Wilhelm-Maybach-Schule Heilbronn

Projekt „Conqueror“

Geschichtliche Ausarbeitungen und Projektdokumentation

CONQUEROR

im Rahmen des Seminarkurses

„1. Weltkrieg“

Gruppenteilnehmer:

Angelina Heller, Levi Lauer, Matthias Geng, Pherel Fazliu, Pascal Gutsche

Mentoren:

Doktor phil. Steffen Lesle, Nadja Schönfeld, Alexander Racic

Inhaltsverzeichnis

Conqueror:

Wir haben uns im Sommer 2022 zusammengesetzt und haben beschlossen, dass wir ein Spiel machen wollten. Damit man ein Spiel machen kann, braucht man eine „Engine“, die sich um die „lästigen Dinge drumherum kümmert“. Dazu gehört auch die Kommunikation mit der Grafikkarte, Tastatur und auch mit der CPU auf einem ganz anderen Effizienzniveau.

Da Pascal und Pherel sehr daran interessiert waren, haben sie sich entschlossen eine eigene Engine zu schreiben. Anfangs gab es sehr viele Probleme, da nur sehr wenig Wissen im Bereich C++ vorhanden war. Es waren lediglich Konzepte aus Java und C bekannt. Die Sprache war nach ein paar Monaten kein Problem mehr. Was zum größeren Problem wurde, war die Kommunikation mit der Grafikkarte, denn dort mussten sie sich einlesen und viele mathematische Konzepte nachvollziehen, um dann erfolgreich geometrische Formen auf dem Bildschirm anzuzeigen.

Bis dann aber das Game-Team, welches aus Levi, Matthias und Angelina besteht, das Spiel umsetzen konnte, vergingen weitere Monate, da das Core-Team die Engine sehr effizient gemacht und erweitert hat.

Die Engine selbst ermöglicht den Game-Team ein Spiel mit Leichtigkeit zu erstellen. Die Umfangreiche Komplexität der vorhandenen Spiel-Engines auf dem Markt, wie beispielsweise Unity oder Unreal Engine waren nicht für diese Arbeit gedacht.

Eine Engine muss in der Lage sein, dem Benutzer die Möglichkeit zu geben, sehr leicht auf die Tastatur zuzugreifen, und auch Objekte auf dem Bildschirm zu erzeugen. Diese Objekte können divers koloriert werden, und auch Texturen, beinhalten. Außerdem werden Objekte auf dem Bild andauernd verschoben und die Farben werden zu jedem Zeitpunkt geändert. Die Engine des Core-Teams muss damit klarkommen. Diese Herausforderung wurde angenommen und gemeistert.

Planung:

Planungsstart und Grobe Idee

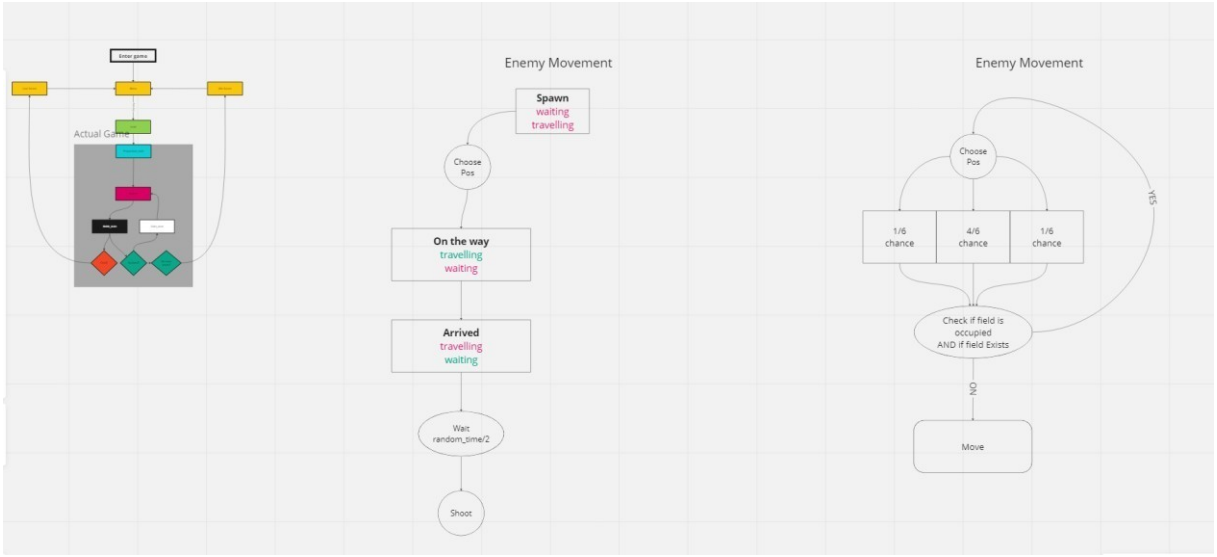
Im Dezember 2022 Sollte mit der Planung des Spiels begonnen werden. Da Pherel und Pascal mit dem Erstellen der Engine beschäftigt waren, und Levi und Matthias beide bereits Erfahrung im Erstellen von Spielen hatten, übernahmen sie in Enger Zusammenarbeit mit Angelina, welche für die Grafische Darstellung zuständig und somit auch wichtig für die Planung war.
Die Erste Idee an der Gearbeitet wurde war ein Spiel des „grand strategy“ Spiele genere das üblich für Kriegsspiele wie Hearts of Iron IV oder Europa Universalis IV. Geplant war also eine Große Europakarte und Strategie auf grobem Niveau. Diese Idee musste jedoch relativ früh aufgegeben werden, da es schlichtweg zu wenig Zeit gab, um ein solches Spiel mit einer eigenen Engine und einem relativ kleinen Team zu realisieren.
Die Zweite Idee, an der wir auch festhielten, war die eines reinen Verteidigungsspiels. Das Ziel sollte sein so lange wie möglich gegen Wellen von Gegnern zu überleben und dabei eigene Truppen zu steuern. Dies ermöglichte einen Bezug zum 1. Weltkrieg die Möglichkeit das Projekt in der Vorgegebenen Zeit fertig zu stellen.

Planungswerkzeuge

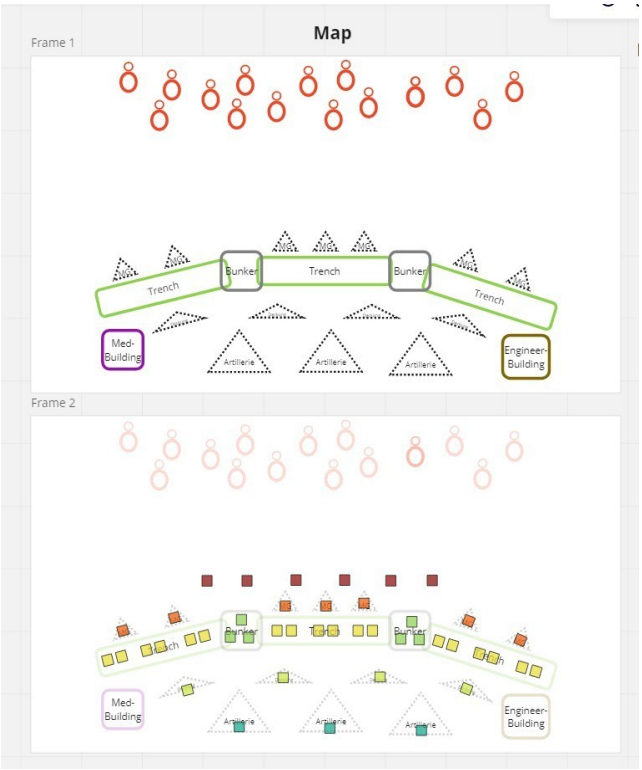
Da die Planung des Projektes nicht von einem allein vollzogen wurde, und wir die Ideen, die wir umsetzten, wollten auch für die nicht mitplanenden visualisieren wollten, musste ein Tool zur Planung her. Nachdem viele Optionen angeschaut wurden, haben wir uns für das online Tool „Miro“ entschieden (miro.com). Diese Website ermöglichte uns kostenfrei und in zeitgleicher Zusammenarbeit das Projekt zu planen, visualisieren und auch für längere Zeit die Ideen zu speichern. Mirko an sich ist ein relativ einfaches Programm, mit dem man Objekte zeichnen, texte Schreiben und Objekte verbinden kann. Dies ermöglichte uns jedoch eine übersichtliche Planung der Spielmechaniken zu machen, die zwar nicht eins zu eins am ende so übernommen wurde, jedoch ein Start bietet.

Grundidee Planung

Die genauere Planung des Projekts erfolgte zwar teils auch noch während des erstellen des Spiels, jedoch sollte zumindest eine Anfangsrichtung festgelegt werden und eine Basis, auf der man aufbauen kann. Die Idee selbst stand recht schnell fest, da sowohl die Grundidee als auch das Spielziel / Spielprinzip nicht sehr komplex waren.

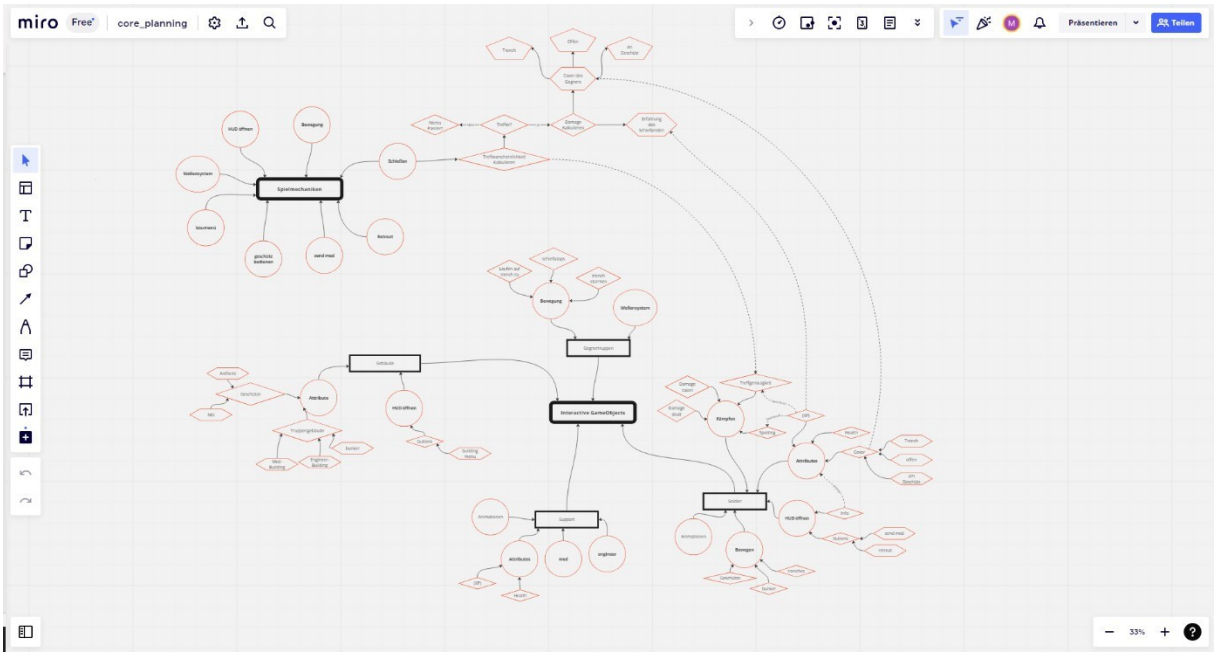


Die Karte und der Grobe Ablauf des Spiels sollten Erstmal nur Anhaltspunkte zum Orientieren sein, an denen man sich ein wenig Festhalten kann. Diese Mechaniken wurden auch erstmal so ins Spiel übernommen, danach jedoch nach Bedarf ein wenig bearbeitet.



Planung der Spielmechaniken

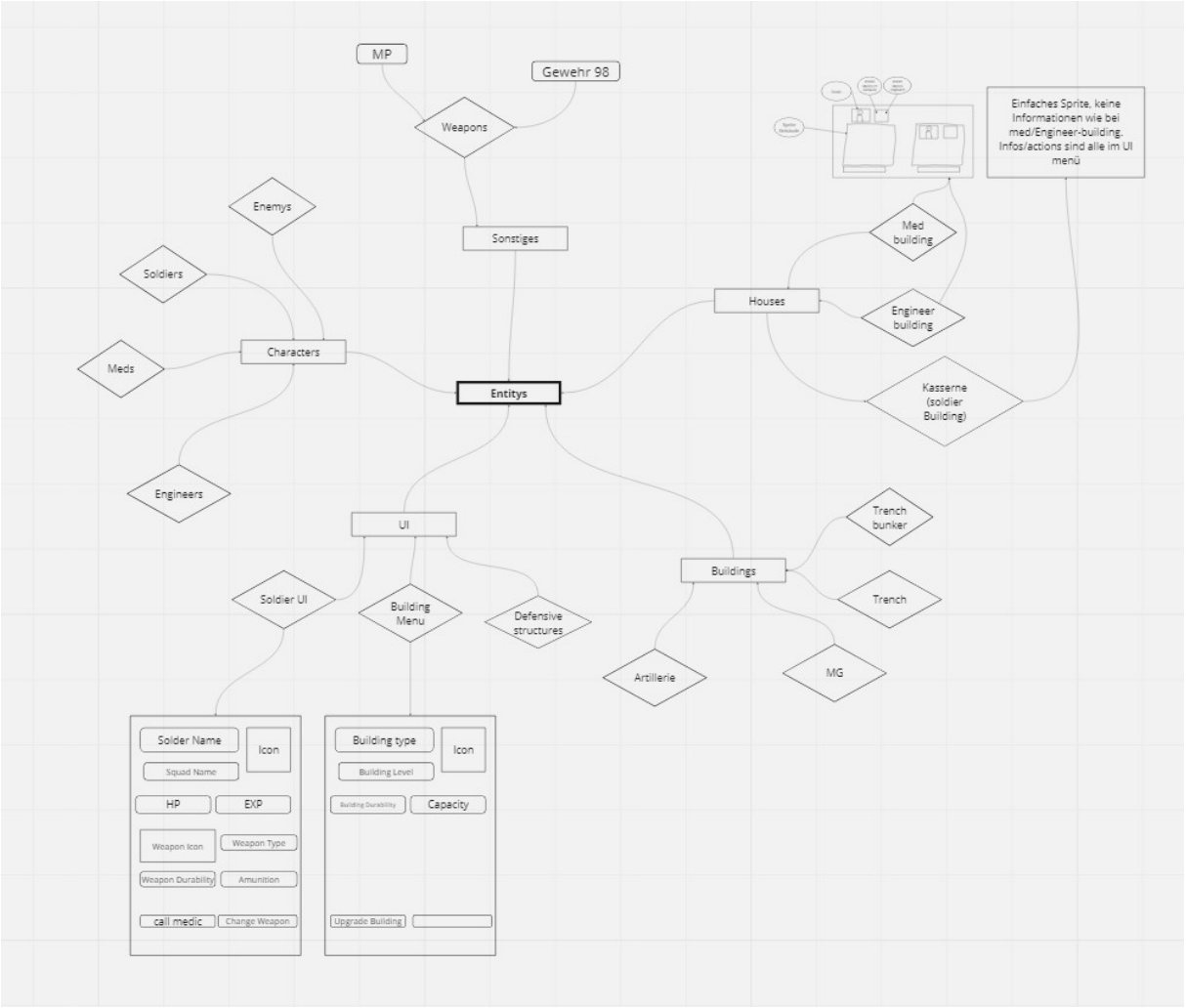
Da nun also die ersten Ansätze des Spiels klar waren, musste man sich über die Spielmechaniken Gedanken machen. Hier war vor allem wichtig, dass man weiß welche Mechaniken und welche Game Objekte miteinander funktionieren, oftmals wiederverwendet werden und welche Game Objekte was können sollen.



Die Grafik passte hier sehr gut zur späteren Umsetzung, da wir mit game components gearbeitet haben, welche eine Mechanik darstellten und zu den jeweiligen Objekten zugeordnet wurden.

Entity Planung

Zuletzt mussten nun noch die einzelnen Objekte genauer zugeordnet werden um ein wenig Übersicht zu schaffen.



Dies sollte nur eine grobe Übersicht geben, um unteranderem den Umfang des Spiels widerzuspiegeln.

Planung der Gebrauchten Grafiken

Dies war mit der Schwierigste teil der Planung. Die Grafiken, beziehungsweise Sprites, waren nicht von Anfang an benötigt und auch nicht direkt testbar, da erst am Grundprinzip des Spiels gearbeitet wurde. Anfänglich gab es zwar einige Ideen, die auch umgesetzt und von Angelina angefertigt wurden, jedoch führte das leider zu ein paar unbrauchbaren Texturen oder Texturen die später überarbeitet werden mussten. Hauptsächlich wurde der Gebrauch einer Textur während des Erstellens des Spiels festgestellt, da einige Objekte oder einfach der gebrauch einiger Texturen anfangs nicht klar war. Hier muss man ein großes Lob an Angelina

aussprechen, die alle Anfragen mit wenig Problemen sehr schnell fertigstellen konnte, und jegliche Fehler in kurzer Zeit behob.

Grafikimplementierung

Die Grafiken wurden ausschließlich von Angelina erstellt, jedoch brauchte es jemanden der diese ins Spiel implementiert. Dank der Engine von Pherel und Pascal war das für einzelne Sprites und Bilder kein großes Problem, da diese mit wenig Mühe eingefügt werden konnten. Das Größere Problem lag bei der Nutzung von Sprite Sheets, welche jedoch wichtig für Animationen waren.

Sprite Sheets

Ein Sprite Sheet ist eine Sammlung von Sprit in einer einzelnen Datei. Diese musste vom Programm nun aufgeteilt und nummeriert werden, um des einzelnen Sprites einzeln auszuwählen. Beispielsweise musste die Laufanimation des Untenstehenden Charakters in gleichgroße Abschnitte ausgeschnitten werden, die jedoch von Sheet zu Sheet anders sind.

Dies alles ermöglichte eine Rechnung, die mit den OpenGL texcoords ein Teil eines Bildes ausschneiden kann.

```
Sprite::Sprite(glm::vec4 color, Shr<Texture> texture, float spriteWidth, float
spriteHeight, float paddingWidth, float paddingHeight, glm::vec2 selectedSprite,
bool registerAlphaPixelsToEvent)
	{
		this->textureWidth = texture->GetWidth();
		this->textureHeight = texture->GetHeight();
		this->color = color;
		this->texture = texture;

		// create texcoords
		texcoords[3] = { ((spriteWidth + paddingWidth) * selectedSprite.x) /
textureWidth, (textureHeight - ((spriteHeight + paddingHeight) *
selectedSprite.y)) / textureHeight };
		texcoords[2] = { ((spriteWidth + paddingWidth) * (selectedSprite.x +
1)) / textureWidth, (textureHeight - ((spriteHeight + paddingHeight) *
selectedSprite.y)) / textureHeight };
		texcoords[0] = { ((spriteWidth + paddingWidth) * selectedSprite.x) /
textureWidth, (textureHeight - ((spriteHeight + paddingHeight) *
(selectedSprite.y + 1))) / textureHeight };
		texcoords[1] = { (spriteWidth + paddingWidth) * (selectedSprite.x +
1) / textureWidth, (textureHeight - ((spriteHeight + paddingHeight) *
(selectedSprite.y + 1))) / textureHeight };
	}
```

Die Koordinaten sind hierbei die ecken der einzelnen Sprite Kasten, dessen Größe in den Variablen spriteWidth und spriteHeight angegeben werden. Der Vector selectedSprite beinhaltet eine x und y Koordinate, welches das aktuell ausgewählte Sprite angibt. Dies braucht man, da diese Funktion in einem Loop ist, und so lange durchgeführt wird, bis kein Platz mehr für eine weitere Texturbox existiert.

```
SpriteSheet::SpriteSheet(glm::vec4 color, Shr<Texture> texture, float
spriteWidth, float spriteHeight, float paddingWidth, float paddingHeight,
glm::vec2 selectedSprite, bool registerAlphaPixelsToEvent)
		: color(color), texture(texture), spriteWidth(spriteWidth), sprite-
Height(spriteHeight),
		paddingWidth(paddingWidth), paddingHeight(paddingHeight), regis-
terAlphaPixelsToEvent(registerAlphaPixelsToEvent)
	{
		Init(texture, selectedSprite);
	}

void SpriteSheet::Init(Shr<Texture> texture, glm::vec2 selectedSprite)
	{
		this->textureWidth = texture->GetWidth();
		this->textureHeight = texture->GetHeight();
		this->spriteColumns = textureWidth / this->spriteWidth;
		this->spriteRows = textureHeight / this->spriteHeight;

		// sprites are inverted, because of the opengl and stbi axis
conflict
		this->selectedSprite.x = spriteColumns - selectedSprite.x;
		this->selectedSprite.y = spriteRows - selectedSprite.y;

		for (int row = 0; row <= spriteRows; row++)
		{
			std::vector<Shr<Sprite>> rowvec;
			for (int column = 0; column <= spriteColumns; column++)
			{
				rowvec.push_back(MakeShr<Sprite>(color, texture,
spriteWidth, spriteHeight, paddingWidth, paddingHeight, glm::vec2(column, row),
false));
			}
			Sprites.push_back(rowvec);
		}
		ChangeCoords();
	}
```

Diese Funktion ist die erste Funktion, die nach dem Konstruktor für das Sprite Sheet Objekt aufgerufen wird. Die ersten vier Zeilen in der Init Funktion berechnen die Anzahl an Texturboxen in einem Sprite Sheet, indem sie die komplette breite der Textur durch die breite eines Sprites teilen. Der for loop darunter springt nun jedes Mal in die vorherige Sprite Funktion und verändert nur den selectedSprite Vector, welcher dort mit den Texturbreiten multipliziert wird. Dadurch wird beim 1. Sprite der linke obere texcoord *0 genommen, was bedeutet, dass er bei dem Pixel 0,0 startet und somit in der komplett linken oberen ecke ist. Die Variablen an den jeweilig anderen punkten bekommen immer ein Inkrement auf den selectedSprite Vector an der benötigten Koordinate, abhängig davon, ob sie oben rechts, unten links oder unten rechts sind.

