TU Hamburg-Harburg – Prozedurale Programmierung
Gruppe #37: Florian J******, Marc L******, Marcel H******, WS 2010/2011

Das C-Projekt:

c37 (bessere Idee?)

Entwurf!

Spezifikation

Das Programm *c37* ist ein kleines Abenteuerspiel. Ohne in einem weiteren Kontext zu stehen, übernimmt man die Kontrolle über eine Spielfigur, die sich durch eine zweidimensionale Ebene bewegen kann. Die Spielwelt wird dabei aus einer Perspektive beobachtet, die senkrecht zur Spielwelt steht. Zur Darstellung der Spielfigur und der Umwelt werden dabei Zeichen aus dem ASCII-Zeichensatz und für die angenehmere Betrachtung und Diversifikation der Spielelemente zusätzlich Farben verwendet. Damit orientiert sich *c37* an der Familie der *Rogue-like*-Spiele. Die Steuerung der Spielfigur erfolgt durch die Tastatur, detaillierte Informationen bezüglich der Interaktion des Spielers mit der Umwelt finden über eine Textausgabe am Spielfeldrand statt. Interaktionen sind beispielsweise das Öffnen einer Tür oder Aufheben eines Gegenstands. *c37* ist grundsätzlich nicht auf bestimmte Plattformen beschränkt,

verwendet jedoch für Tastaturevents und den graphischen Ausgabemodus die *Simple DirectMedia Layer*-Bibliothek, ist also auf deren Verfügbarkeit angewiesen. Das Format, in dem die Karten abgespeichert sind, ist die *JavaScript Object Notation* (JSON). Zum Parsen dieser wird eine frei verfügbare Fremdkomponente von *json.org* verwendet. Der Grund für die Verwendung von JSON liegt darin, dass dieses Format im Gegensatz zu etwa XML wenig Overhead besitzt, einfach per Hand editierbar ist und eine Implementation eines eigenen Text- oder Binärformats einen höheren Zeitaufwand und größere Risiken unentdeckter Sicherheitslücken darstellt.

Design

Programmarchitektur

Programmablauf

- Das Programm erhält als ersten Parameter beim Aufruf den Namen einer Karte, welche sich im Verzeichnis *maps* befindet.
- Fehlermeldungen werden nach *stderr* geschrieben.
- Ist diese nicht aufzufinden, startet das Programm nicht. Andernfalls wird die Kartendatei eingelesen: Die Karte wird im Speicher angelegt, die einzelnen Spielfelder erhalten jeweils die Kartenkomponente (Wände, Böden, Türen, ...), die in der Datei angegeben wurde. Weiterhin wird eine Spielfigur gesetzt und Gegenstände platziert.
- Sollte die Kartendatei nicht richtig ausgelesen werden können, bricht das Programm ab.
- Wenn die Karte komplett konstruiert worden ist, wird sichergesetellt, dass I/O-Operationen wie das Warten auf Tasturevents und die graphische Ausgabe funktionieren. Andernfalls bricht das Programm ab.
- Solange das Programm nicht per ESC-Taste abgebrochen wird, findet ein Loop über relevante Events statt, die Logik des Spiels wird auf vorhandene Komponenten angewendet und die Ausgabe aktualisiert.
- Mit dem Drücken der ESC-Taste wird der Loop abgebrochen und vom Programm genutzte Ressourcen werden freigegeben.

zentrale Komponenten

- Maploader - Das Konstruieren von nutzbaren Karten anhand der Vorgaben in der Kartendatei.
- Logikschleife - Interaktion des Spielers mit der Spielfigur und der Umwelt.
- Ausgabe - Darstellung eines vorgefertigten Feldes mit Zeichen- und Farbinformationen.

Ein- und Ausgabe

- Dateisystem (Eingabe) - Das Erstellen einer Karte anhand einer abgelegten Datei, Benutzung eines JSON-Parsers.
- Tastatur (Eingabe) - Steuerung des Programmablaufs sowie der Spielfigur, Benutzung von SDL.
- Bildschirm (Ausgabe) - Ausgabe des Spielgeschehens, Benutzung von SDL.
- Dateisystem/ostream (Ausgabe) - Fehlermeldungen, die in den *stderr*-Stream geschrieben werden.

