Houston Showdown

Projektdokumentation

TowerDefence Computerspiel basierend auf der OpenSource Engine IrrLicht3D

vorgelegt von

Dominik Hübner
Matrikelnummer 18942
dh028@hdm-stuttgart.de

Inhaltsverzeichnis

Einleitung

Ursprung und Spielprinzip

TowerDefence wird dem Genre der Strategiespiele zugeordnet, wenn auch mit stark vereinfachtem Spielablauf. Der Spieler muss versuchen, mittels selbstschießender Türme, ein relativ komplexes Labyrinth aufzubauen, welches von den Gegner, den sog. Creeps, durchquert werden muss. Ziel der Creeps ist es, von einem zu Beginn definierten Punkt in der Umgebung, die Karte zu überqueren. Da diese durch, die vom Spieler erbauten, Türme versperrt ist, muss ein wesentlich längerer Pfad zurückgelegt werden und der Gesamtschaden, welcher auf die Gegner im Laufe der Zeit verteilt wird, steigt.

Seine Ursprung fand diese Art von Spielen als Modifikation von StarCraft (erschienen 31.März 1998), einem Echtzeitstrategiespiel von Blizzard Entertainment. Ausgeliefert mit einem, für die damalige Zeit, mächtigen Map-Editor, war die Fangemeinde in der Lage den ursprünglichen Spielablauf soweit zu verändern, dass die daraus entstanden TowerDefence Modifikationen mit dem ursprünglichen Spiel beinahe nichts mehr gemein hatten.
Da Blizzard Entertainment auch bei Warcraft3, einem Strategiespiel in 3D Grafik, einen Map-Editor in ähnlichem Umfang auslieferte, wurden auch hier diverse TowerDefence Modifikationen erstellt und das Spielprinzip entwickelte sich zu einem eigenen Subgenre.
Heute wird diese Art von Spielen vor allem als Flash-Game produziert und hat seinen erfolgreichsten Vertreter mit Desktop TowerDefence gefunden. Auch der signifikante Rückschritt in den Bereichen Grafik und Sound, lässt keinen Zweifel daran aufkommen, dass TowerDefence zu den meistgespielten Casual-Games gehört.

Spielformen

Durch das relativ flexible Spielprinzip, haben sich mit der Zeit mehrere Spielformen entwickelt. Jede von diesen fordert von dem Spieler in

gewissen Ausmaßen andere Geschicklichkeiten, sowohl im Einzelspieler- als auch Mehrspielerbetrieb. Zu den erfolgreichsten gehören:

1. *Einzelspieler*

    a. *Classic Tower Defence*
    Die Gegner laufen einen festgelegten, jedoch nicht bebaubaren Weg entlang. Der Spieler kann Türme nur an der Seite dieser Pfade bauen, was den Schwerpunkt auf die Aufwertung der Türme, sowohl in zeitlicher Abfolge, als auch Aufwendung von nötigen Mitteln legt.

    b. *Mazing TD*
    Mittels Pathfinding können sich Gegner frei auf dem Spielfeld bewegen und der Spieler erhält die Freiheit Türme nach seiner eigens entwickelten Strategie zu platzieren, was die Schwierigkeit weniger auf die Aufwertung der Türme, als die Planung eines möglich komplexen Labyrinths verlagert.

2. *Mehrspieler*

    a. *Tower Wars*
    In diesem Spielmodus nehmen mehrere Spieler teil, welche auf voneinander abgetrennten Spielbereichen ihre Türme errichten können. Nur wenige Gegner werden von der Spiellogik selbst erstellt, mehr liegt es an den gegeneinander antretenden Spielern ihre finanziellen Mittel dazu einzusetzen, dem Gegner feindliche Einheiten zu entsenden.

    b. *Cooperative TD*
    Ähnlich der Einzelspielervariante Maze können Türme frei gebaut werden und Gegner wählen selbstständig ihren Pfad. Im Unterschied dazu nehmen aber mehrere Spieler teil, die über einen gemeinsamen Geldpool verfügen und sich die Lebenspunkte teilen.

Modifikation

In der hier vorliegenden Implementierung eines TowerDefence Spiels wurde hinsichtlich der Steuerung einem neuen Ansatz nachgegangen. Da sich die nötigen Steuerbefehle auf das Auswählen eines Turmtyps, das

Platzieren des Turms auf dem Spielfeld und gegebenenfalls das Entfernen eines Turms beschränken, scheint die Steuerung mittels Tastatur und Maus als zu aufwändig. Durch die Verwendung eines sog. Game-Controllers, wie er normalerweise bei Spielekonsolen verwendet wird, wird auch der Grundidee von Casual-Games genüge getan, den Aufwand für schnelle Spielerfolge möglichst gering zu halten und dennoch einen möglichst hohen Faktor an Spielspaß, in einem angenehmen Umfeld, zu ermöglichen. Das Verwenden von Maus und Tastatur, was den Spieler zwingend an einen Tisch bindet, widerstrebt diesem Prinzip.