Animationen

Laufanimation

Die Laufanimation war die erste Animation im Spiel. Sie nutzte die Sprite Sheets und deren durchnummerierte Boxen um durch die einzelnen Sprites nach Bedarf durchzugehen.

```
void WalkingAnimation::Start(glm::vec2 indexStart, glm::vec2 indexEnd) {
   int animationLength = indexEnd.x - indexStart.x + 1;
   static int tex = 0;
   static float x = 0.01f;
   x += Application::GetDT() / animationSpeed;

   tex = (int)(x * 100);
   LOG_DEBUG(tex);
   if (tex % (animationLength + 1) == 0)
   {
      x = 0.01f;
      tex = (int)(x * 100);
   }
   gameObject->GetComponent<SpriteSheet>()->ChangeSprite(glm::vec2(tex +
indexStart.x - 1, indexStart.y));
}
```

Das einzige Problem hierbei war das Übliche Problem der Frame Time, was häufig bei Animationen auftritt. Dies ist jedoch mit einer Multiplikation der Deltatime umsetzbar, welche die zeit zwischen den einzelnen Frames darstellt.

Ein weiterer Teil der Laufanimation war auch die Richtung eines laufenden Objektes zu bestimmen.

```
void WalkingAnimation::CalculateDirection() {
   glm::vec2 direction = gameObject->GetComponent<Movement>()->GetDirection();
// Example direction vector

   float angle = std::atan2(direction.y, direction.x); // Calculate the angle in
radians

   // Convert the angle to degrees and shift it to the positive range [0, 360)
   float degrees = std::fmod((std::fmod(angle, 2 * glm::pi<float>()) + 2 *
glm::pi<float>()), (2 * glm::pi<float>())) * (180.0f / glm::pi<float>());

   std::string directionString;


   // Determine the direction based on the angle
   if (degrees > 45 && degrees <= 135)
      Start(indexStartUp, indexEndUp);
   else if (degrees > 135 && degrees <= 225)
      Start(indexStartLeft, indexEndLeft);
   else if (degrees > 225 && degrees <= 315)
      Start(indexStartDown, indexEndDown);
   else
      Start(indexStartRight, indexEndRight);


}
```

Die variable degrees wird einer Rechnung gleichgesetzt, die den direction Vector, welcher die Richtung eines Laufenden Objektes beinhaltet, in grad Zahlen umrechnet. Anschließend ist es nur noch wichtig die Animationsrechnung in der richtigen Richtung auszuwählen.

All dies wurde durch einen Konstruktor gesteuert, der für jedes Objekt die Indizes der Animationen speicherte.

```
character->AddComponent(new WalkingAnimation(glm::vec2(3.0f, 1.0f), glm::vec2(5.0f,
1.0f), glm::vec2(3.0f, 0.0f), glm::vec2(5.0f, 0.0f), glm::vec2(0.0f, 0.0f),
glm::vec2(2.0f, 0.0f), glm::vec2(0.0f, 1.0f), glm::vec2(2.0f, 1.0f), 32,
glm::vec2(3.0f, 1.0f), false));
```

```
WalkingAnimation::WalkingAnimation(glm::vec2 indexStartUp, glm::vec2 indexEndUp,
glm::vec2 indexStartDown, glm::vec2 indexEndDown, glm::vec2 indexStartRight,
glm::vec2 indexEndRight, glm::vec2 indexStartLeft, glm::vec2 indexEndLeft, int
animationSpeed, glm::vec2 standartPosition, bool hasSingleAnimation)
```

Restliche Animationen

Es gab nun ein Problem, da wir nicht nur Laufanimationen hatten, sondern auch Animationen wie das Bauen von einem Gebäude durch einen Ingenieur, oder auch das Explodieren eines Artilleriegeschosses, musste noch ein weiterer Component her. So kam es zum SingleAnimationComponent. Dieser war sowohl dazu gut, um eine Animation für nur einen Zyklus abzuspielen, als auch eine Animation mehrmals durchlaufen zu lassen. Diese nutzte eine etwas andere Rechnung für das Durchlaufen der Animation, die aber ähnlich funktionierte.

```
void SingleAnimation::OnUpdate() {
   if (oneCycle == true) {
      int animationLength = indexEnd.x - indexStart.x + 1;
      static int tey = 1;
      static float y = 0.01f;
      y += Application::GetDT();
      if (y > animationSpeed) {
         tey += 1;
         y -= animationSpeed;
         gameObject->GetComponent<SpriteSheet>()->ChangeSprite(glm::vec2(tey +
indexStart.x - 1, indexStart.y));
      }

      if (tey % (animationLength + 1) == 0)
      {
         oneCycle = false;
         StopAnimation();
         y = 0.0f;
         tey = 1;
      }
   }
}
```

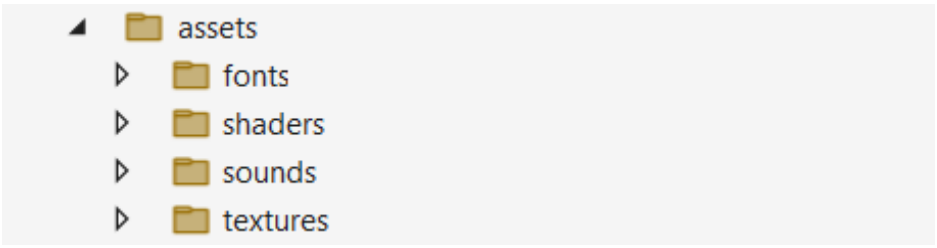
In diesem Fall wird y immer mit der Delta time erhöht und sobald es einen Frame hat, welcher den gewünschten wert überschreitet, wurde das nächste Sprite aufgerufen. Dies machte das Kontrollieren der Geschwindigkeit um einiges einfacher, da man nun einen Sekundenwert festlegen konnte.

Spiel-Code

Einleitung

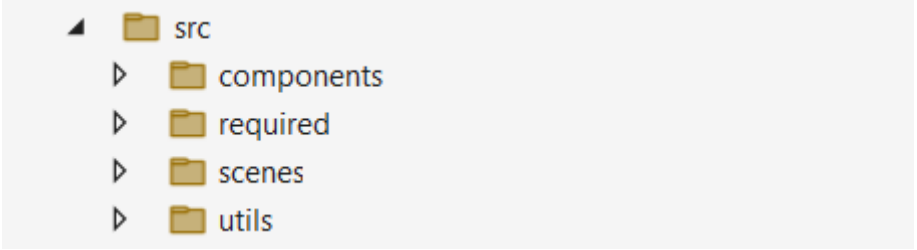
Im Folgenden wird die grundlegende Game-Engine Logik und interne Funktionsweise mitsamt dem Code erklärt, um einen möglichst guten Einblick in die Entwicklung des Spiels zu schaffen. Das Spiel an sich, die Entwicklung und Planung sowie der historische Hintergrund wird von Matthias und Levi erklärt und die Game-Engine an sich von Pascal und Pherel. Ein Teil der Engine-Logik ist aber auch für die Verständlichkeit des Spiel-Codes und -funktionalität nötig, der hier erläutert wird. Die Texturen

Interne Projektstruktur

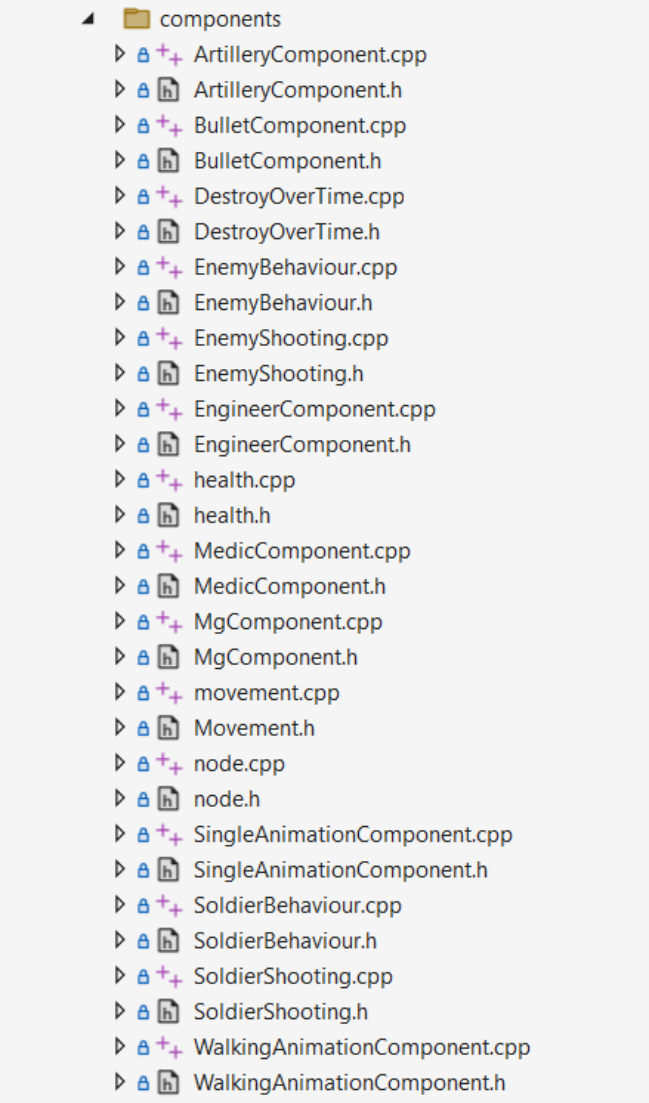


Da es mehrere Klassen und Dateien mit unterschiedlichen Funktionsweisen der Engine in unserem Spiel gibt, haben wir diese aufgeteilt. In unserer Visual-Studio Projektmappe befinden sich unter anderem der Ordner „conquerer“, in dem sich alle Dateien für das Spiel befinden. In diesem befinden sich wiederum die Ordner „assets“ und „src“. In „assets“ sind Dinge wie Texturen, Shader, Sounds oder Schriftarten gespeichert.

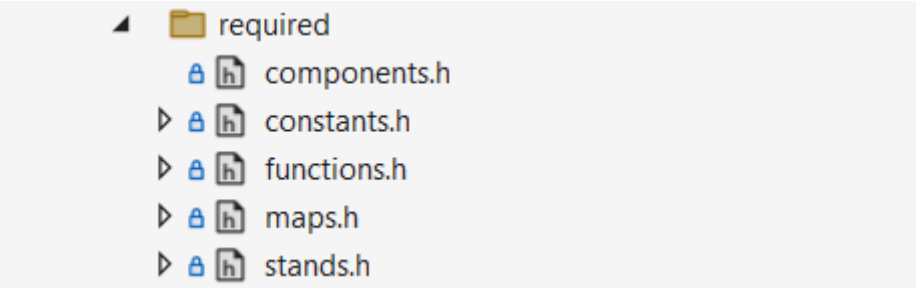
In „src“ sind alle C++ und Header Dateien, die für das Spiel an sich gebraucht werden, sprich der gesamte Code des Spiels, aufgeteilt in die Components („components“), Dinge wie Spielvariablen oder nützliche Funktionen („required“), Szenen („scenes“) und Auslagerungen von bestimmtem Code, der sonst nirgends wirklich hineingepasst hat oder sich für eine Auslagerung eignet („utils“).



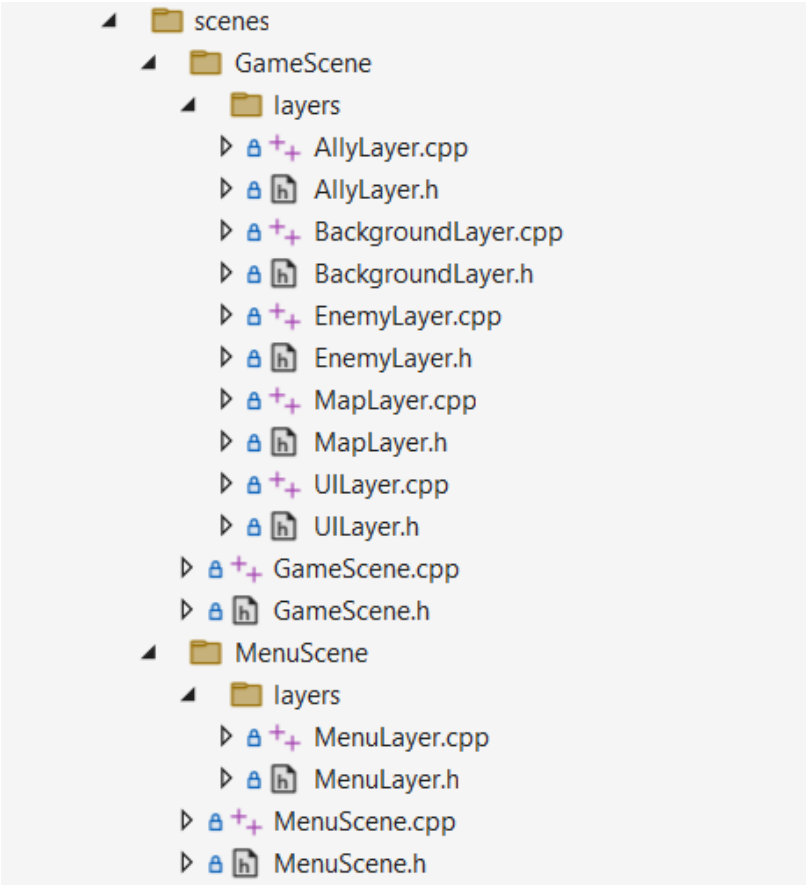
Der „components“-Ordner:



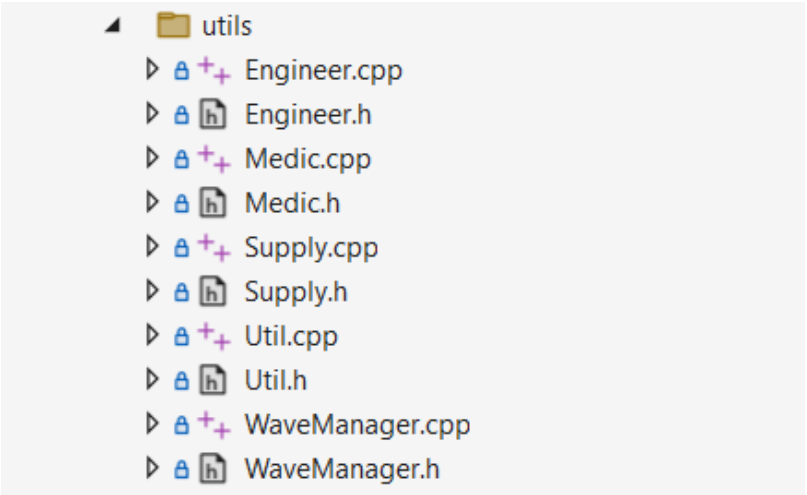
Der „required“-Ordner:



Der „scenes“-Ordner:



Der „utils“-Ordner



Erklärung der hierfür wichtigen Engine-Funktionen

Um meine folgenden Erklärungen richtig verstehen zu können, muss man gleichzeitig auch die wichtigsten Funktionen und Klassen der Engine-Funktionalität verstehen. Das sind z.B. die Components und deren OnStart(), OnStop() und OnUpdate() Funktionen, die die Hauptlogik des Spiels repräsentieren. Die OnStart() Funktion wird sofort nach dem Erstellen eines Components aufgerufen und eignet sich somit, bestimmte Variablen zu initialisieren oder z.B. eine Start-Position eines GameObjects festzulegen. Im Gegenteil dazu steht die OnStop() Funktion, die nach dem Entfernen eines Components als dessen letzte Funktion ausgeführt wird, was in den meisten Fällen beim Löschen eines GameObjects auftritt. In dieser Funktion bietet sich es an, z.B. einen Ton nach dem Sterben eines Soldaten abzuspielen oder etwas aus einer Liste zu entfernen, da es sonst wahrscheinlich zu Problemen kommt. Am wichtigsten ist letztendlich aber OnUpdate(), die im Gegensatz zu den anderen beiden Funktionen jeden einzelnen Frame ausgeführt wird, was heißt, dass wenn man im Spiel 60FPS hat, diese Funktion genau 60 mal pro Sekunde durchlaufen wird. In ihr befindet sich der Code, der das Verhalten eines Components steuert.
Ein ebenfalls wichtiges Konzept sind die Layers, denn sie beinhalten die GameObjects, die ihnen „zugeteilt“ wurden. Sie werden dafür gebraucht, um die Reihenfolge festzulegen, in dem alle GameObjects gerendert werden. So kann man z.B. eine BackgroundLayer erstellen, dort alle GameObjects zuweisen, die später im Hintergrund sein sollen und diese BackgroundLayer einer Scene zuweisen (was ebenfalls im Folgenden erklärt wird). Damit jetzt andere GameObjects über diesem Hintergrund angezeigt werden, kann man z.B. noch eine ForegroundLayer erstellen und diese in einer Scene nach der BackgroundLayer hinzufügen, wodurch sie jeden Frame nach der BackgroundLayer gerendert wird. Dies ist nur ein Beispiel, um das Konzept zuverständlichen, man kann also so viele Layers erstellen wie nötig und diese unterschiedlich benennen.
Nun bleiben noch Scenes, die man sich ähnlich vorstellen kann wie Layers, nur dass Scenes einfach gesagt Layers beinhalten statt GameObjects. Außerdem haben Scenes auch eine OnStart(), OnStop() und OnUpdate() Funktion, von denen man Gebrauch machen kann. In unserem Fall haben wird dort z.B. die Kamerasteuerung implementiert da es sich nicht angeboten hat, dafür einen Component zu erstellen. Scenes kann man sich eine Art kleinere Instanzen des Spiels vorstellen, von denen immer eine ausgeführt werden kann. Beispiele dafür

sind ein Hauptmenü, das Spiel an sich oder ein GameOver-Menü. Sie können intern im Code geladen werden, z.B. dass wenn man den „Start“-Knopf drückt, die SpielScene geladen wird.

