Final Project - MPCS 51040

(Issued: Tuesday November 30, 2021; Last modified: November 30, 2021)

Deadline: Sunday December 12, 11:59pm CST

General Instructions

These are general instructions which typically apply to every assignment, where apprioriate, and **apply unless explicitly stated otherwise**.

If you have questions about this assignment, please ask for clarification on Piazza. **Any instructor clarifications on Piazza are assumed to be known, and apply to this assignment**.

Compiling

Your code must compile with `gcc -std=c11 -Wall -Werror -pedantic`. There should be **no warnings or errors** when compiling with gcc (as installed **on `linux.cs.uchicago.edu`**).

Your code will be tested on `linux.cs.uchicago.edu` and must compile and run there!

Handing in

To hand in your final, you need to commit all requested files (***with correct filenames***!) to your personal git repository. **Make a subdirectory called 'final' and place your files under that directory** (or a subdirectory thereof, as specified in the problem assignment). Don't forget to commit and push your files! You can check on `http://mit.cs.uchicago.edu` to make sure all files were committed to the repository correctly.

The deadline for this final is Sunday December 12, 11:59pm CST. Any changes made after the deadline are not taken into account.

Code samples

This document, and any file you might need to complete the final can be found in the git repository `https://mit.cs.uchicago.edu/mpcs51040-aut-21/mpcs51040-aut-21/`.

Grading

The following applies unless specified otherwise in the problem assignment. Your code will generally be graded based on the following points (in order of descending importance):

- Correctness of the C code: **there should be no compiler errors or warnings** when compiling **on linux.cs.uchicago.edu**. There should be no memory leaks or other problems (such as those detected by valgrind).
- Be prepared to handle errors, for example when opening files.
- Correctness of the solution. Your code should implement the required functionality, as specified in this document.
- Code documentation. Properly documented code will help understand and grade your work.

- Code quality: your code should be easy to read and follow accepted good practices (avoid code duplication, use functions to structure your program, . . .). This includes writing portable code (which will work on both 32 bit and 64 bit systems for example).
- Your code should not have hardcoded limits (such as the maximum input set size or the maximum number of elements, . . .), *unless it is explicitly specified in the assignment that this is allowed.*
- Global variables or static local variables **are not allowed** unless explicitly mentioned otherwise.
- Efficiency: your code should not use more resources (time or space) than needed.

Some items might be marked as 'optional' or 'extra credit'. When correctly completing these tasks, the points obtained will go towards mistakes made elsewhere in the **same** assignment.

1: Spatial Index

Problem Description

This task is an exercise on:

- Function pointers and late(runtime) binding.
- Unit testing
- Spatial indexing (and trees, depending on implementation choices)

In addition, this is building block for the final project.

Task 1

No makefile is provided. It is up to you to write a suitable makefile. As usual, make sure to accurately capture all the dependencies between the files of your project and to compile with appropriate compiler flags (`-std=c11 -g -Wall -pedantic`).

Your makefile should also have a ' `clean`' target (which removes any object files or binaries built by the makefile). The default target (named `test`) should be to build the unit test for task 1 and task 2 and execute it. In other words, issuing the `make` command without other options should build and run both unit tests.

The first unit test binary should be called `util_test`; the second should be `spatial_index_test` (with equally named targets for the makefile). Note that the latter make targets should only build the binary. They should not automatically run them. The `test` target should build both **and** execute both tests.

Task 2

Before we can build our spatial indexing library, we first need to provide some helper code to comfortably describe the spatial concepts. Take a look at `util.h` and implement the corresponding `util.c` file.

Note:

- Carefully read the comments about the coordinate system.

- Make sure you understand exactly how a rectangle is defined, in particular which points are included (i.e. inclusive or exclusive boundary) and what the consequences are for a rectangle defined by a single point.
- Many functions in `util.h` will be used in the final project. Others are mainly useful in the unit test. Make sure to use them!
- While the `rectangle` structure is in the public header, **it should be considered private**. This means **it is not allowed to directly access the members of the `rectangle_t` structure**.
- Pay attention to the ordering requirement for the two points defining a rectangle. While this is an implementation detail (i.e. this is not exposed to the public API, in particular because no direct access to the `rectangle_t` members is allowed), this requirement is there to help you.
- The hardest function to implement is `rectangle_intersect`. It might be a good idea to write a thorough unit test (i.e. evaluating all possible cases) first before implementing the function. (See the comments in the header on how to validate your function in the unit test).

