Entwicklung einer betriebssystemunabhängigen 2D Game Engine

Driton Gjuraj
InfB - 6
Matrikelnummer: 00097085

***Abstract*—Alle 2D Computerspiele besitzen bis zu einem gewissen Punkt die gleiche Codebasis. Diese ist für die grundlegenden Funktionen wie Grafik und Soundausgabe, Berechnung der Physiksimulation und Verwaltung der menschlichen Eingabe verantwortlich. Der dafür verantwortliche Programteil wird auch Game Engine genannt. Wird diese Gameengine von Anfang an betriebsystemunabhängig entwickelt, so können auch die Spiele mit deutlich geringerem Aufwand betriebsystemunabhängig gemacht werden, da die meisten Spielfunktionen, auf die Basisfunktionen der Game Engine zurückgeführt werden können. Die grundlegenden Komponenten einer Game Engine, also die Grafik, Sound, Physik und Eingabe werden dazu von Anfang an mit betriebssystemunabhängigen Programmiersprachen und Bibliotheken geplant und entwickelt. Die Game Engine soll unter Windows, Linux und MacOS voll funktionfähig sein**

I. Einleitung

Jedes 2D Computerspiel benötigt den vom Aufbau her gleichen Basis Code. Es müssen Bilder auf dem Bildschirm angezeigt werden, Töne abgespielt werden, die Physik simuliert und die Eingabe des Spielers erkannt werden. Dieser Programteil kann in einer vom konkreten Spiel unabhängigen Game Engine entwickelt werden, da es sich um einige Basisfunktionen handelt auf die jedes Spiel zurückgreift. Sobald man eine Game Engine hat, können sich die Entwickler des Spiels voll und ganz auf die Logik des Spiels konzentrieren und müssen sich nicht um die Umgebung kümmern. Die Game Engine bietet also eine gewisse Abstraktionsebene zum Betriebssystem und vereinfacht somit die Entwicklung deutlich, da die gesamte Kommunikation mit dem Betriebsystem von der Game Engine übernommen wird. Heutzutage ist es üblich, dass Spiele für mehrere Platformen zur Verfügung gestellt werden, z.B. Windows, MacOS, Linux und die verschiedenen Konsolen wie die PS4, XBox usw. Da jede Platform ihr eigenes Betriebssystem hat, bzw. ist, läuft auch die Kommunikation mit dem Betriebsystem nach anderen Richtlinien und es gibt unterschiedliche APIs. Für unsere Game Engine bedeutet das, dass man den Code nicht einfach auf eine neue Plattform übertragen kann und direkt kompilieren kann. Da die Spiele größtenteils über die Game Engine mit dem Betriebsystem kommunizieren, würde eine betriebssystemunabhängige Game Engine die Spiele automatisch auch betriebssystemunabhängig machen. Der mehraufwand in der Entwicklung dafür, wird mit einer deutlich erhöhten Verbreitungsmöglichkeit belohnt, da die Spiele auf mehrern Plattformen laufen. In diesem Paper sollen die Grundlagen die für die Entwicklung einer betriebssystemunabhängigen 2D Game Engine nötig sind genauer erläutert werden ohne in die Ebene der konkreten Implementierung zu gehen.

II. Betriebssystemunabhängigkeit im Detail

In diesem Paper wird unter einer betriebssystemunabhängigen Game Engine, eine Game Engine verstanden die auf den verbreitetsten Computer Betriebssystemen Windows, Linux und MacOS lauffähig ist. Spielekonsolen werden nicht explizit behandelt, jedoch gelten auch dort ähnliche Prinzipien. Unter Umständen müssen andere Bibliotheken verwendet werden.

