Project 308: Augmented Reality Mario Kart

by

Joseph Abad

David Allender

Joryl Calizo

Ryan Gaspar

Gavin Lee

Senior Project

COMPUTER ENGINEERING DEPARTMENT

ELECTRICAL ENGINEERING DEPARTMENT

California Polytechnic State University

San Luis Obispo

June 2011

Table of Contents

I. Acknowledgements

We would like to thank Nintendo for coming out with such an awesome video game, as well as Dr. John Oliver for giving us the opportunity to go through with this project. We would also like to thank Cal Poly for its learn-by-doing philosophy because without that, none of this would ever be possible.

II. Introduction

Mario Kart is a popular go-kart racing game developed by Nintendo. The premise of the game is simple: drive a go-kart along a racetrack and reach the finish line before the other players. What makes this game unique, however, is the inclusion of weapons, traps, and other projectiles that a player can use to gain an advantage in the race.

We have taken on the challenge of not only recreating this amazing game, but using the art of Augmented Reality to fully immerse the player in the full experience. Rather than play the game on a television screen with a video game controller, the player can physically sit in a moving go-kart, drive along a track, and allow the virtual game to unfold on a mounted viewing screen. By overlaying HUD information and other graphics on the screen, the player can feel as though he or she is interacting with the virtual game.

Because of the scale of this project, the different components of the project are divided amongst five team members. These categories, in no particular order, are as follows: Game Mechanics, Hardware Mounting, Image Overlay, Kart Electronics, Networking, Positioning.

III. Requirements

A. Game Mechanics

To reproduce the original Mario Kart game as closely as possible, software for a server must be written that can run in the background and maintain the game state. The “karts” in the game should refresh their positioning information using data received from the on-kart electronics. When a player speeds up or turns their go-kart, these actions are reflected by the corresponding kart in the virtual world.

For debugging, the server will have the option of receiving positioning information from the keyboard rather than a physical kart. This feature allows the game to be tested even if the electronics on one go-kart fails. This functionality can be further developed as a secondary input to the game, allowing players to play the game via a USB controller or other computer peripherals.

One of the unique features of Mario Kart is that in addition to the racing aspect of the game, players also have access to a variety of items that they can use to gain an advantage. These items include projectiles that can be fired at an opposing player or static traps that can slow down other karts that drive over them. Project 308 will include support for a subset of these items, and the game must keep track of items currently active on the grid.

Finally, the server must relay information about the game state to the laptop and microcontroller on each kart. The server is responsible for constructing data packets containing this information and using the networking class functions to transmit the data.

B. Hardware Mounting

Each go-kart requires hardware to be mounted in such a way that the end-user will be able to drive the kart’s without any hardware interfering with his or her driving experience. Hardware that must be mounted includes a laptop, camera, the micro-controllers, and kart electronics.

For debugging purposes, the hardware must be situated in such a way that developers can quickly get to each part. This means that the each piece of equipment that is placed on the kart must be completely modular. Modularity in a design means that each piece of equipment is its own entity. While debugging, each node should be able to taken out by itself.

C. Image Overlay

Since the karts can not throw actual shells or run over actual game objects, the image overlay is required to render these game objects to allow the user to visualize the actual game. The driver also needs to have a sense of what is going on during the race so a heads up display is also required to keep the player

informed of how he is doing in the race. The main requirements of a racer in the Mario Kart game are as follows:

- Position - Player's position in the race (1st, 2nd, 3rd, etc.)
- Lap - The current lap the player is in.
- Inventory - What item the player has.

In term's of the augment reality world, Mario Kart has a set amount of weapons that make it distinguishable from any other racing game. To give the player's a Mario Kart type of feel we will reproduce these items in the augmented reality world for racer's to visualize. The set of core augmented reality objects we would like to produce are:

- Shells
- Item Box
- Banana

D. Speed Regulation

This segment of the project will connect the software-side of the video game to the hardware-side of it. Whatever happens in the virtual realm of the video game will be physically shown with this speed regulation. In order to do this, there must be a medium in between the laptop and the kart that will be able to:

- pick speed regulation data from a network packet
- translate that data to determine the speed of kart
- modify the speed of the kart

Since the kart's motor is electrical, the "medium" for this design will be a micro-controller. A micro-controller can communicate to a laptop via a serial connection, as well as manipulate analog voltages to modify the kart's speed.

The main task is to design a system so that the karts will not go its normal speed, unless called upon. The kart's maximum speed should be reserved for the "gold star effect". The go-kart must run on four different speeds: 100% (gold star effect), 75%(in-game normal speed), 50%(slow speed), 0%(completely stopped).

E. Networking

Communication is a vital part for a large integrated system. The karts will have to be able to send messages to the main game server as well as receive messages. Once each kart gets localized information from the various sensors that are attached, it has to send it to the main server. As of Spring 2011, testing the game has been made with only two karts, but the framework can support many more devices and karts. The user must specify how many players that are going to be participating in the game, two is the default, and only choice.

Initially, every kart must connect to the server in order for the game to start. Once the game begins,

then the various machines send status packets to each other the specified information about it's system.

The packets must be sent at a frequency in which there will be minimal latency between each of the systems. This means that each packet must be sent and received multiple times within one second. 10hz is an ideal rate at which data will be transferred at to maximize the efficiency of the game framework.

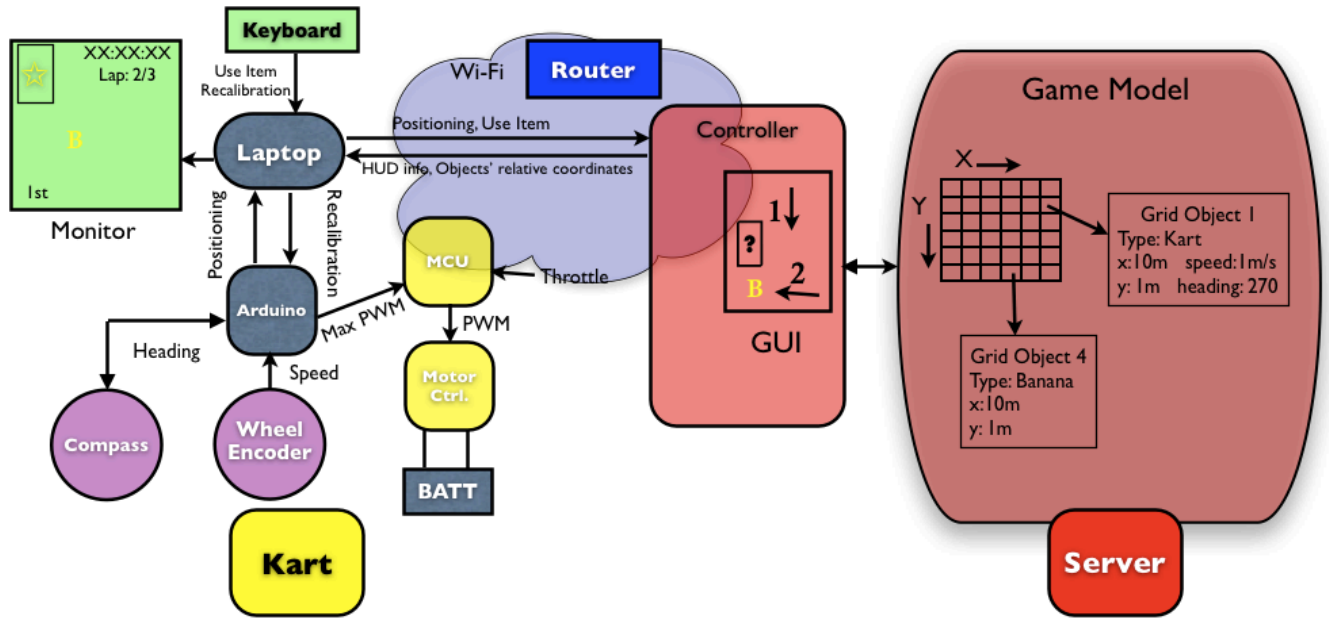
In order for each machine to connect, the server would need to setup the socket communication for each device. The server then waits on a "connect" method until each device sends an initial packet to the server.

F. Positioning

In Mario Kart, go-karts race to the finish around a track littered with obstructions and items that can be used as weapons to slow down the opponent or gain an advantage such as power-ups and speed boosts. Since it is impossible for us to physically create these items and weapons, we have taken these things in a virtual world and requires a known location of what we can physical have in the real world: go-karts and a track. The approximate position of the specified go-kart, its opponents, and the items must all be known in order for the game to be properly played.

