LunarLander

Felix von Harrach, 2022

Inhaltsverzeichnis

1 | Themenstellung

Ziel ist eine Nachstellung des Arcade-Spiels **“Lunar Lander”** (1979) mit einigen kreativen Freiheiten.

Darunter zB. die dynamische Verringerung des Gewichts des Landers, je mehr Treibstoff verbraucht wird und ein Player-System, in dem Spielsessions einem Spieler zugewiesen und gespeichert werden.

Ein Spieler kann Highscores erreichen und diese auch brechen.

Die Terraingeneration soll **zufällig, prozedural und seedabhängig** sein, um Tuple, bei Bedarf **wiederholbare** Terrain-Generation zu erlauben.

Wird der Lander erfolgreich **gelandet**, d.h. er landet senkrecht, mit einer Senkrechtgeschwindigkeit von weniger als 10 und einer Horizontalgeschwindigkeit von weniger als 3 auf einer Landeplattform, bekommt er **je nach Plattformgröße** Treibstoff und Punkte (je kleiner, desto mehr!).

Stürzt er ab, bekommt er weniger Punkte.

Verlässt der Lander das Spielfeld und bleibt zu lange außerhalb des Spielfelds, wird dies als **Mission-Fehlschlag** angesehen; es passiert dasselbe wie bei einem Crash.

Der Lander soll entsprechend mit Sprites **animiert** werden.

Falls der Lander in der nächsten “Mission” ohne Treibstoff starten würde, wird das Spiel stattdessen **beendet** und der Score usw. **gespeichert**.

Das “Software-Paket” sollte **erweiterbar** und **modular** sein, um das Bearbeiten und Hinzufügen von Programmabläufen zu vereinfachen.

Zudem soll das Projekt gewissermaßen als **Proof of Concept** fungieren, dass **Visual Studio Code** sich im Schulunterricht tatsächlich verwenden lässt.

Soll plattformunabhängig sein: Solange Jython in irgendeiner Form installiert ist, läuft‘s. Ich arbeite mit der Bibliothek jGameGrid.

Soll ein Hauptmenü haben, von dem aus einige Spieleinstellungen eingestellt werden können.

2 | Versionierung

Im folgenden erläutere ich meine Versionierung, wie ich sie auch in meiner Github-Repository (*https://github.com/flexxyfluxx/LunarLander, unter* Releases) vorgenommen habe:

2.1 | Version 0.0.0

In dieser Version habe ich erstmal nur den Lander implementiert, da dieser ein zentraler, wenn nicht der zentralste, Aspekt des Spieles ist.

Anmerkung: Ich verwende über das ganze Projekt eine Tickrate von 100tps, da damit die Rechnungen am einfachsten sind und 10/20/50tps evtl. einfach nicht schön aussehen. Da ich die Geschwindigkeiten und auch die Schwerkraft in m/s speichere, muss ich also bei jeder Beschleunigung oder Bewegung erstmal durch 100 teilen.

Zuerst habe ich die Momentangeschwindigkeit des Landers auf ihre x- und y-Vektoren aufgeteilt, da ich wusste, dass ich das sowieso irgendwann tun müsste, wenn ich wirklich omnidrektionale Bewegung ermöglichen wollte.

Dann habe ich Schwerkraft implementiert, indem ich den Lander jeden Tick ein bisschen nach unten beschleunige., bzw. jeden Tick ein Hundertstel Mondschwerkraft von der y-Geschwindigkeit abziehe (negative y-Geschwindigkeiten entsprechen bei mir einer Beschleunigung nach unten).

Danach habe ich eine einfache Steuerung des Landers implementiert: Der Schub lässt sich mit W und S hoch-/runterregeln und der Lander lässt sich mit A und D drehen. Die „absolute“ Beschleunigung lässt sich dann anhand der Formel F=m*a, umgeformt zu a=F/m, berechnen: Die Kraft ist hierbei eine Konstante, die ich mit dem momentan eingestellten Schub, der zwischen 0 und 1000 liegt, multipliziere; die Masse ist gleich der momentanen Treibstoffmasse plus die „Trockenmasse“ des Landers, wobei der Treibstoff in dieser Version noch nicht verbraucht wird.

