**CSCI 4611: Programming Interactive Graphics and Games**

# Assignment 2: Car Soccer

**Handed out:** Tuesday, September 22

**Worksheet Due:** Sunday, September 27 (11:59 pm)

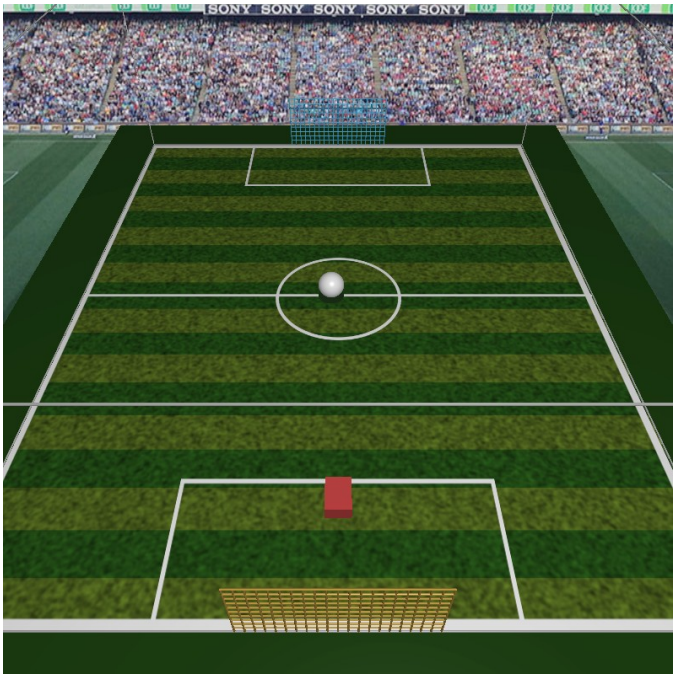
**– With 48-hour Grace Period:** Tuesday, September 29 (11:59 pm)

**Program Due:** Sunday, October 4 (11:59 pm)

**– With 48-hour Grace Period:** Tuesday, October 6 (11:59 pm)

## Introduction

To render compelling 3D computer graphics and games in real time, all graphics data has to be sent to the GPU using a graphics API such as OpenGL or DirectX. We will be using the MinGfx API in this course, and this assignment will be your first significant experience with it. You'll use the MinGfx library to work with 3D points and vectors in C++. These will help you to compute the positions, velocities, and other quantities needed to make a simple 3D interactive game based on the 2015 game *Rocket League* (http://rocketleague.psyonix.com/).




Above is a screen shot from an example of the game you will create. The playing field, or pitch, is rendered out of 3D boxes and line segments with some image textures mapped onto them. The car is drawn using simple 3D primitives; in my implementation, it's just a box. The ball is a sphere. The car can be made to move around using the arrow keys. When the car hits the ball, the ball reacts in a "physically plausible" way, updating its current velocity based on the direction it was hit and the velocity of the car at the time of impact.

In this assignment, you will learn to:

- Use the MinGfx graphics toolkit to build a 3D graphics program of your own.
- Draw simple 3D geometry using MinGfx.
- Work effectively with 3D points, vectors, and other "graphics math" primitives.
- Balance the tradeoffs between realism and effective game play by simulating physics in a "plausible" but not necessarily 100% realistic way.
- Successfully program a first interactive 3D graphics game.

## Worksheet

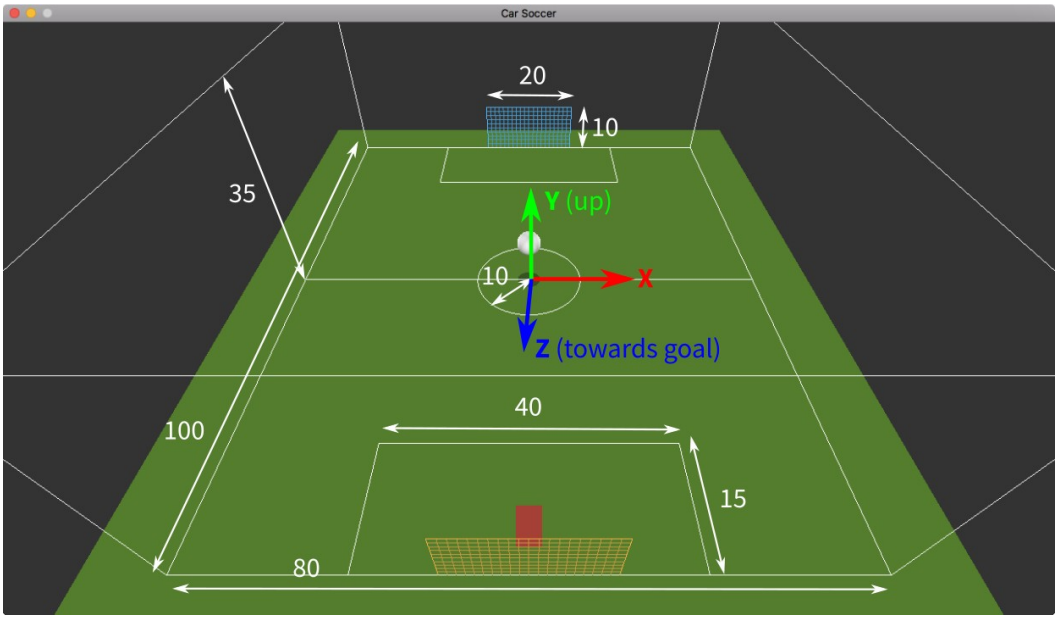
As with every assignment in this class, there will be a worksheet component to help you get started thinking about some of the concepts you'll need while working on the program itself. **Don't put the worksheet off until the last minute and be left with only a week to finish the program!** The worksheet is located in the class support code at **worksheets/a2_carsoccer.md**. Please follow the instructions in the getting-started repository in order to pull the latest support code. Edit **a2_carsoccer.md** and commit and push it to turn the worksheet in.

