Documentation

User Documentation

Start The Game

To build and run the game:

```
michael@michael-ubuntu: make all
michael@michael-ubuntu: ./saucer
```

Once in the game, a splash screen is initiated with the controls and goal of the game. The goal is to prevent the saucers (moving left to right) from escaping. Users can pick up health and rocket care packages that spawn randomly and take the form of '@' and '*'.

The launcher is controlled with the arrow keys, and rockets are launched with the spacebar. The game is setup to handle single key input. That is to say, holding spacebar and an arrow key at the same time will only register once, so you cannot fire and move. You must stop moving and then fire. The launcher can move left and right, but never fully outside the bounds of the screen.

The rate in which the saucers launch increases as the score increases. The rate is defined as:

```
2 seconds - score * 0.0005 seconds
```

New Features

In addition to the original design specifications, some further features were added.

Random care packages spawn every DELIVERPACKAGE seconds. Initially this is 2 seconds, however as the score increases, the spawn rate of care packages will also increase. For further details see how the timing thread is handled. The code to implement the care packages also allowed an easy method to dynamically increase the rate that saucers spawn such that as the score increases, saucers spawn at a faster rate.

The game continues forever. Initially the game would end after a certain number of saucers were launched. The main game loop writes over the array position of rockets or saucers that have collided to allow the game to continue without dynamically allocating and freeing memory. The only limitation is that a maximum of 20 rockets and 20 saucers can occupy

the screen at any one time. The game is only terminated when the user enters ‘q’, has less than 0 rockets or has run out of lives.

Developer Documentation

Code Structure

The code is modulated in three .c files (saucer.c, draw.c and threads.c). Saucer.c handles the main game loop, keyboard input, and the thread loops. Draw.c is responsible for drawing game objects to the ncurses window. Threads.c handles the creation and cancellation of pthreads as well as the creation of saucer and rocket structures. Global variables, definitions, and function prototypes are defined in saucer.h.

Four important arrays exist.

```
pthread_t rthreads[MAXTHREADS];
pthread_t sthreads[MAXTHREADS];

struct object rockets[MAXTHREADS];
struct object saucers[MAXTHREADS];
```

The original implementation of the program used a linked list to dynamically allocate memory. However freeing the the lists was troublesome within the critical sections and would often result in deadlocks.

The current approach overwrites items in the list after a collision has been detected through the following code:

```
/* Array position */
int i = id % MAXTHREADS;
```

The array position rolls over after MAXTHREADS has been reached. This allows a maximum (as it is currently defined) of 20 rockets and 20 saucers on the screen at any one time. Id is passed into the method as either the number of rockets or saucers launched.

With the exception of the care packages and the launcher, onscreen objects in the game take the form of:

```
struct object {
        int x;
        int y;
        int type;
        int velocity;
        int direction;
```

```
        int id;
        int collision;
};
```

There existed enough similarities between rockets and saucers that a single object struct could be used to define both. Type is defined as either ROCKET or SAUCER in the header file and velocity is set at one. Directions are also defined in the header file and are different for saucers and rockets. Id represents an object's position in the rockets and saucers array as outlined above.

While implementing the game, I switched from clearing the screen every one second to deleting only elements that exist on the screen. Clearing the screen every one second caused the screen to flicker. The current implementation uses the velocity of an object to retrieve the old coordinates and overwrite the ncurses window with blank spaces in that position.

Best efforts have been taken to follow a javadoc commenting style in the source code to complement the design document. However the comments have not been compiled into an html page.

Threads

There exist four types of threads in Saucer.

Timing Thread

```
void *time_handler();
```

Timing thread handles the creation of saucers and care packages in the game. The thread is initiated from the main thread and runs an infinite loop or until the game's exit conditions are satisfied.

Timing thread's sleep time is dynamically adjusted based off the games score. The higher the score gets, the less sleep time occurs. This results in more saucers being launched and more care packages being delivered.

Rocket Thread

```
void *update_rocket(void *arg);
```

Rocket threads handle updating the position of a single rocket. If it is detected that rocket->collision is TRUE, the thread exits. Collision detection is set in the saucer thread which is read in the rocket thread through the use of a mutex.

Rocket thread sleeps for 1 second * a rocket's velocity which is defined as one in the header file. After the sleep the old rocket is deleted from the ncurses window and a new rocket is drawn.

Takes as an argument, an object struct which holds the coordinates of a rocket.

Saucer Thread

```
void *update_saucer(void *arg);
```

The heaviest in terms of work, the saucer thread handles collision detection between rockets and saucers and the drawing of saucers to the ncurses window. If it is detected that a collision occurred, the score is incremented and the thread is closed.

The thread uses locks to access:

```
struct object rockets[MAXTHREADS];
struct object saucers[MAXTHREADS];
```

Sleep time is the same as that of the rocket thread and sleeps for 1 second * a saucers velocity.

Main Thread

The main thread on which all other threads spawn. Handles the main game loop that polls for keyboard input and the drawing of the launcher and care packages to the ncurses window.

Critical Section

All draws to ncurses and access to the rocket and saucers struct arrays must first go through a lock. I suspect that multithreading is actually slower than single threading in this application since the same arrays are iterated over multiple times in each thread instead of once in a single thread for collision detection.

All global variables take the form of sig_atomic_t integers. Specifically, the number of rockets fired, saucers launched, score, lives and rockets remaining.