Local information of the go-karts in real time is key in incorporating and integrating all parts collectively. However, unlike global positioning systems that have global coverage, local positioning methods will be used in order to obtain useful data. The kart's position determines whether or not item boxes have been obtained, where to place new items, and the position of used items. Several local positioning systems were considered while designing this project and will be later discussed.

IV. Design



System Block Diagram

A. Game Mechanics

i. Game Objects

The game needs to support a variety of different items; therefore, the obvious place to start is with a base class for a simple grid object. This class contains the features common to every object on the grid:

- x and y-coordinates (in 0.1m units)
- size / radius of object (for handling collision detection, in 0.1m units)
- strength of object (for resolving collisions, unitless value)
- ID (unique integer value for each grid object)
- object type (enumerated data type representing object)

Because numerous grid objects have mobile properties, a mobile grid object class with the base grid object class as its parent. Aside from the class members listed above, mobile objects also have the following information:

- speed (in 0.1m/s units)

- heading (in 1° units)

All objects are either a descendant of the mobile object class or extend the base grid object class directly. The only exception is the shell object class, which can be further divided into green and red shells, each of which function differently.

The following is a list of all the grid objects supported in the game:

- Wall (immobile object, represents outer boundaries)
- Green shell (basic projectile, bounces off of walls)
- Red shell (projectile with targeting capabilities; tracks nearest opponent)
- Banana (trap, slows down kart when hit)
- Item box (immobile object, gives kart new item when hit)
- Kart (mobile object, can carry an item)

As mentioned above, a new enumerated data type is used to represent the different objects used in the game, such as karts, weapons, item boxes, etc. As the game is further developed, items can be added or removed from this list without affecting existing code. In addition to the grid objects above, the following items are also included:

- Mushroom (temporarily increases speed of kart)
- Star (in addition to speed boost, temporarily renders kart invincible to traps and projectiles)
- Red/Green Shell x3 (Creates protective barrier around kart, can be launched as projectiles)

ii. Grid

To represent each grid object's position in relation to the next, a grid keeps track of every active object using its x and y-coordinate. This grid class represents the main data structure for adding, removing, and accessing each of the grid objects in the game. Any instance of the base grid object class or its descendants can be added to this data structure via a class member function, and by passing a pointer to an object and a new x and y-coordinate, the object's position in the grid and the object's own class members are changed accordingly.

At minimum, the grid class supports the following functions:

- *addObject* (adds a grid object to the data structure if x and y-coordinate are not occupied)
- *removeObject* (removes a grid object from data structure if present)
- *moveObject* (moves grid object to new coordinates)
- *get* (get pointer to grid object at specified coordinates)
- *isOccupied* (checks if coordinates are occupied)
- *getCount* (get number of objects in grid)

iii. Game Model

The heart of the game requires software that maintains the game state even without continued input from the user. This allows mobile objects such as projectiles to update their position at a constant rate until they are removed from the game due to a collision. The *update_mobile* function, when called within a timing-sensitive thread, handles the movement of all mobile grid objects.

As mentioned in the previous paragraph, there is a function called *find_collisions* that detects when two grid objects are "too close." *find_collisions* compares the two objects using their radii and determines if the two circles overlap. In the event of a collision, the strength values of each object are compared, and the object with the lower value is removed from the game. If both strengths are equal, both items are removed. The following cases are exemptions or extensions to this rule:

- Wall vs Wall
  Two wall objects will always have the same strength value, but neither should be removed. Optimally, walls should be placed such that they never overlap, but poor map design may cause this problem, and including this case prevents walls from disappearing prior to the start of the game.

- Kart vs Wall
  If a go-kart collides with a wall in the virtual world, neither object should be removed from the grid. This case will occur if the kart travels out of bounds; the kart's maximum speed should be decreased as a penalty until the kart returns to the track. Unfortunately, this case may also occur if the positioning system fails, so care must be taken to ensure that the hardware always returns valid coordinates.

- Kart vs Kart
  If two karts are determined to be within collision distance from each other, chances are the karts have already collided in the real world, in which case the karts will naturally stop moving. Regardless, the karts should not be removed from play, as this case may also occur due to a positioning error.

- Kart vs Projectile or Trap
  Because the strength value of a kart is higher than that of any weapon, the item will always be removed from the game. However, some adverse effects will happen to the kart as a penalty for colliding with the item. The kart's max speed will temporarily decrease; a thread must be spawned that changes the speed for a few seconds, returning the value to normal before exiting.

- Kart vs Item Box
  Similar to the previous case, an item box will always be removed when a kart collides with it. In this case, however, the kart is rewarded with an item if the kart is currently unarmed.

The function *add_object* places a new object on the grid. After checking to ensure that the given coordinates are unoccupied, an instance of the correct grid object is created and added to the grid data structure. When the new object is a projectile, the object's heading is automatically set to target the nearest kart. When a player's kart is armed with an item and wants to use it, the *use_item* function is called. *use_item* is very similar to *add_object*, but when the algorithm selects a target for projectiles, the player's kart is exempt.

When a kart collides with an item box, the game arms that kart with an item if one is not already available. Normally, a random item is selected from the list of items currently supported, but flags can be used to prevent some items from being chosen. This feature can prevent a player too far in the lead from receiving too many speed boosts; likewise a player in last place will not receive traps since no opponents are behind him or her.

The game class keeps track of other important information, such as each player's lap count, position in the race, and max allowed speed. Functions are available to change these values, and the function for changing max speed can be called in the context of a thread to make speed changes temporary.

Before the game begins, each kart's lap count is at zero. When the *start_game* function is called, all lap counts are set to one simultaneously. This change will be reflected in the packets sent to the kart laptops; until the game starts, the motors can be forced into braking mode.

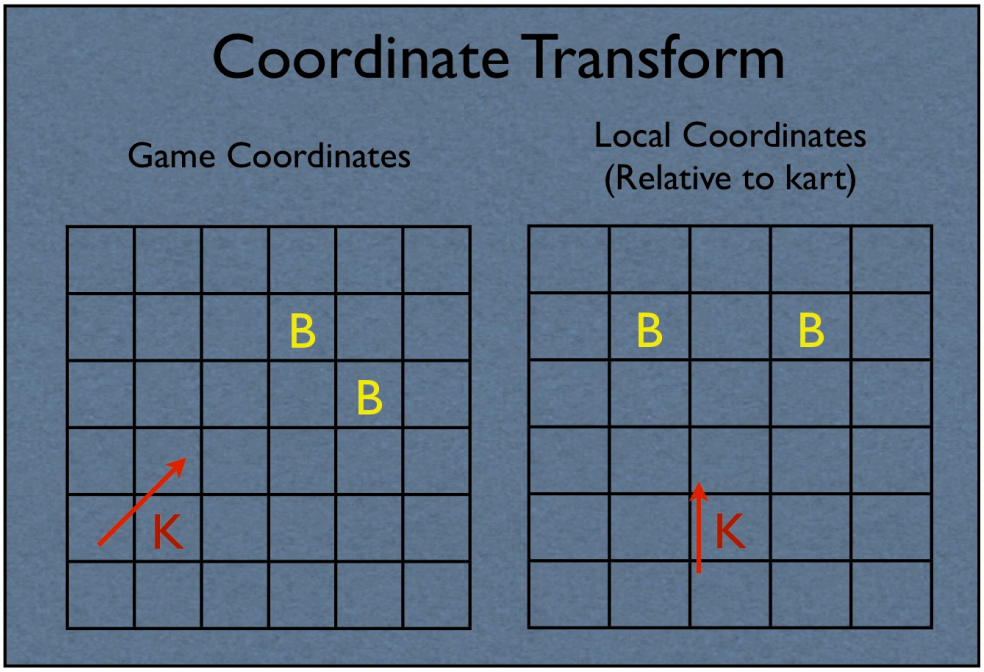
iv. Controller

The game model maintains the game state internally, but another system must be responsible for passing it information from the players or from keyboard inputs by a user at the server. The controller bridges the connection between the game and the user(s). The controller receives information about the game state and passes the relevant information to each of the laptops.