Eng verwoben mit der Frage der Betriebssystemunabhängigkeit ist auch die der Platformunabhängigkeit. Da Linux ein Betriebssystem ist, welches auch auf nicht x86/x64 Architekturen läuft, muss man im Vorfeld entscheiden auf welchen Prozessorarchitekturen man die Game Engine verwenden will. Das liegt unter anderem daran, dass die verschiedenen Plattformen unterschiedliche Endians verwenden, d.h. die Reihenfolge der Bytes, die ein und die selbe Zahl repräsentieren ist unterschiedlich. Für viele alltägliche Programmieraufgaben mag das unwichtig erscheinen, da immer nur die gesamte Zahl betrachtet wird, bei einer Game Engine sieht es jedoch anders aus. Laden wir beispielsweise ein Bild im RGBA Bitmap Format in den Ram, so besitzt jedes Pixel genau einen 32 Bit Integer mit 4 Bytes von denen jeweils einer für Rot, Grün, Blau und Alpha steht. Möchten wir dieses Format jetzt in BGRA umwandeln, muss auf das jeweils richtige Farbbyte zugegriffen werden, um es umzuspeichern. Je nachdem ob wir uns auf Little Endian oder Big Endian befinden, greifen wir für Rot entweder auf das erste oder letzte Byte zu. Entscheidet man sich also für die Unterstützung mehrerer Architekturen, so muss vor jedem Bytezugriff je nach Big oder Little Endian anders vorgegangen werden.

Nicht nur in der Hardware gibt es einiges zu beachten, sondern auch in der Software. Nicht jede Bibliothek existiert für alle gewünschten Betriebssysteme. Nun muss man entweder entscheiden von vornherein eine unabhängige Bibliothek zu verwenden, oder je nach System eine andere zu verwenden. Verwendet man verschiedene Bibliotheken unter verschiedenen Systemen, ist der Entwicklungsaufwand der Engine deutlich höher, da nicht nur eine Abstraktion der Betriebssysteme stattfindet, sondern auch eine der Bibliotheken selbst. In einigen Fällen kann es sich jedoch trotzdem lohnen, weiteres dazu in den jeweiligen Kapiteln.

III. Die Wahl der Programmiersprache

Es existiert eine Vielzahl an verschiedenen Programmiersprachen mit denen man auf verschiedenen Betriebssystemen entwickeln kann, von denen jede ihre Vor und Nachteile hat. Grundsätzlich kann man die Programmiersprachen in Zwei Kategorien einteilen.

A. Kompiliert in Maschienencode

Bei diesen Programmiersprachen (z.B. C,C++) wandelt der Compiler den Source Code in eine direkt ausführbare Binärdatei um. Da wir je nach Betriebssystem zwangsläufig unterschiedlichen Code brauchen, braucht man für jedes Betriebssystem einen eigenen Code. Wie das genau umgesetzt wird, bleibt dem Programmierer überlassen. Möglich sind in C/C++ Präprozessor Befehle, die je nach Betriebssystem einen gewissen Code kompilieren und den Teil für ein anderes Betriebssystem weglassen.

B. Kompiliert in eine Zwischensprache

Bei diesen Programmiersprachen (z.B. Java, .Net Sprachen) wandelt der Compiler den Source Code in eine Binärdatei, die für eine virtuelle Maschiene ausführbar ist, welche auf dem Zielsystem läuft. Die Betriebssystemunabhängigkeit ist dadurch automatisch mit dem gleichen Code gegeben, es müssen keine unterschiedlichen Source Codes verwendet werden. Eine automatische Plattformunabhängigkeit ist dadurch explizit nicht gegeben, denn um das Little/Big Endian Problem muss sich auch in diesen Programmiersprachen gekümmert werden.

