Inhalt

Dies ist die laufende Projektdokumentation zu *The sinister Labyrinth of Dr. Cubus*.

Aufgaben

Es folgt eine Lister der anstehenden Aufgaben nach Betreuer.

Mazzi

n/a

Leuo

n/a

Beide

n/a

Programmablauf

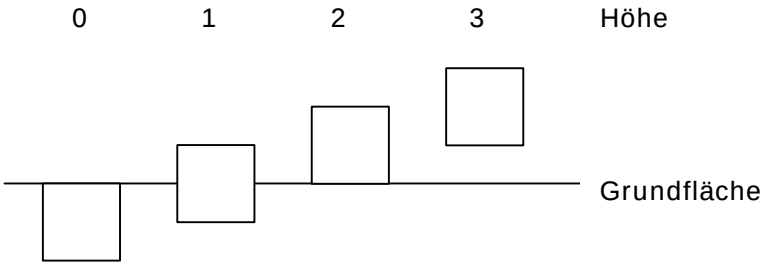
Hier kommen die Diagramme, Grafiken und der Spielablauf hin. Ein Beispiel:

Einige Überlegungen

- Block-Prinzip; Speicherung der Spiele-Welt in einem Array, alle Spielfelder sind quadratisch (von oben)
- Es gibt eine flache „Grund“-Ebene, darauf 2 editierbare Ebenen mit Blöcken, zusätzlich gibt es ein Array, das die Art der Blöcke an einer horizontalen Stelle angibt (z.B. Eis, Wasser, ...)
- Spielsteine sind: Murmel, Gegner; werden in einer Liste verwaltet (mit Position, Richtung, ...), Index 0 = Spiele, Rest=Gegner
- Eventliste: spezifiziert die Sonderdinge, die ein Block auslösen kann (z.B.: Schalter beim drauffahren oder anfahren, Magnet/Saugding, Exit, Start, ...)
- Pro c-File kann mittels #define die Optimierung an- oder abgeschaltet werden.
- Verwendete Software: Visio4, Winword97, VC++5

Repräsentation eines Levels im Speicher

Die Leveldaten bestehen aus zwei 128x128-Feldern, die in je zwei Byte die Höhe und Art des jeweiligen Blocks angeben. Anmerkung zur Höhe:



Genaueres zum Aufbau auf Bit-Ebene siehe main.h.

Aufbau der Leveldateien

In den Leveldateien werden beide Spielhöhen feldweise abwechselnd ("interleaved") gespeichert (2 Byte erste Höhe, 2 Byte zweite Höhe, 2 Byte erste Höhe, usw.). Der Aufbau (Bedeutung der Bits) dieser 2 Byte Felder entspricht der Abbildung im Hauptspeicher. Das Spielfeld wird von links oben beginnend zeilenweise in der Datei abgelegt.

an die strukturbeschreibung der ebenen angeschlossen ist die beschreibung der initialpositionen der entitaeten (id, posx, posy, posz, velx, vely, velz, orientx, orienty, orientz). vor der id der ersten entitaet ist die anzahl der folgenden entitaeten abgelegt.

Grundlegender Spielablauf

Der Main-Loop besteht aus: User-Input abfragen, AI-Aufruf, Spielphysik und Darstellung.

DirectX

Ausgabefunktionen

Zu implementieren ist:
- Grafikmodus initialisieren & einschalten
- Grafikmodus ausschalten (Programmende)
- Page Flip
- Pixel setzen
- Pixel auslesen/abfragen
- Page löschen

Eingabefunktionen

Zu implementieren ist:
- Maus initialisieren
- Tastatur initialisieren
- Maus abschalten (Programmende)
- Tastatur abschalten(Prg.ende)
- getch(); auf eine Taste warten
- readkey(); ist gerade eine Taste gedrückt? Welche?
- direction(); in welche Richtung navigiert der Benutzer gerade seine Kugel?