The controller is used to initialize the game by querying the user for starting settings (e.g., number of players, grid size, etc.). Once all devices are connected to the network, the controller sends the game the requested settings and calls the game's *start_game* function.

The network objects allow the computers mounted on the karts to connect to the main server, and the controller keeps track of each of these connections. When the electronics on the kart send fresh positioning data, the controller determines if the data is valid and passes it on to the game to change the kart's position in the virtual world. Aside from the positioning data, the kart computer also sends commands to use an item if the player has one available.

The controller knows the location of each kart in relation to the other objects on the grid. Using trigonometry and proper thresholding, the controller can calculate the coordinates of each object relative to the kart's own coordinate system. The packet sent to the laptop combines this data with the other information displayed on the HUD for the image overlay software to decode.



The controller provides a user interface for manipulating the game state and printing information to stdout. The following commands can be selected via keyboard:

- Display list of grid objects
  The controller displays positioning information about each active object on the grid. For karts, extra information such as the current item and lap count are also included.

- Display occupancy grid
  The controller prints a text-based representation of a 2D overhead view of the game.

- Add objects
  The user can manually add an object to the grid. The controller calls the *add_object* function of the game class, which automatically provides projectile items with a target.

- Change kart places
  This command increases or decreases the position of a player, shifting the places of all other players accordingly. Because grids and tracks are not pre-defined, the game has no way of automatically determining which kart is in each position.

- Increment lap count
  Until hardware or software is in place to detect when a kart crosses the starting line, a user at the server needs to manually press a key when a kart has moved on to the next lap.

In case the hardware fails on one of the karts, the controller can reroute positioning information to be received from the keyboard. The user can control the movement of an in-game kart or use an item with key presses, and the communication with the game class remains identical.

B. Hardware Mounting

The main requirement to meet is to have a mounting system which can house all electronics including the laptop. The proposed design is to have a all electronics in the front. This will make soldering much simpler because there will less wires to connect together. Also, the mounting must be secure so none of the parts fly off of the kart while drivers are in the game.

Steel and aluminum will be used because they are both very sturdy metals. There are currently no laptop mounts for go-kart applications so a laptop mount must be manufactured. A foundation for the laptop mount must also be manufactured.

Originally, the micro-controller and cameras was to be mounted on top of the laptop mount, but it was not an elegant design. The final proposed design was to have the camera and micro-controllers underneath the laptop mount. This will hide most of the 18 gauge wire that will be running from micro-controller to micro-controller.

i. Foundation Options

The original design for this project was to weld metal tubes onto a cylindrical hole that was already connect to the front of the kart. Unfortunately nobody in the group had welding experience or a red card from the Cal Poly machine shop. Welding the metal tubes would also mean the laptop mount would also need to be welded on as well.

Another possibility was to have a stand alone clamp that would clamp onto the base of the steering shaft. This would theoretically work, but it would not be a sturdy solution. The sturdiest solution would to have the laptop mount sit on three points.

The proposed design is to place two threaded rods in the two cylindrical holes. These threaded rods will then be bolted down, and a screw will be drilled into the rod and the whole. This screw will safely secure the rod from moving up and down. the bolts will secure the rod from moving from side to side.

ii. Laptop Mount Option

The proposed laptop mount will be made out of aluminum sheet metal. Sheet metal is easy to bend and cut. The metal will also be bent on three corners. Bending the corners will also make the design sturdier compared to a flat piece of sheet metal.

iii. Camera and Micro-controller Option

Since the camera and micro-controller must be underneath the laptop mount, a new mount must be implemented for these two components. The proposed idea is to mount two metal brackets underneath the laptop mount. These two brackets will not be readily available at local hardware stores so the brackets will have to be custom made at a machine shop. The brackets will be screwed onto the laptop

mount. Also, rubber must be applied at the bottom of these mounts to be prevent ground issues.

C. Image Overlay

i. Design Choices

	Options	Strengths	Weaknesses
Displaying AR World	Computer Monitor attached to laptop	-Easy to Mount -Plenty of control on how to place the view	-Needs to be powered -Puts more weight on the kart
	Laptop Monitor	-Self Powered -Support for keyboard inputs -Lightweight	-Harder to mount
Operating System	Windows	-Widely used operating system -multiple support for graphics drivers -multiple support for audio drivers	-Not much experience on programming on this Operating system
	Mac OSX	-Support for POSIX library -All members have experience on programming with it	-Difficult to find driver support
Programming Language	Java	-Multiple libraries available -Class based programming	-Harder to integrate with other members who are programming in C++ -Not much experience on programming with it
	C/C++	-Multiple libraries available -Easily integrated with other member's code -Plenty of experience working in C	-Not much experience on creating class based programming in C++

		-Class based programming	
	Objective C	-Class based programming -Multiple libraries available -Superset of C	-Little experience on it
	Perl/Python	-Able to implement classes quickly	-Little experience on it
Graphics Library	**OpenGL**	-Plenty of experience on it -opensource	
	Direct X	-Widely used graphics library	-No experience on it
Pulling Camera Frames	VideoMan Library	-Works easily with opengl	-No experience on it -Extra features not needed
	OpenCV	-Support for image processing	-Hard to integrate with opengl
Playing Sound	**Simple and Fast Media Library**	-Supported on OSX -Easy to load wav files -Easy to use -Easy control over sound files	-No experience on it -No support for playing mp3 files
	OpenAL	-Works perfectly with OpenGl	-Not much support on OSX
	AFPlay funcion on OSX	-Easy to use	-Needs to start a new process -No control over it
Handling Multiple Events	Spawning Multiple Threads or Processes	-Easy to implement -Not have to worry about blocking calls	-Could get messy -Hard to track multiple threads and processes
	Have one process only and make sure everything is performed sequentially	-Easy to maintain -Cleaner than spawning multiple threads	-Make sure no functions are blocking -Harder to implement

ii. Image Layers

The image overlay can be broken down into three layers where each layer represents a seperate visualization of the game. Using this layer system avoids the need for threads because it will require each layer to use a step function that will be called after that layer is setup. The first layer is the actual camera stream of what the kart is looking at while the race is going on. On top of the camera stream layer is the layer for the heads up display. This layer shows the current player's race information. The third layer is reponsible for rendering the objects that should be seen in the Mario Kart world. This image layer design was chosen so that a layer can be easily modified without touching the other two layers. For example, if a different style of a hud is preferred than only HUD layer needs to be changed and the other layers would still be able to work.
With the image layer design we can also have a running program even if one of the layers stop working.

a. Camera Layer

This layer's main responsibilty is to pull frames from the camera that will be displayed on the screen. The camera layer class will need to pull frames continously to resemeble a real time video feed. To make this class abstract we will just implement a step function that will have to be called continously in the higher level classes or else the video will be choppy. The pseudo code for this function will be as follows:

```
- void step()
{
        if(camera is available)
        {
                frame = (pull frame from camera);
        }
        function(display camera frame on screen);
}
```

b. HUD Layer

Figure C.i.b

The hud layer is designed so that it closely resemblies the hud of the actual Mario Kart game. As you can see from figure C.i.b the hud will need to show the player's position(1), current weapon(2), race time(3), and lap(4). All of this information is received from the game mechanics side through the network implentation. The hud items on 5, 6, and 7 can be implemented in future versions of the game (see future implementations section). The hud layer will also have a step function that should be called continuously along with the step function of the camera layer.

c. Object Layer

The object layer is to provide support for the different objects that will be seen in our 3D Mario Kart. These items are shells, banana peels, and item boxes. It will be the game mechanic's responsibility to notify the program where each object is located relative to the kart's current position. To match the design of the two other layers, this class will also have a step function that should be called along with the other layers.

iii. Image Layer Controller

This image layer controller is responsible for coordinating the layers together along with the packets received from the network. This design implementation is modeled from Apple's Model-View-Controller design implementation (see developer.apple.com). Most of the work will be done in this class to ensure that all the other classes are properly coordinated with each other. This class will also have a step function that will be called continuously in the main opengl loop.