Components

Movement

Der Movement-Component ist für die gesamten Bewegungen im Spiel zuständig und somit eine der Grundlagen, weswegen ich diesen als erstes entwickelt habe. Im Laufe der Entwicklung wurde dieser wie alle anderen ein paar Mal verändert bzw. umgeschrieben. Dieser Component wird so benutzt, dass ein anderer Component desselben GameObjects eine der zwei Funktionen SetTrackingPos() oder SetTargetPos() aufruft, um sein eigenes Ziel zu setzen. Daraufhin bewegt sich das GameObject entweder (wenn SetTargetPos() aufgerufen wird) zu einem festen Punkt, oder folgt seinem Zielobjekt dauerhaft (wenn SetTrackingPos() aufgerufen wird). Probleme gab es anfangs mit den float-werten (Fließkommazahl Werte), da dort bei den Nachkommastellen immer bestimmte Abweichungen bzw. Ungenauigkeiten auftreten. Dadurch sind die GameObjects immer leicht in eine Richtung gedriftet und nicht an ihren Zielkoordinaten angekommen, was wir mithilfe von Rundung gelöst haben. Gerundet wird hier mit der Funktion RoundVec2(), die sich in einer anderen Datei befindet, welche später noch erklärt wird.

```
static bool CoordRoundVec2(glm::vec2 targetPos, glm::vec2 pos, float speed)
{
	glm::vec2 diff = { targetPos.x - pos.x, targetPos.y - pos.y };
	if (abs(diff.x) < speed /	&& abs(diff.y) < speed /	) return true;
	return false;
}

void Movement::OnUpdate() {


	if (tracking_position != nullptr)
	{
		target_position = *tracking_position;
		move = true;
	}

	if (CoordRoundVec2(target_position, gameObject->transform.position,
movement_speed))
	{
		gameObject->transform.position = target_position;
		move = false;
		isArrived = true;
	}

	if (move)
```

```
                MoveTo(target_position, movement_speed);
}

void Movement::SetTrackingPos(glm::vec2* pos)
{
        tracking_position = pos;
        isArrived = false;
}

void Movement::SetTargetPos(glm::vec2 pos)
{
        target_position = pos;
        tracking_position = nullptr;
        move = true;
        isArrived = false;
}

void Movement::MoveTo(glm::vec2 target_pos, float speed) {

        direction = target_pos - gameObject->transform.position;  // calculate

        gameObject->transform.position += direction * speed * Application::GetDT();

        RoundVec2(gameObject->transform.position);
}
```

Node

Da man im Spiel bestimmte Ansatzpunkte braucht, wo man seine Charaktere hinbewegen kann, werden hier sogenannte Nodes (Felder) verwendet. Sie werden als Quadrate auf dem Schlachtfeld dargestellt, die man anklicken kann und deren Position so als Ziel des Movement-Components festgelegt wird. Außerdem sollen alle Felder einer von mehreren Stellungen angehören, die später auch noch genannt werden. Von diesen Stellungen hängt z.B. die Trefferwahrscheinlichkeit der Gegner ab. Sehr wichtig ist ebenfalls, dass nicht mehrere Charaktere einen Node besetzen können, was hier mit dem Attribut is_occpuied geregelt wird.

```
Node::Node(std::vector<GameObject*>* stand)
        :stand(stand)
{
        is_occupied = false;
        contains_soldier = false;
}

void Node::OnStop() {
        if (stand == trench_stand.stand) {
                for (size_t i =   ; i < trench_nodes.size(); i++) {
                        if (trench_nodes.at(i) == gameObject) {
                                trench_nodes.erase(trench_nodes.begin() + i);
                                break;
                        }
                }
        }
        else if (stand == hiding_stand.stand) {
```

```
		for (size_t i =   ; i < hiding_nodes.size(); i++) {
			if (hiding_nodes.at(i) == gameObject) {
				hiding_nodes.erase(hiding_nodes.begin() + i);
				break;
			}
		}
	}
}
```

EnemyBehaviour

Wie man am Namen wahrscheinlich schon erkennen kann, ist dieser Component für das Verhalten der Gegner zuständig. Dieses besteht aus: zu einem Punkt im EnemyGrid laufen (wird später noch erklärt); Nach der Ankunft eine regulierte zufällige Zeit warten; Auf einen Charakter schießen.

```
void EnemyBehaviour::OnStart() {
	move_component = gameObject->GetComponent<Movement>();

	dt_time_counter =      ;

	is_waiting = false;
	time_over = true;
	time_running = false;

	TryLeaveSpawnPos();
}

void EnemyBehaviour::OnUpdate() {
	if (onSpawnPos)
	{
		TryLeaveSpawnPos();
	}
	// handles the complete movement and shoot behaviour of the enemies
	if (is_waiting) {
		dt_time_counter += Application::GetDT();
		if (time_over) {
			is_waiting = false;
			time_running = false;


			if (y_index != enemy_grid_y -   ) {
				ChoosePosAndMove();
			}

		}
		else {
			if (!time_running) {
				time_over = false;
				time_running = true;

				// random calculation of time to wait
				time_to_wait = RandomF(min_enemy_waiting_time,
max_enemy_waiting_time) * game_time_factor;
				//LOG_DEBUG("time_to_wait: {0}", time_to_wait);
```

```
                        }
                        else {
                                if (dt_time_counter >= time_to_wait) {
                                        time_over = true;
                                        dt_time_counter =       ;

                                        gameObject->GetComponent<EnemyShooting>()-
>Shoot();
                                }
                        }
                }
        }
        else {
                if (gameObject->transform.position == move_component-
>GetTargetPos()) {
                        is_waiting = true;
                        time_over = false;
                }
        }

}


void EnemyBehaviour::ChoosePosAndMove() {

        // first calculate random number between 0 and enemy_random_movement_sum to
select the position the enemy should move to, based on their probabilities
        uint32_t random = RandomInt(   , enemy_random_movement_sum);

        // if he is at the bottom of the grid, do nothing
        if (y_index == enemy_grid_y -   ) return;

        GameObject* move_node_go;

        int32_t x_offset =   ;
        int32_t y_offset;

        // calculates the probabilities for each move-possibility and sets the
move_node_go to the randomly selected node
        if (random <= enemy_move_left_probability && x_index !=   ) {
                move_node_go = enemy_grid.at(x_index -   ).at(y_index +   );
                x_offset = -  ;
                y_offset =   ;
        }
        else if (random <= enemy_move_left_probability +
enemy_move_right_probability && x_index != enemy_grid_x -   ) {
                move_node_go = enemy_grid.at(x_index +   ).at(y_index +   );
                x_offset =   ;
                y_offset =   ;
        }
        else{
                move_node_go = enemy_grid.at(x_index).at(y_index +   );
                y_offset =   ;
        }

        // if this node is already occupied, do nothing
        if (move_node_go->GetComponent<Node>()->is_occupied) return;
        enemy_grid.at(x_index).at(y_index)->GetComponent<Node>()->is_occupied =
false;
        // removes the gameobject from the array
        for (size_t i =   ; i < enemy_grid_x; i++) {
                if (enemy_stands[y_index][i] == gameObject) {
```

```
                        enemy_stands[y_index][i] = nullptr;
                }
        }


        x_index += x_offset;
        y_index += y_offset;

        // now move to the selected node and occupy it
        move_component->SetTrackingPos(&move_node_go->transform.position);
        enemy_grid.at(x_index).at(y_index)->GetComponent<Node>()->is_occupied =
true;
        enemy_stands[y_index][x_index] = gameObject;
        node = move_node_go;

}

void EnemyBehaviour::TryLeaveSpawnPos()
{
        x_index = RandomInt(  , enemy_grid_x -   );
        y_index =   ;

        GameObject* move_node_go = enemy_grid.at(x_index).at(y_index);
        if (move_node_go->GetComponent<Node>()->is_occupied) {
                onSpawnPos = true;
                return;
        }
        move_component->SetTargetPos(enemy_grid.at(x_index).at(y_index)-
>transform.position);

        enemy_stands[y_index][x_index] = gameObject;
        move_node_go->GetComponent<Node>()->is_occupied = true;
        onSpawnPos = false;
        node = move_node_go;
}
```

SoldierBehaviour

Der SoldierBehaviour-Component steuert ähnlich wie der EnemyBehaviour-Component das Verhalten seines GameObjects. Dieser funktioniert aber ein wenig anders wie der des Gegners, da Soldaten nicht von alleine laufen sollen, sondern nur durch eine Aktion des Benutzers, was in diesem Fall ein Klick auf den Soldaten den man bewegen will mit einem anschließendem Klick auf eine Feld ist. Gleich bzw. nur leicht verändert wie bei den Gegnern ist das Schussverhalten. Wenn ein Soldat auf einem geeigneten Feld steht, sucht er sich erst ein Ziel aus und schießt dann in bestimmten Zeitabständen, die auch reguliert zufällig bestimmt werden, bis dieses tot ist.

```
void SoldierBehaviour::OnStart() {

        // start configuration
        on_spawn_pos = true;
        travelling = false;
        gets_healed = false;
```

```
	time_to_wait = RandomF(min_soldier_shoot_waiting_time,
max_soldier_shoot_waiting_time) * game_time_factor;
	dt_counter =	;
}

void SoldierBehaviour::OnStop() {
	// delete the gameobject from the recent stand
	if (!stand) return;
	for (size_t i =	; i < stand->size(); i++) {
		if (stand->at(i) == gameObject) {
			stand->erase(stand->begin() + i);
			break;
		}
	}
}

void SoldierBehaviour::OnUpdate() {

	if (!can_shoot) return;

	// if he's not walking
	if (!travelling) {

		// if he is not on the waiting stand
		if (this->stand != waiting_stand.stand) {

			// if time is not over => increase dt-time-counter by dt
			if (dt_counter < time_to_wait) {
				dt_counter += Application::GetDT();
				//ld("dt_counter: {0} ; time_to_wait: {1}", dt_counter,
time_to_wait);
			}

			else {
				if (on_spawn_pos) {
					if (SoldierTryMoveToWaitingNode()) {
						on_spawn_pos = false;
					}
					else {
						RestartTimer();
					}
				}
				else if(this->stand != bunker_stand.stand){
					gameObject->GetComponent<SoldierShooting>()-
>Shoot();
					RestartTimer();
				}

			}
		}
	}

	// if he arrived at the new node => the stand the node it's in get's
extended by the gameobject
	else {
		if (gameObject->transform.position == target_position) {
			travelling = false;
			current_node->contains_soldier = true;
			if (current_node->stand == mg_stand.stand || current_node-
>stand == artillerie_stand.stand) {
				can_shoot = false;
			}
```

```
                        else {
                                can_shoot = true;
                        }
                        this->stand->push_back(gameObject);
                        RestartTimer();
                }
        }
}

void SoldierBehaviour::SoldierMove(GameObject* move_node) {

        if (gets_healed) return;

        // get the for this function required components
        Movement* move_component = gameObject->GetComponent<Movement>();
        current_node = move_node->GetComponent<Node>();

        // if the target node is the same it is already on or it is already
occupied (position-check is needed because clicking a node twice with the same
selected gameobject will make it unoccupied)
        if (move_component->GetTargetPos() == move_node->transform.position ||
current_node->is_occupied) return;

        // make the clicked node occupied and the recent unoccupied
        current_node->is_occupied = true;

        old_node->GetComponent<Node>()->is_occupied = false;
        old_node->GetComponent<Node>()->contains_soldier = false;

        // move the gameobject
        move_component->SetTrackingPos(&move_node->transform.position);
        this->target_position = move_node->transform.position;
        travelling = true;

        // delete the gameobject from the recent stand
        for (size_t i =   ; i < stand->size(); i++) {
                if (stand->at(i) == gameObject) {
                        stand->erase(stand->begin() + i);
                        break;
                }
        }

        // used for accessing the recent stand and node
        this->stand = move_node->GetComponent<Node>()->stand;
        this->old_node = move_node;
}

bool SoldierBehaviour::SoldierTryMoveToWaitingNode() {

        // choose a free waiting node
        for (auto& node : waiting_nodes) {
                if (!node->GetComponent<Node>()->is_occupied) {

                        // move the gameobject to the chosen waiting node
                        gameObject->GetComponent<Movement>()->SetTrackingPos(&node-
>transform.position);
                        this->target_position = node->transform.position;
                        node->GetComponent<Node>()->is_occupied = true;
                        this->stand = waiting_stand.stand;
                        this->stand->push_back(gameObject);
                        this->old_node = node;
```

```
                        return true;
                }
        }
        return false;
}

void SoldierBehaviour::RestartTimer() {
        dt_counter =    ;
        time_to_wait = RandomF(min_soldier_shoot_waiting_time,
max_soldier_shoot_waiting_time) * game_time_factor;
}

void SoldierBehaviour::FreeNode() {
        current_node->is_occupied = false;
        current_node->contains_soldier = false;
        current_node = nullptr;
}
```

MedicComponent

Der MedicComponent besteht gleich aus zwei Klassen, dem MedicCharacter und dem MedicBuilding. Das MedicBuilding ist einfach dafür da, die Anzahl der Medics zu , verwalten und diese auch loszuschicken. Der MedicCharacter ist ein wenig komplizierter. Hier kann man wieder einen kleinen Vergleich zum SoldierBehaviour ziehen, nur dass ein Arzt sich den Soldaten als Ziel heraussucht, in dessen GUI (grafische Benutzeroberfläche) der „Arzt rufen“-Knopf geklickt wurde. Sobald er an seiner Zielposition angekommen ist, fängt er an, den Soldaten zu heilen, wobei die Zeit abhängig von dessen Gesundheitspunkten ist. Nach diesem Prozess kehrt er wieder zum Arzt-Gebäude zurück und verschwindet darin.

```
void MedicBuilding::SendMedic() {

        if (available_medics <= 0) return;
        available_medics--;

        gameScene->GetActiveCharacter()->GetComponent<SoldierBehaviour>()-
>MedicSent();
        gameScene->allyLayer->CreateMedic(gameObject->transform.position);

        if (gameScene->GetActiveBuilding() == gameScene->mapLayer->medicBuilding) {
                gameScene->uiLayer->DeactivateBuildingUI();
                gameScene->uiLayer->ActivateMedicBuildlingUI();
        }
}

void MedicBuilding::IncreaseAvailableMedics() {
        available_medics++;
        if (gameScene->GetActiveBuilding() == gameScene->mapLayer->medicBuilding) {
                gameScene->uiLayer->DeactivateBuildingUI();
                gameScene->uiLayer->ActivateMedicBuildlingUI();
        }
}

MedicCharacter::MedicCharacter(GameObject* medic_building)
```

```
	:medic_building(medic_building)
{
	healing_target = gameScene->GetActiveCharacter();
	healing_target_position = healing_target->transform.position +
medic_healing_position_offset;
	healing = false;
	going_back = false;
	dt_counter = 0.0f;
	heal_time = 0.0f;
}

void MedicCharacter::OnStart() {
	gameObject->GetComponent<Movement>()-
>SetTargetPos(healing_target_position);
}

void MedicCharacter::OnUpdate() {

	if (going_back) {
		if (gameObject->transform.position == medic_building-
>transform.position) {
			medic_building->GetComponent<MedicBuilding>()-
>IncreaseAvailableMedics();
			LOG_DEBUG("medic reached medic-building");
			delete gameObject;
		}
		return;
	}

	// if the soldier is already dead
	if (!healing_target->GetComponent<SoldierBehaviour>()) {
		// go back
		LOG_DEBUG("soldier to heal just died");
		going_back = true;
		gameObject->GetComponent<Movement>()-
>SetTrackingPos(&medic_building->transform.position);
		return;
	}

	if ((gameObject->transform.position != healing_target_position) ||
going_back) return; // if he has not arrived yet or is going back
	if (!healing) {		// if he arrived and is not healing already
		LOG_DEBUG("medic just arrived at soldier to heal");
		healing = true;
		heal_time = (soldier_health - healing_target-
>GetComponent<Health>()->GetHp()) * waiting_time_per_hp * game_time_factor;
	}
	else if (healing) {
		if (dt_counter >= heal_time) {
			// healing is over
			LOG_DEBUG("medic just finished healing");
			healing = false;
			going_back = true;
			healing_target->GetComponent<Health>()->GetHealed();
			gameObject->GetComponent<Movement>()-
>SetTrackingPos(&medic_building->transform.position);
			healing_target->GetComponent<SoldierBehaviour>()->MedicLeft();
			if (gameScene->GetActiveCharacter() == healing_target) {
				gameScene->uiLayer->DeactivateCharacterUI();
				gameScene->uiLayer->ActivateSoldierUI();
			}
		}
```

```
                dt_counter += Application::GetDT();
        }
}
```

EngineerComponent

Der EngineerComponent ist gleich aufgebaut wie der MedicComponent, er besteht also aus den zwei Klassen EngineerBuilding und EngineerBehaviour. EngineerBuilding funktioniert genau gleich wie MedicBuilding, nur dass Ärzte mit Mechanikern getauscht wurden und noch die Verwaltung der Anzahl an Maschinengewehren und Artillerie dazukommt, doch der EngineerCharacter weist grundlegende Unterschiede auf. Er ist im Spiel dafür da, Maschinengewehre und Artillerie zu bauen, die dann als Unterstützung im Kampf dienen. Dazu sucht er sich nach einem Klick auf den jeweiligen Knopf in der grafischen Benutzeroberfläche des EngineerBuildings einen zufälligen seiner zugeteilten Felder aus (bei Maschinengewehren sind es die Graben-Felder, bei Artillerie sind es die Deckungs-Felder) und baut dort sein jeweiliges Gerät hin.

```
EngineerCharacter::EngineerCharacter(bool mg_artillery)
{
        engineer_building = gameScene->mapLayer->engineerBuilding;
        this->mg_artillery = mg_artillery;
        building_node = ChooseBuildingNode();
        building_node->GetComponent<Node>()->is_occupied = true;
        building_node_position = building_node->transform.position +
engineer_building_position_offset;
        isBuilding = false;
        going_back = false;
        dt_counter =      ;
}

void EngineerCharacter::OnStart() {
        gameObject->GetComponent<Movement>()->SetTargetPos(building_node_position);
}

GameObject* EngineerCharacter::ChooseBuildingNode() {
        std::vector<GameObject*> nodes = mg_artillery ? trench_nodes :
hiding_nodes;
        std::vector<GameObject*> valid_nodes;
        for (auto& node : nodes) {
                if (!node->GetComponent<Node>()->is_occupied)
valid_nodes.push_back(node);
        }

        return valid_nodes.at(RandomInt(  , valid_nodes.size() -   ));
}

void EngineerCharacter::OnUpdate() {

        if (going_back) {
                if (gameObject->transform.position == engineer_building-
>transform.position) {
```

```
                        engineer_building->GetComponent<EngineerBuilding>()-
>IncreaseAvailableEngineers();
                        delete gameObject;
                }
                return;
        }

        if (gameObject->transform.position != building_node_position || going_back)
return;         // if he has not arrived yet or is going back return
        if (isBuilding)
        {
                if (dt_counter >= building_time) {
                        gameObject->GetComponent<SingleAnimation>()->StopAnimation();
                        isBuilding = false;
                        going_back = true;
                        gameObject->GetComponent<Movement>()-
>SetTrackingPos(&engineer_building->transform.position);

                        if (mg_artillery) {
                                GameObject* mg_node = gameScene->mapLayer-
>CreateNode(building_node->transform.position, mg_stand);
                                gameScene->mapLayer->CreateMg(building_node-
>transform.position + mg_position_offset, mg_node);


                        }
                        else {
                                GameObject* artillery_node = gameScene->mapLayer-
>CreateNode(building_node->transform.position, artillerie_stand);
                                gameScene->mapLayer->CreateArtillery(building_node-
>transform.position + artillery_position_offset, artillery_node);

                        }
                        delete building_node;
                }
                else {
                        gameObject->GetComponent<SingleAnimation>()-
>PlayAnimation(false);
                }
                dt_counter += Application::GetDT();
                return;
        }
        // if he arrived and is not building already
        isBuilding = true;
        dt_counter =      ;
}
```