Projektvorfeld

Aufgabenstellung

1. Kennenlernen der Engine mit mehreren Übungsprogrammen
2. Entwerfen der Datenstrukturen
3. Implementierung
    a. Logik
    b. Grafiken
    c. Microsoft XInput Driver einbinden
4. Modeling
5. Texturierung
6. Test und Debugging

Story

Die Welt wird von Aliens angegriffen und die Erdbevölkerung versucht mit letzter Kraft einen Schutzschild aufzubauen. Nahe Houston wird der dazu benötigte Generator erbaut, welcher von den Außerirdischen als erstes Ziel ihrer Invasion auserkoren wurde. Mit aller Kraft versuchen diese den Fortschritt der Bauarbeiten zu verhindern. Der Spieler muss nun mit einem möglichst komplex aufgebauten Labyrinth aus Verteidigungstürmen versuchen die Angreifer zurückzuhalten, bis das Schutzschild fertig gestellt ist und die eigentliche Gegenwehr beginnen kann.

Projektverlauf

Kennenlernen der Engine

Zu Beginn der Projektarbeit wurden 2 kleine Programme geschrieben um die Engine IrrLicht3D, welche bisher nur aus den Spezifikationen bekannt war, kennenzulernen. Zum einen eine Art „HelloWorld“-Programm, welches lediglich ein animiertes Model darstellt und zum anderen eine Erweiterungen dieses Programmes, welches das geladene Model skaliert und innerhalb der Szene bewegt. Da keine Kollisionsabfrage auf der Ebene der Models stattfindet, waren damit auch die für das Spiel nötigen Funktionen abgedeckt. Grundlage für alle mit der Engine erstellten Projekte ist die Device. Sie stellt die Brücke zum Betriebssystem (Unterstützung von Linux, MacOS X und Microsoft Windows) dar. Die Grafikschnittstelle kann aus OpenGL, DirectX 8, DirectX 9 und 2 Softwarerenderern gewählt werden, unter der Vorraussetzung das eine Unterstützung von Seiten des Betriebsystems gegeben ist. Der Programmcode ist in folgende Namespaces unterteilt:

Namespace	*Erklärung*
Irr	Allumfassender Namepace
Core	Datentypen wie Vektoren, Ebenen, Listen, aber auch Dimensionsangaben wie die Fenstergröße bzw. Auflösung
Scene	SceneNodes und Animationen
Video	Klassen, die direkt mit den Grafiktreibern arbeiten
IO	Schreiben und Lesen von Dateien
GUI	Klassen zur Interfaceerstellung

Tabelle 1: Namespaces des IrrLicht3D Engine

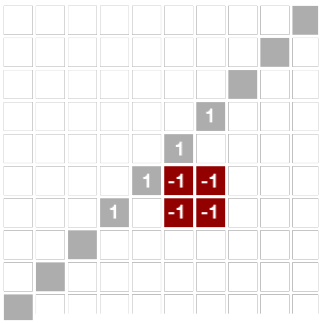
IrrLicht3D ist stark objektorientiert. So werden zum Beispiel das Model und die Animation einer Spielfigur jeweils als eigene Objekte in einem weitern Objekt - der SceneNode - komponiert. Wie auch hier wurde das Composite Design Pattern an vielen weiteren Stellen der Engine verwendet (z.B. GUI).

Entwurf der Datenstrukturen

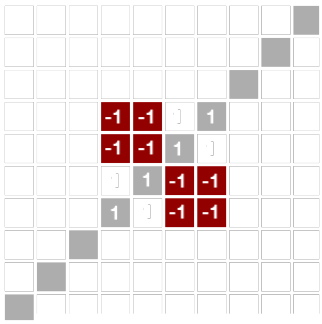
Die Spielfläche wurde als zweidimensionales Array angelegt. Die einzelnen Felder können mit 2 Werten belegt werden.

- 1: Feld begehbar und bebaubar
- -1: Feld belegt

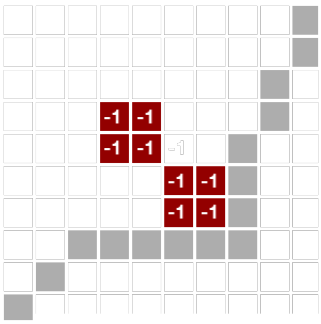
Die Datenstruktur greift aktiv in das Spielgeschehen ein. Sollte der Fall eintreten, dass 2 Türme diagonal zueinander gebaut wurden, entsteht der optische Eindruck, dass der Weg blockiert ist. Der Algorithmus zur Pfadfindung erkennt dies jedoch nicht und die Karten muss einen anderen Wert wiedergeben, als eigentlich hinterlegt.



Weg frei!



Karte muss reagieren



Weg frei!

Ein erbauter Turm belegt jeweils 2x2 Felder auf der 32x32 großen Fläche. Somit können 16 Türme sowohl in der Breite, als auch der Länge der Karte erstellt werden. Dies ermöglicht einen versetzten Bau der Türme und erweckt somit den Eindruck frei bauen zu können.
Eine weitere grundlegende Datenstruktur sind die Wegpunkte, welche von den Gegner abgelaufen werden. Nachdem die Einheit ihren Weg zum Zielpunkt mittels A*-Algorithmus berechnet hat, wird eine verkettete Liste angelegt, die an jedem Listenelement eine horizontale und vertikale Koordinate enthält. Somit wird lediglich bei der Routenberechnung auf die eigentlichen Umgebungsdaten zugegriffen.

