Matt Britt
CS162
12/5/2017

FINAL PROJECT: SNAKE

OVERVIEW

Snake is an implementation of the classic snake game in the terminal. The player controls a snake that must eat numbers to grow and win the game. If the player hits an obstacle, a life is lost. Lose all lives or run out of time and the game ends.

GOALS

Two goals are given for this program design:

1. Design a game with pointer linked spaces
2. Use object oriented programming concepts

REQUIREMENTS

This project must:

1) Be a single player, test-based game where the player can move through spaces and accomplish goals.
2) Space class – must represent a space the player can occupy.
   a) must have 4 Space pointers to link to other spaces
   b) must be abstract, containing one or more virtual functions
   c) unused pointers must point to null.
   d) Game must have at least 6 spaces
3) Gameplay:
   a) game must have a theme and a goal that the player can achieve
   b) game must keep track of which space that player is in
   c) game must have a container that has items needed to complete game
   d) game must have time/step limit
   e) user must be able to interact with parts of the space structure and not just collect items to complete the game
4) Interface:
   a) game must begin by declaring the goal
   b) game must not contain free-form input
   c) game must provide a menu option for each scenario of the game
   d) a map is not required, but encouraged

DIAGRAM

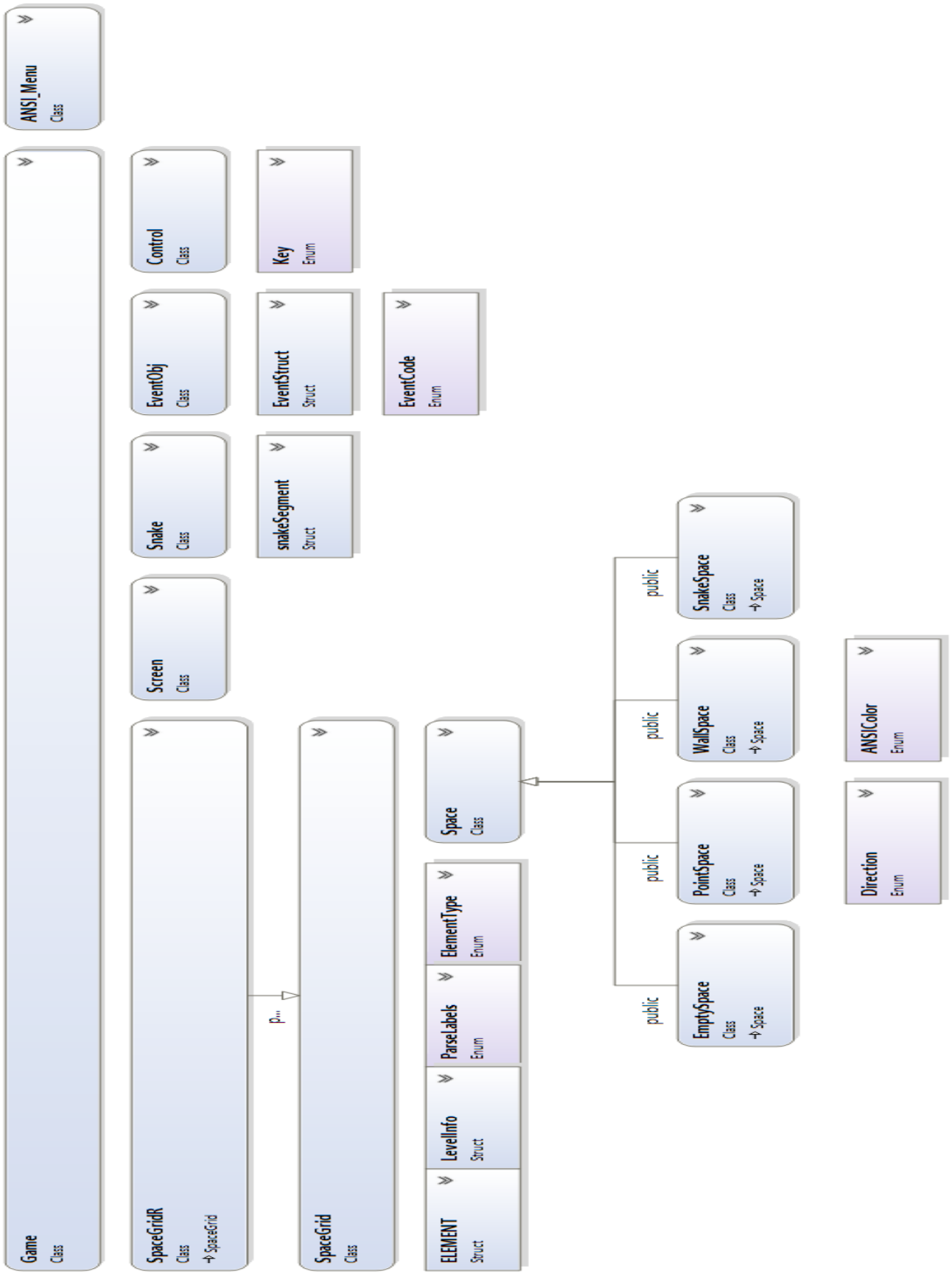

Figure 1 Basic Hierarchy Diagram For Snake Program

DESIGN

Extensibility:

The game loads text files that represent levels, giving the option of creating new levels without hard-coding them into the executable. This requires loading and parsing text files into memory structures.

State Management:

A simple event system is used to manage game state. Modules can create and react to events allowing communication between disparate portions of the program. This allows for non-blocking operations such as movement while still maintaining the time countdown.

Control Input:

Polling has been implemented to watch for keyboard input at the terminal without needing the Enter key to be pressed. Polling is accomplished by setting the terminal to "stty raw" mode and polling stdin to make sure that the input buffer contains key data before attempting to retrieve it. This allows for non-blocking terminal input. The technique is an extension of that used in Terry Quest by Harlan James.

Screen Output:

ANSI control codes are used to position and colorize screen elements. The Space class tracks if it needs to be updated and the Screen class only outputs data for Spaces needing updates. This minimizes the amount of data sent over the network (which is the main chokepoint preventing smooth gameplay). React.js was the inspiration for this functionality.

Object Oriented Programming:

The EventObj class relies on a static member variable to track state across program objects.