Programm-Header

globals.h

Die Headerdatei *globals.h* übernimmt mehrere Aufgaben. Eine davon ist es, alle nötigen System-, Library- und Programm-Header zu laden. Andererseits werden hier interne Konstanten gesetzt, die technische Angelegenheiten betreffen als auch solche, die ein aufwendigeres, externes Konfigurationssystem für die Spiellogik ersetzen. Außerdem werden alle globalen Variablen hier angelegt, sodass diese unabhängig von einer bestimmten Objektdatei nutzbar sind.
Dies ist eine Übersicht von verwendeten internen Konfigurationsparametern:

Konstante	Wert	Zweck
OUTPUT_IN_GLYPHS_X	int	vertikale Anzahl an auszugebenden Zeichen
OUTPUT_IN_GLYPHS_Y	int	horizontale Anzahl an auszugebenden Zeichen
MESSAGE_STREAM_LIMIT	int	max. Anzahl an gespeicherten Nachrichten

Eine Übersicht von semantischen Konfigurationsparametern:

Konstante	Wert	Zweck
VISUAL_SQUARE	int	Kantenlänge des Quadrats (quasi Sichtweite), das bei der Erkundung von unbekannten Bereichen verwendet wird.
NORTH	$2^3(int)$	4. Bit steht für die Himmelsrichtung Norden
EAST	$2^2(int)$	3. Bit steht für die Himmelsrichtung Osten
SOUTH	$2^1(int)$	2. Bit steht für die Himmelsrichtung Süden
WEST	$2^0(int)$	1. Bit steht für die Himmelsrichtung Westen

memory.h

In dieser Headerdatei werden essentielle Funktionen aus der *stdlib* zur Reservierung von Speicherbereichen gewrappt. Im Falle fehlgeschlagener Speicherallokation wird das Programm beendet!

Listing 1: ex_calloc

```
1 void* ex_calloc(size_t num, size_t size);
```

Verwendung: Reserviert *num* mal Speicher der Größe *size* und setzt alle Bytes auf 0. Schlägt dies fehl, beendet sich das Programm.

Parameter:

- num - Anzahl der Speichereinheiten
- size - Größe der Speichereinheit

Rückgabe: Zeiger eines unspezifizierten Typs auf den allozierten Speicherbereich.

Listing 2: ex_malloc

```
1 void* ex_malloc(size_t size);
```

Verwendung: Reserviert Speicher von der Größe *size*. Schlägt dies fehl, beendet sich das Programm.

Parameter:

- size - Größe des zu reservierenden Speicher in Bytes

Rückgabe: Zeiger eines unspezifizierten Typs auf den allozierten Speicherbereich.

Listing 3: ex_realloc

```
1 void* ex_realloc(void* ptr, size_t size);
```

Verwendung: Vergrößert oder verkleinert den Speicher von *ptr* auf *size* Bytes. Ist *ptr* NULL, verhält sich die Funktion wie *ex_malloc*. Schlägt dies fehl, beendet sich das Programm.

Parameter:

- ptr - Adresse des Speichers, dessen Größe verändert wird.
- size - neue Speichergröße in Bytes

Rückgabe: Zeiger eines unspezifizierten Typs auf den neu- oder reallozierten Speicherbereich.

game_messages.h

Da das Spiel etwas mehr als eine graphische Ausgabe benötigt, um sinnvoll mit dem Spieler zu interagieren, können verschiedene Vorgänge und Events dem Spieler mehr berichten.

Listing 4: message

```
1 void message(const char* msg);
```

Verwendung: Fügt den String *msg* an das vordere Ende des Nachrichtenstreams ein. Sind dadurch mehr Nachrichten enthalten als in *MESSAGE_STREAM_LIMIT* zugelassen, werden Nachrichten am hinteren Ende gelöscht.

Parameter:

- msg - Nachricht, die dem Spieler mitgeteilt werden soll.

Rückgabe: -

Listing 5: flush_messages

```
1 void flush_messages();
```

Verwendung: Leer den Nachrichtenstream.