Implementierung

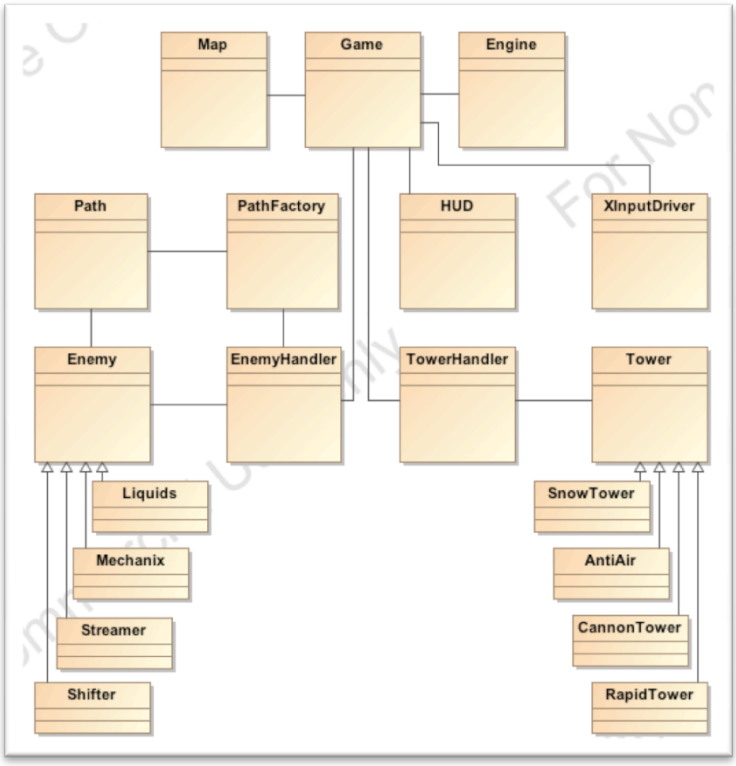


Abbildung 1: Klassendiagramm zu Houston Showdown

Das eigentliche Spiel wurde, soweit es möglich war, von der Engine getrennt. Lediglich die gegebenen Funktionen zum Laden und Anzeigen von Grafiken bzw. Models, sowie zum Bewegen und Animieren dieser wurden verwendet. Ein Wechsel zu einer anderen Engine ist daher ohne großen Aufwand zu bewerkstelligen.

Grundlagen

A-Algorithmus*

Die Implementierung dieses Algorithmus war eine der größten Aufgaben im Laufe dieser Projektarbeit, da er für die Pfadsuche der Gegner, der Grundlage des ganzen Spiels, verantwortlich ist. Er gehört zur Gruppe der „best-first“ - Suchalgorithmen für Graphen.

Datenstruktur	***Erklärung***
OPEN – Liste	Liste der Knoten, die überprüft werden müssen
CLOSED – Liste	Liste der Knoten, die nicht mehr überprüft werden müssen
G – Kosten	Kosten, die aufgebracht werden müssen um diesen Konten zu erreichen
H – Kosten	Geschätzte Kosten, um von diesen Knoten das Ziel zu erreichen
F – Kosten	Gesamtkosten aus G - und H - Kosten

Tabelle 2: Datenstrukturen des A*-Algorithmus

1. *Lege* **OPEN-Liste** an.
2. *Lege* **CLOSED-Liste** an.
3. *Füge* Startknoten in die **OPEN-Liste** ein.
4. *Wiederhole* bis **Zielknoten** gefunden:
    a. *Wähle* Knoten **n** aus **OPEN-Liste** mit geringsten **F-Kosten.**
    b. *Wenn* n der **Zielknoten** ist:
        i. Breche ab, **Zielknoten** gefunden.
    c. *Sonst*:
        i. *Füge* **n** in die **CLOSED-Liste** ein.
        ii. *Wiederhole* für alle Knoten **tmp** die an **n** anliegen:
            1. *Wenn* **tmp** nicht begehbar oder sich bereits auf der **CLOSED-Liste** befindet, ignoriere diesen.
            2. *Wenn anderenfalls* sich **tmp** noch nicht in der **OPEN-Liste** befindet füge diesen hinzu. Speicher in **tmp n** als **Elternknoten** und die berechneten **G-, H- und F-Kosten.**
            3. *Wenn anderenfalls* sich **tmp** bereits auf **OPEN-Liste** befindet, aber die bereits gespeicherten **G-Kosten** höher sind, setze **n** als neuen **Elternknoten** von **tmp** und aktualisiere die **G-, H- und F-Kosten.**
5. *Wenn* der **Zielknoten** gefunden wurde, gehe rekursiv alle gespeicherten **Elternknoten** vom **Zielknoten** zurück und speichere zurückgelegten **Pfad**.
6. Kehre gespeicherten **Pfad** um und erhalte Pfad von ***Start- zu Zielknoten.***

Pseudocode 1: A*-Algorithmus

Vorteile

Der A*-Algorithmus ist vollständig und optimal unter der Vorraussetzung, dass die Heuristik, welche die H-Kosten der Knotenpunkte ermittelt, zulässig ist.
Eine Heuristik ist zulässig wenn ihre Schätzung der Weglänge in dem Intervall [0;k] liegen, wobei k die tatsächlichen Kosten zum Zielpunkt sind. Folglich darf die Heuristik in keinem Fall ein überschätztes Ergebnis liefern.

Nachteile

Zur Ermittlung eines Pfades wird viel Speicher benötigt, da alle überprüften Knoten in den Listen gehalten werden müssen.
Ein weiteres Problem liegt bei der Sortierung der OPEN-Liste. Jede Iteration des Algorithmus verlangt nach dem Knoten mit den jeweils geringsten F-Kosten (siehe a. in Pseudocode 1: A*-Algorithmus), was zur Folge hat, dass an dieser Stelle über die Geschwindigkeit der Pfadermittlung entschieden wird.