One important function this class will have is a timer function. This timer function will return the current runtime of the program. With this timer function it will be very easy to measure time intervals. The code for calculating the frames per second of the graphics would be as follows:

```
static int lastTFrame = (get current runtime of program);
static int frameCounter = 0;

if((get current runtime of program - lastTFrame >= 1000)
{
   fps = frameCounter;
   frameCounter = 0;
   lastTFrame = (get current runtime of program);
   //cout << fps << endl;

}else
{
   frameCounter++;
}
```

The ability of solving time intervals allows us to coordinate the changes of the heads up display and the augmented reality objects with the real world's real time.

iv. Windows Operating System

Even though the other members chose the OSX as their operating system, the image overlay was first built and tried on Windows. The decision to go with this was mainly because of the past experience of using opengl on Windows. There was also plenty of support and tutorials with using opengl and a couple of sound libraries on Windows. When it came down to integrating with the networks section of the project the networking could not work because the member doing the networks did not have any experience on networking with Windows. In order to compensate for this, the image overlay system was eventually ported to OSX, with some adjustments, to make it easier on the member that is doing the networking for the whole system.

v. VideoMan Library

The second main reason Windows was chosen in the first place was because the VideoMan library worked perfectly with Windows. The VideoMan library was made especially for augmented reality and computer vision programs. VideoMan pulled frames from the camera constantly while having an OpenGL layered on top of it for graphics. When it came to porting the whole image overlay system to a unix operating system, VideoMan would no longer work because there is currently no driver support for a unix operating system with the library. The VideoMan library was discarded to fix this problem and instead a function was created that would continuously pull frames from the camera with OpenCV. For more information on VideoMan refer to the bibliography section of this report.

vi. Playing a Sound

Trying to find a sound library that would work with a unix based machine was a little bit hard due to the lack of support for drivers. OpenAl was first method to be tried because XCode came prebuilt with the OpenAL library and OpenAL was said to work perfectly with OpenGL. The problem that we ran into with

this design is that OSX no longer supported the ALUT library which was needed to load and play sound files. There was a way to load sound files into OpenAL but it required a lot of work that would require on decoding sound files. The next method that was tried was to just call seperate process and make a process call to OSX's afplay command. This worked out pretty good except for the fact that we would run into multiple forks when dealing with multiple sounds. Finally a sound library was found that worked perfectly with OSX and that was the Simple and Fast Multimedia Library. An example of playing a sound file is the following:

```
sf::SoundBuffer itemreel.LoadFromFile("./sounds/itemreel.wav");
sf::Sound scrollWepSound.SetBuffer(itemreel);
scrollWepSound.Play();
```

D. Speed Regulation

i. Backround Information

The Honda Minimoto go kart is an electrical go-kart specifically designed for children. Maximum speed for these karts is around 12mph based on wheel encoder calculations. There are three main sections for the kart, which include: Throttle, Minimoto Controller, and a DC Motor. As the throttle is pressed voltage rises and this voltage gets passed into the Minimoto Controller. The controller then converts this analog voltage and spins the DC motor. An important concept to note is that as the voltage from the throttle increases, the go-kart will accelerate; vice versa, as voltage decreases the go-kart will decelerate.

The voltage rail for this kart from the throttle to the minimoto controller is around 4.5 V.

ii. Micro-controller Input Signal

The throttle acts like a potentiometer. Increasing resistance will increase voltage. This is exactly what happens when stepping on the throttle. As the throttle gets pressed, voltage increases. However, a micro controller can only read a digital value (either 5V or 0V). In order to read this throttle value, there must be an Analog to Digital conversion. After conversion, the throttle will be read as a value from 0-255. This 8-bit number is used to determine the speed of the kart.

iii. Micro-controller Output Signal

The output of the micro controller must be an analog voltage. A micro controller is only able to output a TTL low or high. In order to output an analog voltage, the MCU must use a digital potentiometer or a pulse-width modulated signal.

A digital Potentiometer will be able to take in a digital number and translate this digital number to an analog voltage. A pulse-width modulated signal is essentially a duty-cycle varied digital voltage.

Either adjusting the value of the digital potentiometer or the duty cycle will be two viable options for speed regulation.

iv. Communication

Communication with the computer and micro-controller, in general, can be made via a communication interface like SPI,I2C, etc. Since the arduino is already used for Positioning, it would make sense to send speed data from the arduino. For positioning, the compass and the wheel encoder sends its data to a serial monitor. This serial monitor displays the speed as well as compass data. The Serial monitor, for speed regulation, will be used to send data to the AVR chip. Pushing specific keyboard keys will limit the speed of the kart.

Since we will be using the Arduino for communication purposes, this data must be sent over to the AVR. In order to do this, two data bits from the arduino will be sent to the AVR as inputs. Two bits are only necessary because there are only four different speeds to choose from.

E. Networking

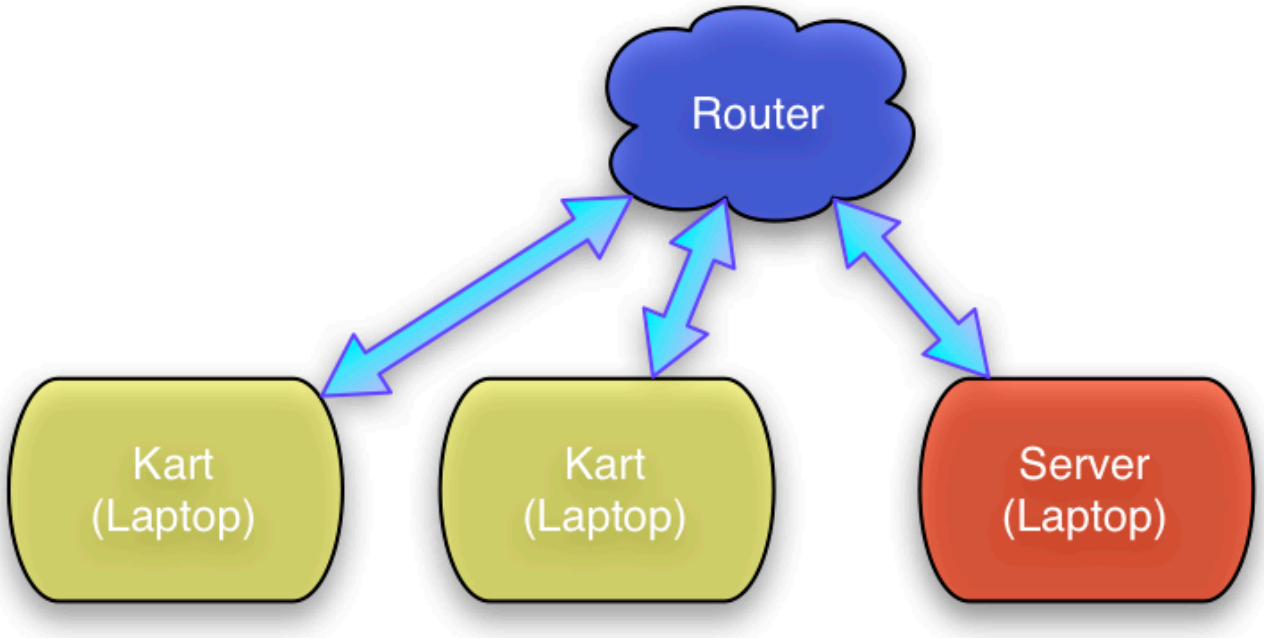


Figure X: Network Diagram

i. Background

In order for each device to communicate with each other, there has to be a certain protocol used. Initially the communication protocol that was considered to be used was the ZigBee Specification,

which uses the IEEE 802.15.4 standard. This standard specifies the physical layer, as well as media access control for low-rate wireless personal area networks. This entails that the protocol makes data communication seamless to transfer from device to device. Neither side, client nor server, would have to worry about addressing the individual layers in order to receive and decode the packets. We would just need a ZigBee device on the kart, as well as one on the main server. This would have been ideal for the initial designs, but since there were going to be computers mounted on the karts, it was decided that using a ZigBee device would just add another component to the system.

This protocol is used by the xBee 1mW chip antenna. It has a 2.4 gHz transmit frequency, but is bottlenecked by the uart connection on the chip. All that needs to be connected to the xBee is a 3.3 V power line, GND, and the DOUT and DIN pins for the UART connection.

Finally, a consensus was reached after some amount of research that each member of the team did. The xBee idea was scrapped, only to be replaced by the IEEE 802.11G standard. This lessened the complexity of the project, because it took out another hardware to software component that needed to be integrated.

It was not in the initial design to include a fully functional computer on the go-karts due to the power issue, as well as the bulkiness of placing an actual computer on the kart, even if it were to be a netbook. After further research and meetings, a consensus was made to include a fully functional computer. This allows the networking part of the project to work off of the regular computer's wireless card, and use the IEEE 802.11G standard. Luckily, the computers that were used were Apple's Macbook Pro line of laptops, so there was plenty of power as well as battery life for our implementation.