You also need to write a unit test (**using cunit**) **which exercises each function** of `util.h`. Note that the unit test cannot assume any internal/non-public details, and so is limited to testing the public API. Write the unit test in `util_test.c` (and update the `makefile` if needed). An easy way to get started is to take one of the provided unit tests from earlier homeworks and use it as template.

Task 3

For this task, you will build an API for indexing a set of 2D shapes. The index allows (possibly efficient) retrieval of any element(shape) in the index which intersects with a given rectangle or point.

Examine `spatial_index.h`. **This file should not be modified**. Complete the corresponding `spatial_index.c` file.

For this first implementation, **performance is not a priority, only correctness is**. However, your code should satisfy all of the requirements below:

- Your remove function should actually remove the elements from your the data structure. (Don't just mark them as deleted.) In other words, **the memory usage of your data structure should go down as you remove words from the index**.
- You are not allowed to use any global or static local variables for the implementation.
- Your implementation should not have hardcoded limits, for example on the maximum number of elements.

The API does not assume anything about the elements stored within the index, other than that a bounding rectangle can be obtained for each element, and that it is possible to determine if a given rectangle intersects the element. (Note that it is possible for a point to intersect with the bounding rectangle of an element, but not actually intersect with the element itself, for example if the element represents a circle).

To keep the code reusable, as usual, elements are represented by `void *` pointers. When creating the index, two functions are provided which, for an element, allow obtaining a bounding rectangle or to perform the intersection test respectively.

Some possible implementation options:

- Store each element in a (dynamically sized) array of element pointers.

- Alternatively, use a linked list to store the elements.

Task 4 - OPTIONAL

Once your code from task 3 passes the unit test, you can try to implement the same functions described in task 3, but this time with an additional goal of reducing the execution time.

Implement your code as follows:

- Your optimized implementation should use the same `spatial_index.h` file (if for some reason you need to modify the file, please ask on the forum).
- Implement your optimized code in `spatial_index_opt.c`.
- Add another `make` target (named `spatial_index_test_opt` which should **run the same `spatial_index_test`** unit test but instead of linking with `spatial_index.c` it should link with `spatial_index_opt.c`.
- Add this additional target to your `test` target as well.

Completing this task will generate a bonus of up to 30% (of this task's score), which can offset any points lost *elsewhere in the final*. In order to get the maximum bonus, your code must

- pass all unit tests (yours and mine),
- must be error and warning free when compiling and running under valgrind,
- and the time complexity of your **`si_intersect` function must be better than** $\mathcal{O}(n)$.

Handing in

The following files need to be submitted, **in directory final**.

- `Makefile`
- `util.c` (you are **not** allowed to modify `util.h` *other than to complete the inline functions*.
- `spatial_index.c`
- The unit tests `util_test.c` and `spatial_index_test.c` (and any additional files you might have created for the unit test).
- `spatial_index_opt.c`, if you implemented the optional task 4.

2: Breakout Game

Goal of this final

This finalallows you to demonstrate your C programming skills. The following concepts and skills will be needed to complete the task:

- Dynamic memory allocation
- Implementing polymorphic behavior (objects) in C
- Makefiles and multi-file C programs

- Splitting programs into logically separate concepts and creating a corresponding header and implementation file
- Debugging

Spatial Index

Note that this final**requires** you to use your spatial index code and utility functions from the before.

The Breakout Game

The goal of this finalis to implement the well known breakout game (`https://en.wikipedia.org/wiki/Breakout_(video_game)`). Basically, the game consists out of a paddle (controlled by the player), a ball, a rectangular game area bounded on the top, left and right, and blocks within this area.

The game is won when all blocks are destroyed, and lost if the ball is able to leave the game area. When a ball collides with one of the blocks (or the walls), it bounces off them. Depending on the type of block, one or more collisions causes the block to be destroyed (and removed from the game area).