Health

Der Health-Component dient dazu, die Gesundheit der Charakter im Spiel zu überwachen und in dem Fall, dass diese unter 0 sinkt, zum Sterben zu bringen. Dabei müssen ein paar Dinge beachtet werden, wie z.B., dass wenn das sterbende GameObject ein Gegner ist, überprüft wird, ob alle anderen Gegner tot sind um die nächste Welle einzuleiten. Der WaveManager-Component wird später ebenfalls noch erklärt.

```
bool Health::TakeDamage(float damage) {
        hp -= damage;
```

```
		if (gameScene->GetActiveCharacter() == gameObject) {
			gameScene->uiLayer->DeactivateCharacterUI();
			if (gameObject->HasTag("soldier")) {
				gameScene->uiLayer->ActivateSoldierUI();
			}
			else if (gameObject->HasTag("medic")) {
				gameScene->uiLayer->ActivateMedicUI();
			}
			else if (gameObject->HasTag("engineer")) {
				gameScene->uiLayer->ActivateEngineerUI();
			}
		}
		if (hp <= 0.0f) {
			if (gameScene->GetActiveCharacter() == gameObject) gameScene-
>SetActiveCharacter(nullptr);
			if (gameObject->HasTag("soldier")) {
				// get node and unoccupy it
				gameObject->GetComponent<SoldierBehaviour>()->FreeNode();
				gameScene->uiLayer->DeactivateCharacterUI();
				Util::enemyTable.erase(Util::enemyTable.find(gameObject));
				gameScene->mapLayer-
>CreateDeadBody("Anims/Soldier/soldier_dead.png", gameObject-
>transform.position);
				Supply::CheckForGameOver();

			}
			else if (gameObject->HasTag("enemy"))
			{
				if (gameObject->GetComponent<EnemyBehaviour>()->GetNode() !=
nullptr)
					gameObject->GetComponent<EnemyBehaviour>()->GetNode()-
>GetComponent<Node>()->is_occupied = false;
				if (Util::soldierTable.count(gameObject))
				{
					auto it = Util::soldierTable.find(gameObject);
					ASSERT(!(*it).first->IsDeleted(), "")
					Util::soldierTable.erase(it);
				}
				gameScene->mapLayer-
>CreateDeadBody("Anims/Enemy/french_dead.png", gameObject->transform.position);

				gameScene->waveManager->CheckForEnemiesDead();
			}
			delete gameObject;
			return true;
		}
		return false;
		// return true means dead
}
```

WaveManager

Der WaveManager-Component verfügt über den gesamten Code, der das Wellensystem des Spiels steuert. Jede Welle dauert eine bestimmte Zeit an und wenn diese Zeit vorbei ist, sowie alle Gegner tot sind, wird die Pause-Phase gestartet. Diese ist dafür da, seine Charaktere neu aufzustellen, seine Soldaten zu heilen oder z.B. seine Aufstellung zu verändern.

```
WaveManager::WaveManager(GameScene* gameScene)
	: gameScene(gameScene)
{
	global_dt_counter =       ;
	spawn_dt_counter =       ;
	cooldown_state = true;
	just_started = true;
	enemies_are_dead = true;
	cooldown_duration = start_preparation_time * game_time_factor;
	wave_duration = start_wave_duration * game_time_factor;
	spawn_interval = enemy_start_spawn_interval * game_time_factor;
}

void WaveManager::OnUpdate() {

	float dt = Application::GetDT();
	global_dt_counter += dt;

	if (cooldown_state) {
		if (global_dt_counter >= cooldown_duration) {
			LOG_DEBUG("cooldown over - wave state");
			cooldown_state = false;
			enemies_are_dead = false;
			global_dt_counter =       ;
			if (just_started) {
				just_started = false;
				return;
			}
		}
		return;
	}

	if (global_dt_counter <= wave_duration) {
		if (spawn_dt_counter >= spawn_interval) {
			gameScene->enemyLayer->CreateEnemy("enemy",
glm::vec2(RandomF(enemy_grid_startpos.x - enemy_spawn_random_x_radius,
enemy_grid_startpos.x + enemy_spawn_random_x_radius), enemy_spawn_y_position));
			spawn_dt_counter =       ;
		}
		else {
			spawn_dt_counter += dt;
		}
	}
	else {
		if (!enemies_are_dead) return;
		LOG_DEBUG("Wave over - cooldown state");
		cooldown_state = true;
		gameScene->uiLayer->ActivateSupplyMenuUI();
		global_dt_counter =       ;
		wave_duration *= wave_length_gradient;
		spawn_interval *= enemy_spawn_interval_gradient;
		return;
	}
}

void WaveManager::CheckForEnemiesDead() {
	if (gameScene->enemyLayer->GetGameObjectsByTag("enemy").size() >    ) return;

	enemies_are_dead = true;
}
```

SoldierShooting

Da es im Spiel einen Health-Component gibt, muss es selbstverständlich auch Schieß-Components geben, die mit dem Health-Component interagieren. Diese sind SoldierShooting oder EnemyShooting. Sie regeln, wann geschossen werden kann, wer abgeschossen wird, ob der Schuss je nach der Trefferwahrscheinlichkeit ein Treffer war oder nicht und instanziieren schließlich eine Kugel, die wiederum den Bullet-Component enthält. Dieser wird später erklärt.

```
void SoldierShooting::Shoot() {

	for (uint8_t i = 0; i < max_soldier_lock_target_tries; i++) {
		GameObject* target = LockTarget();
		if (!target) continue;

		glm::vec2 distanceVec2 = target->transform.position - gameObject-
>transform.position;
		float dist = sqrt(distanceVec2.x * distanceVec2.x + distanceVec2.y *
distanceVec2.y);

		glm::vec2 pos = target->transform.position;
		float distScale = dist / bulletInaccuracyMultiplicator;

		if (RandomInt(-dist, 2) < 0 && bulletDistanceMoreInaccuracy)
		{
			float randomX = RandomF(-1.0f, 1.0f) * distScale;
			pos.x += randomX;
		}


		gameScene->CreateBullet(gameScene->allyLayer, target, gameObject-
>transform.position, pos);

		break;
	}

}

GameObject* SoldierShooting::LockTarget() {
	// check if target already exists
	GameObject* target = GetTarget();
	if (target) {
		return target;
	}

	// choose a random row of enemies
	std::vector<GameObject*> enemies_in_row;
	int row = RandomInt(0, SumTo(enemy_grid_y));
	int prob = 0;
	for (int i = 1; i <= enemy_grid_y; i++) {
		prob += i;
		if (row <= prob) {
			row = i - 1;
			hit_probability = i;
			break;
		}
```

```
		}
		//LOG_TRACE(row);
		for (int i = 0; i < enemy_grid_x; i++) {
			if (enemy_stands[row][i]) {
				enemies_in_row.push_back(enemy_stands[row][i]);
			}
		}

		// if no enemies in row, return false
		if (enemies_in_row.empty()) {
			return nullptr;
		}

		// set random enemy in row as target and return true
		target = enemies_in_row[RandomInt(0, enemies_in_row.size() - 1)];
		ASSERT(target->HasTag("enemy"), "")
		Util::soldierTable[target].push_back(this);

		return target;
}

GameObject* SoldierShooting::GetTarget() const
{
		for (auto [key, val] : Util::soldierTable)
		{
			for (const SoldierShooting* ss : val)
			{
				if (ss == this) return key;
			}
		}
		return nullptr;
}
```

EnemyShooting

EnemyShooting macht der Logik nach das gleiche wie SoldierShooting, nur dass natürlich die Charaktere des Spielers als Ziele genommen und abgeschossen werden.

```
void EnemyShooting::Shoot() {

		// he has max max_enemy_lock_target_tries tries, if no character is found
=> cancel shooting
		for (uint8_t i = 0; i < max_enemy_lock_target_tries; i++) {
			GameObject* target = LockTarget();
			// if a target is found => instantiate bullet and calulate whether
it will hit or not, stop target-searching-process
			if (!target) continue;
			//LOG_DEBUG("LockTarget() returned true");


			glm::vec2 distanceVec2 = target->transform.position - gameObject-
>transform.position;
			float dist = sqrt(distanceVec2.x * distanceVec2.x + distanceVec2.y *
distanceVec2.y);

			glm::vec2 pos = target->transform.position;
			float distScale = dist / bulletInaccuracyMultiplicator;
```

```
                if (RandomInt(hit_probability - 9, hit_probability + 1) < 0 &&
bulletDistanceMoreInaccuracy)
                {
                        float randomX = RandomF(-1.0f, 1.0f) * distScale;
                        pos.x += randomX;
                }

                gameScene->CreateBullet(gameScene->enemyLayer, target, gameObject-
>transform.position, pos);

                break;

        }
}


GameObject* EnemyShooting::LockTarget() {
        GameObject* target = GetTarget();
        if (target) {
                return target;
        }

        const std::vector<GameObject*>* chosen_stand = nullptr;

        // Wähle eine zufällige Reihe aus den schießbaren Ständen
        int random = RandomInt(0, choose_probability_sum);
        uint32_t prob = 0;

        size_t i = 0;
        for (; i < shootable_stands.size(); i++) {
                prob += *shootable_stands[i].choose_probability;
                if (random <= prob) {
                        chosen_stand = shootable_stands[i].stand;
                        break;
                }
        }

        // Überprüfe, ob die gewählte Reihe Charaktere enthält
        if (chosen_stand && chosen_stand->empty()) {
                return nullptr;
        }

        // Wähle zufällig einen Charakter in der gewählten Reihe als Ziel
        target = chosen_stand->at(RandomInt(0, chosen_stand->size() - 1));
        hit_probability = *shootable_stands[i].hit_probability;
        Util::enemyTable[target].push_back(this);
        return target;

}

GameObject* EnemyShooting::GetTarget() const
{
        for (auto [key, val] : Util::enemyTable)
        {
                for (const EnemyShooting* ss : val)
                {
                        if (ss == this) return key;
                }
        }
        return nullptr;
}
```

MgComponent

Den MgComponent kann man sich im Grunde genommen vorstellen wie einen Soldaten ohne SoldierBehaviour, da die Maschinengewehre im Spiel an bestimmten Positionen stehen und sehr schnell auf Gegner schießen, wenn sie von einem Soldaten bedient werden. Sie haben also eine bestimmte Magazingröße und schießen so lange, bis ihr Magazin leer ist, um wieder nachzuladen. Da die Feuerrate viel höher als bei normalen Soldaten ist, muss natürlich die Genauigkeit und der Schaden verringert werden, damit die Maschinengewehre nicht zu stark sind.

```
void MgComponent::OnStart() {
	target = nullptr;
	dt_counter = 0.0f;
	ammo = mg_magazin_size;
}

void MgComponent::OnUpdate() {

	if (!own_node->contains_soldier) return;

	dt_counter += Application::GetDT();

	if (ammo <= 0) {
		if (dt_counter >= mg_reload_time * game_time_factor) {
			ammo = mg_magazin_size;
			dt_counter = 0.0f;
		}
		gameObject->GetComponent<SpriteSheet>()->ChangeSprite(glm::vec2(0.0,
0.0));
		return;
	}


	if (dt_counter < mg_shoot_interval * game_time_factor) {
		return;
		gameObject->GetComponent<SpriteSheet>()->ChangeSprite(glm::vec2(0.0,
0.0));
	}

	dt_counter = 0.0f;


	glm::vec2 pos = gameObject->transform.position;

	float lengthLeft = glm::length(enemy_grid[0][0]->transform.position.x -
pos);
	float lengthRight = glm::length(enemy_grid[enemy_grid_x - 1][0]-
>transform.position.x - pos);
	float length = lengthLeft < lengthRight ? lengthRight : lengthLeft;

	float lengthToNearestEnemy = -1.0f;
	float angleToAim = 0.0f;
```

```
	for (auto vec : enemy_stands)
	{
		for (auto* gm : vec)
		{
			if (gm == nullptr)
				continue;

			if (lengthToNearestEnemy < 0)
			{
				lengthToNearestEnemy = glm::length(gm->transform.position - pos);
				angleToAim = Util::VectorAngle(gm->transform.position - pos);
				continue;
			}

			if (glm::length(gm->transform.position - pos) < lengthToNearestEnemy)
			{
				lengthToNearestEnemy = glm::length(gm->transform.position - pos);
				angleToAim = Util::VectorAngle(gm->transform.position - pos);
			}
		}
	}

	if (lengthToNearestEnemy < 0)
		return;

	const float angle = RandomF(angleToAim - mg_inaccuracy, angleToAim + mg_inaccuracy);

	glm::vec2 calcPos = Util::VectorAngle(angle);
	ammo--;
	gameScene->CreateBullet(gameScene->allyLayer, nullptr, gameObject->transform.position, gameObject->transform.position + calcPos * glm::vec2(length, length));
	gameObject->GetComponent<SpriteSheet>()->ChangeSprite(glm::vec2(1.0, 0.0));

}

GameObject* MgComponent::LockTarget() {

	// check if target already exists
	GameObject* target = GetTarget();
	if (target) {
		return target;
	}

	// choose a random row of enemies
	std::vector<GameObject*> enemies_in_row;
	int row = RandomInt(0, SumTo(enemy_grid_y));
	int prob = 0;
	for (int i = 1; i <= enemy_grid_y; i++) {
		prob += i;
		if (row <= prob) {
			row = i - 1;
			hit_probability = i;
			break;
		}
	}
```

```
	//LOG_TRACE(row);
	for (int i = 0; i < enemy_grid_x; i++) {
		if (enemy_stands[row][i]) {
			enemies_in_row.push_back(enemy_stands[row][i]);
		}
	}

	// if no enemies in row, return false
	if (enemies_in_row.empty()) {
		return nullptr;
	}

	// set random enemy in row as target and return true
	target = enemies_in_row[RandomInt(0, enemies_in_row.size() - 1)];
	ASSERT(target->HasTag("enemy"), "");
		//Util::soldierTable[target].push_back(this);

	return target;
}

GameObject* MgComponent::GetTarget() const
{
	//for (auto [key, val] : Util::soldierTable)
	//{
	//	for (const MgComponent* ss : val)
	//	{
	//			if (ss == this) return key;
	//	}
	//}
	return nullptr;
}
```

BulletComponent

Da die Geschosse im Spiel alle einzelne GameObjects darstellen, muss ihnen beim Abschießen dieser Component hinzugefügt werden. Er ist dafür zuständig zu prüfen, ob jemand getroffen wurde und wenn ja, ihm Schaden hinzuzufügen.

```
static bool CoordRoundVec2(glm::vec2 targetPos, glm::vec2 pos)
{
	glm::vec2 diff = { targetPos.x - pos.x, targetPos.y - pos.y };
	if (abs(diff.x) < 0.25f && abs(diff.y) < 0.25f) return true;
	return false;
}

BulletComponent::BulletComponent(GameObject* target, glm::vec2 pos)
	: target(target), pos(pos) { }

void BulletComponent::OnUpdate()
{
	bool del = false;
	Movement* movement = gameObject->GetComponent<Movement>();
	gameObject->transform.rotation = -Util::VectorAngle(movement-
>GetDirection());
	if (target == nullptr)
	{
		for (auto vec : enemy_stands)
		{
```

```
                        for (auto* gm : vec)
                        {
                                if (gm != nullptr && gm->GetComponent<Health>() !=
nullptr && CoordRoundVec2(gameObject->transform.position, gm-
>transform.position))
                                {
                                        gm->GetComponent<Health>()-
>TakeDamage(mg_damage);
                                        delete gameObject;
                                        return;
                                }
                        }
                }
        }

        if (movement->IsArrived())
        {
                if (target != nullptr && gameObject->transform.position == target-
>transform.position)
                {
                        if (target->HasTag("enemy"))
                                target->GetComponent<Health>()-
>TakeDamage(soldier_damage);
                        if (target->HasTag("soldier"))
                                target->GetComponent<Health>()-
>TakeDamage(enemy_damage);
                }
                delete gameObject;
        }
}
```

ArtilleryComponent

Die Artillerie ist wie das MG ein „Gebäude“, das der Mechaniker bauen kann, nur dass es anders funktioniert. Es schießt nämlich Geschosse auf das Feld des Gegners, wo in einem Radius von 3 x 3 Feldern Schaden zugefügt wird. Gegner im mittleren Feld bekommen dabei Extraschaden.

```
ArtilleryComponent::ArtilleryComponent(GameObject* own_node)
        :own_node(own_node->GetComponent<Node>())
{
        dt_counter = 0.0f;
}

void ArtilleryComponent::OnStart() {
        reload_time = RandomF(artillery_min_reload_time, artillery_max_reload_time)
* game_time_factor;
        LOG_DEBUG("reload_time: {0}", reload_time);
}

void ArtilleryComponent::OnUpdate() {

        if (!own_node->contains_soldier) return;

        dt_counter += Application::GetDT();
        if (dt_counter >= reload_time) {
                Shoot();
```

```
                dt_counter = 0.0f;
        }
}

// makes damage in ring form
void ArtilleryComponent::Shoot() {
        // choose a random middle-node at the enemy grid
        uint8_t target_x = RandomInt(0, enemy_grid_x - 1);
        uint8_t target_y = RandomInt(0, enemy_grid_y - 1);
        GameObject* target_node = enemy_grid.at(target_x).at(target_y);

        gameObject->GetComponent<SingleAnimation>()->PlayAnimation(true);
        LOG_DEBUG("ARTILLERY SHOOT");

        GameObject* explosionb = new GameObject("bum", Transform(target_node-
>transform.position, artillery_explosion_size), ProjectionMode::PERSPECTIVE);
        explosionb->AddComponent(new SpriteRenderer(glm::vec4(white_color),
Geometry::RECTANGLE));
        explosionb->AddComponent(new DestroyOverTime(artillery_explosion_lasting));
        gameScene->allyLayer->AddGameObjectToLayer(explosionb);

        GameObject* explosion = new GameObject("bum", Transform(target_node-
>transform.position, artillery_explosion_size), ProjectionMode::PERSPECTIVE);
        explosion->AddComponent(new
SingleAnimation(DataPool::GetTexture("Anims/artillery_explosion.png"), 136.0f,
136.0f, 16.0f, 16.0f, glm::vec2(0.0f, 0.0f), glm::vec2(2.0f, 0.0f),
artillery_explosion_anim_speed, glm::vec2(0.0f, 0.0f),
DataPool::GetTexture("Anims/artillery_explosion.png"), 136.0f, 136.0f, 16.0f,
16.0f));
        explosion->AddComponent(new DestroyOverTime(artillery_explosion_lasting));
        explosion->GetComponent<SingleAnimation>()->PlayAnimation(true);
        gameScene->allyLayer->AddGameObjectToLayer(explosion);

        glm::ivec2 top_left = glm::ivec2(target_x - 1, target_y - 1);
        // loops through a field that starts at the top left of the randomly chosen
middle-node
        for (uint8_t x = 0; x < 3; x++) {

                // if out of x-bounds
                if (top_left.x < 0 || top_left.x + x >= enemy_grid_x) continue;
                for (uint8_t y = 0; y < 3; y++) {

                        // if out ob y-bounds
                        if (top_left.y < 0 || top_left.y + y >= enemy_grid_y)
continue;

                        // make damage to every gameobject in this field
                        int actual_x = top_left.x + x;
                        int actual_y = top_left.y + y;
                        GameObject* hit_enemy = enemy_stands[actual_y][actual_x];
                        if (hit_enemy != nullptr) {
                                // crit hit on enemies that stand on the middle-node
                                if (x == 1 && y == 1) {
                                        hit_enemy->GetComponent<Health>()-
>TakeDamage(artillery_critical_damage);
                                }
                                else {
                                        hit_enemy->GetComponent<Health>()-
>TakeDamage(artillery_normal_damage);
                                }
                        }
                }
```

```
        }
        reload_time = RandomF(artillery_min_reload_time, artillery_max_reload_time)
* game_time_factor;
}
```

DestroyOverTime

Dieser Component ist ausschließlich dafür zuständig, das eigene GameObject nach einer festgelegten Zeit (die man beim Hinzufügen selbst bestimmten kann) zu zerstören. Er wird z.B. einer Leiche hinzugefügt, die nach einer bestimmten Zeit vom Schlachtfeld verschwinden soll.

```
DestroyOverTime::DestroyOverTime(float time)
        :time(time) { }

void DestroyOverTime::OnStart() {
        time *= game_time_factor;
}

void DestroyOverTime::OnUpdate() {
        dt_counter += Application::GetDT();

        if (dt_counter < time) return;

        delete gameObject;
}
```

