Programmierung:

Grundlegende Struktur:

Programmieren in C++:

Warum C++?

C++ ist eine von [Bjarne Stroustrup](https://de.wikipedia.org/wiki/Bjarne_Stroustrup) bei [AT&T](https://de.wikipedia.org/wiki/AT%26T) entwickelte Programmiersprache, sie gilt als Erweiterung der Sprache C. (Wiki) C++ ermöglicht dem Nutzer eine Maschinennahe Programmierung und dadurch auch hohe Effizienz. Dies ist gerade bei der leistungsaufwändigen Spieleprogrammierung relevant.

C++ und OOP:

Die Sprache unterstützt, anders als C, objektorientierte Programmierung (OOP). Das bedeutet das der Code praktisch in kleinerer abschnitte unterteilt werden kann, diese abschnitte nennt man Klassen. (Bessere Erklärung einfügen)

Wenn der Code, der sich in den Klassen befindet, ausgeführt werden soll, muss eine Instanz (Kopie des Codes) erstellt werden. Je nach dem ob der Code im Stack oder im Heap allokiert werden soll, wird anderer Code verwendet sprich: „Class ClassInstanceAtStack;“ oder „Class\* ClassInstanceAtHeap = new Class()“

Wobei „new“ einen Pointer\* initiiert und die Größe der Klasse in Bytes im Heap freigibt. (Dabei wird in der Pointer Variable die Speicheradresse gespeichert). Um den Speicher wieder Freizugeben muss anschließend „delete“ aufgerufen werden um den Speicher freizugeben. Um den new/delete aufwand zu umgehen werde ich im Programm smart-pointer\* verwenden.

std::shared\_ptr und std::unique\_ptr: (Aus der offiziellen Dokumentation zitieren)

Um das Prozedere um new/delete bei der Erstellung von Pointern\* zu umgehen wurden in C++XX (Recherchieren) Smart-Pointer hinzugefügt diese automatisieren die Speicher zu- und Abweisung bei Pointern. Es gibt 3 verschiedenen Variationen dieser Pointer: unique\_ptr, shared\_ptr und weak\_ptr. std::unique\_ptr kann nicht kopiert werden. (z.B in einem Vector gespeichert werden oder als Parameter einer Funktion verwendet werden).

std::shared\_ptr hingegen kann kopiert werden. Dies kommt aber mit einem Overhead\* da Kontrollspeicher mit überblick über erstellte Referenzen initialisiert wird. Prinzipiell wird der Speicher erst freigegeben wenn keine Referenz zum shared\_ptr existieren. Sollte eine Referenz zu einem shared\_ptr benötigt werden welche nicht als eigenständige Kopie gelistet werden soll kann std::weak\_ptr verwendet werden.

(Evtl. C++ Logo einfügen?)

Da ich zu beginn der Arbeit schon grundlegende Kenntnisse in C++ habe, liegt die Wahl der Programmiersprache nahe.

Da ich bereits vor

C++ Spracheigenschaften:

SMFL:

Das Spiel:

Klassen:

GameWindow:

MenuWindow:

MapManager:

Player:

Enemy

Item

Gun

Perlin Noise:

* Theorie
* NoiseGenerator

Warum Perlin Noise?:

Die Karte des Spiels wird zufällig generiert, um aber zu vermeiden das diese unnatürlich wirkt, wird ein Algorithmus zu prozeduralen Generation genutzt, namentlich Perlin noise\*. Das der Prozess des Map-Designs dabei völlig wegfällt, ist im Rahmen einer Einzelentwicklung ein großer Vorteil, da dieser sonst einen aufwändigen Bestandteil der Entwicklung darstellt. Für dieses Projekt wird eine Perlin Noise ähnliche funktion verwendet.

Konzept:

Das Prinzip hinter Perlin noise ist grundsätzlich mehrere Funktionen mit jeweils unterschiedlichen Amplituden und Schwingungsdauer (Ausdehnung an der x- bzw. y-Achse) zu addieren und so ein Rauschen zu erzeugen. Hierbei werden üblicherweise Sinuswellen verwendet. „Eine Sinuswelle in ihrer Grundform ohne vertikale Verschiebungen besteht aus einer Reihe von Wellenperioden, die jeweils zwischen einem Start- und einem Endwert von Null wechseln.“\* Die addierten wellen werden Oktaven genannt.

Implementierung:

Die Implementierung wurde für die Entwicklung des Spiels von „One Lone Coder“ zum größten Teil übernommen.

