Programmieren in C

Prof. Dr. Peter Thiemann
Hannes Saffrich
Sommersemester 2021

Universität Freiburg
Institut für Informatik

Übungsblatt 7

Abgabe: Montag, 14.06.2021, 9:00 Uhr morgens

Hinweis: Es gibt einen kleinen Bug im bereitgestellten Code. Siehe Forum im Raum *Ankündigungen*.

In diesem Übungsblatt sollen Sie mit der TUI library, die Sie im vorherigen Übungsblatt entwickelt haben, ein Spiel programmieren:

Eine Demo-Video des fertigen Spiels finden Sie unter https://www.youtube.com/watch?v=H4KgUTdogHQ.

Hinweis: Sollte die Seitenzahl des Übungsblattes zu hohem Blutdruckführen, so können Sie beruhigt aufatmen: dies liegt großteils an den vielen Bildern, den ausführlichen Hinweisen und daran, dass die letzten beiden Aufgaben Bonusaufgaben sind. Machen Sie sich einen Tee und lesen Sie sich das Blatt zunächst ein mal komplett durch :)

`Vec*` mit `void*`-Elementen.

In unserem Spiel müssen wir die aktuell bestehenden Asteroiden, Projektile, Powerups, und Explosionspartikel verwalten. Deren Anzahl steht nicht im voraus fest, sondern ändert sich während dem Spielen je nach Spielverhalten. Wir brauchen also ein dynamisches Array wie den `Vec` aus Blatt 5.

Das Problem dabei ist aber, dass unser `Vec` so implementiert ist, dass er nur mit Elementen vom Typ `int` umegehen kann. Ein Asteroid wird aber zum Beispiel durch seine Position beschrieben - also einem Wert der Struktur

```
typedef struct Int2 {  /* from game_lib.h */
  int x;
  int y;
} Int2;
```

Um jetzt nicht für jeden Element-Typ die `vec.c` und `vec.h` kopieren und abändern zu müssen bedient man sich in C dem folgenden Trick: Anstatt die Elemente direkt im `Vec` zu speichern wie in Blatt 5, speichert man Zeiger auf die Elemente im `Vec`. Zeiger haben stets die gleiche Größe auch wenn die Daten auf die sie zeigen unterschiedlich groß sind: ein `Int2` ist doppelt so groß wie ein `int`, aber ein `Int2*` ist gleich groß wie ein `int*`, da beide Zeiger lediglich eine Speicheradresse sind.

Um sich bei dem `Vec` nicht auf einen bestimmten Zeiger-Typ festlegen zu müssen, gibt es in C den Typ `void*`. Der Typ `void*` steht für Zeiger, für die der Compiler nicht weiß welchen Typ die Daten haben auf die die Zeiger zeigen. Jeder Zeiger lässt sich in C automatisch zu einem `void*` konvertieren:

```
int   x = 5;
int*  p = &x;
void* q = p;  /* Make the Compiler forget that `p` points to an `int`. */
```

Die beiden Zeiger `p` und `q` enthalten dabei genau die gleiche Adresse - lediglich der Typ verändert sich.

Analog lässt sich ein `void*` zu einem beliebigen Zeiger Typ zurück konvertieren:

```
int* p2 = q; /* Promise the Compiler that `q` points to an `int`. */
assert(*p2 == *p);
```

Da der Compiler vergessen hat, dass `q` auf einen `int` zeigt, erfolgt die Rückkonvertierung auf eigene Gefahr! Zum Beispiel hält uns der Compiler nicht davon ab folgenden Code zu kompilieren:

```
double* p3 = q; /* Lie to the Compiler that `q` points to a `double`. */
```

Der Compiler glaubt uns hier einfach, dass `q` auf einen `double` zeigt, obwohl `q` eigentlich auf einen `int` zeigt. Der Wert von `*p3` ist also keineswegs `5.0` sondern eine schwachsinnige Zahl, da die einzelnen Bits des `int` so gelesen werden als würden sie einen `double` beschreiben. Da ein `double` mehr Speicher braucht wie ein `int`, wird dabei sogar über die Bits des `int` hinausgelesen (Speicherzugriffsfehler).