Once the wireless communication protocol was finalized, it was necessary to find a fitting design method to be used for the actual communication. UDP multicast was the initial implementation for that. After much research, it was apparent that much more work would need to be put in to broadcast a packet on all the open ports, as well as consume much more network traffic than necessary.

For the final implementation of this project, unicast UDP communcation was used. This way each device connects to the main server, and the server only sends messages to the devices that need to be told. One example would be if one physical kart connected, and one ghost kart connected, then it would just send packets to the physical kart instead of the ghost kart. This is a practical method because it not only reduces network traffic, but also decreases processing power of the server.

Our network infrastructure can support as many players as possible playing, but the game mechanics framework only supports 2 players at the moment. This can easily be changed in the code to handle more players.

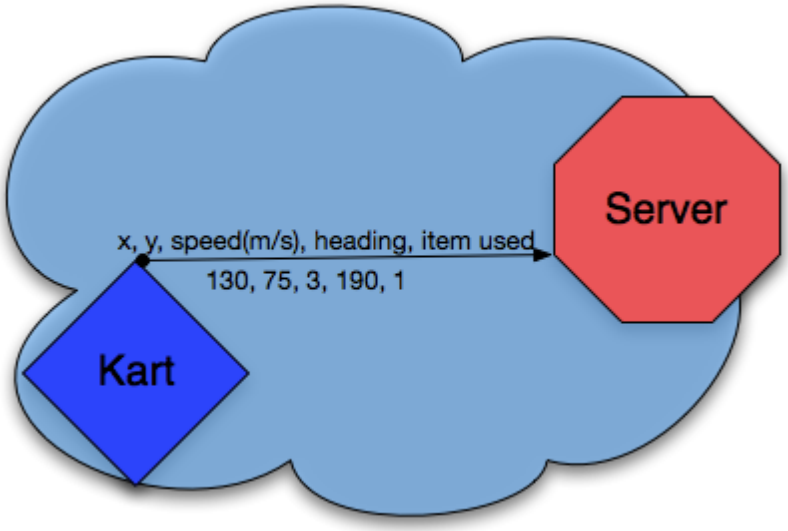
ii. Game Server

Since the game uses a client and server communication model, there needs to be specific responsibilities for the server. In the initial stages of development, the code was not modular, so it was almost like running a C program. Very procedural and not as efficent and easy to use as if it were ported to C++. After modularizing the code, a networking class was created. The user first has to initialize a new networking object by passing in what type of object that is trying to initialize, as well as what player number it is connecting to. In this case the server waits for the client, or kart's connection. Once

the kart sends an initial packet to the server, a connection has been made. This way there is seamless communication on the port number to where the kart has been initialized. Once all karts have been initialized, the game mechanics know that the game should start, and the game startup sequence begins.

The information that the packet contains varies from device to device. Because no image processing is done on the kart side due to its computational requirements, the server sends each kart information about what it should be overlaying. The kart then processes this information and places object images onto the screen in as described in the packet structures below. There is also other information that gets piggy-backed on each packet. One example of another packet item would be the max speed that the kart is allowed to run at.

iii. Game Kart(Client)



In the kart's networking side, the user creates a new networking object, just as it does for the Server type, but it's type now is Kart. The one caviat with this design is that the user must specify in code what the server's IP address is. This needs to happen because the user cannot just communicate with any other device, it must be the specific to the server to ensure unicast communication. Also in the initialization, the user must specify a player number. The port on which the connection is being made on is dependent on the player number. This ensures no two karts connect on the same port.

The initial connection packet that the kart sends is very arbitrary and doesn't need to contain any specific data. The main purpose that the initial packet is sent, is to let the server know the return information to where the return packet should be sent. The kart also sends various information to the server, such as it's own velocity, heading, and whether or not an item has been used. The karts sends this information to the server as soon as it processes the server information that gets received.

iv. Packet Structure

The packet structure for each respective device is as designed below:

a. Server to First-Person View (Padded with "zero" elements)

Format: H1,H2,H3,H4,H5,H6,H7,OC,O1T,O1x,O1y,O2T,O2x,O2y, ...

(HUD info)
H1: (uint8_t) Kart No. (1 or 2)
H2: (item_values_t) Current item (see items.h)
H3: (item_values_t) Current shield (STAR, REDSHELL, GREENSHELL, or NO_ITEM)
H4: (item_values_t) Shield count (0,1,2,3, or STAR)
H5: (uint8_t) Lap No. (0 to 3)
H6: (uint8_t) Place (1 or 2)
H7: (uint8_t) Max speed (for speed regulation, 0 to 100%)

(Object info)
OC: Unused

O1T: (item_values_t) Object 1 type
O1x: (uint8_t) x-coordinate (0.1m units)
O1y: (uint8_t) y-coordinate (0.1m units)

O2T: (item_values_t) Object 2 type
O2x: (uint8_t) x-coordinate
O2y: (uint8_t) y-coordinate
...

---

b. Server to Overhead View (NOT padded with "zero" elements)

Format: H,K1i,K2i,F,U,O1T,O1x,O1y,O2T,O2x,O2y, ...
OR
Format: H,K1i,K2i,F,U,O1C,O1T,O1x,O1y,O2C,O2T,O2x,O2y, ...

Example: "SO,1,3,A,0,1,50,50,3,100,100,6,75,25"

(Header)
H: (char*) "SO"

(HUD info)
K1i: (item_values_t) Kart 1 item
K2i: (item_values_t) Kart 2 item

(Format)
F: Format (determines what type of packet is received)
  A: All items in string. Clear list before adding each item.
  C: Changes only.

(Unused)
U: Unused

*For format A:
O1T: (item_values_t) Object 1 type
O1x: (uint8_t) x-coordinate (0.1m units)
O1y: (uint8_t) y-coordinate (0.1m units)

O2T: (item_values_t) Object 2 type
O2x: (uint8_t) x-coordinate
O1y: (uint8_t) y-coordinate

*For format C:
O1C: (char) Change (A: Add object, R: Remove object)
O1T: (item_values_t) Object 1 type
O1x: (uint8_t) x-coordinate (0.1m units)
O1y: (uint8_t) y-coordinate (0.1m units)

O2C: (char) Change (A, R)
O2T: (item_values_t) Object 2 type
O2x: (uint8_t) x-coordinate
O1y: (uint8_t) y-coordinate
...

---

c. Overhead View to Server (NOT padded with "zero" elements

Format: H,N,U,K1n,K1x,K1y,K1h,K1s,K1u, ...

Example: "O,2,0,1,123,102,197,0,1,2,54,67,349,0,0"

(Header)
H: (char) 'O'

(Overview)
N: (unsigned int) Number of karts

(Unused)
U: Unused

(Kart info)
K1n: (unsigned int) Kart No. (1 or 2)
K1x: (unsigned int) x-coordinate
K1y: (unsigned int) y-coordinate
K1h: (unsigned int) heading
K1s: (unsigned int) speed
K1u: (bool) use item
...

____

d. Server to RC Controller

Format: H,U,K1n,K1s,K2n,K2s, ...

Example: "SRC,0,1,0,2,1"

(Header)
H: (char*) "SRC"

(Unused)
U: Unused

(Kart info)
K1n: (unsigned int) Kart No. (1 or 2)
K1s: (unsigned int) Speed regulation (see config.h)

K2n: (unsigned int) Kart No. (1 or 2)
K2s: (unsigned int) Speed regulation
…

e. Pre-loaded Map Data (command-line parameter)

Format: H,U,Dc,Dr,OC,O1T,O1x,O1y,O2T,O2x,O2y, ...

(Header)
H: (char) 'M'

(Unused)
U: Unused

(Dimensions)
Dc: (unsigned int) Number of columns
Dr: (unsigned int) Number of rows

(Object info)
OC: (unsigned int) Object count

O1T: (item_values_t) Object 1 type
O1x: (uint8_t) x-coordinate (0.1m units)
O1y: (uint8_t) y-coordinate (0.1m units)

O2T: (item_values_t) Object 2 type
O2x: (uint8_t) x-coordinate
O2y: (uint8_t) y-coordinate
...

f. Server to 2D Map (NOT padded with "zero" elements)

Format: H,K1i,K2i,Dc,Dr,O1T,O1x,O1y,O2T,O2x,O2y, ...