Required

Functions

In functions.h befinden sich ein paar nützliche Funktionen wie, die im Spiel öfters mal gebraucht werden. RoundVec2() rundet den gegebenen Vektor auf zwei Nachkommastellen, RandomInt() gibt einen zufälligen Integer Wert zwischen den beiden gegebenen Zahlen zurück und RandomF() hat dieselbe Funktionsweise, nur mit float als Datentypen. Die Funktion SumTo() gibt einfach die Summe von 0 bis zu der gegebenen Zahl zurück.

```
inline void RoundVec2(glm::vec2& vec) {                    // written by chatGPT 0_0
        // rounds to 2 decimal places
        vec.x = std::roundf(vec.x * powf(10, 2)) / powf(10, 2);
        vec.y = std::roundf(vec.y * powf(10, 2)) / powf(10, 2);

}

inline int RandomInt(int from, int to) {                   // written by chatGPT 0_0
        std::random_device rd;  // Use a true random number source
        std::mt19937 generator(rd());  // Use Mersenne Twister algorithm
        std::uniform_int_distribution<> distribution(from, to);  // Distribute
numbers evenly
```

```
	return distribution(generator);  // Generate the random number and return
it
}

inline float RandomF(float min_float, float max_float) {  // written by chatGPT
0_0
	static bool initialized = false;
	if (!initialized) {
		srand(time(nullptr)); // set seed based on current time, only once
		initialized = true;
	}

	float random = ((float)rand()) / (float)RAND_MAX; // generate random float
between 0 and 1
	float range = max_float - min_float; // calculate range
	return (random * range) + min_float; // scale and shift the random number
to the desired range
}

inline int SumTo(int n) {
	return n * (n + 1) / 2;
}
```

Stands

Die stands.h Datei beinhaltet alle Variablen bzw. Listen, die die Besetzung der Stellung der eigenen und der gegnerischen Charaktere beinhalten. Außerdem sind hier einfach gesprochen die „Baupläne“ für alle Felder und deren Eigenschaften definiert.

```
struct Stand {
	std::vector<GameObject*>* stand;
	uint8_t* choose_probability;
	uint8_t* hit_probability;
	glm::vec4* color;
};

inline Stand front_stand = { new std::vector<GameObject*>(),
&front_choose_probability, &front_hit_probability, &node_front_color};
inline Stand mg_stand = { new std::vector<GameObject*>(), &mg_choose_probability,
&mg_hit_probability, &node_mg_color };
inline Stand trench_stand = { new std::vector<GameObject*>(),
&trench_choose_probability, &trench_hit_probability, &node_trench_color };
inline Stand hiding_stand = { new std::vector<GameObject*>(),
&hiding_choose_probability, &hiding_hit_probability, &node_hiding_color };
inline Stand artillerie_stand = { new std::vector<GameObject*>(),
&artillerie_choose_probability, &artillerie_hit_probability,
&node_artillerie_color };

inline Stand bunker_stand = { new std::vector<GameObject*>(), nullptr, nullptr,
&node_bunker_color };
inline Stand waiting_stand = { new std::vector<GameObject*>(), nullptr, nullptr,
&node_waiting_color };

inline std::vector<Stand> shootable_stands = {
	front_stand, mg_stand, trench_stand, hiding_stand, artillerie_stand
};
```

```
inline GameObject* enemy_stands[enemy_grid_y][enemy_grid_x];
inline std::vector<GameObject*> waiting_nodes;
inline std::vector<GameObject*> trench_nodes;
inline std::vector<GameObject*> hiding_nodes;
```

Maps

In maps.h sind alle Variablen definiert, die für die Spielkarte gebraucht werden. Das sind in diesem Fall z.B. die Anordnung der Felder oder bestimmter Sprites, die auf der Karte zu sehen sein sollen, aber sonst keine Funktion erfüllen.

```
inline float tr_wi = trench_node_size.x;
inline glm::vec2 mid_tr_start = glm::vec2(-4.0f * tr_wi, -4.0f);
inline glm::vec2 mid_tr_end = glm::vec2(4.0f * tr_wi, -4.0f);

inline std::vector<std::pair<std::vector<glm::vec2>, Stand>> standard_map =

{
	{
		{
			glm::vec2(-6.0f, -12.0f),
			glm::vec2(-5.0f, -12.0f),
			glm::vec2(-4.0f, -12.0f),
			glm::vec2(-6.0f, -11.0f),
			glm::vec2(-5.0f, -11.0f),
			glm::vec2(-4.0f, -11.0f)
		}
		,waiting_stand
	},
	{
		{
			glm::vec2(-9.0f, -4.5f),
			glm::vec2(-9.6f, -5.0f),
			glm::vec2(-10.2f, -5.5f),
			glm::vec2(-10.8f, -6.0f),
			glm::vec2(-11.4f, -6.5f),
			glm::vec2(-12.0f, -7.0f),
			glm::vec2(-12.6f, -7.5f),

			glm::vec2(-4.0f * tr_wi, -4.0f),
			glm::vec2(-3.0f * tr_wi, -4.0f),
			glm::vec2(-2.0f * tr_wi, -4.0f),
			glm::vec2(-tr_wi, -4.0f),
			glm::vec2(0.0f, -4.0f),
			glm::vec2(tr_wi, -4.0f),
			glm::vec2(2.0f * tr_wi, -4.0f),
			glm::vec2(3.0f * tr_wi, -4.0f),
			glm::vec2(4.0f * tr_wi, -4.0f),

			glm::vec2(9.0f, -4.5f),
			glm::vec2(9.6f, -5.0f),
			glm::vec2(10.2f, -5.5f),
			glm::vec2(10.8f, -6.0f),
			glm::vec2(11.4f, -6.5f),
			glm::vec2(12.0f, -7.0f),
			glm::vec2(12.6f, -7.5f)
```

```
                }
                ,trench_stand
        },
        {
                {
                        glm::vec2(-3.0f, -5.5f),
                        glm::vec2(-1.0f, -5.5f),
                        glm::vec2(1.0f, -5.5f),
                        glm::vec2(3.0f, -5.5f)
                }
                ,front_stand
        },
        {
                {
                        // implement bunker_node size
                        glm::vec2(mid_tr_start.x - 2.5f, -3.5f),
                        glm::vec2(mid_tr_start.x - 2.5f, -4.5f),
                        glm::vec2(mid_tr_start.x - 1.5f, -4.5f),
                        glm::vec2(mid_tr_start.x - 1.5f, -3.5f),

                        glm::vec2(mid_tr_end.x + 2.5f, -3.5f),
                        glm::vec2(mid_tr_end.x + 2.5f, -4.5f),
                        glm::vec2(mid_tr_end.x + 1.5f, -4.5f),
                        glm::vec2(mid_tr_end.x + 1.5f, -3.5f)
                }
                ,bunker_stand
        },
        {
                {
                        glm::vec2(-3.5f, -7.75f),
                        glm::vec2(-8.0f, -8.5f),
                        glm::vec2(8.0f, -8.5f),
                        glm::vec2(3.5f, -7.75f),
                        glm::vec2(0.0f, -8.5f)
                }
                ,hiding_stand
        }
};

inline std::vector<std::pair<std::vector<Transform>, std::string>>
standard_map_sprites =
{
        {
                {

                }
                ,
        }
};
```

Constants

In constants.h sind alle Variablen definiert und initialisiert, die das gesamte Spiel steuern. Man soll sie also einfach anpassen können, um das Spiel später balancen (die Mechaniken “ausbalancieren“) zu können und nicht ewig in einzelnen Code-Abschnitten suchen muss.

```
// scenes
inline GameScene* gameScene;
inline MenuScene* menuScene;

// enemy movement grid
inline std::vector<std::vector<GameObject*>> enemy_grid;

// camera movement
inline float camera_scroll_speed = 28.0f;
inline float camera_move_speed = 2.0f;
inline float min_camera_z_pos = 2.0f;
inline float max_camera_z_pos = 15.0f;
inline glm::vec2 min_camera_positions = glm::vec2(-7.0f, -6.0f);
inline glm::vec2 max_camera_positions = glm::vec2(7.0f, 5.0f);

// enemy grid
inline const int enemy_grid_x = 15;
inline const int enemy_grid_y = 9;
inline float enemy_grid_offset = 1.0f;
inline glm::vec2 enemy_grid_startpos = glm::vec2(0.0f, 5.0f);

// enemy behaviour
inline float min_enemy_waiting_time = 0.5f;
inline float max_enemy_waiting_time = 2.0f;
inline float enemy_movement_speed = 1.0f;
inline uint8_t max_enemy_lock_target_tries = 3;
inline glm::vec2 enemy_scale = glm::vec2(0.9f, 1.2f);
inline uint8_t enemy_random_movement_sum = 5;
inline uint8_t enemy_move_left_probability = 1;
inline uint8_t enemy_move_mid_probability = 3;
inline uint8_t enemy_move_right_probability = 1;
inline float enemy_spawn_y_position = 17.0f;
inline float enemy_spawn_random_x_radius = 10.0f;

// character
inline float soldier_movement_speed = 1.2f;
inline float medic_movement_speed = 1.4f;
inline float engineer_movement_speed = 3.9f;
inline glm::vec2 character_scale = glm::vec2(1.3f, 1.6f);
inline glm::vec2 dead_body_size = glm::vec2(1.6f, 1.4f);
inline float dead_body_lasting_time = 6.0f;

//bullet
inline float bullet_speed = 30.0f;
inline glm::vec2 bullet_scale = glm::vec2(0.25f, 0.5f);
inline glm::vec4 bullet_color = glm::vec4(0.1f, 0.1f, 0.1f, 1.0f);

//soldier behaviour
inline float min_soldier_shoot_waiting_time = 0.7f;
inline float max_soldier_shoot_waiting_time = 1.0f;
inline uint8_t max_soldier_lock_target_tries = 2;
inline uint8_t soldier_miss_points = 1;
inline glm::vec2 soldier_spawn_pos = glm::vec2(0.0f, -18.0f);

// stands (probabilities have to be choose_probability_sum in total)
inline uint8_t choose_probability_sum = 100;
inline uint8_t front_choose_probability = 45;
inline uint8_t mg_choose_probability = 20;
inline uint8_t trench_choose_probability = 15;
inline uint8_t hiding_choose_probability = 8;
inline uint8_t artillerie_choose_probability = 12;
```

```
// shooting
inline uint8_t max_hit_probability = 10;
inline uint8_t front_hit_probability = 8;
inline uint8_t mg_hit_probability = 6;
inline uint8_t trench_hit_probability = 5;
inline uint8_t hiding_hit_probability = 4;
inline uint8_t artillerie_hit_probability = 6;

// damage
inline uint8_t soldier_damage = 35;
inline uint8_t enemy_damage = 35;

// health
inline float enemy_health = 100.0f;
inline float soldier_health = 100.0f;
inline float medic_health = 50.0f;
inline float engineer_health = 200.0f;

// bullet trace
inline glm::vec4 trace_color = glm::vec4(1.0f, 0.0f, 0.0f, 0.8f);
inline float min_inaccuracy = 0.8f;
inline float max_inaccuracy = 1.4f;
inline float trace_thickness = 0.3f;
inline float trace_lasting = 0.15f;
inline bool bulletDistanceMoreInaccuracy = true;
inline float bulletInaccuracyMultiplicator = 10.0f;

// node
inline glm::vec2 node_size = glm::vec2(0.5f, 0.5f);
inline float node_alpha = 0.8f;
inline glm::vec4 node_front_color = glm::vec4(0.9f, 0.0f, 0.0f, node_alpha);
inline glm::vec4 node_mg_color = glm::vec4(0.9f, 0.2f, 0.0f, node_alpha);
inline glm::vec4 node_trench_color = glm::vec4(0.9f, 0.0f, 0.0f, node_alpha);
inline glm::vec4 node_hiding_color = glm::vec4(0.5f, 0.0f, 0.5f, node_alpha);
inline glm::vec4 node_artillerie_color = glm::vec4(0.6f, 0.4f, 0.2f, node_alpha);
inline glm::vec4 node_bunker_color = glm::vec4(0.0f, 0.9f, 0.0f, node_alpha);
inline glm::vec4 node_waiting_color = glm::vec4(0.0f, 0.4f, 0.4f, node_alpha);

// time
inline float game_time_factor = 3.0f;

// medics
inline glm::vec2 medic_healing_position_offset = glm::vec2(0.5f, 0.0f);
inline float waiting_time_per_hp = 0.03f;

// engineer
inline glm::vec2 engineer_building_position_offset = glm::vec2(0.5f, 0.0f);
inline float building_time = 2.5f;

// buildings
inline glm::vec2 building_size = glm::vec2(3.0f, 3.0f);

// wave
inline float start_preparation_time = 4.0f;
inline float start_wave_duration = 4.0f;
inline float wave_length_gradient = 1.5f;
inline float enemy_start_spawn_interval = 2.0f;
inline float enemy_spawn_interval_gradient = 0.9f;

// stocks
inline unsigned int start_soldier_stock = 5;
inline uint8_t start_engineer_stock = 3;
```

```
inline uint8_t start_medic_stock = 3;

// mg
inline glm::vec2 mg_size = glm::vec2(0.6f, 0.8f);
inline unsigned int mg_magazin_size = 50;
inline float mg_reload_time = 3.0f;
inline float mg_shoot_interval = 0.05f;
inline float mg_miss_points = 5;
inline float mg_damage = 15;
inline float mg_inaccuracy = 10.0f;
inline glm::vec2 mg_position_offset = glm::vec2(0.0f, 0.4f);

// artillery
inline glm::vec2 artillery_size = glm::vec2(0.7f, 0.9f);
inline float artillery_min_reload_time = 0.5f;
inline float artillery_max_reload_time = 2.5f;
inline float artillery_normal_damage = 70.0f;
inline float artillery_critical_damage = 100.0f;
inline glm::vec2 artillery_explosion_size = glm::vec2(3.0f, 3.0f);
inline glm::vec2 artillery_position_offset = glm::vec2(0.0f, 0.4f);
inline int artillery_shoot_anim_speed = 15;
inline int artillery_explosion_anim_speed = 15;
inline float artillery_explosion_lasting = 3.0f;

// ui fonts
inline std::string ui_font_family = "PixeloidMono.ttf";
inline glm::vec4 ui_font_color = glm::vec4(1.0f, 1.0f, 1.0f, 0.8f);

// character ui color
inline glm::vec4 white_color = glm::vec4(1.0f, 1.0f, 1.0f, 1.0f);

// character and building ui transform
inline glm::vec2 ui_building_background_size = glm::vec2(0.3f, 0.73f);
inline glm::vec2 ui_background_size = glm::vec2(0.3f, 0.6f);
inline glm::vec2 ui_character_position = glm::vec2(0.8f, 0.0f);
inline glm::vec2 ui_building_position = glm::vec2(-0.8f, 0.0f);
inline Transform ui_header_transform = Transform(glm::vec2(0.0f, 0.57f),
glm::vec2(0.9f, 0.25f));
inline Transform ui_header_name_transform = Transform(glm::vec2(0.0f, 0.0f),
glm::vec2(0.23f, 0.67f));
inline Transform ui_header_building_name_transform = Transform(glm::vec2(0.0f,
0.0f), glm::vec2(0.14f, 0.71f));
inline Transform ui_character_icon_transform = Transform(glm::vec2(0.0f, 1.1f),
glm::vec2(0.3f, 0.32f));
inline Transform ui_building_icon_transform = Transform(glm::vec2(0.0f, 1.1f),
glm::vec2(0.3f, 0.24f));
inline Transform ui_medic_button_transform = Transform(glm::vec2(0.0f, -0.5f),
glm::vec2(0.5f, 0.22f));
inline Transform ui_first_button_transform = Transform(glm::vec2(0.0f, -0.2f),
glm::vec2(0.5f, 0.22f));
inline Transform ui_second_button_transform = Transform(glm::vec2(0.0f, -0.7f),
glm::vec2(0.5f, 0.22f));
inline Transform ui_building_count_transform = Transform(glm::vec2(0.0f, 0.15f),
glm::vec2(0.1, 0.1));
inline Transform ui_hp_icon_transform = Transform(glm::vec2(-0.3f, -0.01f),
glm::vec2(0.25f, 0.2f));
inline Transform ui_health_text_transform = Transform(glm::vec2(0.25f, -0.01f),
glm::vec2(0.12f, 0.13f));

// supply menu ui color
inline glm::vec4 ui_supply_menu_background_color = glm::vec4(0.5f, 0.5f, 0.5f,
0.6f);
```

```
inline glm::vec4 ui_choice_field_color = glm::vec4(0.2f, 0.2f, 0.2f, 0.6f);
inline glm::vec4 ui_choice_field_button_color = glm::vec4(0.8f, 0.6f, 0.4f,
0.7f);

//supply menu ui transform
inline glm::vec2 ui_supply_menu_background_size = glm::vec2(1.5f, 1.44f);
inline Transform ui_supply_menu_text_transform = Transform(glm::vec2(0.0f, 0.8f),
glm::vec2(0.04f, 0.105f));
inline Transform ui_supply_menu_text2_transform = Transform(glm::vec2(0.0f,
0.575f), glm::vec2(0.03f, 0.05f));
inline glm::vec2 ui_left_choice_field_position = glm::vec2(-0.45f, -0.18f);
inline glm::vec2 ui_right_choice_field_position = glm::vec2(0.45f, -0.18f);
inline glm::vec2 ui_choice_field_size = glm::vec2(0.32f, 0.58f);
inline Transform ui_choice_field_text_transform = Transform(glm::vec2(0.0f,
0.9f), glm::vec2(0.1f, 0.16f));
inline Transform ui_choice_field_count_transform = Transform(glm::vec2(0.0f,
0.5f), glm::vec2(0.11f, 0.14f));
inline Transform ui_choice_field_button_transform = Transform(glm::vec2(0.0f, -
0.63f), glm::vec2(0.7f, 0.2f));
inline glm::vec2 anfordern_button_size = glm::vec2(0.18f, 0.62f);

// supply menu
inline uint8_t min_soldiers_choice = 2;
inline uint8_t max_soldiers_choice = 5;
inline uint8_t soldier_increase_by_wave = 3;

// map sprites
inline glm::vec2 background_tile_size = glm::vec2(30.0f, 15.0f);
inline glm::vec2 hiding_sprite_size = glm::vec2(1.6f, 0.85f);
inline glm::vec2 hiding_sprite_node_offset = glm::vec2(0.0f, 0.8f);

// nodes
inline glm::vec2 trench_node_size = glm::vec2(1.2f, 1.0f);
inline glm::vec2 waiting_node_size = glm::vec2(0.5f, 0.5f);
inline glm::vec2 hiding_node_size = glm::vec2(0.5f, 0.5f);
inline glm::vec2 bunker_node_size = glm::vec2(0.5f, 0.5f);
inline glm::vec2 artillery_node_size = glm::vec2(0.5f, 0.5f);
```