Um unseren `Vec` nun so umzuschreiben, dass er statt `int`-Elementen `void*`-Elemente enthält, reicht es aus in der `vec.c` und `vec.h` jedes Wort `"int"` durch das Wort `"void*"` zu ersetzen:

```
/* Old Declarations from sheet 5: */
struct Vec {
  int* data;       /* dynamic memory area containing elements of type int */
  size_t length;   /* how many elements are currently stored in data */
  size_t capacity; /* how many elements can be stored in data */
};
bool vec_push(Vec* xs, int x);  /* push an int element into the vector */
int* vec_at(Vec* xs, size_t i); /* return a pointer to the int element at index i */

/* New Declarations from sheet 7: */
struct Vec {
  void** data;     /* dynamic memory area containing elemnts of type void* */
  size_t length;   /* how many elements are currently stored in data */
  size_t capacity; /* how many elements can be stored in data */
};
bool vec_push(Vec* xs, void* x);  /* push a void* element into the vector */
void** vec_at(Vec* xs, size_t i); /* return a pointer to the void* element at index i */
```

Da wir in unseren Anwendungen stets Pointer im `Vec` ablegen wollen, die mit `malloc` angelegt wurden, schreiben wir die `vec_pop`-Funktion noch so um, dass sie beim Entfernen eines Elements, das Element für uns automatisch freigibt.

```
void vec_pop(Vec* xs) {
  xs→length--;
  void* x = *vec_at(xs, xs→length);
  free(x);
  if (xs→length * 2 == xs→capacity) {
    vec_set_capacity(xs, xs→length);
  }
}
```

Um zu verhindern, dass wir vor `vec_free` die Elemente des `Vec` noch von Hand entfernen müssen, rufen wir die neue `vec_pop` einfach in der `vec_free` auf:

```
void vec_free(Vec* xs) {
  while (vec_length(xs) > 0) {
    vec_pop(xs);
  }
  free(xs→data);
  free(xs);
}
```

Wir können den `Vec` dann wie folgt verwenden:

```
Vec* asteroids = vec_new();

/* Create a new asteroid. */
Int2* position = malloc(sizeof(Int2));  /* (1) */
*position = (Int2) { .x = 42, .y = 23 };
vec_push(asteroids, position);

/* Use the asteroid somewhere. */
Int2* position = *vec_at(asteroids, 0) /* recall vec_at returns void** */

/* Remove the asteroid, e.g. after it got hit by a projectile. */
vec_pop(asteroids); /* automatically calls free for the malloc in (1) */
```

In Ihrem Repository finden Sie die Dateien `vec.h` und `vec.c`, die den Vec aus der Musterlösung von Blatt 5 wie eben beschrieben abgeändert haben. Zusätzlich gibt es noch eine neue Funktion `void vec_remove(Vec* xs, size_t i)`, die das Element mit Index `i` aus `xs` entfernt und mit `free` dessen Speicher freigibt. Beim Entfernen werden alle Elemente mit Index größer `i` um eins nach links verschoben: hatte ein nachfolgendes Element also zuvor den Index `j`, so hat es danach den Index `j-1`. Die `vec_length(xs)` verringert sich dadurch um 1 (wie bei `vec_pop`).

Entfernen von Elementen aus einem `Vec*`.

Nehmen wir mal an wir haben einen `Vec* xs` und wir möchten dessen Elemente in einer Schleife durchlaufen und dabei bestimmte Elemente aus `xs` entfernen. Ein naiver Ansatz hierfür würde wie folgt aussehen:

```
for (size_t i = 0; i < vec_length(xs); i++) {
  if (element_should_be_removed) {
    vec_remove(xs, i);
  }
}
```

Der Code ist aber fehlerhaft! Wenn die Bedingung in der `if`-Verzweigung wahr wird und wir mit `vec_remove` das Element an Index `i` entfernen, dann rücken alle nachfolgenden Elemente eins nach links. Das Element das also vorher an Index `i+1` war ist nach dem `vec_remove` an Index `i`. Vor dem nächsten Schleifendurchlauf wird aber trotzdem `i++` ausgeführt, wodurch wir bei jedem `vec_remove` das jeweils nächste Element überspringen.

Das Problem verschwindet, wenn man den `Vec` stattdessen rückwärts durchläuft:

```
size_t i = vec_length(xs);
while (i > 0) {
  i--;
  if (element_should_be_removed) {
    vec_remove(xs, i);
  }
}
```

Wenn hier ein Element entfernt wird, dann verändert sich nur der Index von denjenigen Elementen die man bereits durchlaufen hat.