EmptySpace, WallSpace, PointSpace and SnakeSpace are all derived from the Space class. Space pointers are used to add and move spaces between objects. These objects have a virtual checkEvent() function that is used to react to game conditions set with EventObj.

SpaceGridR inherits SpaceGrid. SpaceGrid manages a dynamic memory object consisting of a "grid" of pointer linked Space objects. SpaceGridR allow the restoration of a space to it's original conditions by managing foreground and background SpaceGrid objects.

TESTING

Game:

Test Case	Input Values	Driver Functions	Expected Outcomes	Observed Outcomes
Snake collides with wall		Snake move, checkEvent	Snake loses life and stops moving	Snake loses life and stops moving
All lives lost	Lives < 1	Game	Game ends in loss	Game ends in loss
Dynamic memory is deleted/freed on exit		main	Valgrind finds 0 memory leaks	No mem leaks possible
Snake collides with PointSpace		checkEvent, Game	Score increased, snake lengthens, New point created	Score increased, snake lengthens, point created
9th point collected		checkEvent, Game	Game ends in win	Game ends in win
Time runs out	Time < 1	Game	Game ends in loss	Game ends in loss
Snake collides with itself(another SnakeSpace)		Snake move, checkEvent	Snake loses life and stops moving	Snake loses life and stops moving
Player tries to move snake backwards into itself		Snake move	Ignore input and keep snake movement going in same direction	Ignore input and keep snake movement in same direction

Note:

This program was tested on FLIP with PUTTY version 0.68. Putty implements all ANSI terminal control codes used by this project. Default Putty window (80 columns, 24 rows) displays correctly. Different terminal programs may produce different results.

REFLECTION

This was a fun, yet challenging project. Getting the non-blocking input figured out took a considerable amount of time, trial and error. This was part of my pre-project test program I used to determine if my goal of fluid movement was feasible over a terminal connection to the FLIP server.

With a project of this complexity, it was imperative that I tested thoroughly and often. Having many modules that are small and simple helped when I was trying to track down errors. Too often I wrote several functions at once and wasted time tracking down simple issues. I'm starting to understand the one module, one function paradigm.

This was the first project in cs162 that I had no significant troubles with memory management. The lessons learned in the previous projects and labs have helped me to understand where the problems usually arise. I also found I could better visualize how everything needed to fit together to get the results I wanted.

For this final project I wanted to challenge myself and explore some ideas I've had as we learned the cs162 material this quarter. I feel that I succeeded in achieving both the given requirements and my personal goals for this project.