Parameter: -

Rückgabe: -

output_buffer.h

Diese Header-Datei definiert die Kachel für den Ausgabepuffer. Diese ist quasi eine sehr beschnittene Spielkartenkachel, die nur noch Informationen enthält, die für die Ausgabe wichtig sind und unabhängig von Spiellogik und Ausgabegerät formuliert ist.

Listing 6: BufferTile

```
1 typedef struct BufferTile {
2         char glyph;
3         unsigned long color;
4 } BufferTile;
```

Elemente der Struktur:

- glyph - Zeichen, das dargestellt werden soll.
- color - 32-bit Farbinformation im Format R8G8B8A8

item.h

Hier wird eine Struktur definiert, die Gegenstände in der Welt darstellen können, die entweder herumliegen oder aufgenommen sind. Dieser Header lädt zusätzlich Item-Property-Structs aus dem Verzeichnis *items*.

Listing 7: Item

```
typedef struct Item {
        char* id;
        char* name;
        unsigned long color;
        int weight;
        int value;
        char* type;
        void* properties;
} Item;
```

Elemente der Struktur:

- id - eindeutiger Name jedes Items in der gesamten Spielwelt
- name - Name des Items in der Spielwelt
- color - 32-bit Farbinformation im Format R8G8B8A8.
- weight - Gewicht des Gegenstands
- value - Wert des Gegenstands
- type - Präzisierung des Typs, den dieses Item darstellt (z.B. Trank oder Waffe).
- properties - Zeiger auf ein *Property*-struct, das abhänging von *type* erstellt wird.

Listing 8: spawn_uses_item

```
void spawn_uses_item(Spawn* spawn, Item* item);
```

Verwendung: Führt abhänging vom Handelnden *spawn* und dem benutzten Gegenstand *item* einen Effekt aus.

Parameter:

- spawn - Benutzer des Items
- size - benutztes Item

Rückgabe: -

spawn.h

Ein *Spawn* ist die Struktur, die grundsätzlich alle Informationen enthält, die für spielbare und nicht-spielbare Figuren benötigt wird.

Listing 9: Spawn

```
typedef struct Spawn {
        char* id;
        char* name;
        unsigned int x,y;
        char direction;
        char glyph;
        char npc;
        char humanoid;
        unsigned int max_hp, hp;
        char* type;
        void* properties;
        Item* inventory;
        unsigned int inventory_size;
} Spawn;
```

Elemente der Struktur:

- id - eindeutiger Name jedes Spawns in der gesamten Spielwelt
- name - Name des Spawns in der Spielwelt
- x, y - Koordinaten des momentanen Aufenthalts
- direction - Blickrichtung
- glyph - für die Anzeige zu verwendendes ASCII-Zeichen
- npc - Flag, ob dies die Spielfigur des Spielers ist.
- humanoid - Flag, ob die Spielfigur humanoid ist.
- max_hp - maximale Anzahl an Lebenspunkten
- hp - aktuelle Anzahl an Lebenspunkten
- type - Präzisierung der Gegnerart
- properties - eine von *type* abhängig geladene Struktur für zusätzliche Eigenschaften
- inventory - Ein Array, in dem Besitztümer liegen.
- inventory_size - die Anzahl der Besitztümer

Listing 10: spawn_spawn_collision

```
void spawn_spawn_collision(Spawn* spawn_a, Spawn* spawn_b);
```

Verwendung: Würfelt das Geschehen aus, das eintritt, wenn Spawns *spawn_a* und *spawn_b* aufeinandertreffen.

Parameter:

- spawn_a - erster Spawn
- spawn_b - zweiter Spawn

Rückgabe: -

Listing 11: explore_area

```
1 void explore_area(Spawn* spawn);
```

Verwendung: Aktualisiert den Sichtbereich um *spawn* herum, wenn dieser Neuland betritt.

Parameter:

- spawn - Spawn, dessen Sichtbereich erweitert wird.

Rückgabe: -

Listing 12: spawn_action

```
1 void spawn_action(Spawn* spawn);
```

Verwendung: Würfelt die nächste Aktion aus, die *spawn* tun soll, z.B. Gehen oder Warten.