Scenes

GameScene

Die GameScene ist die eigentliche Spiel-Szene und steuert somit alle zugehörigen Layers und Dinge wie das Kameraverhalten oder den WaveManager.

```
GameScene::GameScene() {
    // set background color
    backcolor = glm::vec4(0.8f, 0.8f, 0.8f, 1.0f);

    backgroundLayer = new BackgroundLayer();
    mapLayer = new MapLayer();
    enemyLayer = new EnemyLayer();
    allyLayer = new AllyLayer;
    uiLayer = new UILayer();

    waveManager = new WaveManager(this);
}
```

```
void GameScene::OnStop() {
    RemoveLayer(backgroundLayer);
    RemoveLayer(mapLayer);
    RemoveLayer(enemyLayer);
    RemoveLayer(allyLayer);

    RemoveOverlay(uiLayer);
}
void GameScene::OnStart() {

    Supply::Init();

    AddLayer(backgroundLayer);
    AddLayer(mapLayer);
    AddLayer(enemyLayer);
    AddLayer(allyLayer);

    AddOverlay(uiLayer);
}

extern float con;

void GameScene::OnUpdate() {
    CameraMovement(Application::GetDT());
    waveManager->OnUpdate();
}

void GameScene::CameraMovement(float dt) {
    if (Input::IsKeyPressed(KEY_S) && camera->position.y >
min_camera_positions.y)
        camera->position.y -= camera_move_speed * dt;
    if (Input::IsKeyPressed(KEY_W) && camera->position.y <
max_camera_positions.y)
        camera->position.y += camera_move_speed * dt;
    if (Input::IsKeyPressed(KEY_D) && camera->position.x <
max_camera_positions.x)
        camera->position.x += camera_move_speed * dt;
    if (Input::IsKeyPressed(KEY_A) && camera->position.x >
min_camera_positions.x)
        camera->position.x -= camera_move_speed * dt;
}

bool GameScene::OnMouseScroll(MouseScrolledEvent& e)
{
    //Checking if camera z position is inside wanted bounds
    if (GetCamera()->position.z > max_camera_z_pos && e.GetYOffset() > 0) return
true;
    else if (GetCamera()->position.z < min_camera_z_pos && e.GetYOffset() < 0)
return true;
    GetCamera()->position.z += camera_scroll_speed * Application::GetDT() *
e.GetYOffset();

    return true;
}

bool GameScene::GameObjectPressed(GameObjectPressedEvent& e) {

    return true;
}

GameObject* GameScene::CreateBullet(Layer* layer, GameObject* target, glm::vec2
startPos, glm::vec2 targetPos) {
```

```
    GameObject* bullet = new GameObject("bullet", Transform(startPos,
bullet_scale));
    bullet->AddComponent(new CircleRenderer(bullet_color, 1.0f, 0.005));
bullet->AddComponent(new Movement(bullet_speed, targetPos));
    bullet->AddComponent(new BulletComponent(target, targetPos));
    bullet->AddTag("bullet");
    layer->AddGameObjectToLayer(bullet);
    return bullet;
}
```

Layers:

AllyLayer

In der AllyLayer sind alle Funktionen definiert, die verbündete GameObjects (Soldaten, Mechaniker, Ärzte, Maschinengewehre, Artillerie) erstellen. Außerdem wird hier mithilfe der GameObjectPressed-Eventfunktion geschaut, welcher Charakter angeklickt wurde und welche Aktionen daraus folgen.

```
GameObject* AllyLayer::CreateSoldier(glm::vec2 position) {
        GameObject* character = new GameObject("soldier", Transform(position,
character_scale));
        character->AddComponent(new SpriteSheet(glm::vec4(1.0f, 1.0f, 1.0f, 1.0f),
DataPool::GetTexture("/Anims/Soldier/German_Soldier_MG-0001.png"), 200.0f,
304.0f, 16.0f, 16.0f, glm::vec2(0, 0)));
        character->GetComponent<SpriteSheet>()->ChangeSprite(glm::vec2(3, 0));
        character->AddComponent(new Movement(soldier_movement_speed));
        character->AddComponent(new Health(soldier_health));
        character->AddComponent(new SoldierBehaviour());
        character->AddComponent(new SoldierShooting());
        character->AddComponent(new WalkingAnimation(glm::vec2(3.0f, 1.0f),
glm::vec2(5.0f, 1.0f), glm::vec2(3.0f, 0.0f), glm::vec2(5.0f, 0.0f),
glm::vec2(0.0f, 0.0f), glm::vec2(2.0f, 0.0f), glm::vec2(0.0f, 1.0f),
glm::vec2(2.0f, 1.0f), 10, glm::vec2(3.0f, 1.0f), false));
        character->AddTag("soldier");
        character->onlyLayerReceive = true;
        AddGameObjectToLayer(character);
        return character;
}

GameObject* AllyLayer::CreateMedic(glm::vec2 position) {
        GameObject* character = new GameObject("medic", Transform(position,
character_scale));
        character->AddComponent(new SpriteSheet(glm::vec4(1.0f, 1.0f, 1.0f, 1.0f),
DataPool::GetTexture("Anims/Med/med_walk_fixed-0001.png"), 112.0f, 304.0f, 24.0f,
16.0f, glm::vec2(0, 0)));
        character->AddComponent(new Movement(medic_movement_speed));
        character->AddComponent(new Health(medic_health));
        character->AddComponent(new MedicCharacter(gameScene->mapLayer-
>medicBuilding));
        character->AddComponent(new WalkingAnimation(glm::vec2(0.0f, 1.0f),
glm::vec2(2.0f, 1.0f), glm::vec2(0.0f, 0.0f), glm::vec2(2.0f, 0.0f),
glm::vec2(3.0f, 0.0f), glm::vec2(5.0f, 0.0f), glm::vec2(3.0f, 1.0f),
glm::vec2(5.0f, 1.0f), 16, glm::vec2(0.0f, 1.0f), false));
        character->AddTag("medic");
        character->onlyLayerReceive = true;
        AddGameObjectToLayer(character);
```

```
		return character;
}

GameObject* AllyLayer::CreateEngineer(glm::vec2 position, bool mg_artillery) {
		GameObject* character = new GameObject("engineer", Transform(position,
character_scale));
		character->AddComponent(new SpriteSheet(glm::vec4(1.0f, 1.0f, 1.0f, 1.0f),
DataPool::GetTexture("Anims/Engineer/engineer_walk_sideways.png"), 112.0f,
304.0f, 24.0f, 16.0f, glm::vec2(0, 0)));
		character->AddComponent(new Movement(engineer_movement_speed));
		character->AddComponent(new Health(engineer_health));
		character->AddComponent(new EngineerCharacter(mg_artillery));
		character->AddComponent(new WalkingAnimation(glm::vec2(0.0f, 0.0f),
glm::vec2(2.0f, 0.0f), glm::vec2(0.0f, 1.0f), glm::vec2(2.0f, 1.0f),
glm::vec2(0.0f, 0.0f), glm::vec2(2.0f, 0.0f), glm::vec2(0.0f, 1.0f),
glm::vec2(2.0f, 1.0f), 10, glm::vec2(0.0f, 1.0f), true));
		character->AddComponent(new
SingleAnimation(DataPool::GetTexture("Anims/Engineer/engineer_build.png"),
168.0f, 304.0f, 16.0f, 16.0f, glm::vec2(0.0f, 0.0f), glm::vec2(3.0f, 0.0f), 10,
glm::vec2(0.0f, 1.0f),
DataPool::GetTexture("Anims/Engineer/engineer_walk_sideways.png"), 112.0f,
304.0f, 16.0f, 16.0f));
		character->AddTag("engineer");
		character->onlyLayerReceive = true;
		AddGameObjectToLayer(character);
		return character;
}

bool AllyLayer::KeyReleased(KeyReleasedEvent& e) {
		if (e.getKeyCode() == KEY_SPACE) {
			Supply::TryCallSoldier();
			return true;
		}
		return false;
}

bool AllyLayer::GameObjectPressed(GameObjectPressedEvent& e) {

		GameObject* clicked_character = e.GetGameObject();

		if (clicked_character == gameScene->GetActiveCharacter()) {
			gameScene->SetActiveCharacter(nullptr);
			gameScene->uiLayer->DeactivateCharacterUI();
			return true;
		}
		if (clicked_character->HasTag("soldier") && !clicked_character-
>GetComponent<SoldierBehaviour>()->on_spawn_pos) {
			gameScene->SetActiveCharacter(clicked_character);
			gameScene->uiLayer->DeactivateCharacterUI();
			gameScene->uiLayer->ActivateSoldierUI();
		}
		else if (clicked_character->HasTag("medic")) {
			gameScene->SetActiveCharacter(clicked_character);
			gameScene->uiLayer->DeactivateCharacterUI();
			gameScene->uiLayer->ActivateMedicUI();
		}
		else if (clicked_character->HasTag("engineer")) {
			gameScene->SetActiveCharacter(clicked_character);
			gameScene->uiLayer->DeactivateCharacterUI();
			gameScene->uiLayer->ActivateEngineerUI();
		}
		else {
```

```
                return false;
        }
        return true;
}
```

MapLayer

In der MapLayer sind all die Funktionen verfügbar, die etwas mit der Karte zu tun haben (Erstellen von Feldern, der Karte an sich, dem Gegner-Feld und den Gebäuden).

```
void MapLayer::OnAttach()
{
        CreateGameMap(standard_map, standard_map_sprites);
        CreateEnemyGrid(enemy_grid_x, enemy_grid_y, enemy_grid_offset,
enemy_grid_startpos);

        Layer* layer = this;
        medicBuilding = Medic::AddBuilding(Transform(glm::vec2(11.0f, -11.5f),
building_size), start_medic_stock);
        engineerBuilding = Engineer::AddBuilding(Transform(glm::vec2(-11.0f, -
11.5f), building_size), start_engineer_stock);
}

GameObject* MapLayer::CreateNode(glm::vec2 position, Stand& node_stand) {

        glm::vec2 node_scale = glm::vec2(node_size);
        std::string sprite_path = "node_placeholder.png";

        GameObject* node_go = new GameObject("node", Transform(position,
node_scale));
        node_go->AddComponent(new Node(node_stand.stand));

        node_go->AddTag("move_node");

        // TODO: change sprite_path and node_scale each
        if (node_stand.stand == waiting_stand.stand) {
                waiting_nodes.push_back(node_go);
                node_go->RemoveTag("move_node");
                //sprite_path = "";
                node_scale = waiting_node_size;
        }
        else if (node_stand.stand == trench_stand.stand) {
                trench_nodes.push_back(node_go);
                sprite_path = "Map/bunker_tiles_top.png";
                node_scale = trench_node_size;
        }
        else if (node_stand.stand == mg_stand.stand) {
                sprite_path = "Map/bunker_tiles_top.png";
                node_scale = trench_node_size;
        }
        else if (node_stand.stand == artillerie_stand.stand) {
                //sprite_path = "";
                node_scale = artillery_node_size;
        }
        else if (node_stand.stand == bunker_stand.stand) {
                //sprite_path = "";
                node_scale = bunker_node_size;
        }
        else if (node_stand.stand == hiding_stand.stand) {
                hiding_nodes.push_back(node_go);
```

```
                CreateMapSprite("Map/walls_frontview.png", Transform(position +
hiding_sprite_node_offset, hiding_sprite_size));
                //sprite_path = "";
                node_scale = hiding_node_size;
        }

        node_go->AddComponent(new SpriteRenderer(white_color,
DataPool::GetTexture(sprite_path), 1.0f, Geometry::RECTANGLE));
        node_go->transform.scale = node_scale;

        node_go->onlyLayerReceive = true;
        AddGameObjectToLayer(node_go);

        return node_go;
}

void MapLayer::RotateTrenchNodes() {
        for (size_t i = 0; i < 7; i++) {
                trench_nodes.at(i)->transform.rotation = 45.0f;
        }
        for (size_t i = trench_nodes.size()-7; i < trench_nodes.size() ; i++) {
                trench_nodes.at(i)->transform.rotation = -45.0f;
        }
}

void MapLayer::CreateDeadBody(std::string sprite_path, glm::vec2 position) {
        GameObject* sp = new GameObject("map_sprite", Transform(position,
dead_body_size));
        sp->AddComponent(new SpriteRenderer(glm::vec4(1.0f, 1.0f, 1.0f, 1.0f),
DataPool::GetTexture(sprite_path), 1.0f, Geometry::RECTANGLE));
        sp->AddComponent(new DestroyOverTime(dead_body_lasting_time));
        this->AddGameObjectToLayer(sp);
}

void MapLayer::CreateMapSprite(std::string sprite_path, Transform trans) {
        GameObject* sp = new GameObject("map_sprite", trans);
        sp->AddComponent(new SpriteRenderer(glm::vec4(1.0f, 1.0f, 1.0f, 1.0f),
DataPool::GetTexture(sprite_path), 1.0f, Geometry::RECTANGLE));
        this->AddGameObjectToLayer(sp);
}

void MapLayer::CreateGameMap(std::vector<std::pair<std::vector<glm::vec2>,
Stand>>& stands_with_nodes, std::vector<std::pair<std::vector<Transform>,
std::string>>& map_sprites) {
        // for every stand (vector of its node-positions)
        for (auto& node : stands_with_nodes) {

                Stand s = node.second;

                // for every node-position in that stand
                for (auto& pos : node.first) {
                        // create a node belonging to the stand with the desired
position
                        CreateNode(pos, s);
                }
        }

        for (auto& pair : map_sprites) {
                for (auto& sprite: pair.first) {
                        GameObject* sp = new GameObject("map_sprite", sprite);
                        sp->AddComponent(new SpriteRenderer(glm::vec4(1.0f, 1.0f,
1.0f, 1.0f), DataPool::GetTexture(pair.second), 1.0f, Geometry::RECTANGLE));
```

```
                        this->AddGameObjectToLayer(sp);
                }
        }

        RotateTrenchNodes();
}


// TODO: Change to vec2 2D-vector, cubes are placeholders for debugging and
visualisation
void MapLayer::CreateEnemyGrid(const uint8_t x_size, const uint8_t y_size, const
float offset, const glm::vec2 mid_pos) {

        std::vector<std::vector<GameObject*>> grid;  //create 2D vector
        float cube_rad = 0.25f;
        glm::vec2 whole_len = glm::vec2(x_size * cube_rad * 2.0f + (x_size - 1) *
offset, y_size * cube_rad * 2.0f + (y_size - 1) * offset);                      //length of all
cubes + offsets between
        glm::vec2 start_pos = glm::vec2(mid_pos.x - whole_len.x / 2.0f, mid_pos.y +
whole_len.y / 2.0f);                    // starts top left

        for (size_t x = 0; x < x_size; x++) {
                std::vector<GameObject*> y_row; // create new y row vector
                for (size_t y = 0; y < y_size; y++) {
                        y_row.push_back(new GameObject(std::to_string(x) +
std::to_string(y),
                                Transform(glm::vec2(start_pos.x + x * (offset +
cube_rad * 2.0f), start_pos.y - y * (offset + 2.0f * cube_rad)),
                                        glm::vec2(2 * cube_rad))));                     // add
gameobjects to it

                        y_row.at(y)->AddComponent(new SpriteRenderer(glm::vec4(0.0f,
0.0f, 1.0f, 0.0f), Geometry::RECTANGLE));                       // add SpriteRenderer
                        y_row.at(y)->AddComponent(new Node(nullptr));
                        //y_row.at(y)->AddTag("move_node");
                        AddGameObjectToLayer(y_row.at(y));
                }
                grid.emplace_back(y_row); // add y_row to grid at index x
        }
        enemy_grid = grid;
}

GameObject* MapLayer::CreateBuilding(Transform transform, std::string type) {
        Layer* layer = this;
        GameObject* building = new GameObject(type + "-building", transform);

        if (type == "medic") {
                building->AddTag("medic_building");
                building->AddComponent(new SpriteRenderer(glm::vec4(1.0f, 1.0f,
1.0f, 1.0f), DataPool::GetTexture("Buildings/med_tent.png"), 1.0f,
Geometry::RECTANGLE));
        }
        else if (type == "engineer") {
                building->AddTag("engineer_building");

                building->AddComponent(new SpriteRenderer(glm::vec4(1.0f, 1.0f,
1.0f, 1.0f), DataPool::GetTexture("Buildings/engineer_tent.png"), 1.0f,
Geometry::RECTANGLE));
        }
        else {
                LOG_WARN("WARNING: probably no existing type given when creating a
building");
```

```
		}
		building->onlyLayerReceive = true;
		AddGameObjectToLayer(building);

		return building;
}

GameObject* MapLayer::CreateMg(glm::vec2 mg_position, GameObject* mg_node) {

		GameObject* mg = new GameObject("mg", Transform(mg_position, mg_size));
		mg->AddComponent(new SpriteSheet(glm::vec4(1.0f, 1.0f, 1.0f, 1.0f),
DataPool::GetTexture("Buildings/mg_animation.png"), 80.0f, 224.0f, 0.0f, 0.0f,
glm::vec2(0, 0)));
		mg->GetComponent<SpriteSheet>()->ChangeSprite(glm::vec2(0, 0));
		mg->AddComponent(new
SingleAnimation(DataPool::GetTexture("Anims/mg_animation.png"), 104.0f, 296.0f,
16.0f, 16.0f, glm::vec2(0.0f, 0.0f), glm::vec2(3.0f, 0.0f), 10, glm::vec2(0.0f,
0.0f), DataPool::GetTexture("Anims/mg_animation.png"), 104.0f, 296.0f, 16.0f,
16.0f));
		mg->AddComponent(new MgComponent(mg_node));
		AddGameObjectToLayer(mg);

		return mg;
}

GameObject* MapLayer::CreateArtillery(glm::vec2 artillery_position, GameObject*
artillery_node) {

		GameObject* artillery = new GameObject("artillery",
Transform(artillery_position, artillery_size));
		artillery->AddComponent(new SpriteSheet(glm::vec4(1.0f, 1.0f, 1.0f, 1.0f),
DataPool::GetTexture("Buildings/artillerie_front_north.png"), 248.0f, 400.0f,
16.0f, 16.0f, glm::vec2(0.0f, 0.0f)));
		artillery->GetComponent<SpriteSheet>()->ChangeSprite(glm::vec2(0.0f,
0.0f));
		artillery->AddComponent(new ArtilleryComponent(artillery_node));

		artillery->AddComponent(new
SingleAnimation(DataPool::GetTexture("Buildings/artillerie_front_north.png"),
248.0f, 400.0f, 16.0f, 16.0f, glm::vec2(0.0f, 0.0f), glm::vec2(2.0f, 0.0f),
artillery_shoot_anim_speed, glm::vec2(0.0f, 0.0f),
DataPool::GetTexture("Buildings/artillerie_front_north.png"), 248.0f, 400.0f,
16.0f, 16.0f));

		AddGameObjectToLayer(artillery);
		return artillery;
}

bool MapLayer::GameObjectPressed(GameObjectPressedEvent& e) {
		// check if e belongs to allylayer - otherwise return false

		GameObject* clicked_mapobject = e.GetGameObject();

		if (clicked_mapobject == gameScene->GetActiveBuilding()) {
			gameScene->uiLayer->DeactivateBuildingUI();
			gameScene->SetActiveBuilding(nullptr);
			return true;
		}

		if (clicked_mapobject->HasTag("move_node")) {
```

```
                if (gameScene->GetActiveCharacter() == nullptr || !gameScene-
>GetActiveCharacter()->HasTag("soldier") || clicked_mapobject-
>GetComponent<Node>()->is_occupied) return false;

                gameScene->GetActiveCharacter()->GetComponent<SoldierBehaviour>()-
>SoldierMove(clicked_mapobject);
                return true;
        }
        else if (clicked_mapobject->HasTag("medic_building")) {
                gameScene->uiLayer->DeactivateBuildingUI();
                gameScene->uiLayer->ActivateMedicBuildlingUI();
        }
        else if (clicked_mapobject->HasTag("engineer_building")) {
                gameScene->uiLayer->DeactivateBuildingUI();
                gameScene->uiLayer->ActivateEngineerBuildingUI();
        }
        else {
                return false;
        }

        gameScene->SetActiveBuilding(clicked_mapobject);

        return true;

}
```

EnemyLayer

In der EnemyLayer ist lediglich eine Funktion zum Erstellen der Gegner definiert, da hier nicht mehr benötigt wird.

```
void EnemyLayer::Update(const float dt)
{

}

void EnemyLayer::OnEvent(Event& event)
{
        EventDispatcher dispatcher(event);
        dispatcher.dispatch<KeyPressedEvent>([this](KeyPressedEvent& e)
        {
                if (e.getKeyCode() == KEY_G)
                {
                        CreateEnemy("enemy", glm::vec2(RandomF(enemy_grid_startpos.x -
enemy_spawn_random_x_radius, enemy_grid_startpos.x +
enemy_spawn_random_x_radius), enemy_spawn_y_position));
                        return true;
                }
        return false;
        });

}

GameObject* EnemyLayer::CreateEnemy(std::string name, glm::vec2 spawn_pos) {
        GameObject* enemy_go = new GameObject(name, Transform(spawn_pos,
enemy_scale));

        enemy_go->AddTag("enemy");
```

```
        enemy_go->AddComponent(new SpriteRenderer(glm::vec4(1.0f, 0.0f, 0.0f,
1.0f), Geometry::RECTANGLE));                    // TODO: Change to sprite_path
        enemy_go->AddComponent(new Movement(enemy_movement_speed));
        enemy_go->AddComponent(new EnemyBehaviour());
        enemy_go->AddComponent(new EnemyShooting());
        enemy_go->AddComponent(new Health(enemy_health));

        AddGameObjectToLayer(enemy_go);

        return enemy_go;
}
```