Wir beginnen mit einem Seed-Array welches mit zufälligen werten zwischen 0 und 1 belegt wird:

fNoiseSeed1D = new float[nOutputSize];

for (int i = 0; i < nOutputSize; i++) { fNoiseSeed1D[i] = (float)rand() / (float)RAND\_MAX; }

An dieser stelle könnte man nun eine Funktionen verwenden um Werte aus dem Seed-Array zu extrahieren. Wir verwende aber eine simple Funktion bei der wir von 1\*S(x, Width) bis 1/log2(width)\*s(x, width/log2(width) runterskalieren. Daher 1.0 \* S(x, 256) \* 0.5 \* S(x, 128) + 0.25 \* S(x, 64) … = P. => ∑i=0, n=log2(width=256) ( 1/2^i \* S(256/2^i)

An jeder stelle s(x,i) interpolieren linear wir mit x mit x+1. (S(x,i) ist periodisch daher bei z.b s(x,4) die stellen NS0, NS4, NS8 usw. Interpoliert). Interpolation bedeutet dabei eine gerade zwischen zwei punkten zu finden. Formal daher: (y1-y0)/(x1-x0) (Wird aufgelöst: Ganze rechnung in LATTEX angeben). Da die Funktionswerte addiert werden repräsentiert der letzte Graph (also S(x,1)) den Perlin-Noise output. Um den Graphen aber zwischen 1 und 0 zu halten normalisieren wir den Perlin-output in dem wir an jeder Stelle durch die summe der Skalierungsfaktoren dividieren (1 + 0.5 + 0.25 …).

(Code Einfügen?)

Ein Bild, das Farbigkeit enthält.

Automatisch generierte BeschreibungEin Bild, das Farbigkeit, Screenshot enthält.

Automatisch generierte Beschreibung

Durch das anhäufen von Oktaven entstehen neue Maximalstellen entlang des Funktionsgraphen, dies führt ebenfalls zu einer generellen Abrundung. Vor allem bei einer Erweiterung in die 2. Dimension werden die Änderung bemerkbar. Ein Bild, das Grün, Farbigkeit, Screenshot, Kunst enthält.

Automatisch generierte Beschreibung

Ein Bild, das Karte, Farbigkeit, Grün, Screenshot enthält.

Automatisch generierte Beschreibung

Für die Erweiterung in die 2. Dimension sind kaum Veränderungen im Code notwendig wir nehmen ein weiteres Sample entlang der y-Achse

int nSampleY1 = (y / nPitch) \* nPitch;

int nSampleY2 = (nSampleY1 + nPitch) % nWidth;

und interpolieren die beiden Sampels:

float fBlendY = (float)(y - nSampleY1) / (float)nPitch;

selbiges passiert auch für die x Koordinate.

Ein Bild, das Farbigkeit, Kunst, Majorelle Blue, Grafiken enthält.

Automatisch generierte Beschreibung

Durch das einsetzten von einem Bias, sprich Dämpfung der Oktaven entlang ihrer Funktionsgraphen. (Obige Abbildung)

## Mapmanager:

Die Klasse **MapMaker** ist essenziell für die Erstellung und Verwaltung der Spielwelt in einem 2D-Spiel. Sie integriert Spieler, Karten, Gebäude, Spawner und weitere Objekte zu einer dynamischen, interaktiven Umgebung. Der Standardkonstruktor initialisiert die Klasse, während die Methode Initialize die Spielwelt einrichtet, indem der Spieler erstellt, die Karte generiert, Gebäude und Spawner initialisiert sowie die Kamera angepasst werden.

Zur Spielmechanik gehören die Methoden Update und Draw. Update aktualisiert in jeder Spielschleife die Spielwelt, darunter Spielerbewegungen, Spawner, Gebäude und Kollisionen. Draw rendert alle Objekte auf dem Spielfenster, wobei die Kamera auf den Spieler zentriert wird.

Interne Hilfsmethoden umfassen die Initialisierung von Spawnern (InitSpawner), Häusern (InitHouses) und Gegenständen (InitItems). Zudem aktualisiert UpdateBuildings den Zustand von Gebäuden und Waffen. Für kartenspezifische Logik sorgen Methoden wie CalculatePosition (Berechnung von Objektpositionen), AdjustTileMap (Markierung belegter Kacheln) und Overlapping (Überprüfung von Kollisionen).

Unterstützende Klassen wie **Player**, **Spawner**, **Building**, **Gun**, **MapGenerator** und **CollisionManager** übernehmen spezifische Aufgaben wie die Verwaltung von Spielerbewegungen, Gegnern, statischen Objekten und Kollisionen. Globale Daten wie **GameData** und Ressourcen für Texturen und Schriftarten runden die Funktionalitäten ab, indem sie zentrale Informationen bereitstellen.