In addition to these basic concepts, the game can be made more interesting by adding special blocks. Some examples:

- A block which is only visible some of the time;
- A block which, when destroyed, changes game parameters: for example, it could temporarily increase or decrease the size of the paddle, speed up or slow down the ball, create additional balls, etc.
- ...

The finalis considered complete when the game (the basic concepts) is implemented, using the architecture described later in this document. This means a controllable paddle and a ball which bounces of the blocks in the game area (and destroys them).

Anything extra will yield a higher score (and/or bonus credit). Some examples:

- A level system: once all blocks are destroyed, a new level is created with higher difficulty (more blocks, higher ball speed, smaller paddle, ...)
- Special blocks
- ...

IT IS STRONGLY RECOMMENDED NOT TO ATTEMPT ANY OF THE EXTRA FUNCTIONALITY UNTIL YOU HAVE A WORKING BASIC GAME!!

Overall Design

The code in `game.c` is rather generic, in that it doesn't really assume a specific game. It mostly deals with setting up the screen, keyboard input (which is passed to the paddle object), providing all the game objects with a timer tick, and drawing all objects on the screen.

In a way, this code could be considered to be a simple 'game engine', capable of drawing a collection of objects (stored in a spatial index, to enable efficient collision detection) and (re)moving any object that might have requested so.

This has the benefit of being able to add different behaviors to the game without having to change much in `game.c`.

Objects

Everything which ends up being drawn on the screen **must be** an object (to be more precise, is an instance of `game_object_t`). See `game_object.h` for more details. (This means that **the only things drawn on the screen must be in your spatial data structure**.)

Because of the object oriented style, almost all of the functionality will be in your specific objects. Before you start, consider which visual elements will be in your game, and which object types you will need to represent these.

The only object which is 'special' is the paddle, and only since that will be the object which will be receiving user input (i.e. key left or right press).

You are **required** to implement objects using an external table of function pointers (i.e. all objects of the same type share the same struct containing the function pointers) as was discussed during the lectures. See the example directory in the repository for a complete example.

Screen

To keep the scope and the amount of work minimal, we will be using text graphics only. Routines for getting information on the screen or to manipulate the screen contents can be found in `screen.h`. *You are allowed to modify `screen.h` and `screen.c`.* (For example, you could add support for color – see `man ncurses` for an overview of the capabilities)

Screen size

The size of the screen will depend on the size of the terminal window *when the game is started* – we do not try to support dynamically adjusting the screen size when the terminal size changes.

You have two options:

- Require a certain minimum terminal size for your game, and only use a specific area of the screen (regardless of the terminal size).
- Have a minimum size, but use up to the terminal size, adjusting your game region and level accordingly.

If you choose the first option, **you must**, if the screen size is not acceptable (since it is too small), terminate the screen (`screen_destroy`) and print out an error message indicating what the determined terminal size is and what the minimum required terminal size is.

If you choose to implement the second option, you will receive a bonus (making up for points lost elsewhere).

Memory Leaks

The ncurses library (which is used by `screen.h` and `screen.c` under the covers does not free all of the memory it uses. So, when checking your program with valgrind, feel free to ignore memory leaks *which originate from the a call one of the `screen_xxx` functions.* **All other memory leaks will result in a reduction of your score.**

For more information, see `http://invisible-island.net/ncurses/ncurses.faq.html#config_leaks`.

Example

Take a look at `examples/screen_demo.c`. It (in combination with `screen.h` and `screen.c` shows how to draw and refresh the screen, and how to obtain the screen size . . . The functions in `screen.h` should be self-explanatory.

Caveat

Be very careful with changing elements of your spatial index during iteration (`si_walk`) or while being called from any other `si_xxx` function (such as `si_intersect`).

For example, consider what would happen if from a function called by `si_walk`, a call to `si_remove` was made. `si_remove` would remove an element, but `si_walk` might not realize this and still try to iterate over a now no longer present element.