Die Keypress-Logik hierfür habe ich größtenteils aus meinem ooPong-Projekt übernommen: Ich frage jeden Tick ab, ob relevante Tasten gedrückt werden; wird eine Taste gedrückt, wird der gegenteilige Tastendruck priorisiert, d.h. wenn man W drückt und hält und dann S drückt und hält, wird der S-Keypress priorisiert. Das funktioniert auch andersherum; wenn man S drückt und dann W drückt, wird W priorisiert. Das System habe ich natürlich auch auf die Lander-Drehung angewendet.

Ich benutze kein onKeyPressed/onKeyRepeated oÄ, um konsistentes Keypress-Verhalten zu erlauben.

Wird Leertaste gedrückt, wird der Schub zudem auf den maximalen Wert (1000) gestellt. Dies ändert sich in 0.0.1: Ab 0.0.1 muss man hierfür E drücken; man kann zudem Q drücken, um den Schub abzuschalten.

Um die absolute Beschleunigung dann mit den momentanen Vektorgeschwindigkeiten zu verrechnen, muss die Beschleunigung für die y-Achse erst mit dem Sinus und für die x-Achse mit dem Cosinus der Lander-Ausrichtung verrechnet werden. Das funktioniert, da man sich die Vektorbeschleunigung des Landers auch als rechtwinkliges Dreieck vorstellen kann, wobei die absolute Beschleunigung die Hypothenuse ist und die y-/x-Vektoren davon die Katheten sind. Multipliziert man die Hypothenuse also mit dem Sinus, bzw. Cosinus, erhält man die Katheten.

Ein weiteres Problem, das sich hierbei ergab, war die Limitation der eingebauten gg.Location-Klasse, die nur ganzzahlige Werte speichern kann, wodurch bei der Landerbewegung Rundungsfehler entstehen, die die Steuerung des Landers stark beeinflussen. Ich habe deshalb eine eigene Floacation (Float Location)-Klasse geschrieben, die Floats speichert, aber bei Bedarf auch Ints zurückgeben kann. Ich speichere und verändere also die „wahre“ Position des Landers als Float und aktualisiere jeden Tick die sichtbare Position des Landers auf dessen ganzzahlig gerundete Position.

2.2 | Version 0.1.0

In dieser Version habe ich Terrain-Generation und eine rudimentäre, Proof-of-Concept-mäßige Kollision implementiert, sowie eine LunarGame-Hauptklasse, die ab jetzt sämtliche spielbezogene Programmabläufe steuert.

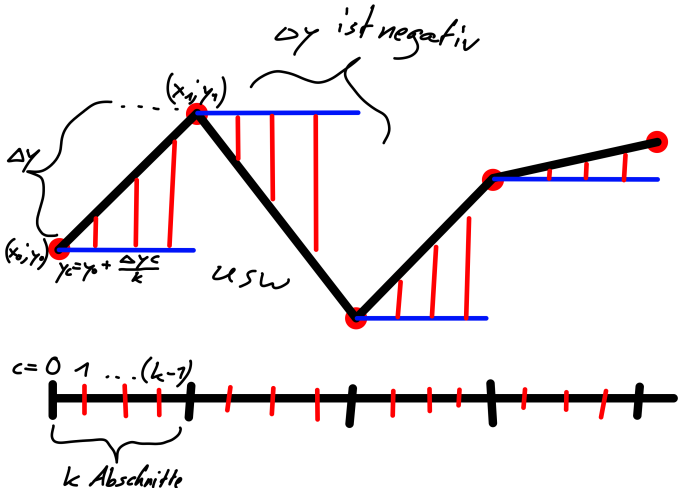
Zur Terrain-Generation: Um die prozedurale, seedabhängige Terraingeneration zu erreichen, die ich vor Augen hatte, habe ich zunächst eine Klasse Terrain erstellt, die einen optionalen Parameter Seed nimmt, den er an ein Zufallsgenerator-Objekt der Art random.Random übergibt (Wird kein Seed gegeben, wird eine Zufallszahl als Seed generiert).