Generieren von Zufallszahlen.

Um einen zufälligen `int` zu erzeugen, kann die Funktion `int rand(void)` aus der `stdlib.h` verwendet werden.

Um einen `int` zwischen `0` und `4` zu erzeugen, kann der modulo operator verwendet werden: `int x = rand() % 5;`.

Die Wahrscheinlichkeit, dass `x == 0` gilt ist dabei genau $\frac{1}{5}$.

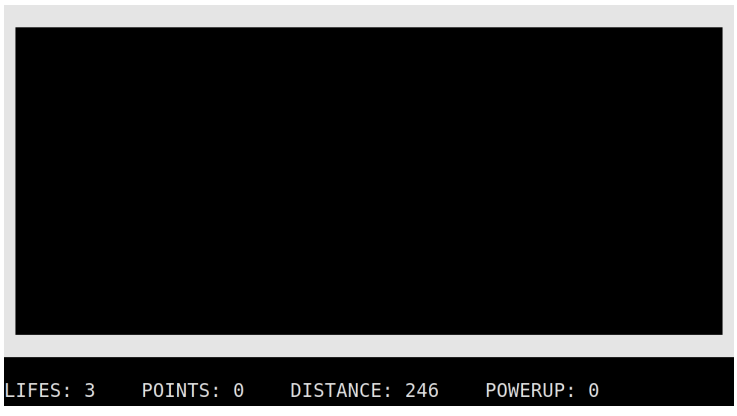
Überblick.

Ihr Repository wurde für dieses Übungsblatt um die folgenden Dateien erweitert:

- Im Verzeichnis `tui` finden Sie die fertig implementierte TUI library aus der Musterlösung von Blatt 6.
- Im Verzeichnis `blatt07` finden Sie die folgenden Dateien:
  - eine vollständige `Makefile`;
  - die Datei `game_lib.h`, die Funktionen deklariert, die Sie in diesem Übungsblatt in der `game_lib.c` implementieren sollen;
  - die Datei `game.c`, die eine fertige `main`-Funktion enthält, welche Ihre Funktionen aus der `game_lib.h` aufruft; und
  - die Dateien `vec.c` und `vec.h`, in denen die `Vec`-Struktur mit `void*`-Elementen definiert wird.

Die Aufgabenteile sind so entworfen, dass Sie vor und nach jedem Aufgabenteil ein funktionierendes Programm haben.

Wenn Sie ohne einen Aufgabenteil zu bearbeiten `make compile` ausführen und das Spiel danach mit `./game` starten, sehen Sie eine Statusanzeige und ein leeres Spielfeld mit weißem Rahmen:



Das Spiel kann durch Drücken der `'q'`-Taste wieder beendet werden.

Nach Aufgabenteil 7.1 wird ein Raumschiff gezeichnet, welches über das Spielfeld bewegt werden kann.

Nach Aufgabenteil 7.2 kann das Raumschiff Projektile abfeuern, die sich über das Spielfeld bewegen.

Nach Aufgabenteil 7.3 werden an zufälligen Positionen Asteroiden generiert, die sich über das Spielfeld bewegen und mit dem Raumschiff und Projektilen kollidieren können.

Nach Aufgabenteil 7.4 (Bonus) werden bei Kollisionen Explosionen gezeichnet.

Nach Aufgabenteil 7.5 (Bonus) werden an zufälligen Positionen Powerups generiert, die vom Raumschiff aufgesammelt werden können und vorübergehend dazu führen, dass das Raumschiff drei Projektile statt nur einem abfeuert.

Sie können bei diesem Übungsblatt selbst entscheiden, ob Sie Ihre Funktionen aus der `game_lib.c` mit Unittests überprüfen wollen oder zwischen den Aufgabenteilen lieber von Hand testen, indem Sie versuchen Ihr Spiel zu spielen. Für den Fall, dass Sie Unittests schreiben wollen, steht eine Datei `game_test.c` zu Verfügung, die Sie ansonsten ignorieren können.

In dem Fall, wo `malloc` einen `NULL`-Pointer zurückgibt, können Sie das Programm einfach mit `exit(1)` beenden.

Aufbau der `game.c`.