Grafikfunktionen

Linie zeichnen
Kreis zeichnen

Raum, Zeit, Geschwindigkeit und Bewegung

In Sinister ist eine Raum-Einheit ident mit einem Einheits-Würfel aus der Level-Karte. So ein Würfel ist also genau eine Einheit lang/breit/hoch.

2-dimensionales Beispiel: Befindet sich ein Spielstein an der Position 0.0/0.0, so ist dessen Mittelpunkt dort, und seine „Extremitäten“ ragen in die benachbarten Blöcken zu gleichen Teilen hinein. Befindet sich der Stein an 0.5/0.5, so ist er exakt in der Mitte des Blockes. Und eine Position von 1.0/1.0 ist demgemäß genau an der diagonal gegenüberliegenden Grenze zum nächsten Block.

Eine Zeiteinheit in der Sinisterwelt entspricht einer zehntel Sekunde. Alle Geschwindigkeitsangaben werden relativ dazu angegeben.

Geschwindigkeit wird in Blöcken pro Zeiteinheit angegeben. D.h. bewegt sich unsere Kugel in eine Richtung (z.B. X) mit der Geschwindigkeit von 0.5, so legt sie in einer Sekunde 5 Blöcke zurück. (0.5 * 10). Jede Spielfigur hat eine Maximalgeschwindigkeit. Vorschlag für die Standardfiguren:

Figur	**Geschwindigkeit pro Zeiteinheit**
Spieler	0.5
Kleine Cubes	0.3
Dr. Cubus	0.1

Jede Spielfigur kann sich aktiv nur mit 2 Freiheitsgraden bewegen (X und Y), man fällt nach den Naturgesetzen. D.h. daß jede Figur frei entscheiden kann, zu welchen Teilen sie ihre Geschwindigkeit auf die 2 Achsen X und Y verteilt. Z.B. kann der Spieler mit „0.5/0.0“ entlang der X-Richtung rollen, mit „0.0/0.5“ entlang der Y oder mit „0.2/0.3“ Diagonal mit etwas mehr Y-Richtung als X. Negative Werte entsprechen einer „Rückwärts“-bewegung.

Die Funktionen, die die Wunschrichung/-Geschwindigkeit für Spieler und AI ermitteln, müssen schon diese aufgeteilten Werte in die Spielsteintabelle eintragen und dafür sorgen, daß sich niemand schneller als erlaubt bewegt.

Die Physikfunktion errechnet dann je nach Zeit und Hindernissen die aktuelle reale Position, und sorgt für „Abbremseffekte“, indem sie die neue Wunschgeschwindigkeit mit der bisherigen Richtung/Geschwindigkeit vergleicht.

Projektdokumentation

3Dgraphicsl.c

void SLdrawCircle();
Zeichnet einen Kreis.

void SLdrawCircleFilled()
Zeichnet einen schwarz gefüllten Kreis bzw. einen „normalen“ Kreis im Falle des Drahtgittermodus.

void SLdrawRect();
Zeichnet ein konkaves Rechteck in beliebiger Rotation auf den Bildschirm.

void SLtransformAndDrawScene()
Rendert die Szene als Wireframe-Grafik am Bildschirm.

3Dgraphicsm.c

void SMDrawLine()
zeichnet beliebige linie

inline void SMDrawLineH()
Zeichnet Linien von links nach rechts <=45°

inline void SMDrawLineV()
Zeichnet Linien von oben nach unten <=45°

ail.c

n/a

aim.c

SMUpdateAIMovement()
Bewegung der künstlichen Mitspieler berechnen

directxl.c

SLinitDirectDraw()
Initialisierung der Direct-X-Grafik-Funktionen (=Fullscreenfenster aufmachen)

SLdrawPixel()
Setzt eine Pixel mit den Koordinaten x/y im Backbuffer auf die (Byte-)Farbe crColor.

SLfiniObjects()
Zerstört die DirectX-Grafikelemente für einen geordneten Programmschluß.

