die Renderer-Klasse auf, um die Texturen für Pacman zu laden. Solange das Spielfenster geöffnet ist, läuft eine kontinuierliche Schleife durch. Dadurch ist es möglich auch mehrere Runden hintereinander zu spielen, ohne das gesamte Programm neu zu starten. Durch die Initialisierung im Konstruktor beginnt das Programm im STATE\_START\_MENU: Eine Anzeige dafür wird von Renderer bereitgestellt. Mehrere mögliche Benutzereingaben werden angezeigt und je nach Input wechselt das Programm in einen neuen Zustand. Im Falle von „ENTER“ wechselt das Programm in STATE\_ENTER\_NAME: Es wird die Funktion getPlayerName() aufgerufen und der Benutzer wird dazu aufgefordert einen Spielernamen einzugeben. Die Funktion gibt am Ende den Boolean true aus, wenn mindestens ein Zeichen als Name eingegeben und die Eingabe bestätigt wurde. Der eingegebene Name wird im string playerName gespeichert und wird für Punktzahl und Leaderboard gebraucht. Ist die Eingabe ebenfalls mit „ENTER“ bestätigt, wechselt das Programm in STATE\_PLAYING: Hier findet die Spielelogik statt. Sofern nicht alle Münzen eingesammelt sind oder gameOver()eingetreten ist, werden die Funktionen der Bewegungen (movePacman() und update()) sowie die Darstellung des Spiels über drawGame() und die Hilfsfunktion spawnFruit() fortlaufend aufgerufen. Während des Spielens kann über Tastendruck von „P“ das Pausenmenü aufgerufen werden. Von hier kommt man entweder über „P“ zurück zum Spiel, über „Q“ ohne Speichern des Spielstandes ins Hauptmenü oder über „ESC“ zum Schließen des Programms. Bevor man aus dem Pausenmenü ins Hauptmenü zurückkehrt, wird zudem eine Warnung ausgegeben. Ist das Spiel zu Ende, wird gegebenenfalls das Leaderboard angepasst und das GameOver-Bild gezeichnet. Aus dem Hauptmenü heraus kann man sich das Leaderboard zudem anzeigen lassen. Beim gesamten Programmende werden alle Ressourcen wieder freigegeben, damit keine Speicherlecks entstehen, und das Fenster wieder geschlossen.

Die Funktion spawnFruit() übernimmt die Früchte-Erzeugung alle 10 Sekunden auf dem Spielfeld, sofern nicht bereits eine Frucht existiert. Dafür wird fortlaufend ein Vektor mit allen leeren Feldern (abgesehen von Pacman’s Position) gefüllt und zufällig eines dieser Felder ausgewählt, um dort die neue Frucht zu platzieren.

**Verbesserungsvorschläge:**

Im Grundprinzip ist Game bereits gut strukturiert und einfach zu handhaben, jedoch könnte man im Rahmen der GameStates eigene Hilfsfunktionen verwenden, welche den Ablauf der jeweiligen Zustände steuern, anstatt wie jetzt diese innerhalb der GameStates selbst zu implementieren.

Eine erweiterte Klassenstruktur wäre hier ebenfalls denkbar um das Programm noch objektorientierter und modularer zu gestalten, wenngleich die Nutzung eines Enums für die GameStates bereits sehr flexibel ist, was neue Spielzustände betrifft.