## Requirements and Grading Rubric

We will post a video of our solution to the assignment so that you have an example of what a successful implementation might look like. We also provide some support code to help you get started. This code includes the routines needed to draw the ball and car.

The game play you are required to implement is simple relative to what it could be. (There's a great opportunity here to go above and beyond the assignment to create a much more compelling game.) In this assignment, all we ask you to do is the following. Have the computer put the ball into play, kicking off the ball from the center of the pitch toward your side whenever you press the space bar. Program support for hitting the ball with the car whenever the two come into contact. You should also detect when the ball hits the goals on either side of the pitch. To do all this, you'll need to calculate collisions between the ball and the ground, walls, and car. Finally, add car-like steering and rotation instead of simply translating the car across the pitch.

Most computer games modeled after real sports are designed to balance the tradeoff between physical realism and game play. A completely realistic simulation of, say, soccer would be quite complex and would make it really difficult to play the game on a 2D computer screen, especially with the limited amount of control possible using a keyboard/mouse or controller input. In our case, the playing field is the size of a real soccer pitch, and the car is about the size of a Mini, but the ball is absurdly large to make it easy to hit. You may also want to have a larger-than-life gravitational acceleration so that things don't stay in the air for an annoyingly long time.




The dimensions of the pitch are shown above in meters, as well as the coordinate system we will use. The support code is set up to work in units of meters, matching the diagram above. The ball itself has a radius of 2.6 meters.

To make the game easier to implement, we will make a few simplifying assumptions about the physics. Specifically, you should follow these guidelines in your code:

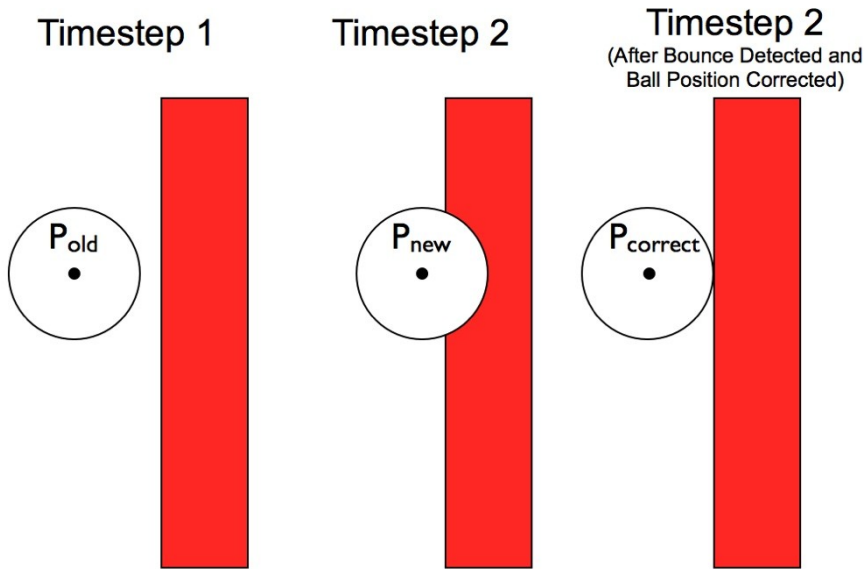
1. To simulate friction, you can simply decrease the speed of the ball a bit when it hits anything. For example, you might make the speed after bouncing 0.8 times the speed before the bounce.

