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

Das C-Projekt:

c37

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, ebenso die darauf aufbauende Erweiterung *SDL_ttf*. 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

Dieser Abschnitt soll einen Überblick über die verschiedenen Programmteile geben und ihre Funktionsweise dokumentieren. Wir behalten uns Änderungen an den Signaturen und der Anzahl an Definitionen bis zur finalen Version vor.

globals.h

Die Headerdatei *globals.h* übernimmt mehrere Aufgaben. Eine davon ist es, alle nötigen System-, Library- und einige 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.
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.

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 4: 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

Listing 5: InterfaceData

```
1 typedef struct InterfaceData {
2         unsigned int player_hp;
3         char* item_name;
4         int item_index;
5         char* message;
6         char last_message;
7  } InterfaceData;
```

Elemente der Struktur:

- player_hp - aktuelle Lebenspunkte des Spielers
- item_name - Name des ausgewählten Items
- item_index - Indexzahl des ausgewählten Items
- message - anzuzeigende Nachricht
- last_message - Flag, ob es sich um die neueste Nachricht handelt.

item.h

Hier wird eine Struktur definiert, die Gegenstände in der Welt darstellt, die entweder herumliegen oder aufgenommen sind. Dieser Header lädt zusätzlich Item-Property-Structs aus dem Verzeichnis *items*. Zusätzlich werden hier die Konstanten für bekannte Item-Typen und Standardinitialisierungswerte gesetzt.

Listing 6: Item

```
typedef struct Item {
        char* id;
        char* name;
        unsigned long color;
        int weight;
        int value;
        unsigned int 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 7: spawn_uses_item

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

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

Parameter:

- spawn - Benutzer des Items
- size - benutztes Item
- map - zur Verwendung in der Ausführungslogik bereitgestellte Spielkarte

Rückgabe: -

Listing 8: free_item

```
void free_item(Item* item);
```

Verwendung: Gibt allen Speicher frei, der für Item-Elemente und *item* reserviert wurde.

Parameter:

- item - Item, dessen Speicher freigegeben werden soll.

Rückgabe: -

spawn.h

Ein *Spawn* ist eine Struktur, die grundsätzlich alle Informationen enthält, die für spielbare und nicht-spielbare Figuren benötigt werden. Zusätzlich werden hier die Konstanten für bekannte Spawn-Typen und Standardinitialisierungswerte gesetzt.

Listing 9: Spawn

```
1  typedef struct Spawn {
2          char* id;
3          char* name;
4          unsigned int x,y;
5          char direction;
6          char glyph;
7          char npc;
8          char humanoid;
9          unsigned int max_hp, hp;
10         unsigned int type;
11         void* properties;
12         Item** inventory;
13         unsigned int inventory_size;
14         unsigned int selected_item;
15 } 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, dessen Adressen auf die Besitztümer zeigen.
- inventory_size - die Anzahl der Besitztümer
- selected_item - Index des momentan ausgewählten Items

Listing 10: free_spawn

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

Verwendung: Gibt allen Speicher frei, der für Spawn-Elemente und *spawn* reserviert wurde.

Parameter:

- spawn - Spawn, dessen reservierte Speicher freigegeben werden soll.

Rückgabe: -

tile.h

Die *Tile*-Struktur beschreibt eine Kachel des Spielfelds, das aufgrund seiner Beschaffenheit für die Spiellogik wichtig ist. Die Art und der Zustand einer Kachel bestimmt, wie diese in der Ausgabe erscheinen wird. Zusätzlich werden hier die Konstanten für bekannte Tile-Typen und Standardinitialisierungswerte gesetzt.

Listing 11: Tile

```
typedef struct Tile {
        char* id;
        unsigned int x,y;
        char spotted;
        unsigned char brightness;
        char glyph;
        unsigned long color;
        unsigned int 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, dessen Adressen auf platzierte Items zeigen.
- number_of_items - die Anzahl der abgelegten Items

Listing 12: 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 13: free_tile

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

Verwendung: Gibt allen Speicher frei, der für Tile-Elemente und *tile* reserviert wurde.

Parameter:

- tile - Tile, dessen Speicher freigegeben werden soll.

Rückgabe: -

map.h

Die Map-Struktur stellt die geladene Karte dar.

Listing 14: Map

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

Elemente der Struktur:

- x, y - Maße der Karte
- name - Name der Karte
- tiles - Eindimensionales Array, das die Spielkacheln enthält.
- spawns - Array, dessen Adressen auf die Spawns zeigen.
- number_of_spawns - Anzahl der Spawns auf der Karte
- msg_hist - Eventlog auf dieser Karte
- latest_msg - letzte Nachricht auf dem Eventlog
- current_msg - Index der aktuell angezeigten Nachricht
- finished - Signalisiert, ob der Ausgang gefunden wurde.

Listing 15: get_player_spawn