Dann generiere ich eine Liste an zufälligen Zahlen, die die Terrainhöhe in regelmäßigen Abständen darstellt. Diese Liste ist natürlich in ihrer Rohform ein wenig… zerklüftet. Um sie auszuglätten, setze ich also jeden Wert in der Liste gleich dem Durchschnitt des Werts selbst, des vorgehenden und des nachgehenden Werts. Damit wird die Varianz bei wiederholtem Durchlaufen immer weiter gesenkt (in der Endversion mache ich insgesamt 13 Durchgänge).

Dieses Konstrukt kann man dann auf ein Gamegrid übertragen, indem man sich die Werte als Punkte vorstellt, deren x-Koordinaten der Index des Wertes, multipliziert mit einer Konstante, ist und diese Punkte verbindet. Damit am rechten Rand keine Lücke entsteht, braucht die Werteliste einen Wert mehr als eigentlich vermeintlich nötig, damit nicht passiert, dass die Funktion am Ende der Liste keine Werte mehr hat und kurz vorm Rand aufhört. Dies setze ich aber erst in Version 0.2.0 um.

Zur Kollision: Jeden Tick prüft das LunarGame-Objekt, ob sich der Lander komplett über dem Terrain befindet. Dazu ist eine vollständige Liste an Terrainhöhen an jeder Stelle nötig.

Hierzu verwende ich lineare Interpolation, um zwischen den geg. Punkten Werte einzufüllen:



Ich addiere zum linken „Randwert“ y_0 jeweils einen Offset von Δy * c / k, wobei Δy = y1 - y0 für jedes Intervall der Länge k, und c dem „momentanen“ Subintervall entspricht. Ist Δy negativ, wird der entstehende Wert y_c mit zunehmendem c immer kleiner.

Damit habe ich an jeder ganzzahligen x-Stelle eine entsprechende Terrainhöhe, dessen Index die x-Stelle ist, und kann sehr einfach prüfen, ob der Lander über dem Terrain ist.

Hier vereinheitliche ich zudem die verwendeten Konstanten in der Config-Klasse, sodass diese aus einer Datei settings.ini gelesen werden, um Magic Numbers (namenlose Fixzahlen im Programm) zu vermeiden und somit den Code verständlicher zu gestalten.

2.3 | Version 0.2.0

In dieser Version habe ich ein HUD implementiert, das veränderliche Werte wie Treibstoffstand, vergangene Zeit, Vektorgeschwindigkeiten usw. zeigt. Dazu habe ich eine LunarGameHUD-Klasse implementiert, die einige GGTextFields zusammenfasst. Ursprünglich erbte sie von gg.Actor, damit sie sich selbstständig jeden Tick aktualisiert, aber ich empfand dies später als unnötig, da ich das HUD auch in LunarGame.act aktualisieren kann.

Des weiteren habe ich die Lander-Terrarin-Kollision verfeinert, indem nicht nur von der Mitte des Landers aus nach Kollision gesucht wird, sondern auch am linken und rechten Rand, damit auch bei steileren Hügeln genaue Kollision möglich ist.

Ich entferne mich in der LunarGame-Klasse zudem wieder vom „return self“-Idiom, zu dem ich zwischenzeitlich übergegangen war, das einem erlaubt, Methodenaufrufe aneinanderzuhängen und gehe wieder dazu zurück, die Methoden einzeln aufzurufen, da das einfach übersichtlicher ist.

Zuletzt wird das Terrain nun genauer gezeichnet, um die tatsächliche Höhe des Terrains besser zu repräsentieren: Ich ziehe nicht mehr einen einzelnen fetten Strich von Punkt zu Punkt, sondern zeichne jeweils fünf einpixelbreite Striche übereinander von Punkt zu Punkt, wobei der oberste Strich auf der tatsächlichen Höhe liegt und die darunter um jw. einen Pixel versetzt werden.

2.4 | Version 0.3.0

In dieser Version existieren nun Plattformen, die ins Terrain „hineingeneriert“ werden, auf denen der Lander landern kann. Hierzu wird bei einer Kollision einfach geprüft, ob die Voraussetzungen für eine Landung (Lander ist vollständig auf der Plattform; Lander steht ausreichend senkrecht; Lander ist langsam genug) gegeben sind und führt dann je nachdem eine Lande-, bzw. eine Absturzfunktion aus.