BackgroundLayer

Die BackgroundLayer ist für die Erstellung des Hintergrundes der Karte, also den Matsch, zuständig.

```
void BackgroundLayer::OnAttach() {
        glm::vec2 background_tile_rad = background_tile_size / 2.0f;
        CreateBackgroundTile(glm::vec2(-background_tile_rad.x,
background_tile_rad.y));
        CreateBackgroundTile(glm::vec2(background_tile_rad.x,
background_tile_rad.y));
        CreateBackgroundTile(glm::vec2(-background_tile_rad.x, -
background_tile_rad.y));
        CreateBackgroundTile(glm::vec2(background_tile_rad.x, -
background_tile_rad.y));
}

void BackgroundLayer::CreateBackgroundTile(glm::vec2 pos) {
        GameObject* tile = new GameObject("background-tile", Transform(pos,
background_tile_size), ProjectionMode::PERSPECTIVE);
        tile->AddComponent(new SpriteRenderer(glm::vec4(1.0f, 1.0f, 1.0f, 1.0f),
DataPool::GetTexture("Map/fullscreen_dirtground.png"), 1.0f,
Geometry::RECTANGLE));
        this->AddGameObjectToLayer(tile);
}
```

UILayer

In der UILayer sind alle Funktionen für das Aktivieren und Deaktivieren der einzelnen grafischen Benutzeroberflächen-Elemente im Spiel definiert, also der Soldaten, Mechaniker, Ärzte, dem Arzt-Gebäude, dem Mechaniker-Gebäude und dem Nachschub-Menü (dieses wird aktiviert, wenn eine Welle abgeschlossen ist). Diesmal wird nur der Header gezeigt, da dies sonst 4 Seiten nicht durchblickbarer Code ist.#

```
class UILayer : public Layer
{
public:

        UILayer();
        ~UILayer() override;
```

```
        void OnAttach() override;
        void OnDetach() override;
        void Update(const float dt) override;
        void OnEvent(Event& event) override {};

        void ActivateSoldierUI();
        void ActivateMedicUI();
        void ActivateEngineerUI();
        void ActivateMedicBuildlingUI();
        void ActivateEngineerBuildingUI();
        void ActivateSupplyMenuUI();
        void DeactivateCharacterUI();
        void DeactivateBuildingUI();
        void DeactivateSupplyMenuUI();

private:
        PictureBox* character_background;
        PictureBox* building_background;
        PictureBox* supply_menu_background;

        int hp;
        Label* hp_text;
};
```

MenuScene

Die MenuScene ist ein simples Hauptmenü mit zwei Knöpfen um das Spiel zu starten oder es zu beenden.

```
MenuScene::MenuScene() {
   backcolor = background_color;

   menuLayer = new MenuLayer();

   CreateElement("title_object_001.png", glm::vec2(0.0f, 3.0f), glm::vec2(11.0f,
3.0f));
   CreateButton("play_object_001.png", glm::vec2(0.0f, -1.0f), "play");
   CreateButton("quit_button.png", glm::vec2(0.0f, -3.0f), "quit");
}

GameObject* MenuScene::CreateElement(std::string sprite_name, glm::vec2 position,
glm::vec2 size) {

   GameObject* element = new GameObject("menu_element", Transform(position,
size));
   element->AddComponent(new SpriteRenderer(glm::vec4(1.0f, 1.0f, 1.0f, 1.0f),
DataPool::GetTexture("MainMenu/" + sprite_name), 1.0f, Geometry::RECTANGLE));
   menuLayer->AddGameObjectToLayer(element);
   return element;
}

GameObject* MenuScene::CreateButton(std::string sprite_name, glm::vec2 position,
std::string action) {

   GameObject* button = new GameObject("menu_button", Transform(position,
button_size));
   button->AddComponent(new SpriteRenderer(glm::vec4(1.0f, 1.0f, 1.0f, 1.0f),
DataPool::GetTexture("MainMenu/" + sprite_name), 1.0f, Geometry::RECTANGLE,
true));
```

```
    button->AddTag(action);
    menuLayer->AddGameObjectToLayer(button);
    return button;
}
```

Layers

MenuLayer

Diese Layer regelt das Laden der richtigen Szene (GameScene), sobald der dafür zuständige Knopf gedrückt wurde.

```
bool MenuLayer::OnGameObjectClick(GameObjectPressedEvent& e)
{
    GameObject* go = e.GetGameObject();

    if (go->HasTag("play")) {
        Application::ChangeScene(gameScene);
    }
    else if (go->HasTag("quit")) {
        Application::GetInstance()->Exit();
    }

    return true;
}

void MenuLayer::OnEvent(Event& event)
{
            EventDispatcher dispatcher(event);
            dispatcher.dispatch<GameObjectPressedEvent>(BIND_EVENT_FN(MenuLayer::OnGame
ObjectClick));
```

Engine

Application

Damit die Engine von diversen Anwendungen, wie beispielsweise das Spiel des Game-Teams auf einfachen Wege benutzt werden kann, verwendet man die Application Klasse. Diese Klasse kümmert sich um alle Initialisierungen und Einstellungen die im Vorhinein getroffen werden müssen, um eine Anwendung Grafik bereit zu machen.

```
#pragma once

#ifdef CORE_PLATFORM_WINDOWS
#ifdef INCLUDE_MAIN

extern core::Application* core::CreateApplication();

int main(int argc, char ** argv)
{
	auto app = core::CreateApplication();
	app->Run();
	delete app;
}

#endif
#endif
```

Wie man erkennen kann, ist dies das Grundlegende Skelett der ganzen Anwendung. Die Application dient hier als Bindeglied zwischen Spiel und Engine.

Die Main-Funktion ist allseits dafür bekannt, dass sie das erste ist, was in einem Computerprogramm aufgerufen wird. Wir deklarieren die Application-Klasse als etwas, welches sich „extern“ also fremd irgendwo im Code des Spieles, befindet. Diese Zeilen erstellen lediglich eine Application, welche vom Spiel definiert wurde, und rufen die gesamte Logik-Pipeline zwischen Spiel und Engine auf.

```
class Conqueror : public core::Application {
public:
	Conqueror() {

	}

	~Conqueror() {

	}

	void Init() override {
		Application::GetWindow()->SetVSync(true);
		gameScene = new GameScene();
		menuScene = new MenuScene();

		DataPool::GetFont(ui_font_family);

		Application::ChangeScene(menuScene);
	}

};

core::Application* core::CreateApplication() {
	return new Conqueror();
}
```

Hier ist die Conqueror Application definiert, und dient als Anwendungsspezifisches Objekt. Das gute hierbei ist, dass wir mit Leichtigkeit auch ein ganz anderes Spiel erstellen können, welches einen viel abstrakteren Initialisierungsvorgang haben kann. Dieser Objekt-Orientierte Anspruch hilft uns dabei, unsere Anwendung universell anwendbar zu machen. Beispielsweise kann man bei einer Kartenapplikation, so nun vor der Initialisierung Karten laden, oder Dateien vorbereiten. Dies löst eine Art unübersichtliche „Bloatware“ und hilft sowohl Benutzer als auch Core-Developer bei Problemfindung.

Die besagte Logik-Pipeline sieht wie folgt aus:

```
Application::Application()
{
	Log::Init();
	CORE_ASSERT(!instance, "application is already instanced!");
	instance = this;

	Core::Init();

	window = Window::Create();
	SetEventCallback(BIND_EVENT_FN(Application::OnEvent));

	Renderer::Init();
}
```

```
while (gameRunning)
{
	window->PollEvents();
	ProcessQueues();

	if (currentScene != nullptr) {
		// request color
		RenderCommand::ClearColor(currentScene->GetBackcolor());


		if (queuedScene != nullptr) {
			currentScene->Stop();

			currentScene = queuedScene;
			currentScene->Start();

			queuedScene = nullptr;
		}

		for (Layer* layer : layerStack)
		{
			if (layer->IsAttached())
				layer->Update(dt);
		}

		currentScene->Update();

		Input::ProcessInput();

	}
	else if (warn) {
		LOG_CORE_ERROR("No Scene exists. Make sure to call Applica-
tion::changeScene() in the 'init' function of your Application class");
		warn = false;
	}

	window->SwapBuffers();

	imguiEnabledBefore = imguiEnabled;
	resizing = false;
}
```

Zuerst wird das Log-System initialisiert, danach wird das interne ID-System gestartet. Dies wird benötigt, da dies jedem Objekt eine einzigartige ID gibt. Durch diese ID können dann wieder diese Objekte eindeutig zugeordnet werden. Danach wird das Window erstellt und den Event-Callback gesetzt. Dieser Callback ist wichtig, da dadurch alle Events (z.B. MouseMoved, KeyPressed oder WindowResized) angenommen und an das Spiel verteilt werden können. Als letztes wird noch der Renderer gestartet.
Als nächstes folgt dann der while-Loop. In dieser Schleife befndet dich das gesamte Programm. Das heisst ein Schleifendurchlauf ist ein Frame. Kurz gesagt, schauen wir, ob eine Szene gerade aktiv ist und wenn dies der Fall ist, dann gehen wir durch diese Szene und aktualisieren alle Objekte und funktionen. In der Szene wird dann auch als letztes der Renderer aufgerufen, der

dann alles auf den Bildschirm zeichnet. Wenn keine Szene aktiv ist, dann geben wir eine Warnung in der Konsole aus.

Szenensystem

Damit der Benutzer, oder auch das Game-Team, das Spiel strukturiert unterordnen kann, in Bereiche wie: Menü, Ladebildschirm, Spielbildschirm, … haben wir ein Szenensystem entwickelt. Dieses System erwartet eine Referenz zu einer Szenen Klasse. Diese Szene wird im spezifizierten Aktualisierungsintervall aufgerufen und erfüllt die Objekterzeugung des Game-Developers. Dies dient lediglich ebenfalls auch für die Übersicht des Spieles.

```
class Scene {
   friend class Application;
public:
   std::vector<GameObject*> gameObjects;

   Scene();
   virtual ~Scene() = default;

   Shr<Camera> GetCamera();
   glm::vec4& GetBackcolor();

   void Start();
   void Stop();


   void AddLayer(Layer* layer);
   void AddOverlay(Layer* layer);
   void RemoveLayer(Layer* layer) const;
   void RemoveOverlay(Layer* layer) const;

protected:
   Shr<Camera> camera = nullptr;
   glm::vec4 backcolor = {};

   virtual void OnStart() = 0;
   virtual void OnStop() = 0;
   virtual void OnUpdate() = 0;
   virtual void OnEvent(Event& e) = 0;
   virtual void Imgui(float dt) {}

private:
   bool isRunning = false;

   void Update();

};
```

Die Szenenklasse ist wie folgt strukturiert. Sie aktualisiert das jeweilige Spiel und ist auch in der Lage Objekte zu Initialisieren. Die Init und Update Logik macht dies möglich. Weitere

Funktionalitäten dienen den Debugging-Zwecken. Jede Szene ist so modifizierbar, dass sie auch ihre eigene Projektion haben kann. Notiz um die Datenverwaltung klarzustellen: Alle GameObjects werden in der Szene gespeichert, und können vom Renderer abgerufen werden.

GameObjects:

GameObjects sind Objekte die dem Game-Team ermöglichen verschiedenste Geometrien auf dem Bildschirm anzuzeigen. GameObjecte sind aber viel zu allgemein um eine bestimmte Form anzuzeigen. Deswegen werden sie begrenzt. Das einzige was gespeichert wird, ist die Transformation, sprich die Position und Skalierung.

Beispiel:

```
GameObject* bullet = new GameObject("bullet", Transform(glm::vec2(0.0f, 0.0f),
glm::vec2(2.0f, 2.0f)));
```

Der erste Parameter dient lediglich nur als Debughilfe.

Will man dieses GameObject aber nutzbar machen, muss man ihm einen Component zuschreiben. Ein Component grenzt das GameObject ein. Beispielsweise sollte man für das Bullet-GameObject den CircleRenderer Component verwenden, da dieses GameObject die Geometrie eines Kreises erhalten soll.

```
bullet->AddComponent(new CircleRenderer(bullet_color, 1.0f, 0.005));
```

Die CircleRenderer Parameter dienen dem Style (Farbe, Radius, Strichstärke).

Components des GameObjects kümmern sich nicht nur um die Form, sondern auch um die Logik. Dies hilft dem Game-Team, GameObjects funktional in ihre Logik-Kategorie einzuordnen. Hier bekommt das Kugel-GameObject die Bewegungslogik mithilfe des Components zugeschrieben.

```
bullet->AddComponent(new Movement(bullet_speed, targetPos));
```

Anhand dieser Beispiele lässt sich zeigen, dass Components die großen Strukturen des Spiele vereinfacht. Diese Technologie wurde vom Core-Team implementiert.

Layers:

Layer sind dazu da, um die Reihenfolge des Anzeigens der GameObjects festzulegen. Der Layer ist sozusagen ein Prioritätsverwalter.

```
void GameScene::OnStart()
{
   Supply::Init();

   AddLayer(backgroundLayer);
   AddLayer(mapLayer);
   AddLayer(enemyLayer);
   AddLayer(allyLayer);

   AddOverlay(uiLayer);
}
```

In dieser Szene werden die Layer hinzugefügt und nach Reihenfolge abwärts Priorisiert. Der backgroundLayer ist der Hintergrund und hat daher die niedrigste Priorität. Beispiel:

Bildbeschreibung: Hier wird ein Objekt des allyLayer über dem backgroundLayer gerendert.

So würde man ein GameObject zu einem Layer hinzufügen:

```
enemyLayer->AddGameObjectToLayer(bullet);
```

Nun wird das Bullet-GameObject mit der Priorität des enemyLayer gerendert.

UI-System:

Damit benutzerfreundliche User-Interfaces gemacht werden können, hat das Core-Team ein UI-System entworfen, welches vom Game-Team verwendet um die Lebensanzeige der Charaktere im Spiel anzuzeigen.


```
character_background = new PictureBox(white_color,
Transform(ui_character_position, ui_background_size),
DataPool::GetTexture("UI/box_1.png"), Type::Rectangle);
```

So ist man in der Lage eine Hintergrundbox zu erstellen.

Die Erstellung dieser Mechaniker-UI würde dann so aussehen:

```
void UILayer::ActivateEngineerBuildlingUI() {
	building_background = new PictureBox(white_color, Transform(ui_building_po-
sition, ui_building_background_size), DataPool::GetTexture("UI/box_2.png"),
Type::Rectangle);

	PictureBox* header = new PictureBox(white_color, ui_header_transform, Data-
Pool::GetTexture("UI/box_small.png"), Type::Rectangle);  // picturebox with pic-
ture of medic-bulding
	Label* name = new Label("AERZTE", ui_font_color, ui_header_build-
ing_name_transform, DataPool::GetFont(ui_font_family), "ui_buildling_name");
	PictureBox* icon = new PictureBox(white_color, ui_building_icon_transform,
DataPool::GetTexture("UI/med_icon_clear.png"), Type::Rectangle);
	header->AddChildObject(name);

	Label* count = new Label(std::to_string(gameScene->mapLayer->medicBuilding-
>GetComponent<MedicBuilding>()->GetAvailableMedics()) + " verfuegbar",
ui_font_color, ui_building_count_transform, DataPool::GetFont(ui_font_family));

	building_background->AddChildObject(header);
	building_background->AddChildObject(count);
	building_background->AddChildObject(icon);

	AddUIObject(building_background, ProjectionMode::SCREEN);
}
```

Texturen:

Hintergründe

Dreckboden

Der Dreckboden ist von Abb. 1 inspiriert worden und war als erstes ein einzelner Block, wie in Abb. 3, um anschließend mit diesem Muster den ganzen Bildschirm auszufüllen (Abb. 2), um das Kreieren eines komplett ausfüllenden Bodens zu vereinfachen und eine simple Basis zu erschaffen auf dem sich die Figuren bewegen können.

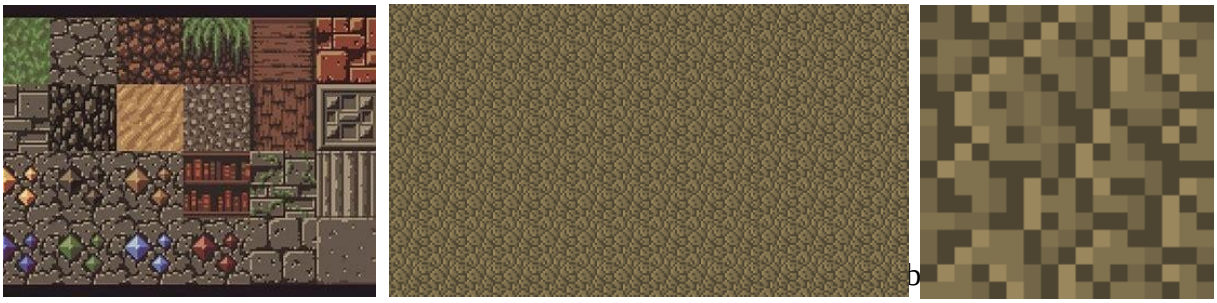
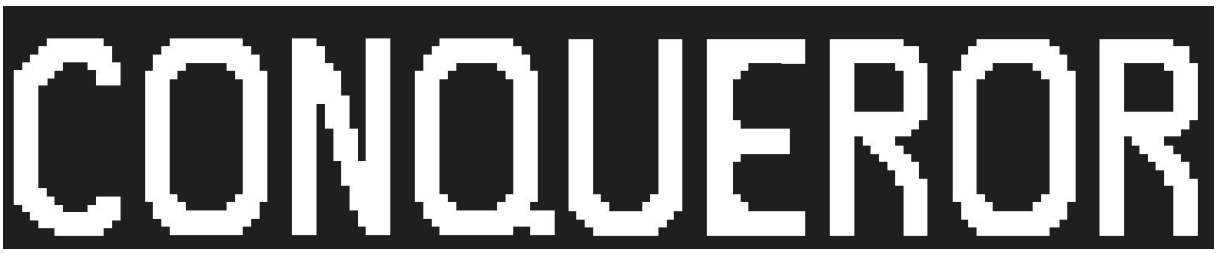


Abb. 3

Start-Bildschirm

Für den Startbildschirm, welcher bei Beginn des Spieles erscheinen soll, bietet eine simple Überschrift und 4 Button für die nötigen Funktionen eine simple Benutzeroberfläche.
Jeder dieser Button hat insgesamt 3 Sprites. Einen Grundsprite, der erscheint sofern der Button nicht mit der Maus berührt wird, einen Sprite, wo die Maus den Button berührt und er etwas vergraut und einen wo der Button angeklickt wird und etwas kleiner wird, wie man es in vielen Spielmenüs kennt.

Überschrift



Play

- Zum Starten des Spieles#

Basis

Mausberührung

Klicken

Settings

- Um verschiedene Einstellungen vorzunehmen

Basis

Mausberührung

Klicken

Credits

- Um Credits anzuzeigen

Basis

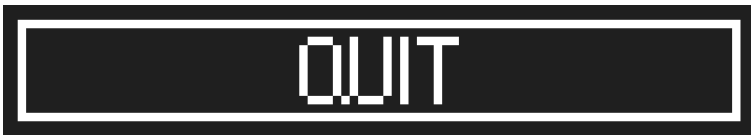
Mausberührung

Klicken

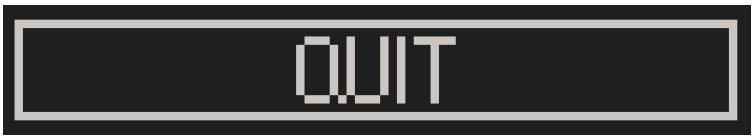
Quit

- Um wieder zum normalen Start-Menü zurückzukehren

Basis



Mausberührung



Klicken

NPC

Da wir uns im Vorhinein für den Pixelstil entschieden hatten und es einfach halten wollten, wurden die NPC an die, des Spieles Stardew Valley inspiriert, was durch die unteren beiden Fotos zu erkennen ist.