## MapGenerator:

Die Klassen **MapGenerator**, **BiomeSetter** und **NoizeGenerator** arbeiten zusammen, um eine detaillierte und performante Karte für ein 2D-Spiel zu erstellen und darzustellen.

Die **MapGenerator**-Klasse erzeugt die Spielkarte, definiert ihre Struktur und übernimmt die Darstellung. Während der Initialisierung werden Kacheln (Tiles) mit einer festen Größe, Positionen und Standardtexturen vorbereitet. Biome und ein Rauschgenerator sorgen für geografische Vielfalt. Bei der Generierung wird jede Kachel mit einem spezifischen Biom und einer entsprechenden Textur ausgestattet. Zur Performance-Optimierung werden nur sichtbare Kacheln basierend auf der Spielerposition gerendert.

Die **BiomeSetter**-Klasse definiert verschiedene Biome wie Wüste, Wald oder Ozean. Basierend auf den vom **NoizeGenerator** erzeugten Rauschwerten wählt sie das passende Biom für jede Kachel aus. Dazu vergleicht sie die Eigenschaften der Biome mit den Rauschwerten und bestimmt das beste Match.

Die **NoizeGenerator**-Klasse erzeugt Rauschmuster, um geografische Eigenschaften der Karte zu modellieren. Sie berechnet für jede Position auf der Karte normalisierte Rauschwerte, die anschließend vom **BiomeSetter** zur Bestimmung des Bioms verwendet werden.

Im Zusammenspiel erstellt der **MapGenerator** eine Kachelstruktur und weist mithilfe des **BiomeSetter** Biome und Texturen zu, wobei die Werte des **NoizeGenerator** die Grundlage für die Biome bilden. Die Darstellung der Karte bleibt durch das Beschränken auf den Sichtbereich des Spielers performant.

## ColliosionManager:

Die Klasse verwaltet eine Liste von Spielobjekten und übernimmt deren Kollisionserkennung. Sie speichert Objekte in einer Liste und ermöglicht das Hinzufügen und Entfernen. Die Hauptaufgabe besteht darin, Kollisionen zwischen den Objekten zu erkennen, indem ihre Kollisionsboxen überprüft und bei Überschneidungen entsprechende Kollisionsevents ausgelöst werden.

Die Methode checkCollisions entfernt zunächst inaktive Objekte aus der Liste, bevor sie die verbleibenden Objekte paarweise auf Kollisionen prüft. Dabei wird die Methode GetBoundingBox verwendet, um die Kollisionsboxen zu erhalten, und intersects, um Überschneidungen zu erkennen. Bei einer Kollision wird die Methode OnCollision für die beteiligten Objekte aufgerufen, um individuelle Reaktionen zu ermöglichen. Mit addObject können neue Objekte zur Liste hinzugefügt werden, während clearInactiveObjects inaktive Objekte effizient entfernt.

Diese Implementierung ist effizient, da inaktive Objekte automatisch entfernt werden und die paarweise Prüfung redundante Vergleiche vermeidet. Die Flexibilität wird durch die individuell anpassbaren Methoden OnCollision und GetBoundingBox gewährleistet.

Potenzielle Verbesserungen umfassen die Optimierung der paarweisen Prüfung bei großen Objektmengen durch Raumaufteilungsalgorithmen wie Quadtrees oder Gitter. Zusätzlich könnten Kollisionen mittels Multithreading parallelisiert und durch Gruppierungen von Objekten (Layer) weiter gefiltert werden, um die Leistung zu steigern.

## Spawner:

Die **Spawner-Klasse** ist eine komplexe und vielseitige Komponente zur Erzeugung und Verwaltung von Gegnern in einem 2D-Spiel. Sie vereint Mechanismen für das dynamische Spawning, die Verwaltung von Gegnerzuständen und deren Interaktion mit der Spielwelt. Im Folgenden wird die Funktionsweise dieser Klasse detailliert beschrieben.

Bei der Initialisierung eines Spawners werden alle wesentlichen Parameter und Eigenschaften festgelegt. Der Konstruktor nimmt die Textur-ID und die Schriftart-ID entgegen, die für die visuelle Darstellung benötigt werden, sowie die Anfangsposition des Spawners und eine eindeutige Spawner-ID. Diese Werte bestimmen das Aussehen und die Platzierung des Spawners in der Spielwelt. Zusätzlich werden die Grundwerte für die Gesundheit, den Schaden und die Spawnrate des Spawners definiert. Der Gesundheitsstatus des Spawners wird als Text vorbereitet, der während des Spiels aktualisiert und angezeigt wird. Die Spawnrate, die standardmäßig auf 2000 Millisekunden gesetzt ist, legt fest, wie oft neue Gegner aus der Warteschlange erzeugt werden.