SLflipPage()

Kopiert den Backbuffer in den Frontbuffer.

extern HWND SLhwnd

Handler auf das Fullscreen-Fenster des Spiels

SLRGB(r, g, b)

Konvertiert RGB-Angaben in einen BYTE-Wert, der als Farbindex für die Palette dienen kann. r kann Werte von 0..6 annehmen, g und b von 0..5. Achtung: diese Wertebereiche sind einzuhalten!

SLRGB256(r, g, b)

Konvertiert RGB-Angaben in einen BYTE-Wert, der als Farbindex für die Palette dienen kann. r, g und b können Werte von 0..255 annehmen, und werden mit Rücksicht auf die 7-6-6-Bit Farbtiefe der RGB-Palette umgerechnet. Wenn möglich, sollte jedoch SLRGB verwendet werden, da ansonsten eine unnötige Umrechnung gemacht wird.

SLdraw3DPixel(x, y, z, r, g, b)

Zeichnet einen Punkt mittel z-Buffer.

SLdrawPixelDither(x, y, r, g, b)

Zeichnet einen Bildpunkt mittels Schwellwertmatrix-Dithering

*SLdrawText(char *theMessage, int x, int y, BYTE r, BYTE g, BYTE b)*

Schreibt den übergebenen Text in den Backbuffer.

SLclearScreen(BYTE color)

Löscht den Bildschim mit der angegeben Farbe.

directxm.c

SMInitializeInput()

DirectInput initialisieren

SMGetMouseState()

Mausposition und gedrückte Maustasten lesen

SMGetKeyboardState()

Keyboardeingabe (Taste) lesen

SMBusyWaitKey()

auf Tastendruck warten

SMFreeInput()

Eingabegeräte freigeben

SMFKeyCompare()

Umwandlung DirectX ⇔ Interne Repräsentation

SMFKeyCompare()

Einlesen der F-Tasten

*SMKBSTATE *kbstate*

enthaelt Tastaturstatus

WORD SMFKeys

enthaelt Status der F-Tasten (Schalterfunktion)

main.c

In main.c befindet sich das Hauptprogramm, das alle notwenigen Komponenten initialisiert, und dann in den Main-Loop geht.

modell.c

unsigned short SLlevelData[2][SMLEVELDIM][SMLEVELDIM]

Enthält die Leveldaten. Bitbelegung siehe modell.h.

SLENTITYSTRUCT SLEntities[SLMAXENTITIES]

Dies ist die Spielsteintabelle. Index 0 enthält den Spieler (Kugel), der Rest ist für Gegner reserviert.

SLCAMERASTRUCT SLCamera

Cameramodell mit Position und Sichtwinkel.

SLinitSinus()

Initialisiert die (Co-)Sinus Tabellen.

modelm.c

int SMnumEntities

anzahl der tatsaechlich verwendeten entities

*SMshowGameField() **

zeichnet Spielfeld

*SMpaintBlock() **

zeichnet einen Block

*SMrefreshVisibilityMask() **

maskieren der nicht sichtbaren Seiten diverser Blöcke

void SMkeepCameraMiddle()

Kameraposition an Spielkugel orientieren

void SMkeepDirectionAndOrientation()

Orientierung und Richtung der Spielkugel an Kamerarotation anpassen

physicsl.c

SLgeneratePhysics()

Adaptiert alle Spielerdaten, indem es ihre physikalisch korrekten neuen Positionen errechnet.

physicsm.c

SMUpdateUserMovement()
Bewegungsdaten des Spielers einlesen und abspeichern

*BOOL SMcheckExit() **
Prüfen ob das Exit-Feld erreicht wurde

SMgeneratePhysics()
Simulation physikalischer Grundgesetze (= Kollision + Umgebungsbedingungen)

SMdetectCollision()
Kollisionserkennung und -behandlung mit Feinden und Umgebung

SMsimulateEnvironment()
Simulation der Umgebungsbedingungen (Erdanziehung)

SMfps()
mißt Frames per Second

ressourcel.c

n/a

ressourcem.c

SMReadLevel()
liest Spiellevel ein