BHPQ (binary heap priority queue)

Eine Implementierung, die alle Elemente vergleicht, würde wesentlich zu lange brauchen (worst case O(n)). Der Ansatz einer „binary heap priority queue“ ist sinnvoll. Durch das sortierte Einfügen in die Liste muss, bei der Entnahme, lediglich ein Zugriff auf das erste Element erfolgen, da sich hier bereits der Knotenpunkt mit den geringsten Gesamtkosten befindet. Implementiert ist diese Art von Listen durch ein Array. Die Elemente, also Knotenpunkte, werden ab Index 1 in das Array mittels Heapsort-Algorithmus einsortiert. Um die geforderte Baumstruktur zu erhalten erfolgen die Zugriffe auf Eltern- bzw. Kindknoten wie folgt:

Gesuchter Knoten	Berechnung des Index
Kindknoten links | rechts	2 * Index(Elternknoten) | 2 * Index(Elternknoten) + 1
Elternknoten	Index(Kindknoten) / 2

Tabelle 3: Berechnung der Knotenindices innerhalb einer BHPQ

Spiel

Die Abarbeitung aller Aufgaben erfolgt innerhalb der Klasse „Game“. Mit der Erstellung eines Objektes dieser Klasse wird das Spiel initialisiert. Alle

für die Engine relevanten Objekte (siehe „Kennenlernen der Engine“) werden vor den eigentlichen Spielbestandteilen erzeugt.

Skybox

Eine Skybox ist ein für Videospiele gängiges Stilmittel, um eine Karte bzw. ein Level größer und detailreicher erscheinen zu lassen, als tatsächlich umgesetzt. Um die für den Spieler begehbaren Bereiche wird eine Box (Skybox) oder Halbkugel (Skydome) gespannt und diese mit einer Textur bespannt. Geeignet sind z.B. Bilder von Himmel, Gebäuden oder Gebirgen um den Eindruck von Ferne zu erwecken. In „Houston Showdown“ besitzt die Skybox eine Himmelstextur.

Terrain

Das Terrain wurde mittels 3D Studio Max erstellt und mit IrrEdit, dem von der Engine gelieferten, mit Details, wie Kisten, Absperrungen oder Ölfässern, angereichert. Das Aussehen des Terrains hat keinen Einfluss auf die Spieldynamik oder den Ablauf.

Kamera und Cursor

Die Position, der zubauenden Türme, wird durch einen Cursor bestimmt. Um diesen über die Spielfläche bewegen zu können wird, statt der gewöhnlichen Maussteuerung, der rechte sog „Thumbstick“ des XBOX 360 Controllers verwendet.
Die Kamera der Engine besteht aus 2 Szenenknoten im Szenengraphen. Der Zielpunkt und die eigentliche Kameraposition hängen als Kindknoten an dem Cursorobjektknoten und richten somit ihre Positionsänderungen nach diesem aus. In Folge dessen muss lediglich die Cursorposition verändert werden um die Kamera neu auszurichten.

Wasser

Das Wasser, das den bebaubaren Bereich umgibt, wird durch einen Shader dargestellt. Dieser wurde von einem User der IrrLich3D Community erstellt und implementiert. Die Einbindung erfolgt während der Erstellung einer Instanz der „Game“-Klasse. Zu beachten ist, dass in der Szene mindestens eine Kamera befindlich sein muss, da der Shader auf diese zurückgreift um sog. „render to texture“-Techniken zu verwenden.

Endlosschleife

Das Spiel selbst läuft in einer Endlosschleife. In dieser wird vor jedem Durchlauf überprüft, ob ein Spielzustand erreicht wurde, in dem der Spieler verloren oder gewonnen hat und in Folge dessen der Spielablauf abbricht. Tritt dieser Fall nicht ein, werden alle übrigen Zustände der Gegner, Türme und HUD aktualisiert, sowie dem Scene- und GUI-Manager die Anweisung gegeben mit dem Rendering zu beginnen.

Trigger

Die Erstellung der Gegner wird durch eine ausgelagerte XML-Datei gesteuert. Zur Ladezeit werden Gegnertyp und Pausen zwischen dem Erscheinen zweier Gegner festgelegt. Zu welchem Zeitpunkt die Gegner im Verlauf des Spiels erscheinen sollen wird durch die Summe der Zeitdifferenzen zwischen den einzelnen Iterationen und den Abgleich mit den Pausenzeiten des letzten und nächsten Gegners bestimmt.

Gegner

Die Gegner werden zentral von einem „EnemyHandler“ verwaltet, welcher die Erstellung, Aktualisierung und Entfernung dieser übernimmt. Sollte ein Gegner eigentlich aktualisiert werden, aber meldet, dass seine Lebensenergie aufgebraucht ist, wird dieser entfernt. Da aber in diesem Moment noch Türme auf den bereits entfernten Gegner zielen könnten, werden diese angewiesen erneut mit der Zielsuche zu beginnen.
Die vier Gegnertypen werden von der Klasse „Enemy“ abgeleitet. Diese stellt grundlegende Funktionen wie Bewegung, Beeinflussung durch Treffer, Überprüfung der Lebensenergie oder Information über den eigenen Zustand und Position bereit.
Die Bewegung erfolgt durch Ablaufen der einzelnen Wegpunkte, die der Gegner durch Übergabe seines Pfades erhalten hat. Die eigentliche räumliche Verschiebung wird durch eine, von der Engine gegebene, Funktion realisiert. Nachdem der nächste Wegpunkt erreicht wurde, wird berechnet ob die Richtung der Bewegung beibehalten wird oder gegebenenfalls das Model neu ausgerichtet oder die Geschwindigkeit angepasst werden muss.