Example: "S2D,1,3,200,250,1,50,50,3,100,100,6,75,25"

(Header)
H: (char*) "S2D"

(HUD info)
K1i: (item_values_t) Kart 1 item
K2i: (item_values_t) Kart 2 item

(Dimension)
Dc: (unsigned int) Number of columns
Dr: (unsigned int) Number of rows

O1T: (item_values_t) Object 1 type
O1x: (uint8_t) x-coordinate (0.1m units)

O1y: (uint8_t) y-coordinate (0.1m units)

O2T: (item_values_t) Object 2 type
O2x: (uint8_t) x-coordinate
O1y: (uint8_t) y-coordinate
…

g. Kart to Server

Format: X, Y, V, H, IU

Example: "120, 70, 6.5, 240, 1"

(X position)
X: (unsigned int) X position in relative game coordinates

(Y position)
Y: (unsigned int) Y position in relative game coordinates

(Velocity)
V: (unsigned long) Speed at which kart is traveling at(in m/s)

(Heading)
H: (unsigned int) Compass reading for where kart is heading(0-359)

(Item Used)

IU: (unsigned int) Item used

F. Positioning

Calculations for localization is done within a microcontroller attached to each go-kart, which then sends this processed information wirelessly back to the server. The kart's position determines whether or not item boxes have been obtained, where to place new items, and the position of used items.

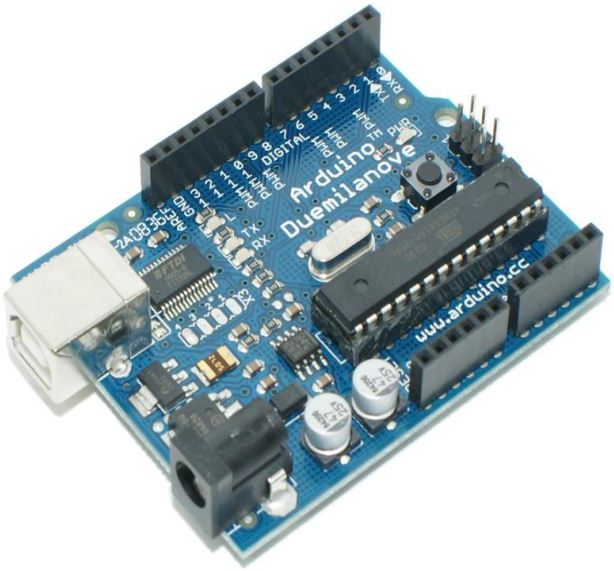


Figure 1. Arduino Uno board with ATmega328 microcontroller

Several methods were considered with all of fall quarter solely dedicated to researching these methods in our EE 523 class. The following sections give a brief overview of each method we researched.

i. Multilateration

Multilateration is widely used to accurately locate aircraft. It is very similar to trilateration, with one small difference in implementation. When using trilateration, transmit and receive times at the mobile and ground nodes are subtracted to compute the RF time of flight. In multilateration, the clock on the transmitting node is removed. Rather than determining a location using time-of-flight (ToF) measurements, time-difference-of-arrival (TDoA) is used instead.

In ToF calculations, distance is used to find locations. This creates intersections between circles or spheres at a radius of the computed distance. However, TDoA will record the arrival times of the RF signal. The arrival times found by each ground station is then compared. If they all record the same time, then the mobile node is an equal distance away from each ground station. Otherwise, the ground nodes with an earlier arrival time will be closer to the mobile node, and vice versa. In either case, the end result is that distances are found.

Because multilateration uses TDoA instead of ToF, adding extra ground stations to reduce errors is no longer a simple drag-and-drop solution, requiring changes to the algorithm for each added station.

ii. Triangulation

Another technique that can be achieved with RF transceivers is triangulation. However, unlike trilateration and multilateration, triangulation does not require more than two ground stations and is not timing critical. Rather, each ground station estimates the direction from which the mobile node's RF signal is approaching. By finding the signal's angle of arrival (AoA) at two separate ground stations, a line from each station can be extrapolated back to the source of the RF signal. The intersection of these two lines represents the approximate location of the mobile node.

As seen in Figure 4, the two stationary nodes are located on a common base line parallel to the horizontal axis. The direction towards the mobile node is represented by an angle from this base line. Along with this base line, the two traced rays form a triangle, which is where the name of the system is derived.

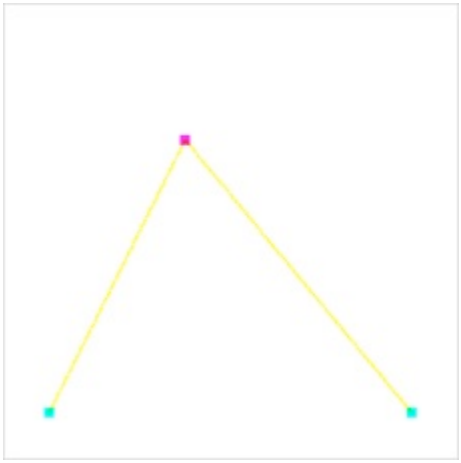


Figure 4. Example of Triangulation

As seen in Figure 4, the two stationary nodes (blue) are located on a common case line parallel to the horizontal axis. The direction towards the mobile node (red) is represented by an angle from this base line. Along with this base line, the two traced rays form a triangle, which is where the name of the system is derived.

iii. Global Positioning System (GPS)

Similar to multilateration, GPS locates a mobile node using multiple stationary nodes with known locations. However, rather than using ground stations, GPS employs a number of satellites orbiting the planet. With its twenty-four strategically placed satellites, any location on the planet theoretically has a direct line-of-sight to at least four of these stations. Thus, a 3-D coordinate can be assigned to any location on Earth. Instead of receiving the positioning signals, the satellites are configured to transmit them instead. The GPS receivers on our cellular devices and vehicles then receive these signals periodically and use them to determine latitude, longitude, altitude, heading, and other useful data. Because the ground receivers do not actually send their own signals, an infinite number of receivers can be used without interfering with each other.

Unfortunately, GPS systems suffer from multiple sources of error. Since the satellites are located in space, they slowly drift over time from their established orbits; although they are programmed to

periodically correct their position, this still causes a slight problem. A more severe problem is clock error both on the ground receiver and satellite ends. Commercially available receivers typically suffer from inadequate clocks (like in trilateration, clocks with GHz precision are expensive). On the other hand, the atomic clocks on each satellite suffer from small clock drifts, which cannot be corrected instantly.

iv. Differential-GPS (D-GPS)

Differential-GPS employs a stationary GPS receiver to determine the error timing errors and transmit correction signals to mobile receivers. Unfortunately, in order for D-GPS to work sufficiently, the object must be close by. Therefore, to cover large geographical areas, multiple ground stations must be placed for full coverage.

The ground node must know first know its exact position to determine timing errors. Normally, the time of flight and the velocity of an RF signal (speed of light) are used in the equation $distance = velocity \times time$ to determine a GPS receiver's distance from a satellite. With information from at least four satellites, the receiver's location can be found using trilateration. However, if the precise location of the receiver is already known, the equation can be used backwards ($time = distance/velocity$). The result is the expected time of flight, and can be subtracted from the measured time of flight to determine the timing error for each satellite transmission. By incorporating error correction into a mobile GPS receiver's calculations, errors are reduced substantially.

v. Dead Reckoning (DR)

Dead reckoning is a method that estimates the present position of an object by using its previous positions. While more advanced methods such as GPS tracking and triangulation have replaced DR, it is still used in aircraft and nautical navigation, autonomous navigation, and robotic positioning. The reason for this is because various elements of the object can be easily found: acceleration, orientation, or velocity. However, by solely relying on past positions to determine its present position, DR can only provide approximations. Also, error in past positions create accumulating problems for future positions. These errors can be due to sudden stops or turns, a miscue in calculating velocity or acceleration, or incorrect orientation values. Therefore, DR calls for precise measurements or frequent recalibration.

The advantage of DR is that unlike the other methods discussed, it does not require GPS satellites or ground stations. All is calculated on through the object and does not need further complex math. By eliminating the need for wireless communication and line-of-sight techniques, DR is an excellent indoor system solution and does not need expensive materials in order to work properly.