Parameter:

- spawn - Spawn, für den eine Aktion gewürfelt wird.

Rückgabe: -

tile.h

Die *Tile*-Struktur beschreibt ein Spielfeld, das aufgrund seiner Beschaffenheit für die Spiellogik wichtig ist. Die Art und der Zustand einer Kachel bestimmt, wie diese in der Ausgabe erscheinen wird.

Listing 13: Tile

```
typedef struct Tile {
        char* id;
        unsigned int x,y;
        char spotted;
        unsigned int brightness;
        char glyph;
        unsigned long color;
        char* type;
        void* properties;
        Item* items;
        unsigned int number_of_items;
} Tile;
```

Elemente der Struktur:

- id - eindeutiger Name jedes Spawns in der gesamten Spielwelt
- x, y - Koordinaten des momentanen Aufenthalts
- spotted - Flag, ob die Kachel bereits erkundet wurde.
- brightness - Helligkeit auf dieser Kachel
- glyph - für die Anzeige zu verwendendes ASCII-Zeichen
- color - für die Anzeige zu verwendende Farbe im Format R8G8B8A8
- type - Präzisierung des Tile-Typs
- properties - eine von *type* abhängig geladene Struktur für zusätzliche Eigenschaften
- items - Ein Array, in dem hier abgelegte Items liegen.
- number_of_items - die Anzahl der abgelegten Items

Listing 14: render_tile

```
void render_tile(BufferTile* bt, Tile* tile);
```

Verwendung: Es wird ermittelt, welche Ausgabeinformationen für *tile* zu verwenden sind (Zeichen, Farbe). Diese werden in die Pufferkachel *bt* geschrieben.

Parameter:

- bt - Pufferkachel, in die die Anzeigeinformation geschrieben wird
- tile - Kachel, für die die Ausgabeinformationen ermittelt werden

Rückgabe: -

Listing 15: spawn_tile_collision

```
void spawn_tile_collision(Spawn* spawn, Tile* tile);
```

Verwendung: Es wird die Aktion ausgeführt, die eintritt, wenn *spawn* die Kachel it betreten möchte. Beispielsweise, ob er diese überhaupt betreten darf.

Parameter:

- spawn - handelnder Spawn
- tile - Kachel, die er betreten möchte.

Rückgabe: -

Listing 16: toggle_tile

```
1 void toggle_tile(Tile* tile);
```

Verwendung: Ändert einen binären Zustand der Kachel *tile*, z.B. Tür auf/zu.

Parameter:

- tile - Kachel, auf der eine Aktion ausgeführt werden soll.

Rückgabe: -

map.h

Die Map-Struktur stellt die geladene Karte dar.

Listing 17: Map

```
typedef struct Map {
        unsigned int x, y;
        char* name;
        Tile* tiles;
        Spawn* spawns;
        unsigned int number_of_spawns;
} Map;
```

Elemente der Struktur:

- x, y - Maße der Karte
- name - Name der Karte
- tiles - Eindimensionales Array, das die Spielkacheln enthält.
- spawns - Array, das die Spawns enthält.
- number_of_spawns - Anzahl der Spawns auf der Karte

Listing 18: get_player_spawn

```
Spawn* get_player_spawn();
```

Verwendung: Ermittelt die Spielfigur.

Parameter: -

Rückgabe: Rückgabe eines Zeigers auf die Spielfigur.

Listing 19: get_spawn_at

```
Spawn* get_spawn_at(unsigned int x, unsigned int y);
```

Verwendung: Liefert, wenn vorhanden, die Spielfigur auf einer Kachel an der gegebenen Position zurück.

Parameter:

- x - x-Koordinate
- y - y-Koordinate

Rückgabe: Rückgabe eines Zeigers auf die Figur auf der Kachel bei (x, y). Oder *NULL*, wenn keine vorhanden oder außerhalb der Kartengrenze.

map_loader.h

Der Map-Loader ist die Komponente, die das Parsen der Kartendatei übernimmt und diese im Speicher zusammensetzt.

