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 ihres Funktionsgraphen. Dadurch sind