To better understand the process of constructing our final positioning system, Table 1 shows all of their advantages and disadvantages

System	Indoor?	Advantages	Disadvantages

Multilateration	Yes	Less clocks to synchronize Does not need to compute time taken from object to ground node	Extra ground stations are harder to incorporate Requires line-of-sight
Triangulation	Yes	Only two ground stations needed Basic calculations	Moving parts introduces mechanical breakdowns Signal strength measurements very susceptible to interference
GPS	No	Easy to access positioning data	Complex data harder to compute Error can be as high as 15m
D-GPS	No	Most GPS transceivers are D-GPS ready Reduces GPS error	Mobile receivers must be near base stations
DR	Yes	Does not require base stations Readily available parts Simple calculations	Incorrect data leads to accumulating error Requires precise measurements

Table 1. Lists pros and cons of all systems

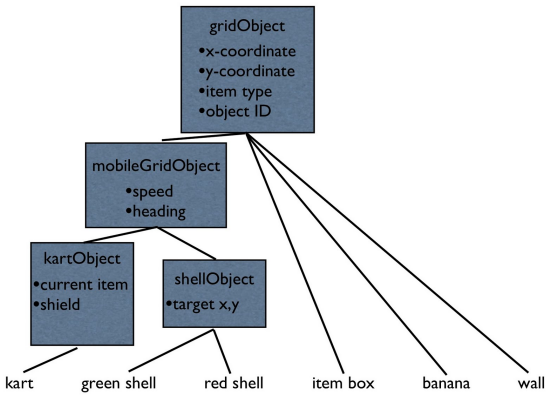
V. Build

A. Game Mechanics

i. Grid Objects

As explained in the Design section, each type of grid object has its own individual class. After some reevaluation, only the major objects such as karts need their own class; while traps may affect players in different ways, this is handled by the game model and is independent of their class representation. The class diagram below is representative of the final structure for the grid objects.

Grid Object Class Heirarchy

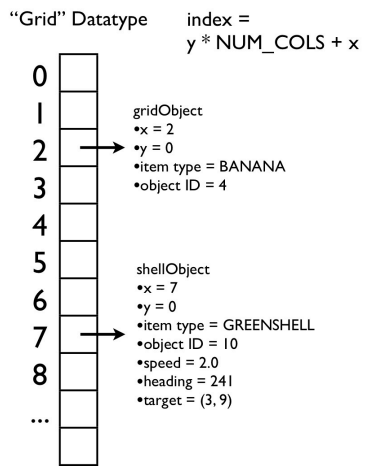


ii. Grid

Originally, the grid was represented as a 2D array of pointers to grid objects (or NULL if a coordinate is unoccupied. However, this approach proved unnecessary, as a 1D array would suffice. The x and y-coordinate simply mapped to a single index using the equation below:

$$index = y*NUM\_COLS + x$$
$$0 < x < NUM\_COLS,\ 0 < y < NUM\_ROWS$$

Similar to a hash function, multiple combinations of x and y values can yield the same index, but by limiting these inputs to fall within the ranges above, every valid pair of x and y-coordinates yields a unique index.



Along with the static 1D pointer array, the grid class also uses another data structure in parallel. A linked list of pointers contains the addresses of each object in the grid. This linked list allows quicker traversal through the grid objects since the 1D array is much larger and contains several NULL pointers.

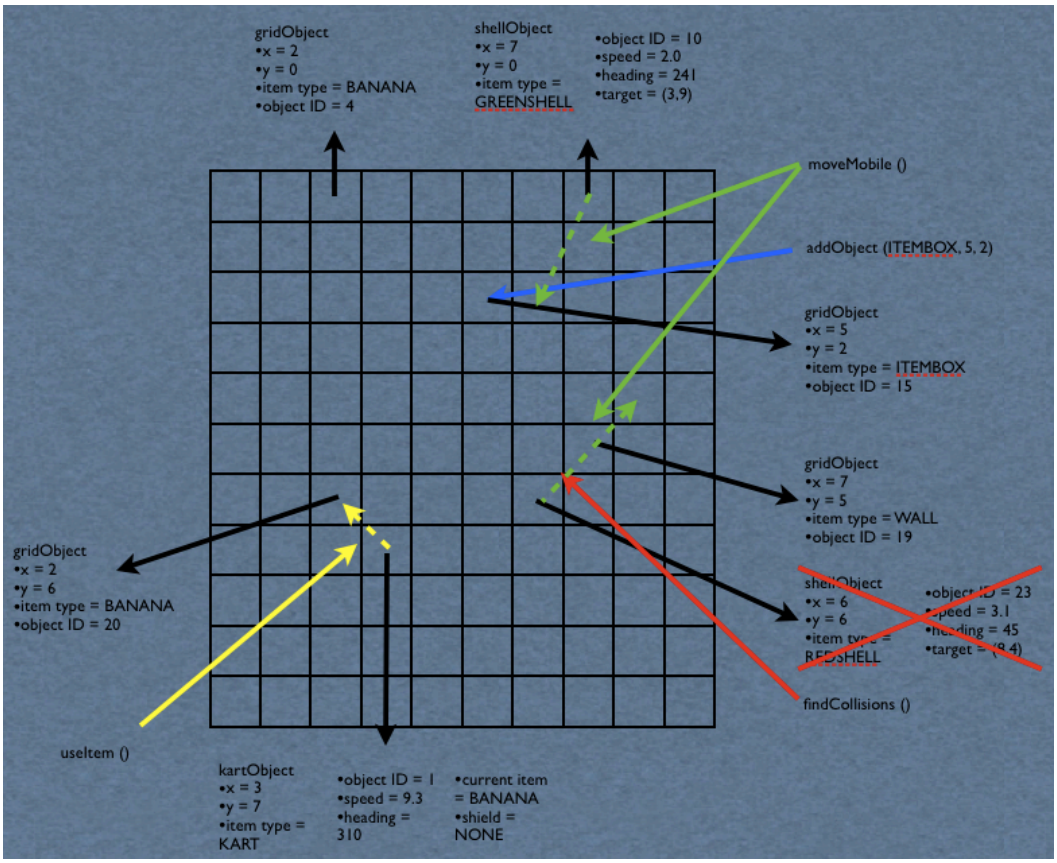
In conclusion, the 1D array may have been unnecessary, as everything can be accomplished using the linked list alone. Additionally, using two parallel data structures requires extra care when an object is removed, as it must be removed from both structures while only de-allocating memory once.

iii. Game Model

The game model manipulates the grid data structure by moving objects to new coordinates and adding or removing objects when necessary. Because every grid object has a method to determine its distance from any other object, the model can easily detect collisions by comparing these distances to a preset threshold. As explained in the Design section, each grid object has a preset strength value, and these different values inherently create a hierarchy when dealing with collisions. However, because there are very few "special" cases, if-else statements are used instead. The object strength parameter, while unused, remains present in case they become useful in future revisions.

Other features discussed in the Design section are not available in the current implementation. For example, there is currently no way to limit which items a kart can receive from an item box. This feature is not included because without a way to automatically detect the place or lap number of either kart, such an unfair advantage should not be given to either player.

The move_mobile function runs into a few problems depending on how much time has elapsed since its last call; since the coordinate system and compass headings are both stored as integers, the calculations may be rounded such that a shell traveling only a few degrees off of the horizontal axis may in fact appear to travel directly along that axis indefinitely.



iv. Controller

Because the grid data structure is stored in the game model, it generates all the necessary data packets directly and passes them along to the controller. The controller is responsible for creating the socket connections via the networking class and sending the correct packets to each device.

The most important functionality added to the controller is a graphical user interface. While the original server required the user to manually type in coordinates for each object to be added, the new GUI displays all objects on a properly sized window, and objects can be added by clicking on the desired grid location. The GUI also provides a feature to change the status of an individual kart (speed, lap number, place, current item, etc.).