Ein zentraler Bestandteil der Klasse ist die Methode Update, die in jedem Spielzyklus ausgeführt wird. Zu Beginn überprüft diese Methode den aktuellen Zustand des Spawners. Sollte dessen Gesundheit auf null gesunken sein, werden alle aktiven Gegner entfernt, und der Spawner wird deaktiviert. Diese Maßnahme stellt sicher, dass ein zerstörter Spawner keine weiteren Gegner generieren kann. Anschließend wird der Text für den Gesundheitsstatus des Spawners aktualisiert, um den Spielern visuelles Feedback über den Zustand des Spawners zu geben.

Der eigentliche Spawning-Prozess wird durch den internen Timer gesteuert, der die Zeit seit dem letzten Spawn misst. Wenn der Timer die festgelegte Spawnrate überschreitet, wird ein neuer Gegner aus der Warteschlange m\_stack entnommen und durch die Methode SpawnNPC aktiviert. Hierbei wird ein neuer Gegner-Typ wie ein Zombie, Skelett oder Bandit erzeugt und an einer zufälligen Position in der Nähe des Spawners platziert. Die Methode CalculatePosition berechnet diese Position basierend auf einem vorgegebenen Radius, sodass die Gegner in einem logischen Bereich um den Spawner erscheinen. Der neu erzeugte Gegner wird dem Kollisionsmanagement hinzugefügt und zur Liste der aktiven Spawns m\_spawn hinzugefügt.

Während des Updates überprüft die Methode außerdem den Zustand aller aktiven Gegner. Gegner, deren Gesundheit auf null gefallen ist, werden aus der Liste entfernt, um unnötige Rechenlast zu vermeiden. Gleichzeitig werden die verbleibenden aktiven Gegner durch interne Aktualisierungen auf den neuesten Stand gebracht, sodass sie weiterhin interagieren und sich in der Spielwelt bewegen können.

Eine weitere wichtige Funktion der Klasse ist OnCollision. Diese Methode definiert das Verhalten des Spawners, wenn er mit anderen Objekten in der Spielwelt kollidiert. Wird der Spawner beispielsweise von einem Projektil getroffen, überprüft die Methode den Typ des Projektils. Handelt es sich um eine Wand\_bullet, wird die Gesundheit des Spawners reduziert, und das Projektil wird deaktiviert. Dieser Mechanismus ermöglicht es Spielern, den Spawner gezielt anzugreifen und ihn letztlich zu zerstören.

Die Darstellung des Spawners und seiner Gegner erfolgt durch die Methode Draw. Diese zeichnet den Spawner nur, wenn er aktiv ist und sich im Sichtbereich des Spielers befindet. Auch die aktiven Gegner werden gezeichnet, wobei deren Position und Status in Echtzeit berücksichtigt werden. Dadurch wird sichergestellt, dass nur relevante Elemente gerendert werden, was die Performance der Spielwelt optimiert.

Ein besonderer Vorteil der Spawner-Klasse liegt in ihrer Fähigkeit, dynamische Herausforderungen zu schaffen. Indem sie Gegner in festgelegten Intervallen und basierend auf einer vorgegebenen Warteschlange generiert, ermöglicht sie die Erzeugung von Gegnerwellen. Spieler können so durch ansteigende Schwierigkeitsgrade oder spezifische Missionsziele wie die Zerstörung eines Spawners gefordert werden.

## Gun:

Die Gun-Klasse erweitert die Funktionalität der Basisklasse Item und dient zur Verwaltung und Implementierung von Schusswaffenmechaniken in einem Spiel. Sie ermöglicht die Erzeugung und Bewegung von Projektilen (Kugeln) sowie deren Interaktion mit der Spielwelt. Die zentrale Aufgabe der Klasse besteht darin, das Schießen, die Positionsberechnung der Kugeln und die effiziente Verwaltung dieser Projektile zu steuern.

Beim Erstellen eines Gun-Objekts initialisiert der Konstruktor die grundlegenden Parameter wie die Textur-ID, Schriftart-ID und die Startposition. Diese Parameter werden an die Basisklasse Item weitergeleitet, um das Objekt in der Spielwelt zu platzieren und visuell darzustellen.