Die `game.c` enthält eine fertige `main`-Funktion und muss von Ihnen nicht mehr bearbeitet werden. Da die `main`-Funktion aber die Funktionen aufruft, die Sie in den folgenden Aufgabenteilen implementieren sollen, ergibt es Sinn zunächst einen Blick in die `game.c` zu werfen.

Die `GameState`-Struktur enthält alle für das Spiel relevanten Daten, wird in der `game_lib.h` definiert und an die Funktionen übergeben, die Sie in den folgenden Aufgabenteilen schreiben sollen.

In jedem Schleifendurchlauf in der `main`-Funktion passieren folgende Schritte:

- Als erstes wird geprüft ob eine Taste gedrückt wurde und wenn ja entsprechend reagiert: Raumschiff bewegen, Projektile abfeuern, etc.
- Danach wird der Spielzustand aktualisiert: die Asteroiden werden bewegt, es wird geprüft ob das Schiff mit einem Asteroiden kollidiert, etc.
- Danach wird der neue Spielzustand im Terminal angezeigt: das Schiff und die Asteroiden an ihrer neuen Position gezeichnet, etc.

Diese Schritte werden so lange wiederholt, bis das Raumschiff mit zu vielen Asteroiden kollidiert ist (game over) oder durch Drücken der `'q'`-Taste, das Spiel beendet wurde.

Beim ersten Durchlesen der `game.c` ist es okay falls Sie noch nicht im Detail verstehen was die Funktionen machen die innerhalb der Schleife aufgerufen werden. Diese

Funktionsaufrufe machen zu Beginn einfach nichts (siehe `game_lib.c`) und zeigen erst dann einen Effekt, wenn Sie den entsprechenden Aufgabenteil bearbeitet haben. Es geht mehr darum ein Gefühl für den grundlegenden Aufbau zu bekommen.

Koordinatensysteme.

In unserem Spiel haben wir zwei verschiedene 2D-Koordinatensysteme, die jeweils die Positionen von Zeichen in der Terminalmatrix beschreiben:

- Das eine Koordinatensystem bezieht sich auf das gesamte Terminalfenster und findet z.B. in `tui_cell_at` Verwendung. Wir nennen diese Koordinaten *Terminalkoordinaten*.
- Das andere Koordinatensystem bezieht sich auf das Spielfeld, also den Bereich innerhalb des weißen Rahmens. Wir nennen diese Koordinaten *Spielfeldkoordinaten*.

Die beiden Koordinatensysteme sind in Abbildung 1 dargestellt.

Die Spielfeldkoordinate `(x,y)` entspricht also der Terminalkoordinate `(x+1, y+1)`.

Um uns das Leben etwas einfacher zu machen, wird in der `game_lib.c` eine Funktion

```
Cell* field_cell_at(GameState* gs, int x, int y);
```

bereitgestellt, die sich wie `tui_cell_at` verhält aber Spielfeldkoordinaten entgegen nimmt. Um das Debuggen etwas zu erleichtern, stellt die `field_cell_at` sicher, dass sich die Koordinaten innerhalb des Spielfeldes befinden und bringt ansonsten das Programm mit einer Stack Trace zum Absturz.

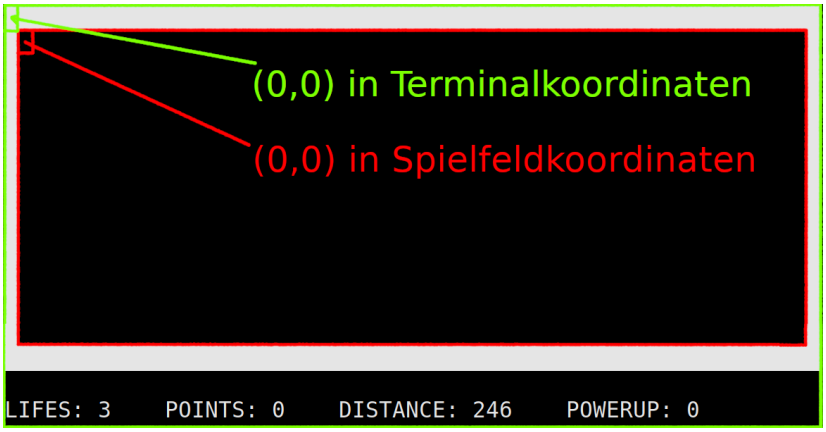


Abbildung 1: Koordinatensysteme

Aufgabe 7.1 (Where is my ship?!; 3 Punkte)

Kein Spiel ohne ein bewegbares Raumschiff...