The same thing is true for changing the bounding box for an element which is currently in the index. Assuming your index is storing elements in a certain order or position (based on their bounding box), a change in that bounding box without telling the index would mean that the element is now no longer in the correct position in the structure. This means it might no longer be found (or worse).

This problem is not specific to our spatial index. For example, imagine a binary search tree. The tree would store `void *` pointers in the nodes, and have access to a function capable of comparing two of these pointers. Once one of these pointers is added to the binary search tree, it is assumed that this ordering will not change! (if it would change without the binary search tree being aware of it, the node would now be in the wrong location in the tree!).

One way to deal with this problem could be to (during iteration or at any time when it is not allowed to modify the properties of the element) build a list (*outside of the spatial index*) of elements that need to be updated (or removed). Then, once it is safe to do so (for example after `si_walk` returns), your code can then examine this list and remove and/or reinsert these elements so that the spatial index can be updated accordingly.

Or, alternatively, depending on how you implemented your spatial index, you could make it safe for modification during `si_walk`. Note that you'll still have to handle the case in which the bounding box of an element was updated.

vararray

As a standalone task, complete `vararray.c`. You are strongly encouraged to use it (for example to implement the set of game objects that need to be removed or moved). You are allowed to modify `vararray.h` as you see fit.

Debugging

The fact that our game uses the whole terminal means that we can't directly debug the program via `gdb`. Otherwise, the output from the game and the gdb output would be mixed, making debugging very hard.

Instead, we'll be using the remote debug functionality of gdb. Instead of having gdb start the program directly, another program (`gdbserver`) will start the program and then wait for commands. We then use `gdb` as usual, but tell it to connect to the running `gdbserver` to control the program that we want to debug.

Steps:

1. Run `gdbserver --multi localhost:12345`
   NOTE: if somebody else is already using port 12345, pick another port between 1025 and 65535.
2. Now open another window (possibly via SSHing to the **same** linux server). You can tell from the prompt where you started `gdbserver` which server you should connect to (for example `linux1.cs.uchicago.edu`).
3. In the second window, start gdb using the `gdb` command (or `cgdb` if you prefer).
4. Now tell gdb that we will be connecting to the remote, that the remote is on the same system, and which program we want to debug.
   - `set sysroot`
   - `file game` (assuming game is the executable you want to debug and that it is in the current directory)
   - `target extended-remote 12345`

Now you can debug as usual. For example, setting a breakpoint on the main function (`break main`) followed by the run command (`r`) will allow you to step through the program. You'll see how the window that is running `gdbserver` will be showing the game output. Note that while the game is in stopped state (because it hit a breakpoint for example) it will not be able to respond to keypressed. However, any input during the stopped state will be processed once the game runs again.

General Notes

- You are allowed to modify all files, however, your code should clearly be based on the provided framework. **You need to document (in `README.md`) why you needed to modify the provided code.**
  - For example, if there is a function from `game.c` you want to use elsewhere, feel free to move it to a common file.
  - You are also allowed to use POSIX.1-2008 (and earlier) functions, but should prefer C11 standard functions if similar functions exist. (See how `game.c` makes the POSIX functions available by defining the correct preprocessor symbol)
  - The code fragments show how I implemented the game and is provided to help you get started. You are not required to follow the same structure (in other words, `COMPLETE ME` is really `POSSIBLY COMPLETE ME`).
- Other usual restrictions still apply, including but not limited to **no global variables**, **no warnings or errors with strict C11 compiler flags and all warnings enabled**.

- The goal of the finalis to demonstrate that you are capable of writing correct and high quality C code. What this means is that the quality of your game (features, visual attractiveness, etc.) is a secondary concern (though a more pleasant game experience will result in a better score). *Do not spend time on optional features or visual quality until the C code is warning, memory leak and error free and until the basics of the game are working. Do not try to optimize the spatial index for performance until your game is working.*
- Linking: you will need to link with `-lncurses` (and possibly with `-lm` if you use any math functions). Depending on your system (if you try on another system than `linux.cs.uchicago.edu`) you might also need other libraries. See `examples/Makefile` for a method to automatically obtain the list of needed libraries.
- Don't forget to review the examples in the example directory.