Die Update-Methode wird in jedem Spielzyklus aufgerufen und erfüllt zwei Hauptaufgaben. Erstens prüft sie in der Methode Shoot, ob die linke Maustaste gedrückt wurde und ein Schuss abgegeben werden kann. Wenn die Mindestzeit zwischen zwei Schüssen verstrichen ist, wird ein neues Projektil erstellt. Dieses wird durch die Spielerposition als Ausgangspunkt und die Mausposition als Zielpunkt definiert. Das Projektil wird anschließend zur Kollisionsverwaltung hinzugefügt, und sein Ziel wird über die Methode CreateBulletTarget festgelegt. Zweitens aktualisiert die Methode CalculatingBullets die Positionen der aktiven Kugeln, entfernt inaktive oder am Ziel angelangte Kugeln und sorgt so für eine effiziente Verwaltung.

Die mathematische Grundlage für die Bewegungsberechnung der Kugeln in CalculatingBullets basiert auf der Richtung, Geschwindigkeit und Zeitdifferenz. Die Bewegungsrichtung eines Projektils wird durch den Vektor berechnet, der vom Zielpunkt (T=(xtarget,ytarget)\mathbf{T} = (x\_{\text{target}}, y\_{\text{target}})T=(xtarget​,ytarget​)) zur aktuellen Position (P=(xcurrent,ycurrent)\mathbf{P} = (x\_{\text{current}}, y\_{\text{current}})P=(xcurrent​,ycurrent​)) führt:

D=T−P=(xtarget−xcurrent,ytarget−ycurrent)\mathbf{D} = \mathbf{T} - \mathbf{P} = (x\_{\text{target}} - x\_{\text{current}}, y\_{\text{target}} - y\_{\text{current}})D=T−P=(xtarget​−xcurrent​,ytarget​−ycurrent​)

Dieser Vektor wird anschließend normalisiert, um eine einheitliche Bewegungsrichtung zu erhalten, unabhängig von der Entfernung zum Ziel. Die Norm (∥D∥\|\mathbf{D}\|∥D∥) wird berechnet als:

∥D∥=(xtarget−xcurrent)2+(ytarget−ycurrent)2\|\mathbf{D}\| = \sqrt{(x\_{\text{target}} - x\_{\text{current}})^2 + (y\_{\text{target}} - y\_{\text{current}})^2}∥D∥=(xtarget​−xcurrent​)2+(ytarget​−ycurrent​)2​

Der normalisierte Richtungsvektor d\mathbf{d}d ergibt sich dann zu:

d=D∥D∥=(xtarget−xcurrent∥D∥,ytarget−ycurrent∥D∥)\mathbf{d} = \frac{\mathbf{D}}{\|\mathbf{D}\|} = \left( \frac{x\_{\text{target}} - x\_{\text{current}}}{\|\mathbf{D}\|}, \frac{y\_{\text{target}} - y\_{\text{current}}}{\|\mathbf{D}\|} \right)d=∥D∥D​=(∥D∥xtarget​−xcurrent​​,∥D∥ytarget​−ycurrent​​)

Die neue Position des Projektils wird schließlich berechnet, indem der normalisierte Richtungsvektor mit der Geschwindigkeit vvv und der verstrichenen Zeit Δt\Delta tΔt multipliziert und zur aktuellen Position addiert wird:

Pneu=P+d⋅v⋅Δt\mathbf{P}\_{\text{neu}} = \mathbf{P} + \mathbf{d} \cdot v \cdot \Delta tPneu​=P+d⋅v⋅Δt

Diese iterative Berechnung sorgt dafür, dass sich die Kugeln kontinuierlich und gleichmäßig in Richtung ihres Ziels bewegen.

Zusätzlich prüft die Methode CalculatingBullets, ob ein Projektil deaktiviert ist oder sein Ziel erreicht hat. Inaktive Projektile werden aus der Liste entfernt, indem sie mit dem letzten Element der Liste getauscht und anschließend gelöscht werden. Kugeln, die ihr Ziel erreicht haben, werden ebenfalls entfernt, sobald ihre Kollisionsbox den Zielpunkt enthält.

Die Methode Draw rendert alle aktiven Projektile auf das Spielfenster. Sie stellt sicher, dass nur relevante Objekte sichtbar sind und dabei die Performance der Spielwelt erhalten bleibt. Für Kollisionen mit anderen Objekten, wie etwa Projektiltreffern, wird die Methode OnCollision verwendet. Trifft ein Projektil auf ein Hindernis, wird dessen Zustand entsprechend angepasst.