Liquids

Der Gegnertyp „Liquid“ soll dem Spieler helfen einen schnellen Einstieg in das Spiel zu finden. Sie bewegen sich nur langsam und sind leicht zu besiegen. Zu Beginn einer Spielrunde dienen diese hauptsächlich dazu möglichst schnell Geld zu verdienen, um ein langes Labyrinth aufzubauen. Später sollen damit kurze Pausen geschaffen werden, die den gesamten Spielablauf etwas entzerren und den Spieler entlasten.
Die Implementierung der Klasse „Liquids“ verändert die Hauptklasse „Enemy“ nur hinsichtlich der Parameter für Lebensenergie und Bewegungsgeschwindigkeit.

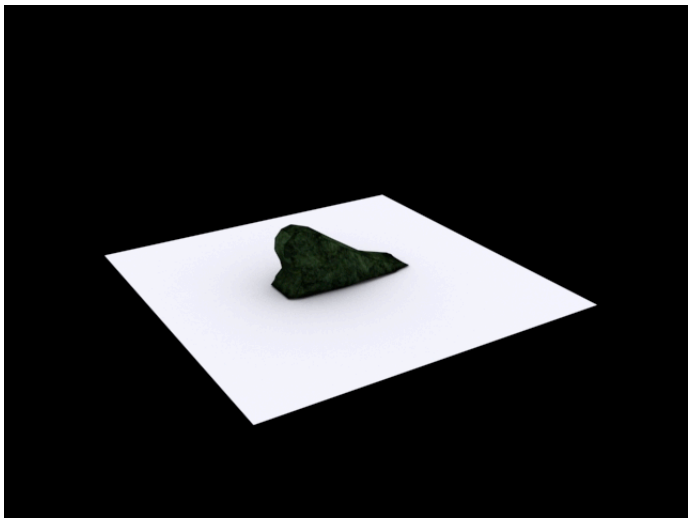


Abbildung 2: Gegnertyp "Liquid"

Streamer

Dieser Typ ist der einzige Gegner mit Flugfähigkeiten. Für den Spieler stellt dieser eine große Herausforderung dar, da er von seinem sonstigen Bauverhalten abweichen muss. Die drei übrigen Typen fordern einen möglichst langen Weg, während hierbei eine möglichst flächige Verteilung gefordert wird, da die Türme überflogen werden.
Die „Enemy“ – Klasse wurde hinsichtlich der Bewegungsfunktion abgeändert. Der Streamer erhält keinen Pfad bei seiner Initialisierung, sondern fliegt in jedem Fall direkt von Start- zu Zielpunkt. Um den Spielablauf etwas zu vereinfachen muss die Zeit die benötigt wird um von Feld zu Feld zu gelangen erhöht werden.

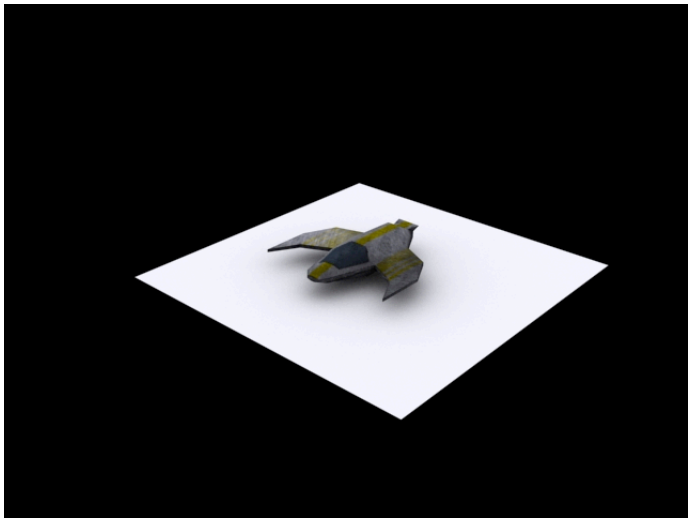


Abbildung 3: Gegnertyp: "Streamer“

Mechanix

Mechanix bilden die Gegnerklasse mit den meisten Lebenspunkten. Der Spieler wird daher dazu gezwungen den Gegnerfluss durch „Freezer“ – Türme zu verlangsamen. Die Implementierung weicht wiederum nur kaum von der „Enemy“-Klasse ab und beschränkt sich auf Parameter.

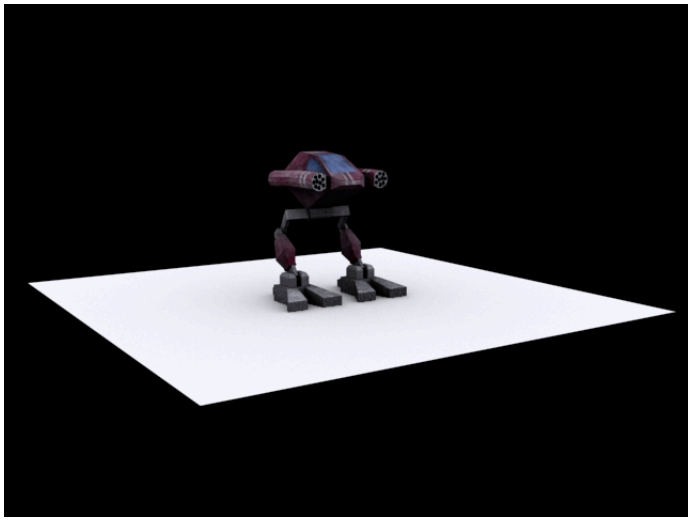


Abbildung 4: Gegnertyp "Mechanix"

Shifter

Hierbei handelt es sich um eine unsichtbare Gegnerklasse, die nur von den „Rapid“-Türmen getroffen werden kann. Im Übrigen unterscheidet sich der „Shifter“ nicht von den Gegnertypen „Liquid“ und „Mechanix“.