Der Hauptvorteil der in Maschienencode kompilierten Programmiersprachen ist die Effizienz. Da der Code direkt für die jeweilige Plattform und Betriebssystem kompiliert wurde, läuft dieser in einer sehr hohen Geschwindigkeit. Oft sind diese Sprachen jedoch schwieriger zu lernen und brauchen wie oben erwähnt, für jedes Betriebssystem individuellen Code, der Entwicklungsaufwand ist also höher. Genau anders herum ist es bei den in Zwischensprache kompilierten Sprachen, diese sind weniger Effizient in der Laufzeit, jedoch deutlich komfortabler zum entwickeln.
Welche Programmiersprache sollte man nun also nun wählen? Darauf gibt es keine eindeutige Antwort, denn Sie hängt vollkommen von der Abwägung der Vor/Nachteile ab. Auf einem modernen PC laufen selbst sehr komplexe 3D Anwendungen Mühelos, somit sind 2D Berechnungen schon lange kein großes Problem mehr. Möchte man also hauptsächlich Heimcomputer als Zielgruppe anvisieren, würde sich der Effizientvorteil von C/C++ kaum bemerkbar machen, jedoch deutlich die Entwicklungzeit erhöhen. In diesem Fall ist die zweite Kategorie, die bessere Wahl. Möchte man beispielsweise einen selbstgebauten Arcade Automaten aus einem alten Raspberry PI basteln, bei dem der Leistungsvorteil bemerkbar wäre, kann es sich lohnen Maschienencode kompilierte Sprachen zu verwenden. Abgesehen von den objektiven Kriterien, hängt die Frage nach der Programmiersprache natürlich auch von den subjektiven Vorlieben des Programmierers ab, sowie von seiner Erfahrung. Möchte man beispielsweise ein einfaches Fenster erstellen, kann man in Java unter jedem Betriebssystem einfach ein JFrame verwenden und die Tastatur auch mit in Java vorhandenen Methoden abfragen. Unter C/C++ ist das Thema schon deutlich schwieriger, da man für jedes Betriebssystem die jeweilige API des Herstellers verwenden muss. Für einen eher unerfahrenen Entwickler wird es sicher die bessere Wahl sein eine Sprache wie Java zu verwenden, da diese bereits Betriebssystem unabhängig ist. In den weiteren Kapiteln liegt deshalb der Fokus auf der Programmiersprache Java.

IV. Grundaufbau einer Game Engine

Eine Game Engine besteht aus mehreren Kern Komponenten, diese sind die Grafik, das Audio und die Logik. Die Grafik Komponente kümmert sich um alles was visuell darstellbar ist. Das fängt an bei der Erstellung und Verwaltung eines Fensters, welches für die Ausgabe verwendet wird. Weiterhin kümmert sich diese Komponente um das Laden, Anzeigen und Verwalten der Bilder für das Spiel, zum Beispiel das Anzeigen eines Hintergrundes und der Spielfigur. Die Audio Komponente kümmert sich um das Laden und Abspielen von Tönen, zum Beispiel wenn der Spieler einen Sprung ausführt. Die Logik Komponente kümmert sich um Eingabe des Spielers, zum Beispiel wenn dieser die Tastatur oder die Maus verwendet. Alle diese Komponenten haben keine festgeschriebene Komplexität und können belibieg erweitert werden. So kann die Grafikkomponente um belibieg viele unterstützte Bildformate erweitert werden oder um Effekte die auf die Bilder hinzugefügt werden können. Dem Ausmaß an Funktionalität sind keine Grenzen gesetzt. [1] Damit ein Spiel diese Komponenten verwenden kann, bedarf es eines Game Loops. Dabei handelt es sich um eine Schleife, die alle benötigten Grundfunktionen der Engine ausführt, gefolgt von den Engine unabhängigen Funktionen des eigentlichen Spiels, das die Game Engine benutzt. Ein vollständiger Ablauf dieser Schleife erzeugt genau ein Bild auf dem Bildschirm und verarbeitet alle relavanten Aktionen. Dieser Ablauf läuft mehrere male pro Sekunde ab.

V. Die Grafik Komponente

Die Grafik Komponente in einer Game Engine ist verantwortlich dafür ein Fenster zu erstellen und ein Bild in diesem anzuzeigen. Das Bild ist eine Zusammensetzung aus mehreren Einzelteilen, die dann das Gesamtbild ergeben. Diese Einzelteile können primitive geometrische Objekte sein, wie z.B. Linien, Kreise, Rechtecke etc. Weiterhin sollte es auch die Möglichkeit geben Bilder in den gängigen Formaten zu unterstützen, damit in Zeichenprogrammen erstellte Bilder verwendet werden können. Bevor es überhaupt soweit ist Objekte auf dem Bildschirm anzuzeigen, muss erst eine geeignete API zur Grafikkarte verwendet, da diese für alles was auf dem Bildschirm angezeigt wird verantwortlich ist.