(a) Implementieren Sie die Funktion `draw_ship`, die das Raumschiff an seiner aktuellen Position zeichnet. Die Farben können Sie frei wählen, aber verwenden Sie die in Abbildung 2 vorgegebene Form und Zeichen, um den Tutoren die Korrektur zu erleichtern. Das Raumschiff soll dabei so gezeichnet werden, dass das in der Abbildung markierte Minus-Zeichen sich genau an der Raumschiff-Position befindet. Es bietet sich an `field_cell_at` statt `tui_cell_at` zu verwenden (siehe Koordinatensysteme).

(b) Ändern Sie die Funktion `handle_input` so ab, dass sich das Raumschiff mit der für Spiele typischen WASD-Steuerung bewegen lässt:

- `'w'` bewegt das Raumschiff eine Zelle nach oben.
- `'s'` bewegt das Raumschiff eine Zelle nach unten.
- `'a'` bewegt das Raumschiff eine Zelle nach links.
- `'d'` bewegt das Raumschiff eine Zelle nach rechts.

Sorgen Sie dafür, dass man das Raumschiff nicht aus dem Spielfeld bewegen kann: es soll stets mindestens eine Zelle frei sein zwischen dem Raumschiff und dem Rand des Spielfelds. Zum Beispiel soll sich das Raumschiff nicht weiter nach oben und nicht weiter nach links bewegen lassen wie es in Abbildung 2 zu sehen ist.

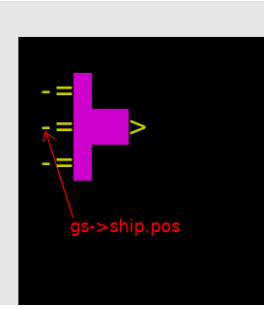


Abbildung 2: Raumschiff

Aufgabe 7.2 (Fire up those plasma cannons!; 4 Punkte)

Die Projektile werden in einem `Vec*` verwaltet - dem `projectiles`-Feld der `GameState`-Struktur. Jedes Projektil wird durch seine Position repräsentiert (`Int2`). Die Positionen sind in Spielfeld-Koordinaten.

(a) Verändern Sie die Funktion `handle_input`, sodass durch Drücken der Leertaste ein neues Projektil an der Position der Spitze des Raumschiffs mit `malloc` erstellt wird (siehe Abbildung 3) und dem `projectiles` Arrays hinzugefügt wird.

(b) Implementieren Sie die Funktion `draw_projectiles`, die jedes Projektil durch eine Zelle mit dem Zeichen `'>'` zeichnet.

(c) Implementieren Sie die Funktion `move_projectiles`, die alle Projektile um eine Zelle nach rechts bewegt. Verlässt ein Projektil das Spielfeld, so soll es aus `projectiles` entfernt werden.

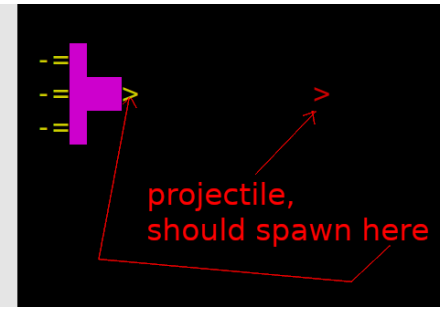


Abbildung 3: Projektile

Aufgabe 7.3 (Asteroids everywhere!!1; 7 Punkte)

Die Asteroiden werden in einem `Vec*` verwaltet - dem `asteroids`-Feld der `GameState`-Struktur. Jeder Asteroid wird durch seine Position repräsentiert (`Int2`). Die Positionen sind in Spielfeld-Koordinaten.

(a) Implementieren Sie die Funktion `spawn_asteroids`, die in der letzten Spalte des Spielfelds neue Asteroiden mit zufälligen `y`-Positionen erstellt und sie `gs->asteroids` hinzufügt. Für jede Zelle in der letzten Spalte des Spielfelds soll mit einer Wahrscheinlichkeit von 2% ein Asteroid erzeugt werden. Dies soll nur passieren, wenn `time_step` durch 5 teilbar ist (jeden fünften Zeitschritt).