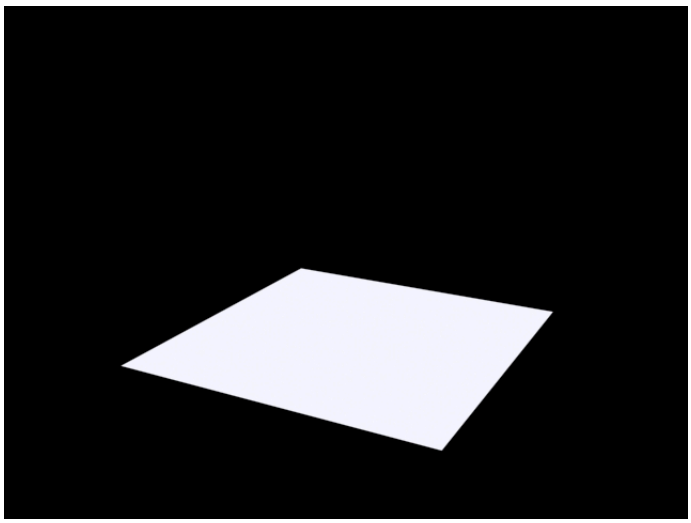
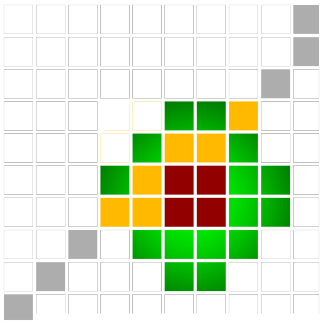


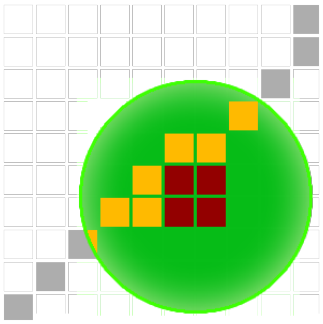
Abbildung 5: Gegnertyp "Shifter"

Türme

Die durch den Spieler erbauten Türme werden, ähnlich wie es der „EnemyHandler“ bei den Gegnern übernimmt, durch die „TowerRegistry“ verwaltet. Die „TowerRegistry“ überwacht ob der Turm, an der gewünschten Stelle, erbaut werden darf. Die Türme dürfen zum Beispiel nicht auf dem Start- bzw. Zielpunkt erbaut werden, da Gegner diese immer erreichen können müssen. Ebenso dürfen keine Gegner eingeschlossen werden, weshalb immer sichergestellt sein muss, dass mindestens ein Weg für jede Einheit begehbar ist. Eine weitere Vorraussetzung für das erfolgreiche Erbauen eines Turms ist, dass der Spieler genügend Geld besitzt. Ein Vergleich erfolgt mit dem Geldwert, den die Klasse „Game“ vorhält. Die Türme entscheiden selbstständig, ob sie einen Gegner anvisieren können, indem sie ihre Fähigkeiten mit den Eigenschaften des Ziels vergleichen. Wurde ein potentielles Ziel gefunden und befindet sich dieses in Reichweite, wird mit dem Beschuss begonnen. Die Reichweite wird im Gegensatz zu der Bewegung der Gegner nicht feldweiße sondern exakt berechnet. Eine feldweiße Überprüfung sieht im Spielverlauf sehr künstlich aus. Alle Türme erben von der Grundklasse „Tower“ und wirken auf diese größtenteils nur durch parametrische Änderungen ein.



Feldweise Zielsuche



Exakte Zielsuche

Rapid

Der erste und auch günstigste Turm im Spiel. Zu Beginn wird dieser erbaut um schnell ein gutes Labyrinth zu erbauen. Dem Spieler wird dies unbewusst auferlegt, indem dieser Turm als einziger die Fähigkeit besitzt auf die unsichtbaren „Shifter“ zu schießen.

Abbildung 6: Turmtyp "Rapid"

Cannon

Dieser Turm hat keine besonderen Fähigkeiten. Er übernimmt die Funktionen der Urklasse „Tower“, besitzt aber eine höhere Feuerkraft bei niedriger Feuerrate.

Abbildung 7: Turmtyp "Cannon"

AntiAir

Der „AntiAir“-Tower hat mit Abstand die größte Schadenswirkung, verteilt diese aber nur auf fliegende Gegner. Durch den Aufpreis, den der Spieler zahlen muss um diesen Turm zu bauen, erspart er sich andere Türme weitflächig verteilen zu müssen und kann dadurch mehr Ordnung und Übersicht auf dem Spielfeld halten.