A. Die Grafik API

Die Grafik API ist die Programmierschnittstelle zur Kommunikation mit der Grafikkarte. Möchte man etwas auf dem

Bildschirm anzeigen, so wird das über die Grafik API gemacht. Vorher muss allerdings ein Fenster erstellt werden, welches die Ausgabe auch anzeigt. Java bietet Methoden und Klassen um geeignete Fenster zu erstellen, z.B. das JFrame. Weiterhin bietet Java vorhandene Methoden um primitive Grafiken in diesem Fenster anzuzeigen. Diese sind jedoch nicht hardware beschleunigt und deshalb deutlich weniger performant als die Grafik APIs. Für eine Game Engine sollte somit auf eine Grafik API zurückgegriffen werden. OpenGL ist eine betriebssystemunabhängige Grafik API und somit bestens für diesen Anwendungsfall geeignet. Mit OpenGL können zum Beispiel einzelne Pixel hardwarebeschleunigt auf dem Bildschirm angezeigt werden. [2]

B. Die Renderpipeline

Wenn man einen Bildschirm genauer betrachtet, bemerkt man das er nur ein Koordinatensystem aus Pixeln ist. Der Ursprung dieses Koordinatensystems liegt üblicherweise nicht in der Mitte des Bildschirms, sondern in der oberen linken Ecke. Die X Achse zeigt und wächst nach rechts und die Y-Achse zeigt und wächst nach unten. In einem 1920x1080 Bildschirm ist die Koordinate ganz oben links (0,0) und die unterste rechte Koordinate somit (1920,1080).
Um nun etwas auf dem Bildschirm anzuzeigen, müssen die Informationen verarbeitet werden und in einen Frambuffer geschrieben werden. Dieser Prozess wird Renderpipeline genannt und besteht aus mehreren Schritten.

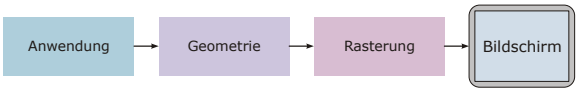

Fig. 1. Renderpipeline Prozess[3]

Grob zusammengefasst liefert die Anwendung liefert die Geometrie Daten. Diese Daten werden dann gerastert, d.h. in Pixel umgewandelt, die auf dem Bildschirm angezeigt werden. Da die einzelnen Geometrien nacheinander geladen werden, entsteht am Bildschirm ein flackern, da nach jedem Update in der Szene neu gezeichnet wird. Zum Beispiel sieht man erst den leeren Hintergrund, dann den Hintergrund und eine Linie und in einem weiteren Schritt noch einen Kreis. Um dieses flackern zu verhindern, wird das gesamte Bild zuerst in einen Puffer gezeichnet und sobald das Bild fertig gezeichnet wurde, wird es in einem Schritt am Bildschirm angezeigt. Dazu wird es in den Framebuffer geladen. Der Framebuffer ist ein Bereich im Speicher, der die direkten Pixeldaten im Bitmap Format des Bildschirms enthält. D.h. jedes Pixel ist in Form eines Farbwertes, meistens Rot, Grün, Blau und Alpha(Transparenz) abgespeichert.[4]