2. To check for collisions between the ball and the car, you should treat them both as spheres, even if the car is actually drawn on the screen as a box (and would be a much more complicated model in a real game). This makes it extremely easy to do collision tests, as we describe below in the Technical Background section. In games, it is typical to test for collisions using such a “collision proxy” whose geometry is much simpler than the rendered model.

3. In real life, when two objects collide, they experience equal and opposite forces, and both their velocities change. To keep things simple, in this assignment we will assume that in a collision between the ball and the car, only the ball’s velocity changes while the car is unaffected.

4. Watch out for virtual balls penetrating other virtual objects, i.e. the ground, the walls, and the car. If you update your simulation once each frame, that means the “time step” of your simulation (i.e., the time interval *dt* between consecutive frames) will be somewhere around 1/30–1/60 second. That’s fast, but still not fast enough to capture the *exact* moment when the ball first makes contact with the car. This means that if you update the position of the ball using $\mathbf{p}' = \mathbf{p} + \mathbf{v}\,dt$, you may have a situation where, for example, the ball was away from a wall at the

last frame, but at the next frame the ball has not only hit the wall but is partially inside it! Of course, this cannot happen in real life, but in computer graphics, a virtual ball could actually penetrate the virtual wall unless you detect this situation and correct for it. The figure below demonstrates this situation. When you detect this situation, you can correct for it by simply setting $p_{new}$ to a value that places the ball just outside the wall, as shown.




A more specific list of requirements follows. We use this list in our grading, so this also serves as a grading rubric. To get a 100% on the assignment, you need to correctly implement everything that is listed here. To get a grade in the "A" range, you need to implement everything in the "C" and "B" ranges, plus some portion of the features in the "A" range, and so on. You'll find that the requirements are ordered very conveniently here, almost like a set of steps that you should take to complete the assignment. We recommend starting at the top and going down the list as you work.

*Work in the "C" Range Will:*

- Draw lines around the boundary of the 3D soccer pitch so that we can see the box that we will be playing inside. Also, draw a grid of lines for each goal. You can use the QuickShapes::DrawLines() routine to do this in MinGfx.
- Make the ball move through the air based on a random initial velocity, and relaunch the ball when the space bar is pressed.
- Update the ball's position and velocity each time step based on the acceleration due to gravity.
- Detect a contact between the ball and the ground, and make the ball bounce in the correct direction. That is, the ball's velocity vector should be reflected about the normal of the ground.

*Work in the “B” Range Will:*

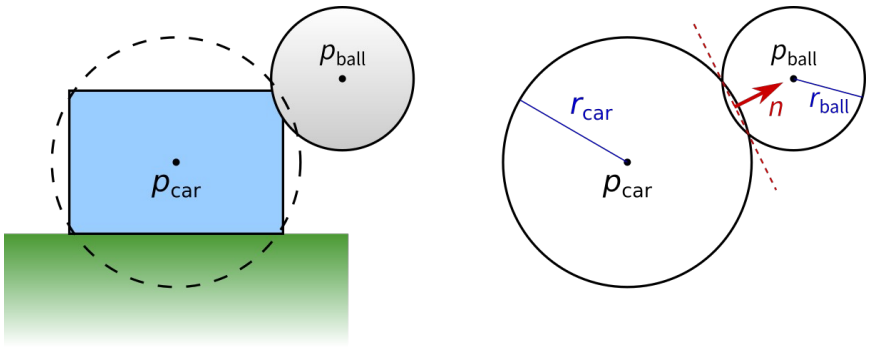
- Detect when the ball hits the walls and the ceiling and make it bounce off them too.
- Use a similar approach to prevent the car from leaving the bounds of the playing area.
- Decrease the speed of the ball when it bounces. This is due to friction and other factors, but you do not need to simulate these; just decrease the speed by some constant factor.

*Work in the “A” Range Will:*

- Respond to a contact between the ball and the car. To do this correctly — relative to the car’s spherical collision proxy, at least — you will need to define a “collision normal”, and reflect the ball’s velocity about the collision normal. See the technical discussion in the next section.
- Incorporate the velocity of the car into the way the ball responds when hit. For example, if you drive really fast into the ball it should go much faster than if you nudge it gently.
- Give the car a more realistic driving model. The car should always move forwards or backwards relative to the direction it’s facing, but never sideways. The up and down arrow keys should change its speed, while left and right should turn it at a rate proportional to its speed. A simple approach for doing this is given at the end of the technical discussion.
- When the ball hits one of the goals, reset the car to the initial position and relaunch the ball from the center of the pitch. You don’t have to animate the ball actually going *into* the goal, just treat them just as special regions marked out on a flat wall.

## Additional Technical Background and Tips

One of the main challenges in this assignment is handling collisions between the ball and the car. For collision purposes, we will approximate the car by a sphere, as shown in the figure below, so we only need to detect whether the two spheres representing the car and the ball are intersecting. Of course, this may result in us detecting collisions when the car’s rendered geometry does not actually hit the ball, or vice versa, but as long as the proxy and the render are not too different it shouldn’t matter too much to the gameplay.