Abb. 4

Abb. 5

Designauswahl

Die Designs der NPC wurden an im ersten Weltkrieg vorkommen Uniformen angelehnt, um eine Parallele zu diesem Ereignis zu schaffen.

Deutscher Soldat

Referenz


Abb. 6


Abb. 7

Britischer Soldat

Referenz


Abb. 8

Französischer Soldat

Referenz

Abb. 9

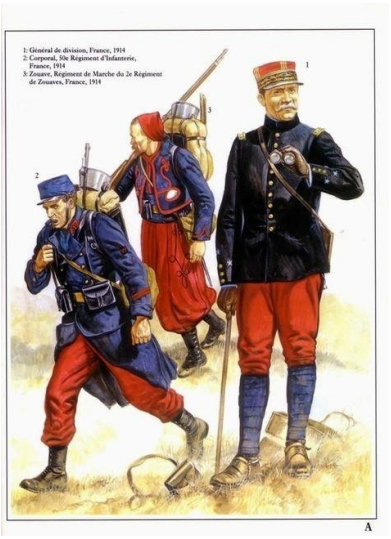


Abb. 10

Ingenieur

Referenz

Abb. 11

Sanitäter

Referenz

Abb. 12

Animation

Jeder NPC brauchte mehrere Sprites, um verschiedene Bewegungen und Aktionen zu visualisieren. Dazu gehören mindestens jeweils drei verschiedene Sprites für das Laufen in jede Himmelsrichtung, die Visualisierung einer Aktion, wie Bauen, Heilen oder das Mitführen einer Waffe und eine Animation für den Tod des jeweiligen NPC. Der deutsche Soldat war der erste NPC der entstand. Seine Grundzüge wurden anschließend auf alle anderen NPCs soweit es ging übertragen.

Tod

Laufen

Maschinengewehr

Heilung

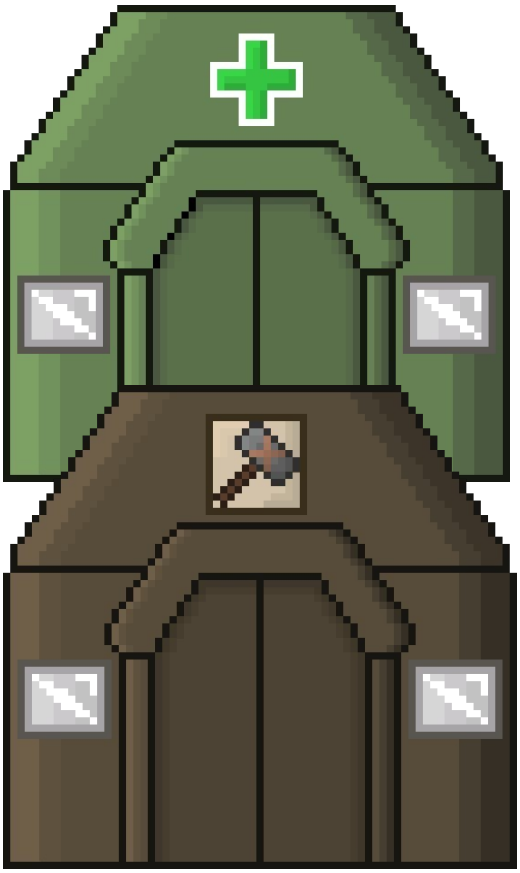
- Mit grünem Glitzern, um den Heileffekt zu visualisieren

Bauen

- Da die Animation ursprünglich zu schnell war und somit unrealistisch wirkte, wird jeder frame doppelt gezeigt.

Gebäude und Objekte

Medizinzelt



Das Medizinzelt stellt den Unterschlupf der Sanitäter dar. Sobald also ein Sanitäter per Sanitäter Button gerufen wird, kommt ein Sanitäter-NPC aus dem Zelt heraus. Das Design wurde von dem Zelt aus Abb. 13 inspiriert.

Ingenieurszelt

Das Ingenieurszelt hat dieselbe Funktion wie das Medizinzelt nur ist es für die Ingenieurs-NPC bestimmt. Genauso wie beim Medizinzelt kommen daher die Ingenieure aus dieser Grafik, wenn man einen Ingenieur zu sich ruft.

Für dieses Zelt wurde, dass Medizinzelt zuerst umgefärbt, um es passender zum Ingenieur zu machen und anschließend anstelle des Medizinkreuzes das Icon für den Ingenieur als Logo genommen.

Referenz

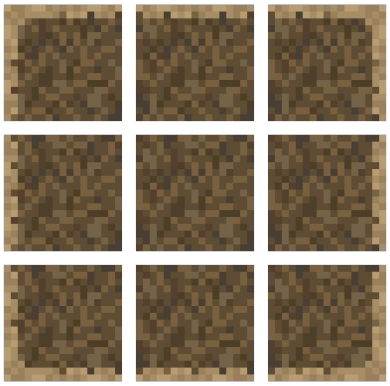
Abb. 13

Bunker (Schützengraben)

Um den Bunker, der eine Art Schützengraben darstellt kenntlich zu machen und einfach gestalten zu können, gibt es Bodensprites für die entsprechenden Flächen. Es existiert ein Sprite

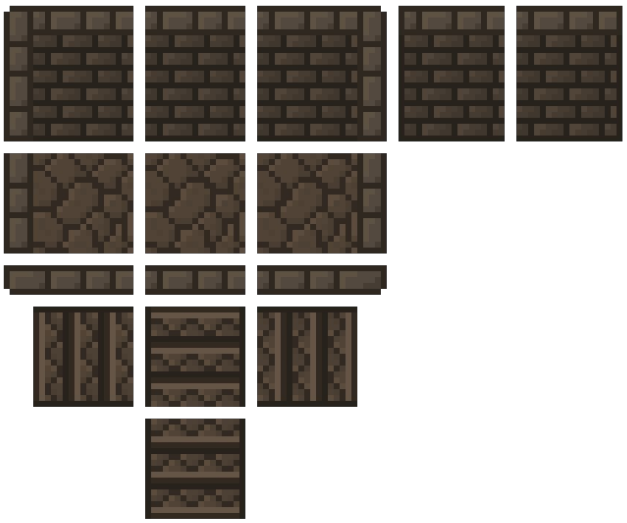
für jede mögliche Richtung und Anordnung. Darunter gibt es die Mittelfläche, Randflächen, Eckflächen und Flächen für engere Wege.

Bunker Tiles Enge Wege



Bunker (unterirdische Basis)

Der unterirdische Bunker, welcher als eine Art Basis dient, lässt sich, ähnlich wie bei den Schützengräben mit verschiedenen Tiles zusammenbauen.

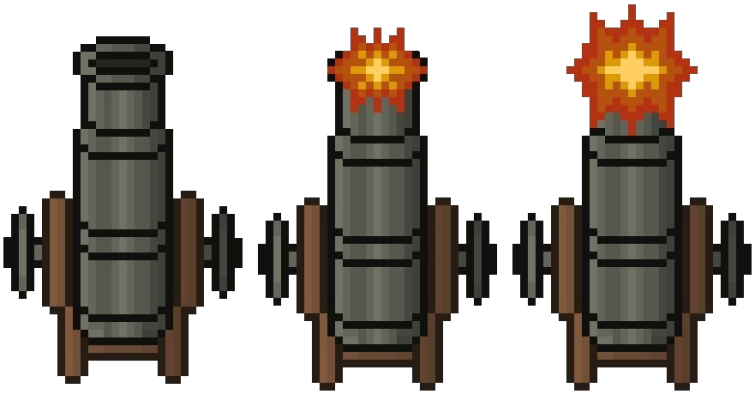


Referenz

Abb. 14

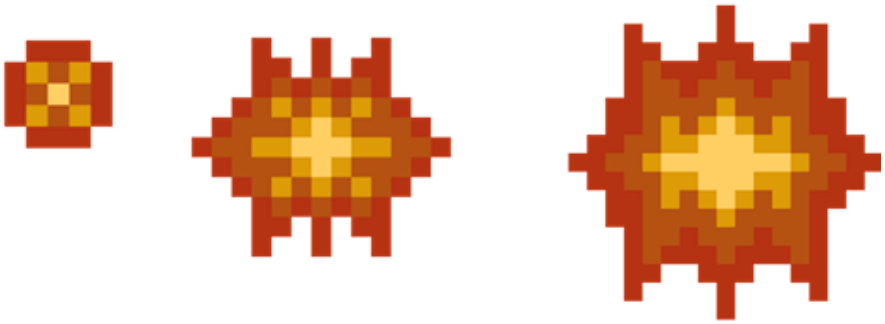
Artillerie

Die Artillerie ist eine Standwaffe die platziert werden kann. Sie feuert Explosionen ab, welche mithilfe der Explosionsanimation visualisiert werden.



Explosion

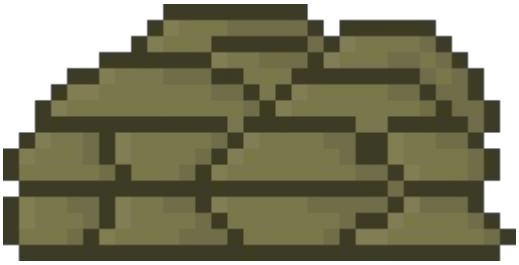
Die Explosionsanimation wird bei Abschuss einer Artillerie oder eines Maschinengewehr angezeigt, um den Schuss deutlicher zu machen. Die Animation enthält drei Sprites, wobei bei einem Abschuss die letzten 2 von rechts angezeigt werden und bei Treffen auf ein Hindernis alle drei.



Sandsäcke

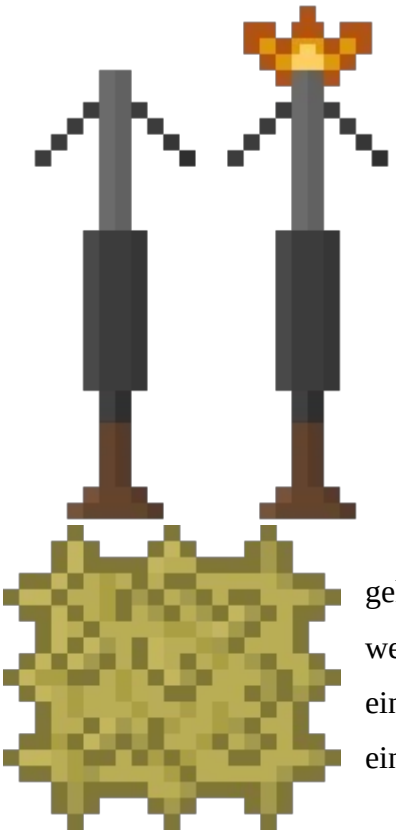
Als Schutzmauer wurde ein Haufen Sandsäcke gepixelt hinter denen Soldaten sich in Deckung begeben können.

Frontansicht



Seitliche Ansicht

Standmaschinengwehr



Das Standmaschinengewehr ist eine kleinere Standwaffe, die von Soldaten bedient werden kann. Bei Abschuss wird ein Sprite mit einer Explosion vorne abgespielt, wie beim rechten Exemplar zu sehen ist.

Aktionsfeld

Das Aktionsfeld, welches hier mit ein wenig Heu gekennzeichnet wird, soll Orte an denen eine Aktion ausgeführt werden kann besser erkennbar machen. Ein Beispiel dafür wäre eine Position vor einem Maschinengewehr. Hier kann man nun einen Soldaten dieser Position zuweisen, damit es dieses benutzt.

Icons

Icon-Holder

Der Icon-Holder bildet die Bedienfläche, bzw. Button, auf der das jeweilige Icon platziert werden kann.

Sanitäter-Icon

Das Sanitäter Icon, welches mit einer Medizinflasche kenntlich gemacht wurde, dient dazu Sanitäter zu rufen

Ingenieur-Icon

Das Ingenieurs-Icon dient dazu Ingenieure zu rufen, um diese bauen zu lassen

Soldat-Icon

Das Soldat-Icon dient dazu Soldaten zu sich zu rufen

Maschinengewehr-Icon

Das Maschinengewehr-Icon dient dazu Aktionen mit den Maschinengewehren auszuführen, wie einen Soldaten zuzuweisen

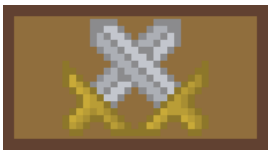
Artillerie-Icon

Das Artillerie-Icon dient dazu Aktionen mit den Artillerien auszuführen, wie einen Soldaten zuzuweisen.

Deckung-Icon

Das Deckung-Icon dient dazu Soldaten sich in Deckung bringen zu lassen.

Welle-Starten-Icon



Dieser Button startet eine Angriffswelle.

Welle-Stoppen-Icon

Dieser Button stoppt eine gestartete Angriffswelle vorzeitig.

Lebensanzeige (HP-Leiste)/Erfahrungsanzeige (EXP-Leiste)

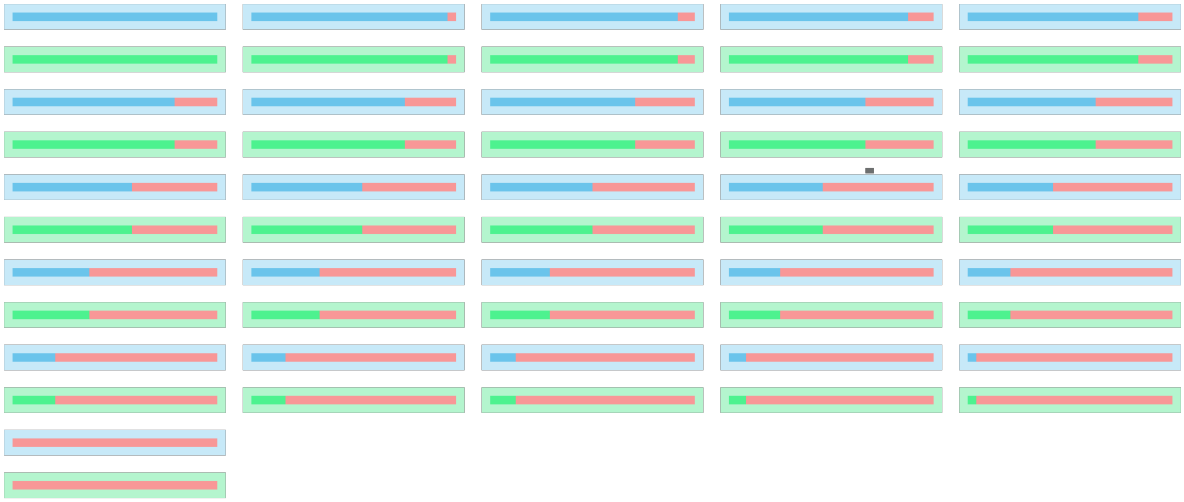
HP-Symbol

Dieses Symbol visualisiert Lebenspunkte (Health Points) der Soldaten. Sind alle HP verbraucht stirbt ein Soldat

XP-Symbol

Dieses Symbol visualisiert Erfahrungspunkte (E(X)perience Points) der Soldaten. Je mehr Erfahrung eine Figur hat desto stärker sind ihre Fähigkeiten

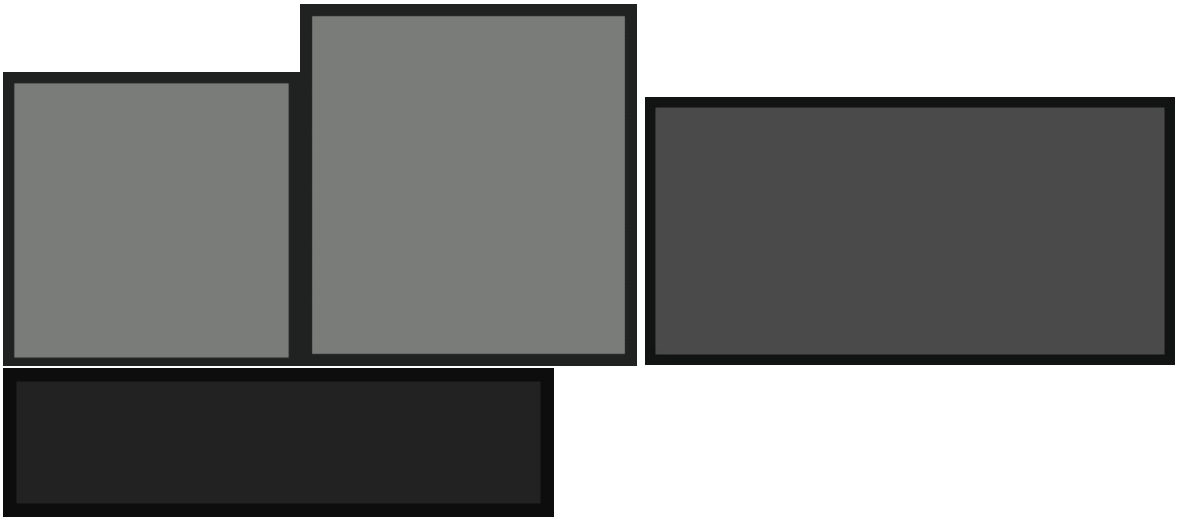
Die Dazugehörigen Anzeigeleisten



Wie zu erkennen ist, sind die jeweiligen Anzeigeleisten in den passenden Farben zu ihrem Symbol gestaltet worden. (Vergleiche HP-Symbol und XP-Symbol). Es gibt einen Sprite für jede Anzahl an Punkten.. Nicht vorhandene Punkte werden in der Leiste rot angezeigt.

Anzeigeboxen

Anzeigeboxen in verschiedenen Größen sollen Platz für unterschiedliche Texte und weitere Buttons geben.



Bildquellen

Abb. 1:

https://th.bing.com/th/id/OIP.1l9yR-8DKxVr8-dXalbBTAHaFF?w=230&h=180&c=7&r=0&o=5&dpr=1.5&pid=1.7

Abb. 4:

https://staticdelivery.nexusmods.com/mods/1303/images/1901/1901-1519085948-652371865.png

Abb. 5:

https://th.bing.com/th/id/OIP.9-7JVqdYxhyvm-ong4gpwAHaEj?w=267&h=180&c=7&r=0&o=5&dpr=1.5&pid=1.7

Abb. 6:

https://gmic.co.uk/uploads/monthly_2019_12/German_soldier_1914_uniform_Colorized_1.png.af2286ac6c4b8a79fcf5e8287c4a0663.png

Abb. 7:

https://i.pinimg.com/474x/2b/b6/d6/2bb6d614ce4bfe8b089730adfbe4f1bc.jpg

Abb. 8:
https://th.bing.com/th/id/R.4a4f567f5a3578660c40ee60bdb50acc?rik=7kRDihEwddfdxQ&pid=ImgRaw&r=0

Abb. 9: https://th.bing.com/th/id/OIP.-Hzb8Br1rJBYU8sU4X_RPwHaMM?w=194&h=320&c=7&r=0&o=5&dpr=1.5&pid=1.7

Abb. 10: https://i.pinimg.com/474x/22/cd/02/22cd02deb2d05c4a44f2a864ca264d3c--s%C3%A9culo-xx-french-army.jpg

Abb. 11:

https://th.bing.com/th/id/OIP.DHnWDv-6Uhjw06jPd628EAHaJY?pid=ImgDet&w=200&h=253&c=7&dpr=1,5

Abb. 12:

https://img.freepik.com/free-vector/pixel-art-set-of-cartoon-doctor-character_41992-1487.jpg?size=338&ext=jpg

Abb. 13:

https://th.bing.com/th/id/OIP.0F8YxXbWEtUpqum0imtXEwAAAA?w=186&h=180&c=7&r=0&o=5&dpr=1.5&pid=1.7

Abb. 14:

https://th.bing.com/th/id/R.7539e65c0a9b179675d224a94ff49924?rik=SG46tcApGEioWA&riu=http%3a%2f%2ftkool.jp%2fmz%2fassets%2fimages%2fgame%2fsample%2fss_04_05.pn

g&ehk=AbVux68VTHSrPkAOU%2fw8Q8kqZAdSLptEQc8G7znnXVs%3d&risl=&pid=ImgRaw&r=0

- Alle restlichen Bilder wurden selbst erstellt.