C. Primitive

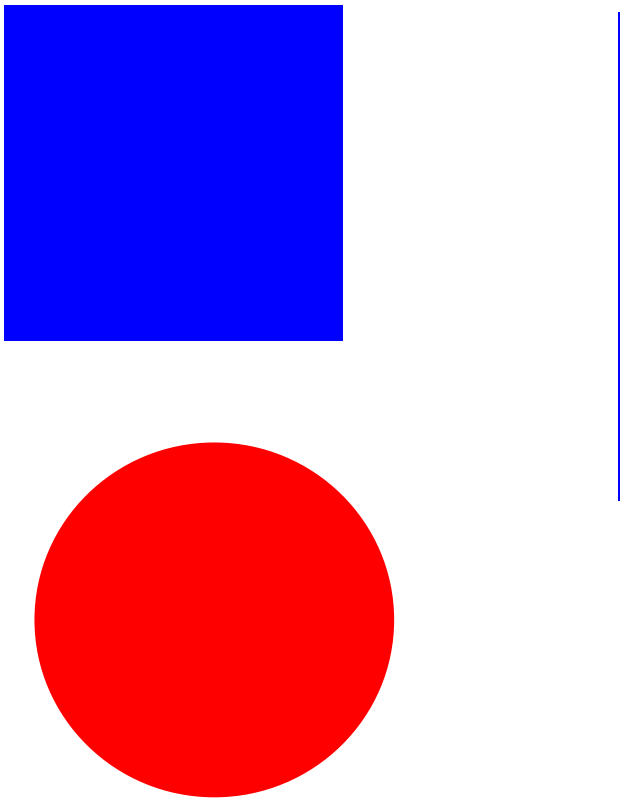

Fig. 2. Primitive 2D Objekte

Primitive 2D Objekte sind, wie der Name schon sagt, einfache zweidimensionale Formen, wie Rechtecke, Kreise und Linien. Diese können mit einer Farbe gefüllt werden, oder einfach nur einen Rahmen ohne Inhalt darstellen.

1) Linie: Eine Linie entspricht mathematisch gesehen einer Geraden, die durch die zwei Punkte P1(x1,y1) und P2(x2,y2) geht und im Interval [x1,x2] liegt. Ein naiver, einfacher und funktionierender Ansatz um eine Linie zu zeichnen wäre es, für jedes X auf diesem Interval mithilfe der Geradengleichung ein Y zu berechnen und in den Framebuffer an die jeweilige Position zu speichern.

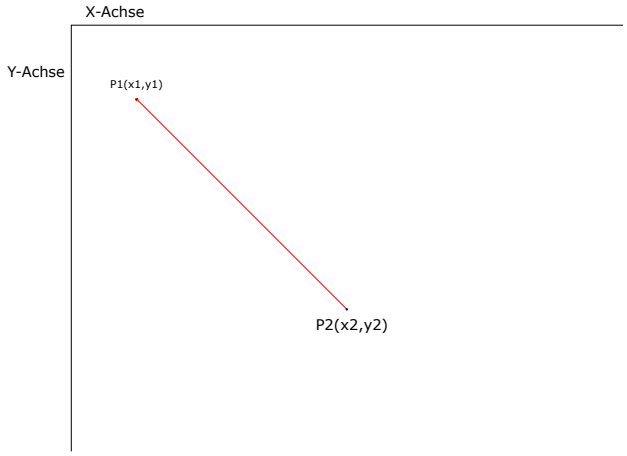

Fig. 3. Linie zwischen P1 und P2

Um die Werte der Linie ausrechnen zu können, muss zuerst die Geradengleichung der Linie mit folgenden drei Formeln aufgestellt werden:

$$y = m \cdot x + c$$
$$m = \frac{y_2 - y_1}{x_2 - x_1}$$
$$c = y_1 - m \cdot x_1$$

Iteriert man mithilfe einer for Schleife zwischen x1 und x2 und berechnet den jeweiligen Y Wert, bekommt man die Punkte der Linie und kann diese Zeichnen.
Wie oben bereits erwähnt funktioniert dieser Ansatz zwar, jedoch ist er nicht besonders effizient, da sehr viele Gleitkomma Berechnungen ausgeführt werden müssen. Eine deutlich effizientere Alternative bietet der Bresenham Algorithmus, dieser wird hier jedoch nicht näher behandelt.
Da wir nun Linien zeichnen können, ist es kein großer Aufwand andere Primitive wie Dreiecke und Rechtecke zu zeichnen, da diese ebenfalls nur aus einfachen Linien bestehen. Der Endpunkt einer Kante ist der Anfagspunkt der nächsten Kante.

2) Kreis: Für einen Kreis bedarf es etwas höherer Mathematik. Kreise haben keinen Start und Endpunkt, sondern nur einen Radius. Möchte man die Koordinaten für die Pixel eines Kreises berechnen, so muss man α über das Interval [0;2π] iterieren und mithilfe von sinus und cosinus die jeweilige X und Y Koordinate der Punkte berechnen und mit einem Offset versehen. Sei K ein Kreis mit dem Radius r der seinen Mittelpunkt im Punkt P(x,y) hat, so gelten folgende Formeln zur Berechung der Pixelkoordinaten für den Winkel α

$$f_x(\alpha) = \cos(\alpha) \cdot r + x$$
$$f_y(\alpha) = \sin(\alpha) \cdot r + y$$