(b) Implementieren Sie die Funktion `draw_asteroids`, die für jeden Asteroiden eine Zelle mit weißer Hintergrundfarbe zeichnet.

(c) Implementieren Sie die Funktion `move_asteroids`, die alle Asteroiden um eine Zelle nach links bewegt. Verlässt ein Asteroid das Spielfeld, so soll er aus `asteroids` entfernt werden. Dies soll nur passieren wenn `time_step` durch 5 teilbar ist (jeden fünften Zeitschritt).

(d) Implementieren Sie die Funktion

```
bool collides_with_ship(Int2 ship_pos, Int2 pos)
```

die überprüft, ob ein Raumschiff an Position `ship_pos` mit einem Asteroiden an Position `pos` kollidiert. Eine Kollision liegt dabei genau dann vor, wenn die Position des Asteroiden mit der Position einer Zelle des Schiffskörpers übereinstimmt. Der Schiffskörper besteht aus den lila-farbenen Zellen in Abbildung 2.

(e) Implementieren Sie die Funktion `handle_asteroid_ship_collisions`, die für jeden Asteroiden überprüft, ob er mit dem Schiff kollidiert. Im Falle einer Kollision, soll der Asteroid aus `gs->asteroids` entfernt werden und `gs->ship.health` um eins verringert werden.

(f) Implementieren Sie die Funktion `handle_projectile_asteroid_collisions`, die für jedes Projektil und jeden Asteroiden überprüft ob diese kollidieren. Im Falle einer Kollision sollen sowohl der Asteroid als auch das Projektil entfernt werden. Für das Abschießen eines Asteroiden werden dem `points`-Feld der `GameState`-Struktur 5 Punkte gutgeschrieben.

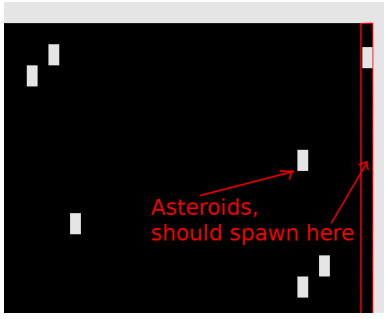


Abbildung 4: Asteroiden

Aufgabe 7.4 (Let's blow some stuff up!; 3 **Bonus**-Punkte)

Die Explosionen werden in einem `Vec*` verwaltet - dem `explosions`-Feld der `GameState`-Struktur. Jede Explosion wird durch ein Wert vom Typ `Explosion` repräsentiert, der aus der ursprüngliche Position (`Int2`) und dem Alter (Anzahl Zeitschritte seid dem Erstellen) der Explosion besteht. Die Positionen sind in Spielfeld-Koordinaten.

(a) Verändern Sie die Funktionen `handle_projectile_asteroid_collisions` und `handle_asteroid_ship_collisions`, sodass bei einer Kollision an der Kollisionsposition eine Explosion erstellt wird und zu `explosions` hinzugefügt wird.

(b) Implementieren Sie die Funktion `move_explosions`, die das Alter jeder Explosion um eins erhöht. Explosionen die älter als 5 Zeitschritte sind werden entfernt.

(c) Implementieren Sie die Funktion `draw_explosions`, die jede Explosion durch 4 Explosionspartikel zeichnet. Zu Beginn befinden sich die Explosionspartikel alle an der Ursprungsposition der Explosion. Mit jedem gealterten Zeitschritt werden die Explosionspartikel weiter vom Ursprung der Explosion entfernt gezeichnet. Jedes der Partikel bewegt sich dabei in eine andere Richtung (oben, unter, links, rechts). Da die Terminalzellen in etwa doppelt so hoch wie breit sind, bewegen sich die horizontal fliegenden Explosionspartikel um zwei Zellen pro Zeitschritt, die vertikal fliegenden aber nur um eine Zelle. Jedes Partikel wird nur gezeichnet, sofern es noch im Spielfeld ist (`is_field_coordinate`).

