# CS 4150/5150: Game Artificial Intelligence

# Project #2

# Collision Handling

# Due: 6:00pm, 2/9/22

## Overview:

Build collision avoidance for *Crash Loyal*, our home-brew clone of the popular mobile game *Clash Royale*. In preparing for this game, you may want to get a copy of *Clash Royale* (free on the Apple or Android store), and spend some time playing around with it, watching how different units interact when they try to move in to the same space.

## Starting Code:

Source code for *Crash Loyal* can be found in the Project2.zip file. The zip includes a Visual Studio 2019 solution. This game is still a work in progress, so if you find any bugs, please let me know! Also, as with project one, I don’t have a working solution for Mac (or Linux) development – Mac users will either need to get things compiling on their own, or find a PC to develop on.

The current interface allows you to drop units by holding a letter key while you left-click. The unit you get depends on the letter you hold:

G – Giant

S – Swordsman

A – Archer

Under the covers, but not currently displayed in the UI, is an elixir mechanic. Like in the game, elixir builds up over time, and each unit costs a different amount of elixir. Each player starts with 8 elixir, and gains 1 elixir every 3 second, up to a maximum of 10 elixir. Constants controlling this can be found in Constants.h. You might, for instance, want to give yourself infinite elixir so that you can spawn lots of units and test how they respond (this is likely something that we will do when testing your code…). There is also a DAMAGE\_MULTIPLIER constant there, which can be used to increase or decrease the amount of damage done universally for units (so setting it to 0 should turn damage off completely). You may find it useful to play with these values, in order to make it easier to test.

The opposing player is currently using a very simple AI I wrote, the logic for which can be found in Controller\_AI\_KevinDill.cpp. Basically, it just waits until it has 9 elixir, and then drops a giant at the bridge and two archers in the back. It’s not very bright, but is a decent placeholder – beating it is certainly possible, but takes a bit of work. Again, you may find it useful to tweak this for testing purposes.

## Project Goals:

Your task is to detect and handle collisions between the NPCs (i.e. swordsmen and archers) and:

* Other mobs
* The three towers at each end of the field
* The river
* The edges of the map

***You are expected to use steering forces (***[***https://www.red3d.com/cwr/steer/***](https://www.red3d.com/cwr/steer/)***).***

Some overall goals (in order of priority):

1. Units should never move off the edge of the map, no matter what.
2. Units should never move faster than their base speed (although they can move slower), no matter what.
3. It’s okay for units to block other friendly units (e.g. an archer caught behind a giant) – this can even be tactically desirable. However, opposing units should always be able to make progress forward – they may be delayed a bit, but they shouldn’t be completely blocked. We don’t want the “wall of giants” to be a valid strategy for blocking the enemy from getting across the river, for instance.
4. To the greatest extent possible given the above constraints, units should avoid moving into the river or interpenetrating (i.e. passing through one another), and when they do it shouldn’t be by as little as possible.

The hooks for handling collisions already exist in the Mob class (/CrashLoyal/src/Mob.h/cpp), and there are some comments with the label “// PROJECT 2: “ to help you figure out where to get started. Note that the interface set up there isn’t sacrosanct – you may need to change it in order to get things working to your satisfaction, and you may also need to change things elsewhere in the code. Those comments are meant to give you a place to start looking, nothing more. With that said, here are some rough ideas of what might need to be done:

* Implement checkCollision(). Some hints:
  + Units are squares.
  + pos is the center of the square.
  + GetSize() is the length of the side of the square
  + The easiest way to check for a collision between two squares is to check whether both the difference in x and the difference in y are less than the average of the sizes. Similar math can help to figure out how far to bump them back.
* Extend checkCollision() so that it returns all mobs/obstacles that are colliding, not just the first one found.
  + Fair warning: Handling situations where there are lots of collisions can be a pain, because fixing one can create others. It doesn’t need to be perfect, but get it as good as you can.
  + You might also want to replace it with a function that returns all nearby obstacles (within some radius), rather than only those with which there is a collision.
* Implement processCollision() so that it uses steering forces to move colliding mobs from on top of one another, and to avoid one another when they’re not already colliding. Some details – you may not get all of these working, but the more, the better…
  + When units have different values from GetMass(), only the lighter (i.e. lower mass) unit should be moved by a collision – so a heavy unit walking forward will push other units back (and thus can’t be blocked my lighter units) and a fast, light unit can be “stuck behind” a heavy unit (especially on the bridge). When a unit is being pushed backwards by an opponent, making it slide a bit to the side (if possible, given the constraints) is a nice touch. In *Clash Royale*, you can try watching the way a Golem pushes enemy units back as an example.
  + When units are moving in approximately the same direction, the one in back should be pushed back, while the one in front continues at the same speed – you shouldn’t be able to have a fast unit “push” a slow unit to make it move faster.
  + When units have the same mass and are moving in approximately opposite directions, they should both be slowed a bit as they slide around one another – but they should never be completely blocked.
  + Getting all these forces to balance is part of the challenge – you won’t be able to satisfy every constraint in every situation, and having some sort of priority system that turns off less important steering forces when they would interfere with more important ones is almost certainly needed. Having overall constraints that limit the final result (e.g. not allowing a unit to move faster than its base speed, not allowing units to move off the edge of the map). Grading will be partly aesthetic – how much of the time does it look and feel “right”?
* Get collisions working with the towers and the river, as well as other units.
  + There are many ways to handle this, but one would be to handle the towers as Mobs that have infinite mass and don’t move, and the river as three Mobs that have infinite mass, don’t move, and are wider than they are tall (i.e. rectangles rather than squares).
  + Right now, there’s a pathing bug where units that have selected an enemy target on the far side of the river move straight across instead of pathing to the bridge. Collision handling should prevent this from happening, although the unit may still get stuck on the river (or move sideways very slowly). Ultimately, most of the work for crossing at the bridge should come from the path planner, with collision handling just handling cases where units get bumped too far to the side.

## Desired Behavior (and grading guidelines):

The assignment will be graded based on how well it works, not on how it is implemented (aside from the expectation that you use steering forces). Grading will be done as follows:

* (30 points) Documentation
* (25 points) The rest of the game continues to work
  + Units can be spawned as described above.
  + Archers and swordsmen should approach to within attack range of their selected enemy, and then stand still while they do damage (unless they are pushed out of the way or have to move to stay in range).
  + Giants behave as above, except that they only attack towers.
* (55 points) Aesthetics – does it look and feel “right.”

## Deliverables:

Your deliverables should be zipped up and submitted via Canvas. They should include:

1. All source code needed to compile and run the game.
2. Written documentation that describes the approach you took. This should specifically include:
   1. A list of the steering forces you used.
   2. Anything you did beyond simply applying steering forces ((e.g. setting hard constraints, prioritizing forces, respecting units’ mass, etc.)
   3. What parts you think work particularly well.
   4. What things you might still like to improve.

In addition, the TAs will schedule a meeting with each of you, during which they’ll play-test your project while discussing your approach with you. During this time you’ll be able to discuss the feel that you were going for as you interpreted the guidelines given above.