Für einen lückenlosen Kreis, sollten die berechneten Koordinaten nicht direkt als Punkte gezeichnet werden, sondern als Linien zwischen zwei benachbarten Werten für α

3) Füllen der Primitive: Bisher wurden nur unausgefüllte Primitive behandelt, oft benötigt man allerdings mit Farbe gefüllte Primitive. Um dies zu erreichen, gibt es unter anderen den sog. Floodfill Algorithmus. Dieser wird hier nicht näher behandelt und kann in der Literatur nachgelesen werden

D. Laden von Bildern

Mit Primitiven kann man, wie der Name schon impliziert, simple Objekte zeichnen. Gerade in Spielen werden aber deutlich komplexere Grafiken, wie z.B. Menschliche Figuren benötigt. Diese nur durch Primitive im Code nachzustellen, wäre sehr aufwändig. Abhilfe schaffen vorgespeicherte Bilder, die in anderen Programmen erstellt wurden. Die gänigsten Formate sind BMP, PNG, TIFF und JPG. Abgesehen von Bitmaps sind die anderen Formate komprimiert. D.h. man kann nicht direkt die Pixeldaten aus der Datei lesen, ohne sie zuerst zu dekomprimieren. [5] Für die Dekompression gibt es viele sehr gute betriebssystem unabhängige Bibliotheken. Diese dekomprimieren die Dateien und liefern die rohen Pixeldaten. Speichert man diese Daten im Ram ab, können sie von dort in den Framebuffer kopiert werden und das Bild wird angezeigt. Da die Bilder in jedem Update angezeigt werden, macht es Sinn diesen Prozess zu automatisieren. Anstatt ständig das Bild manuell neu zeichnen zu lassen, kann man eine Szenen Klasse erstellen. Diese Klasse besitzt eine Liste von Sprites. Sprites sind Objekte mit Bilddaten, die um eine Größe und Position erweitert werden. Läd man ein Bild von der Festplatte, läd man es in Form eines Sprites in die Szene hinein und legt eine Größe und Position fest. In jedem Bildschirm Update, geht die Game Engine alle Sprites durch und zeichnet sie automatisch an die richtige Stelle in der richtigen Größe. Verändert man die Position eines Spriteobjekts, wird sie automatisch beim nächsten Update an der richtigen Position gezeichnet. Die Sprite Klasse kann wieder belibieg erweitert werden, so kann man eine Rotation hinzufügen, verschiedene Farbeffekte und vieles mehr.

VI. Die Audio Komponente

Neben der Grafik ist auch der Ton ein wichtiges Element in jedem Spiel und darf somit in einer Game Engine nicht fehlen. Die gängigsten Ton Formate die unterstützt werden sollten, sind WAV, FLAC, MP3 und OGG. [6] Auch hierfür gibt es wieder gute betriebssystem unabhängige Bibliotheken, mit denen sich die Audiodaten importieren lassen und abspielen.

VII. Die Logik Komponente

Die Logik Komponente kümmert sich um mehrere Dinge, u.a. die Kommunikation mit dem Betriebssystem. Dazu zählt z.B. die Verarbeitung von Ereignissen in Bezug auf das Fenster. Wenn das Fenster vergrößert, verkleinert, verschoben etc. wird, kümmert sich die Logik darum, dass trotzdem weiterhin alles reibungslos abläuft. Neben den Fensterereignissen verwaltet sie auch die menschliche Eingabe über die Maus, Tastatur und evtl. Joysticks und Controller. Weiterhin ist es auch die Aufgabe der Logik eine gewisse Physik zu simulieren, z.B. ob zwei Sprites miteinander kollidieren.

A. Fenstermanagement

Benutzt man als Sprache C oder C++, muss man sich selbst um das Fenstermanagement kümmern. Dazu kann man entweder die betriebssystem spezifischen APIs benutzen, oder man bedient sich einer vorhandenen betriebssystem unabhängigen Bibliothel, die diese Funktionalität bereits bietet. Benutzt man Java, ist diese Funktionalität bereits betriebssystem unabhängig gelöst und das Fenstermanagement ist kein Problem mehr.