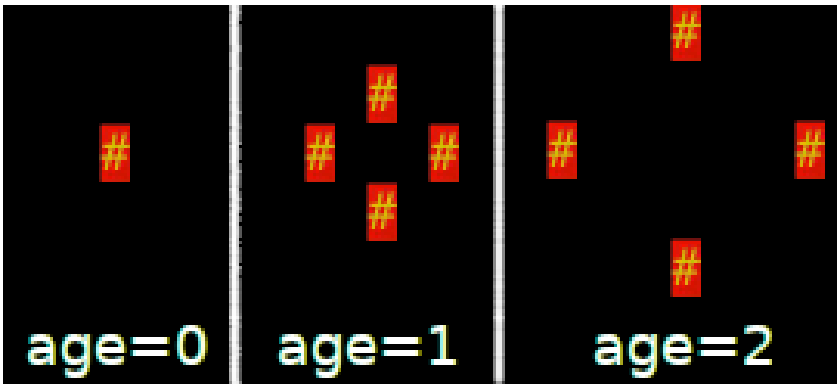


Abbildung 5: Explosionen

Aufgabe 7.5 (Give me some of those yummy powerups!; 3 **Bonus**-Punkte)

Die Powerups werden in einem `Vec*` verwaltet - dem `powerups`-Feld der `GameState`-Struktur. Jedes Powerup wird durch seine Position repräsentiert (`Int2`). Die Positionen sind in Spielfeld-Koordinaten.

(a) Implementieren Sie die Funktion `draw_powerups`, die jedes Powerup als ein grünes `'@'` zeichnet.

(b) Implementieren Sie die Funktion `spawn_powerups`, die mit einer Wahrscheinlichkeit von 0.5% ein Powerup erzeugt und dieses zufällig in der letzten Spalte des Spielfelds platziert und zu `powerups` hinzufügt.

(c) Implementieren Sie die Funktion `move_powerups`, die alle Powerups um eine Zelle nach links bewegt. Verlässt ein Powerup das Spielfeld, so soll es aus `powerups` entfernt werden.

(d) Implementieren Sie die Funktion `handle_powerup_ship_collisions`, die bei einer Kollision eines Powerups mit dem Schiff, das Powerup aus `powerups` entfernt und `gs->ship.powerup_time` auf 1000 setzt. Die `powerup_time` wird in `game.c` in jedem Zeitschritt runtergezählt, sofern sie größer 0 ist. Für's Aufsammeln eines Powerups gibt's 50 Punkte.

(e) Verändern Sie die Funktion `draw_ship`, sodass wenn ein Powerup aktiv ist, das Schiff mit zwei weiteren Plasma Cannons ausgestattet wird.

(f) Verändern Sie die Funktion `handle_input`, sodass wenn ein Powerup aktiv ist, zwei weitere Projektile an den Plasma Cannons erzeugt werden.

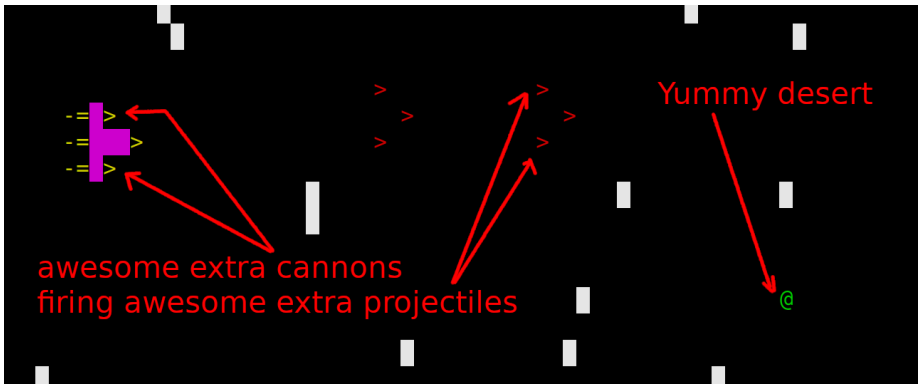


Abbildung 6: Powerups

Aufgabe 7.6 (Erfahrungen; 2 Punkte)

Notieren Sie Ihre Erfahrungen mit diesem Übungsblatt in der Datei `erfahrungen.txt` (benötigter Zeitaufwand, Probleme, Bezug zur Vorlesung, Interessantes, etc.).

Der Zeitaufwand *muss* dabei in der ersten Zeile und in exakt dem folgenden Format notiert werden, da wir sonst nicht automatisiert eine Statistik erheben können:

```
Zeitaufwand: 3:30

<...Andere Erfahrungen...>
```

Die Angabe `3:30` steht hier für 3 Stunden und 30 Minuten.