```
Spawn* get_player_spawn(Map* map);
```

Verwendung: Ermittelt die Spielfigur.

Parameter:

- map - Karte, auf welcher gesucht wird.

Rückgabe: Rückgabe eines Zeigers auf die Spielfigur.

Listing 16: get_spawn_at

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

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

Parameter:

- x - x-Koordinate

- y - y-Koordinate
- map - Karte, auf welcher gesucht wird.

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

Listing 17: explore_area

```
1 void explore_area(Spawn* spawn, Map* map);
```

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

Parameter:

- spawn - Spawn, dessen Sichtbereich erweitert wird.
- map - Karte, auf der der Sichtbereich aufgedeckt wird.

Rückgabe: -

Listing 18: push_msg

```
1 void push_msg(char* msg, Map* map);
```

Verwendung: Fügt den String *msg* als neuesten Eintrag der Message-History hinzu.

Parameter:

- msg - neue Nachricht
- map - die Map, die die History enthält

Rückgabe: -

map_loader.h

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

Listing 19: load_map

```
1 Map* 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 Zeigers auf die erstellte Karte oder NULL wenn unmöglich.

Listing 20: flush_map

```
1 void flush_map(Map* map);
```

Verwendung: Gibt den reservierten Speicher für Karten-Elemente und *map* frei.

Parameter:

- map - Freizugebende Karte.

Rückgabe: -

main.h

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

Listing 21: create_output_buffer

```
1 void create_output_buffer(Map* map, BufferTile* buf, int tiles);
```

Verwendung: Kopiert den Ausschnitt des Spielgeschehens in den Ausgabepuffer.

Parameter:

- map - Spielkarte, die auszugeben ist.
- buf - zu benutzender Ausgabepuffer
- tiles - Anzahl der Tiles des Puffers.

Rückgabe: -

Listing 22: flush_output_buffer

```
1 void clear_output_buffer(BufferTile* buf, int tiles);
```

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

Parameter:

- buf - Ausgabepuffer, der überschrieben werden soll.
- tiles - Anzahl der Tiles des Puffers.

Rückgabe: -

action.h

Dieser Header stellt Funktionen bereit, die bestimmen, wie sich die Spielwelt durch bestimmte Ereignisse verändert.

Listing 23: process_event

```
1 void process_event(SDL_Event *event, Map *map);
```

Verwendung: Verändert die Map entsprechend einem eingehenden Event.

Parameter:

- event - das zu verarbeitende Event
- map - die zu aktualisierende Map

Rückgabe: -

Listing 24: spawn_spawn_collision

```
1 void spawn_spawn_collision(Spawn* spawn_a, Spawn* spawn_b, Map* map);
```

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

Parameter:

- spawn_a - erster Spawn
- spawn_b - zweiter Spawn
- map - zur Verwendung in der Ausführungslogik bereitgestellte Spielkarte

Rückgabe: -

Listing 25: spawn_action

```
1 void spawn_action(Spawn* spawn, Map* map);
```

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

Parameter:

- spawn - Spawn, für den eine Aktion gewürfelt wird.
- map - zur Verwendung in der Ausführungslogik bereitgestellte Spielkarte

Rückgabe: -

Listing 26: spawn_tile_collision

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

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.
- map - zur Verwendung in der Ausführungslogik bereitgestellte Spielkarte

Rückgabe: -

Listing 27: toggle_tile

```
1 void toggle_tile(Tile* tile, Map* map);
```

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

Parameter:

- tile - Kachel, auf der eine Aktion ausgeführt werden soll.
- map - zur Verwendung in der Ausführungslogik bereitgestellte Spielkarte

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 28: output_init

```
1 void output_init(int width, int height, char *mapname);
```

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

Parameter:

- width - Breite der Map in Zeichen
- height - Höhe der Map in Zeichen
- mapname - Name der Map, der in der Titelleiste angezeigt werden soll

Rückgabe: -

Listing 29: output_draw

```
1 void output_draw(BufferTile* buf, int tiles, InterfaceData* id);
```

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

Parameter:

- map - Ausgabepuffer, der gezeichnet werden soll.
- tiles - Anzahl der Tiles des Puffers.
- id - Statistische Werte, die nicht in BufferTile-Form gespeichert sind.

Rückgabe: -

Listing 30: output_clear

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

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

Parameter: -

Rückgabe: -

Listing 31: 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