B. Menschliche Eingabe

Typisch für die menschliche Eingabe sind Tastaturen und Mäuße. Wie beim Fenstermanagement auch gibt es auch hier in Java bereits Methoden, die betriebssystem unabhängig eine Schnittstelle zu diesen Geräten bieten. Diese müssen jedoch ein wenig an die Bedürfnisse einer Game Engine angepasst werden.

1) Maus: Ruft man die Position der Maus ab, ist diese relativ zum Ursprungs des Bildschirms und nicht des Fenster. Koordinaten außerhalb des Fensters, können also negativ oder zu groß sein. Deshalb müssen diese erst in das Fensterkoordinatensystem umgewandelt werden. Dazu müssen lediglich die Koordinaten des Fensters von der Maus Position abgezogen werden und evtl noch ein kleines Offset für den Rahmen des Fensters addiert werden. Da im Vollbild Modus der gesamte Bildschirm genutzt wird, entfällt dort diese Rechnung. Es kann jedoch trotzdem der gleiche Code verwendet werden, da die Fenster Position im Vollbildmodus bei (0,0) liegt und die Subtraktion somit keine Auswirkung hat. Der zusätzliche Code wird also nicht zwingend benötigt. Um die Nutzung der Maus möglichst einfach zu gestallten, empfiehlt es sich die Daten in eine Klasse zu kapseln, da sie dort bis zur Spiellogik gespeichert werden kann und komfortabel aufgerufen werden kann. Für die Maustasten ist die vorgehensweise etwas komplexer. Oft reicht es nicht einfach aus, zu wissen ob mit der Maus geklickt wurde. In vielen Fällen ist es wichtig zu wissen, ob die Mausbuttons gedrückt wurden, gehalten werden oder losgelassen wurden. Für ein Mouse Click, der nur einmal pro Klick aktiv ist, muss man also prüfen, ob die Maus geklickt wurde, und ob bereits ein Engine Update geschehen ist. Um zu prüfen ob die Maus losgelassen wurde, muss einfach überprüft werden, ob die Maus geklickt ist, falls nicht wird in einem weiteren Schritt überprüft ob sie zuletzt noch geklickt war. Das ganze lässt sich mit Variablen gut umsetzen und ist kein kompliziertes Unterfangen.

2) Tastatur: Für die Tastatur gilt das gleiche Prinzip wie für die Maus. Ähnlich wie die Mausbuttons, kann man auch die Tasten einer Tastatur abfragen, da es auch dort einen kurzen Druck, ein halten und ein Loslassen gibt. Speichert man diese drei Werte, für jede gedrückte Taste in eine Liste, so kann das Spiel dann über eine Tastatur Klasse ganz einfach abrufen, ob eine Eingabe durch den Spieler erfolgt ist und entsprechende Aktionen vornehmen. Auch für die Tastatur bietet Java geeignete Methoden, zum Abfragen der Zustände der Tastatur.

C. Physik

Da es sich beim Thema Physik um ein sehr komplexes Thema handelt, wird hier nur ein sehr einfacher Fall einer Kollision zwischen zwei Sprites behandelt.

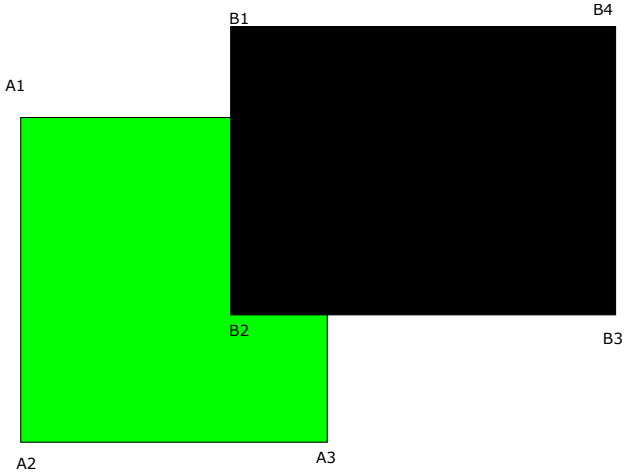

Fig. 4. Kollision zwischen zwei Rechtecken

Jedes Sprite besitzt eine Position, eine Breite und eine Höhe. Ungeachtet des Inhalts, kann dieses Rechteck also Kollisionsbereich gesehen werden. Liegt ein anderes Sprite in diesem Bereich, so kollidieren sie miteinander. In diesem Fall stellt A das eine Sprite dar und B das andere. Die 4 Punkte A1-A4 und B1-B4 stellen die Ecken des Kollisionsrechtecks dar. Damit eine Kollision vorliegt, müssen mindestens ein Punkt des einen Rechtecks im anderen Rechteck liegen. Somit muss folgende Bedingung für mindestens einen Eckpunkt A_i erfüllt sein:

$$(A_i.X >= B1.X \wedge A_i.X <= B3.X)$$
$$\wedge(A_i.Y >= B1.Y \wedge A_i.Y <= B3.Y)$$

Die Engine implementiert nun eine Methode, die zwei Sprites als Parameter annimmt und aus deren Metadaten dann die Punkte berechnet. Ist die Bedingung für einen Eckpunkt wahr, gibt sie true zurück, andernfalls false. Die Reihenfolge der Sprites spielt dabei keine Rolle, da eine Kollision immer in beide Richtungen stattfindet.
Diese Kollisionsüberprüfung ist sehr einfach und sehr performant. Handelt es sich bei den Sprites um Bilder mit Transparenz, wird eine Kollision auch dann erkannt, wenn sich nur die transparenten Bereiche schneiden. Eine Überprüfung die auch die Transparenz miteinbezieht ist deutlich aufwändiger und rechenintensiever zu implementieren. In den meisten Anwendungsfällen, reicht die Rechteck Methode vollkommen aus, trotz Transparenter Bilder.

VIII. Fazit

In diesem Paper wurden die notwendigen Grundlagen und ein kurzer Überblick über die Entwicklung einer betriebssystem unabhängigen Game Engine darstellt ohne dabei zu tief in die konkrete Implementierung zu gehen. Bei der konkreten Entwicklung der Game Engine gilt es viele Entscheidungen

zu treffen, die stark auf Subjektivität beruhen, ein Richtig oder Falsch gibt es nicht. Das Thema betriebssystem unabhängigkeit stellt sich als nicht ganz so einfach heraus, da es viele Dinge zu beachten gibt, die man auf den ersten Blick nicht erkennt. Abhilfe schafft die Programmiersprache Java, sie selbst Betriebssystem unabhängig konzipiert ist. Für eher unerfahrene Programmierer und für einen deutlich verringerten Entwicklungsaufwand ist sie deshalb eher die Programmiersprache der Wahl. Im Bereich der Grafik und der Physik bemerkt man schnell, dass eine gewisse mathematische Fertigkeit für die Entwicklung unerlässlich ist. Generell ist es keine leichte Programmieraufgabe eine Game Engine zu entwickeln, jedoch bekommt man dadurch ein deutlich tiefergehendes Verständnis für die Funktionsweise von Computergrafik und eignet sich nützliche Fähigkeiten an, wenn man die Disziplin besitzt sich in verschieden Bibliotheken und APIs einzulesen.

References

[1] Michael, Alles, was Du über Game Engines wissen musst, Adresse:https://blog.nobreakpoints.com/game-engines/ (besucht am 13.05.2021)

[2] The Khronos Group, OpenGL Overview. Adresse:https://www.khronos.org/opengl/ (besucht am 15.05.2021)

[3] Grafikpipeline, Adresse:https://de.wikipedia.org/wiki/Grafikpipeline (besucht am 15.05.2021)

[4] eTutorial.org, Software engineering and computer games Adresse: http://etutorials.org/Programming/Software+engineering+and+computer+games/Part+II+Software+Engineering+and+Computer+Games+Reference/Chapter+24.+2D+and+3D+graphics/24.2+The+graphics+pipeline/ (besucht am 15.05.2021)

[5] Wintotal, Gängige Bildformate im Vergleich: Diese Unterschiede gibt es Adresse:https://www.wintotal.de/bildformate/ (besucht am 16.05.2021)

[6] Annika Wegerle, Audioformate: Alles, was MusikerInnen wissen müssen, um die richtige Datei zu wählen. Adresse:https://blog.landr.com/de/audioformate-alles-was-musikerinnen-wissen-muessen-um-die-richtige-datei-zu-waehlen/ (besucht am 17.05.2021)