Um die Landeplattformen in die Heightmap des Terrains zu integrieren war nicht viel Arbeit nötig; ich addiere einfach Null zur letzten gegebenen Höhe und iteriere erst nach Ende der Plattform weiter durch das Terrain.

Zu den Landeplattformen: Ich generiere eine Liste an Tupeln, wo jeder Tupel die Position und die Länge einer Landeplattform enthält. Mit gegebenem Abstand zwischen den Terrainpunkten (s.o.) ist es also möglich, einfach eine Liste mit x-Stellen zu füllen, an denen eine Landeplattform vorhanden ist. Ich kann dann schauen, ob die ganzzahlig gerundete x-Position des Landers in der Liste ist, um zu schauen, ob der Lander über einer Plattform ist.

Ich füge außerdem einen Check hinzu, ob der Lander außerhalb des Bildschirms ist. Ist er lang genug außerhalb des Bildschirms, wird ein Mission-Fehlschlag ausgeführt, der äquivalent zu einem Absturz ist. (Der Lander hat den Operationsbereich verlassen.)

Dem Sprite können im Constructor-Aufruf nun auch mehrere Sprites übergeben werden; damit ist die Grundlage für die späteren Animationen gelegt.

2.5 | Version 0.4.0

In dieser Version implementiere ich die Grundlagen für ein Spielersystem. Ich kann Spieler-Objekte erstellen, die in einem globalen Dictionary gespeichert werden. Diese Dictionary wird dann als „Pickle“ (players.pkl) gespeichert, was bedeutet, dass ich bei Beginn des Programms das Dict wieder laden kann und die rohen Spielerobjekte darin bereits vorhanden sind.

Dazu kommen noch einige Verschönerungen, zB. wird das Land/Crash-System ausgebaut, sodass bei erfolgreicher Landung/Absturz tatsächlich was passiert: Bei Absturz bekommt der Spieler 15 Punkte; bei erfolgreicher Landung bekommt er je nach Größe der Plattform Punkte und Treibstoff. Das Terrain wird nun auch vollständig ausgemalt, indem ich von jedem Punkt in der Heightmap einen Strich nach unten ziehe. Das HUD wird nochmal ein wenig verschönert und ich schaffe es endlich, die Raketengrundgleichung korrekt zu implementieren. Bisher hatte ich versucht, rechnerisch innerhalb einer Hundertstelsekunde eine Sekunde an Treibstoff auszustoßen, was zu erheblichen Problemen führte.

2.6 | Version 1.0.0

Diese Version ist die erste, die einen vollständigen Spielzyklus enthält. Das Spiel fragt einen zu Beginn nach einem Namen. Dann spielt man, bis einem der Treibstoff ausgeht; das Spielerobjekt wird in der o.g. Dictionary gespeichert und in die Pickle geschrieben. Das Spielergebnis wird als neue Zeile in eine Datei history.txt geschrieben, in der die Startzeit des Spieles, der Seed, das Smoothing, der Spielername und die erreichten Punkte stehen.

Des weiteren hat der Lander nun ein volles Sprite-Set: Standard-Sprite, jw. 2 Sprites für hohe und niedrige Thrustwerte und 10 Absturz-Sprites. Bei der Absturzanimation wird ein Iterator

120 Ticks lang hochgezählt, während denen die Sprite-ID auf iterator // 10 gesetzt wird, bis der Iterator 109 erreicht. Danach wird der Lander ausgeblendet.

Zu guter Letzt noch der Endscreen! Der Punktestand und Spielername werden noch einmal übersichtlich präsentiert~~, um optimal angeben zu können~~.

2.7 | Version 1.1.0

In dieser Version füge ich ein Hauptmenü hinzu, über das sich, wie bei meinem ooPong-Projekt, einige Einstellungen einstellen lassen. Man kann zudem einen Seed und einen Spielernamen angeben. Wird kein Seed gegeben, wird ein zufälliger Seed gewählt.

Dazu kommt noch die Behebung einiger Schönheitsfehler in der Death Animation, durch die der Lander nicht korrekt ausgeblendet wurde, sondern noch eine Sekunde auf der letzten Sprite verharrte.