Listing 20: load_map

```
1 int load_map(char* name);
```

Verwendung: Öffnet die Datei passend zu *name* im *maps*-Verzeichnis, parst diese und baut die Karte für das Spiel auf.

Parameter:

- name - Der Kartenname, wie sie ohne Dateiendung auf dem Dateisystem abgelegt ist.

Rückgabe: Rückgabe eines Status, 1 bedeutet Erfolg, 0 einen Fehler.

Listing 21: flush_map

```
1 void flush_map();
```

Verwendung: Löscht die Karte und alle verwendeten Komponenten aus dem Speicher.

Parameter: -

Rückgabe: -

main.h

Diese Header-Datei enthält Funktionen, die dem Programmfluss des Programms und der Übersetzung von Karte zu Ausgabe dienlich sind.

Listing 22: pause_game

```
1 void pause_game();
```

Verwendung: Hält das Spielgesehen an.

Parameter: -

Rückgabe: -

Listing 23: continue_game

```
1 void continue_game();
```

Verwendung: Setzt das Spielgesehen fort.

Parameter: -

Rückgabe: -

Listing 24: create_output_buffer

```
1 void create_output_buffer();
```

Verwendung: Kopiert den Ausschnitt des Spielgeschehens in den Ausgabepuffer.

Parameter: -

Rückgabe: -

Listing 25: print_message_box

```
1 void print_message_box();
```

Verwendung: Kopiert einen Textbereich für die Nachrichten in den Ausgabepuffer (und überschreibt damit ggf. Teile der Karte).

Parameter: -

Rückgabe: -

Listing 26: flush_output_buffer

```
1 void flush_output_buffer();
```

Verwendung: Setzt alle Kacheln des Ausgabepuffers auf leere Elemente (Leerzeichen-Glyphen).

Parameter: -

Rückgabe: -

sdl_output.h

Dieser Header stellt Funktionen bereit, die die Ausgaberoutinen von *SDL* wrappen. Er ist prinzipiell so aufgebaut, dass die Ausgabe unabhängig vom graphischen Backend geregelt werden kann.

Listing 27: output_init

```
1 void output_init();
```

Verwendung: Weißt das Ausgabegerät an, nötige Ressourcen für die Ausgabe zu reservieren.

Parameter: -

Rückgabe: -

Listing 28: output_draw

```
1 void output_draw();
```

Verwendung: Weißt das Ausgabegerät an, die im Ausgabepuffer gespeicherten Kacheln zu zeichnen.

Parameter: -

Rückgabe: -

Listing 29: output_clear

```
1 void output_clear();
```

Verwendung: Mit diesem Befehl soll die Ausgabe blank gezeichnet werden.

Parameter: -

Rückgabe: -

Listing 30: output_close

```
1 void output_close();
```

Verwendung: Weißt das Ausgabegerät an, gebrauchte Ressourcen für die Ausgabe freizugeben, da die Ausgabe beendet ist.

Parameter: -

Rückgabe: -

Bewertungskriterien

1. Karten liegen auf dem Dateisystem und sind veränderbar.
2. Karten für dieses Spiel sind über Parameter wählbar.
3. Die Ausgabe unterstüzt bei der Verwendung von SDL farbige Symbole.
4. Es existiert eine Textausgabe, die Informationen über die Interaktionen liefert.
5. Durch Bewegung der Spielfigur werden unbekannte Teile der Karte aufgedeckt.
6. Teile der Karte sind unbeleuchtet und nicht einsehbar.
7. Auf der Karte sind Items platziert, die aufgenommen werden können.
8. Der Gesundheitszustand der Figur ist variabel bis zum Tod.
9. Die Spielfigur kann Türen öffnen.
10. Schalter können Zustände von Wänden oder Türen verändern.

Aufgaben und Zeitplan

Wer?	Arbeitspaket	Tätigkeiten	Datum
Florian	Logik + KI	Interaktionen, Gegner	13.01. - 27.01.11
Marc	Technik	Darstellung, Steuerung	13.01. - 27.01.11
Marcel	Maploader	Karten parsen, konstruieren	13.01. - 27.01.11