In any collision handling implementation, there are two main steps: first, detecting whether a collision has occurred, and second, resolving the collision by updating the positions and velocities of the colliding objects. With spheres, collision detection is easy: two spheres are colliding if the distance between their centers is less than or equal to the sum of their radii. For collision resolution, we approximate the neighborhood of the collision by a plane, giving us a "collision normal", as you can see above. For a collision between spheres, the collision normal is simply parallel to the line joining their centers. Now you must do the following steps:

1. Move $\mathbf{p}_{ball}$ along $\mathbf{n}$ so that the ball is no longer intersecting the "car" sphere, i.e. the distance between $\mathbf{p}_{ball}$ and $\mathbf{p}_{car}$ equals $r_{car} + r_{ball}$.
2. Compute the relative velocity of the ball, $\mathbf{v}_{rel} = \mathbf{v}_{ball} - \mathbf{v}_{car}$.
3. Reflect the relative velocity about the collision normal.
4. Set the new velocity of the ball, $\mathbf{v}_{ball} = \mathbf{v}_{car} + \mathbf{v}_{rel}$.

You will not be graded on how you handle multiple simultaneous collisions, such as when the ball gets squeezed between the car and the wall; this can be a challenge to handle even in real games. We will only grade you on whether you correctly handle cases where the ball collides with only one thing at a time. In any case, we have set the default sizes of the ball and car so that the center of the car is lower in *y* than the radius of the ball, which means the ball will always be able to escape upwards to get out of the "pinch" caused by multiple simultaneous collisions.

As for the car's motion model, many different approaches are possible that vary in realism and playability. Here's a simple one you can use, though you are free to come up with your own. We may store the car's direction simply in terms of its angle in the *xz*-plane, equivalently, how much it is rotated about the *y*-axis. Separately, we store the speed in the forward direction, which may be positive or negative. At each frame, assume there is a thrust force, which is proportional to whether the up or down arrow key is held, and a drag force, which is proportional to the current speed; therefore, increment the speed by (*thrust* − *drag*) *dt*. For turning the car, define the turn rate, which is proportional to whether the left or right arrow key is held, and increment the direction angle by *turnRate* × *speed* × *dt*.

## Above and beyond

All of the assignments in the course will include great opportunities for students to go beyond the requirements of the assignment and do cool extra work. We don't offer any extra credit for this work – if you're going beyond the assignment, then chances are you are already kicking butt in the class. However, we do offer a chance to show off… While grading the assignments the TAs will identify the best 4 or 5 examples of people doing cool stuff with computer graphics. After each assignment, the selected students will get a chance to demonstrate their programs to the class!

There are some great opportunities for extra work in this assignment. Turn this program into a more exciting game! Add some wheels and brake lights to your car. Turn your car into a wheelchair soccer player! Change the camera position to follow the car and/or point to the ball! Add a car for a second player! Draw some fireworks when a goal is scored before resetting the ball! Keep count of the score by drawing some tokens above or next to the goals! (Text is hard to do in plain OpenGL, so don't bother with that.)

## Support Code & Handing In

When you submit your assignment, you should include a README.md file. This file should contain, at a minimum, your name and descriptions of design decisions you made while working on this project. If you attempted any "Wizard" work, you should note that in this file and explain what you attempted.

Now that we are on to using C++ and the MinGfx library, you will be working on this assignment within your course git repo. The first thing you should do is make sure that your repo is setup correctly to run graphics programs. Do that by following the instructions under Initial setup for a new computer. You are in good shape if you have been able to follow through those instructions and get to the point of running the gfxtest program. If you aren't yet to that point, then you need to work on this right away. Try to follow the instructions on your own first. If it doesn't work on your computer, try in one of the CSE labs. If it doesn't work there, come to office hours or email the TAs for assistance immediately. **Don't delay on this or you may get stuck not having time to complete the actual assignment!**

To access the support code for assignment 2, run the instructions under Update support code. Then, instead of building and running gfxtest, do:

```
cd dev/a2-carsoccer

mkdir build

cd build

cmake-gui ..
```

"**Configure**" then "**Generate**" then "**Open Project**"

Now, you should be able to build and run the support code, and you'll be able to start working on the assignment.

To save your code to Github, you need to:

```
git add -A

git commit -m "Your commit message"

git push
```

To hand in your code:

1. Git push your code to the master branch of your Github repository by the deadline. Any commits past the deadline will be assessed according to the late penalties described in the syllabus.
2. Fill out the Assignment Ready-To-Grade Form to let us know your code is ready to be graded.