Abbildung 8: Turmtyp "AntiAir"

Freezer

Im Verlaufe des Spiels wächst die Bedeutung dieses Turms. Durch ihn ist es dem Spieler möglich Gegner drastisch zu verlangsamen. Der Schaden,

welcher durch den „Freezer“-Turm ausgeht, ist kaum von Bedeutung. Da die Gegner aber auch verlangsamt an den umgebenden Türmen vorbei laufen und sich dadurch längere Zeit im Schadensbereich aufhalten steigt der Gesamtschaden beträchtlich.

Abbildung 9: Turmtyp "Freezer"

HUD

Das HUD (Head Up Display) gibt dem Spieler Informationen über aktuell ausgewählte Türme, Geldstand und Lebenspunkte. Die gesamte Anzeige ist als Zustandsautomat realisiert. Alle Werte, die bezifferbar und von Interesse sind, werden als eigener Zustand angesehen. Die Werte werden ziffernweise in ein Array gelegt, jeder Speicherstelle die passende Grafik zugeordnet und danach als gesamtes an der gewünschten Stelle auf dem Bildschirm ausgegeben. Dies entspricht einer eigenen Implementierung eines sog. Bitmap-Fonts.
Für den Spieler ist die bedeutsamste Angabe, welcher Turm aktuell ausgewählt ist und als nächstes erstellt werden würde. Somit bildet die ausgewählte Turmnummer einen eigenen Zustand und beeinflusst die Anzeige bei jedem Zustandswechsel. Am rechten Bildschirmrand werden ständig alle Türme in Grautönen angezeigt, der aktuell ausgewählte Turm in Farbe.

Microsoft XInput Driver

Die Steuerung des Spiels erfolgt mittels XBox 360 Controller. Die Anbindung übernimmt ein Treiber, der seit Version 9 mit dem DirectX SDK ausgeliefert wird.

Die Implementierung erfolgte, ähnlich dem „Head Up Display“, als Zustandsautomat. Mit jedem Tastendruck wird der Zustand geändert und kann durch eine Funktion abgefragt werden. Zu beachten ist, dass nur Zustandsänderungen als Tastendruck interpretiert werden, da die Abfrage in jeder Iteration der Spielschleife erfolgt. Würde dies nicht geschehen, dürfte der Spieler die Taste nur für die Zeit eines Schleifendurchlaufs drücken, um zu verhindern, dass die gewünschte Funktion mehrmals ausgeführt wird.

Modeling

Alle Models des Spiels wurden mit Hilfe von 3D Studio Max 2008 erstellt. Entscheidend war die Abwägung zwischen optischen Eindruck und Performance. Die Polygonzahl der einzelnen Models muss möglichst gering gehalten werden, da davon ausgegangen werden muss, dass alle sich im Spiel befindlichen Grafiken zeitgleich dargestellt werden und dadurch mehr Dreiecke dargestellt werden müssen, als die Grafikkarte unterstützt.

Texturierung

Durch die Texturierung konnte eine höhere Performance erreicht werden, ohne Einbusen des optischen Eindrucks hinnehmen zu müssen. Details wurden durch aufwendigere Texturen und vorberechnete Schatten (shadowmaps) ersetzt. Nach Anpassung der Texturkoordinaten konnten Schatten, welche durch ein Himmelslicht erzeugt wurden, in eine Textur gerendert werden und mit Adobe Photoshop nachbearbeitet bzw. vervollständigt werden.

Abbildung 10: Shadowmap

Abbildung 11: Fertige Textur

Screenshots

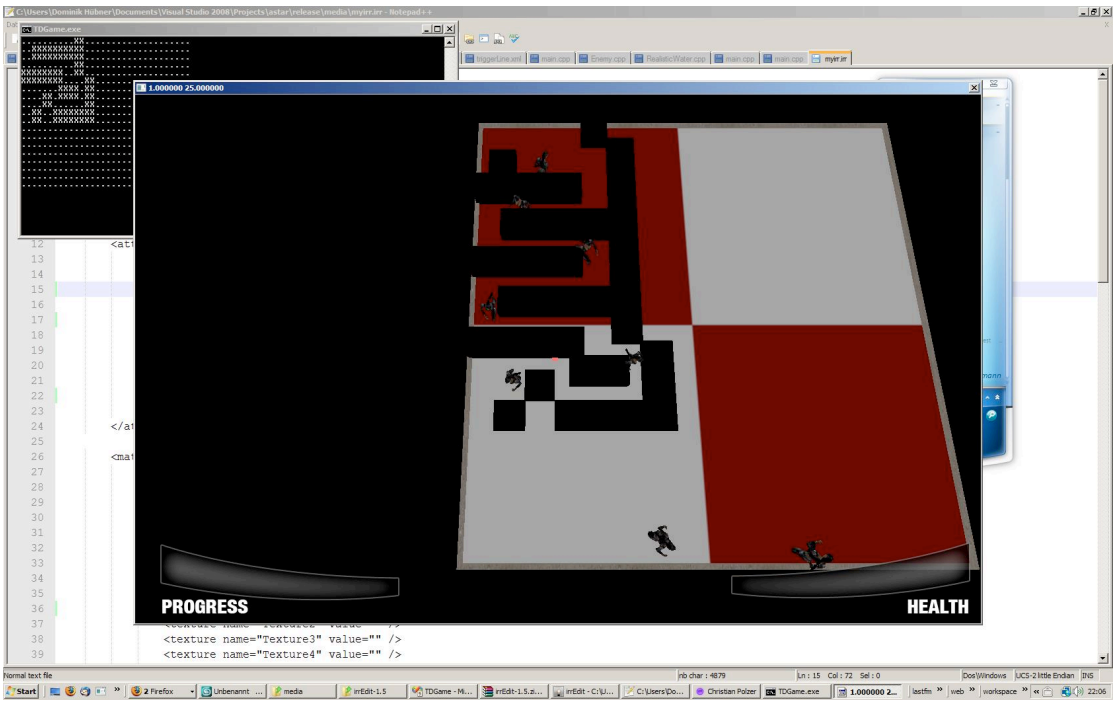


Abbildung 12: Frühe Version des Projekts, im Hintergrund ist die Datenstruktur der Karte zu sehen

