CPSC 427 - A2: Video Game Programming
Game AI and Collision Processing Assignment

September 30, 2021

1 Introduction

The goal of this assignment is to introduce you to basic collision handling and game AI. You will extend the salmon game you made for Assignment 1 with additional features. The assignment includes both a required (80%) and a free-form component (20%). The goal of the latter is to let you experiment with computer graphics and have fun.

2 Template

You should use the same template as for Assignment 1 along with your own Assignment 1 code as a starting point. You will find comments throughout the files to help guide you in the right direction and entry points for this assignment are marked with `TODO A2`. The following classes will be of particular interest.

Collision. You will write additional functions in the `PhysicsSystem` class to test for salmon collisions with walls, making the collisions exact and efficient as discussed below. Detected collisions will be handled in the `WorldSystem`.

AI. To make your fish (and potentially turtles) smarter you will change their motion update mechanism with the `AISystem` (in `ai.h` and `ai.cpp`).

Debug. The `createLine()` in `world_init.cpp` will be useful for drawing debug information on the screen.

3 Required Work (75%)

1. Getting Started (15%):

(a) Commit all your edits from Assignment 1 to git, such that you can track and potentially revert changes. We recommend pushing your code to a **private** git repository, e.g., on GitHub or UBC servers to prevent loss of code upon malfunction of your machine.

(b) Keep a separate copy of your Assignment 1 executable and dlls, to be able to showcase your A1 solutions to TAs on request.

(c) Play the `a2_reference.mp4` video to get a sense of what a possible assignment solution should look like. Note the differences in the character behavior compared to A1.

(d) Reduce the speed and frequency of fish and turtles to make the new features you care about to add easier to see.

(e) Change the movement of the salmon to be consistent with its orientation, so that the mouse position rotates the salmon and the up/down keys move the salmon along the direction it is aligned with.

2. Collisions (35%, prereq Collision Det. lecture):

   (a) Spawn the fish with constant horizontal and a randomized vertical velocity and make them bounce off the walls, flipping their direction of vertical motion. Moreover, prevent the salmon from penetrating the top and bottom walls. Use **exact** collision computation for the salmon (test for collisions between the rotated salmon **mesh** and the walls). Use the salmon's bounding box(es) to make the computation reasonably efficient.

   Note, the mesh vertices of an entity `player` can be retrieved with `registry .meshPtrs.get(player_entity)`. The global position and scale also depends on the position, orientation, and scale properties stored in the `Motion` struct associated with the same entity.

   (b) Implement a debug mode toggle to visualize the impact of collisions as shown in the example video. You will need to write your own functions to show the bounding box of fish and the **exact** vertex positions of the salmon. This will make it easier to identify those colliding with the walls. The `PhysicsSystem::step()` function provides a starting point for visualizing the position and size of all moving objects with lines. You can combine lines to more complicated overlays or write your own primitives. The debug display is triggered with the `d` key.

3. Game AI (30%, prereq AI lecture)

   (a) Make the fish smarter by enabling them to avoid the salmon. The fish should avoiding the salmon by staying at least some minimum distance ϵ away from it. After an encounter, the fish should follow the shortest path to the wall opposite its starting point. The goal path should be updated every X frames. The frequency X with which these paths are recomputed should be user-controllable.

(b) Extend the debug mode from the previous part to visualize the computed paths of each fish on the screen and the spatial data structures (constructed bounding boxes, circles, ...) used to implement this feature. When in debug mode, freeze the view for a few milliseconds after every AI update to make the debug visualizations visible for longer than a single frame. Make sure the freeze happens after the draw call such that the AI debug information is displayed.

4 Creative Part(20%)

The required code changes described so far will let you earn up to 80% of the grade. To earn the remaining 20% to make the game more appealing by implementing one advanced feature. You can also gain bonus points when exceeding our expectations. **Marks for the advanced features will be granted only if both they and all basic features are fully implemented and functional.** Advanced feature suggestions:

1. Making the fish even smarter by accounting not just for the salmon's current position, but also for its anticipated motion trajectory. Similar to the previous part, add an advanced debug mode toggle that visualizes fish paths and spatial data structures used.

2. Avoid salmon/wall penetrations by accurately testing for **expected** collision position and time instead of existing penetrations. Like before, visualize the expected collision locations with an advanced debug mode toggle.

3. Replace the randomly spawned turtles with a single smart turtle that chases the salmon, while at the same time avoiding the fish. The turtles path should be updated every X frames. The frequency X with which the path is recomputed should be user-controllable. Visualize the spatial data structures used to enable this feature and the turtles path with an advanced debug mode toggle.

Use your imagination to make other additions than the ones listed above, however, please make sure you focus on tasks involving collision, geometry, AI, and debugging knowledge. To support both basic and advanced visualization and control features, you need to add a toggle option where the user switches between the two modes by pushing the 'a' and 'b' keys ('a' for advanced mode and 'b' for basic mode; either at startup or during the game).

Document all the features you add in the README.md file you submit with the assignment. Advice: implement and test all the required tasks first before starting the free-form part.

To get full credit you should add at least one of the advanced features above and make it fully functional **and** free from bugs. The grading of additional bonuses, features, and the size of bonuses will be at the marker's discretion. A bonus is given for solutions that go beyond the examples listed above. **Multiple partially implemented features will not receive full credit.**

5 Hand-in Instructions

1. Create a folder called "a2". As for A1, copy all your source files and the CMakeLists.txt as present in the template to this folder (same folder structure; the TA should be able to run CMake and compile). Double check that you include the `shader` folder. Excluded all generated files, such as /build, .vs, /out and the example videos! These would consume a lot of space on our server.

2. In addition, create a README.md file (Markdown language as used on github) that includes your name, student number, and any information you would like to pass on to the marker.

3. The assignment should be handed in with the exact command `handin cs-427 a2`

   This will handin your entire a2 directory tree by making a copy and deleting all sub-directories. If you want to know more about this handin command, use: man handin. You can also use the web interface on your myCS page to upload the assignment.

Recall, do not publish your solution on github or any other place. Neither during the course nor after; both is considered cheating.