# CS 4150/5150: Game Artificial Intelligence

# Assignment #3

# Collision Handling

# Due: 6:00pm, 3/2/20

## 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. 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 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. 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 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.
  + Something like
* Implement processCollision() so that it moves colliding mobs to a position where they are no longer 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 – thus a heavy unit walking forward will push other units back. In *Clash Royale*, you can try watching the way a Golem pushes enemy units back as an example.
  + When units have the same mass and 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.
  + When units have the same mass and are moving in approximately opposite directions, they should both be moved back a bit.
  + If the direction of the impulse that is moving a unit back is straight up/down the screen, you might want to add a little bit of motion to the side, so that the faster unit can eventually get around the slower one. In *Clash Royale*, you can try watching a small unit (like a wizard) trying to get around a big unit (like a Golem) as an example.
* 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. Below is a description of the behavior we’d like, along with point approximations for each item. Partial credit is possible, depending on how well each item works as well as on the overall feel and aesthetics of the result.

* (55 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.
  + Giants behave in the same way, except that they only attack towers.
* (10 points) When units are spawned directly on top of each other (as the opposing player AI does), they currently start out already colliding. They separate as they move toward their goal (i.e. as they move towards the closest enemy or toward the bridge), and thereafter should not overlap.
* (10 points) Units cannot enter the water, either from the shoreline or from the sides of the bridge (giants should be barely small enough to pass one another on the bridge if they move to the side), even when pushed.
* (10 points) When two giants approach one another head-on, they should pass side-by-side. The bridge is just wider than two giants, so there should always be room for this.
* (10 points) Faster moving units (e.g. an archer) should move around slower-moving friendly units (e.g. a giant or swordsman) when neither are in combat. When there’s not room to move around them (e.g. on the bridge) they should instead follow behind – although I’m not sure that that case will arise with the current layout of the map and mix of units.
* (5 points) When one unit (e.g. a giant) is being blocked by an enemy unit (e.g. an enemy swordsman or archer that’s attacking that giant), it should push the enemy backwards and to the side, out of its way.
* (5 points) When multiple melee units all want to attack the same enemy, they can push their fellows a bit to the side to make room, or stand behind them and wait their turn. For example, imagine a giant is attacking a tower while a second giant approaches. The second giant should move up alongside the first, maybe pushing it a bit to the side, maybe not. If a third giant approaches, it should either push between the two, go to one side of them, or stand behind them and wait until there’s room. Any of these solutions is acceptable.
* (5 points) When multiple archers want to attack the same enemy, they should be able to push their fellows forward or to the side enough to get into range, as long as there’s space.

## 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. A written description of the approach you took, what works well, and what you might still like to improve.