Abbildung 13: Stand des Projekts zum Zeitpunkt der Abgabe

Fazit

Wünschenswert wäre es natürlich gewesen, wenn das Spiel fertig gestellt worden wäre. Für eine einzelne Person war dies leider nicht in der vorgegebenen Zeit möglich. Zum Zeitpunkt der Abgabe befindet sich das Spiel in einer Alpha-Version.
Warum das Spiel nicht fertig gestellt werden konnte, hat mehrere Gründe. Da ich vor dieser Projektarbeit keinerlei Erfahrungen im Bereich der Spielentwicklung sammeln konnte, stand ich vor einer Reihe an Herausforderungen, wie z.B. der Arbeit mit einer kompletten Engine bzw. dem generellen Aufbau von Spielen. So mussten viele der Funktionen in kleinen Testprogrammen probeweise implementiert und konnten erst später in das Hauptprojekt aufgenommen werden. Da die Dokumentation der Engine eher einer reinen Funktionsbeschreibung gleicht, artete dies häufig zu einem langwierigen Lernverfahren nach der „trail and error"-Methode aus. Häufig kam es auch vor, dass Features zwar von den Herstellern der Engine versprochen wurden, jedoch die Implementierung nur mangelhaft oder gar nicht abgeschlossen war. Bei der Implementierung der Animationen fiel dies mit am schwersten ins Gewicht. Versprochen wurde der Ladevorgang von animierten Models im Format „*.x" aus dem DirectX SDK. Da ich mich auf diese Aussage verlassen hatte, konnte ich erst in der Endphase des Projekts feststellen, dass dies nur für Models gilt, die durch Skelett-Animation animiert wurden. Somit waren sämtliche Animationsarbeiten grundlegend vergebens, die Zeit für die Implementierung wichtiger Funktionen verloren und die Models wurden ohne Animation verwendet. Eine erneute Animation im „*.md2" Format war sowohl aus Zeitgründen, als auch aus mangelnder Unterstützung der Modelingtools nicht möglich, da das Format aus dem Jahr 1997 stammt und keine Plugins mehr verfügbar sind. Glücklicherweise war dies nur bei den Mechanix, die für die Präsentation an der MediaNight SS2009 durch ein Debug-Model ersetzt wurden, von sehr großer Bedeutung. Im Nachhinein kann ich vom Einsatz von IrrLicht3D abraten und andere Engines, wie Ogre3D, welche ausgereifter sind und größere Communities besitzen, empfehlen. Hier stellt sich der Entwurf, relativ unabhängig von der eigentlichen Engine zu sein, als klarer Vorteil heraus.
Des Weiteren hatte ich nicht erwartet soviel Zeit für die Optimierung von Algorithmen bzw. deren Datenstrukturen aufwenden zu müssen. Allein für die Entwicklung der künstlichen Intelligenz wurde knapp ein Drittel des Semesters aufgewendet.

Trotz einiger Rückschläge und Enttäuschungen, kann ich ein positives Résumé über diese Projektarbeit ziehen. Mir wurde bewusst, wie wichtig es ist größere Funktionen gründlich zu planen und von Anfang an sauber zu implementieren. Zu groß ist die Gefahr einen nicht überprüfbaren Code zu erhalten, der eine Fehlersuche und Fehlerbehebung kaum ermöglicht.
Im Laufe des Projekts konnte ich feststellen, dass meine Kenntnisse der Programmiersprache C++ sich wesentlich erweitert haben und bereits vorhandenes Wissen sich festigte.
Da dies mein erstes Programm in einer solchen Größenordnung war, kann ich es als Grundlage einstufen, die es mir nun ermöglicht, den Aufwand der Softwareentwicklung besser abschätzen zu können und die dafür nötige Zeit besser einzuteilen.

Begriffserklärung

Casual-Game	Computerspiel für Gelegenheitsspieler mit besonders leichtem Einstieg
Creeps	Allgemeine Bezeichnung für Gegner in einem TowerDefence Spiel
Flash-Game	Computerspiel, umgesetzt auf Websites mit Adobe's Flash Technologie
Game-Controller	Eingabeeinheit für Videospiele
IrrEdit	Szeneneditor der IrrLicht3D Engine
IrrLicht3D	Siehe http://irrlicht.sourceforge.net/
Map-Editor	Tool um Spielbereiche für Videospiele zu erstellen
Modifikation	Abänderung von Videospielen, meist durch die Fangemeinde umgesetzt
Shader	Programm, dass auf einer Grafikkarte ausgeführt wird
StarCraft	Strategiespiel von Blizzard Entertainment
Tower	Verteidigungsturm in einem TowerDefence Spiel
Trigger	Funktion, die auf festgelegten Zustand wartet und bei Eintritt ausgeführt wird
WarCraft	Strategiespiel von Blizzard Entertainment