B. Hardware Mounting

For mounting, we wanted to be sure that each kart is able to function normally as well as make sure that each piece of additional hardware did not interfere with driving experience. For safety reasons, we also wanted to be sure that the driver pays attention to the road at all times. Also, we wanted to have all parts safely and securely fastened onto the kart, this was specifically targeted at the MacBook Pro's.

i. Equipment and Materials Used

1	power drill
1	hex key
2	3’x3’ Aluminum Sheet Metal
40	small fitting screws, nuts, and washers
4	⅜ “ Threaded Rods
12	⅜” Nuts , washers, and lock washers
2	L-brackets
4	rectangular brackets
2	small glad-ware boxes
12	banana-to-banana cables
12	banana cable binding posts
12	aligator clips
1	roll of electrical tape
1	10”x10” sheet of rubber
1	epoxy glue
1	5’ roll of velcro

ii. The Foundation

Since we wanted the drivers to pay attention to the road at all times, the most logical place to put the laptop is in front of the steering wheel. But before a laptop can be situated, a strong foundation had to be implemented. This foundation had to be very stable, but also be light enough to not complement the kart’s speed.

The Honda MiniMoto originally had alot of external plastic coverings at the front bumper and side fenders. As we stripped the kart down from these plastic pieces there were two metal cyndrilical holes at the front bumper. Two ⅜ threaded steel poles were inserted into these holes. The inner diameter of the holes was slightly bigger than the pole’s outer diameter so screws were drilled at the base of the holes and poles for added security. Nuts and lock washers were also applied to add extra tension to these poles.

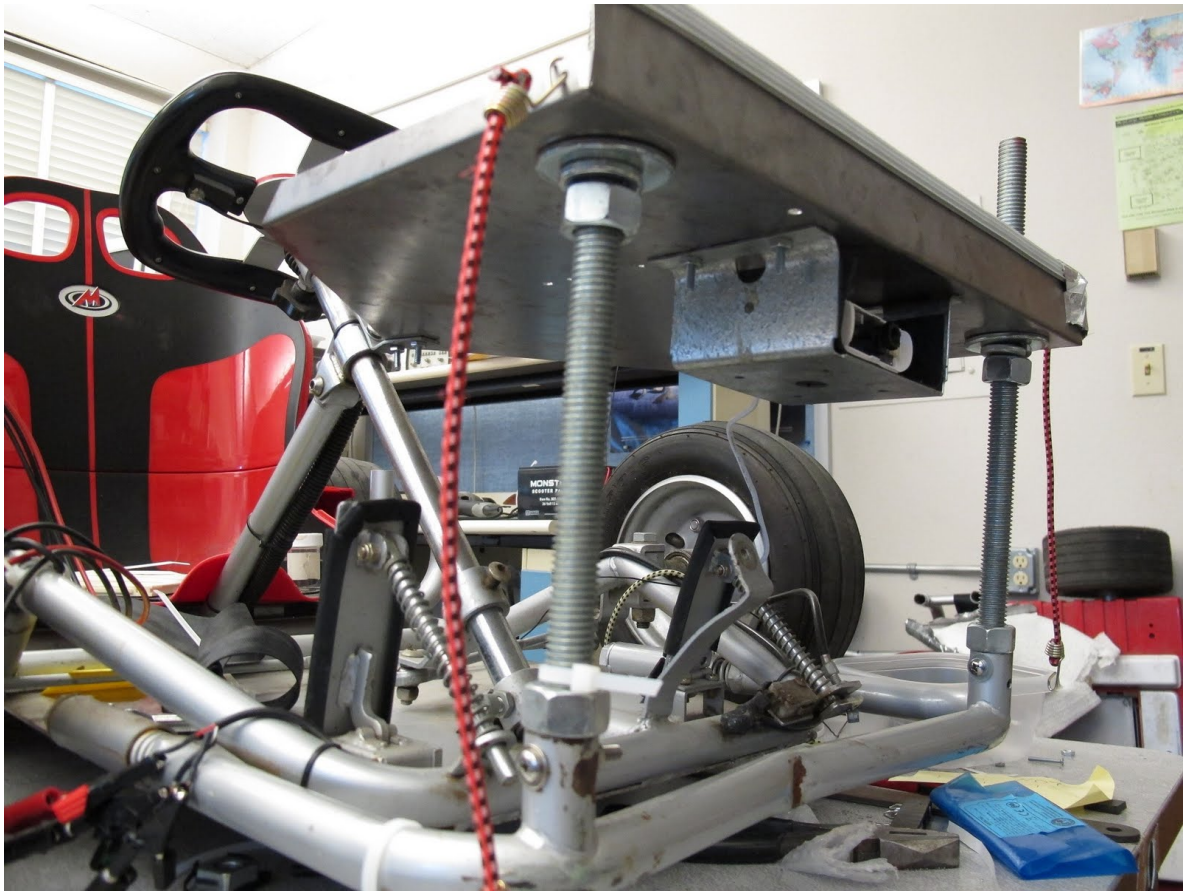


Figure X :The foundation of the mount

iii. Laptop Mounting

As more plastic coverings were stripped off, there was a small threaded area near the steering wheel that could fit something as small as a screw. An L- bracket was placed at this hole and this bracket and the two holes will be able to hold the laptop.

Sheet metal was cut and manufactured at the Industrial Manufacturing Engineering department at Cal Poly. Kevin Williams, an IME professor, assisted in manufacturing the sheet metal base. The sheet metal had to be cut, and folded at the edges. Holes had to also be cut, so that the two threaded rods can go through. Below is a picture of the final design.

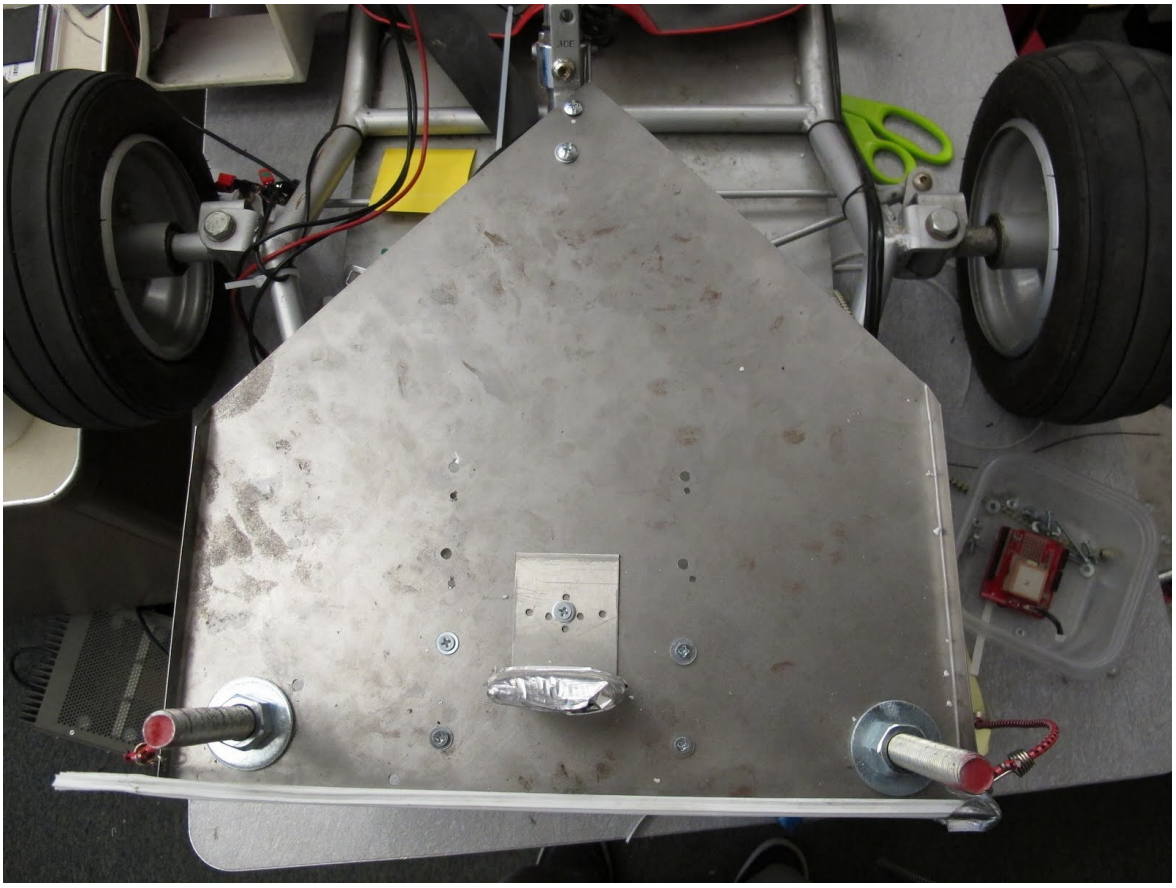


Figure x: An overview image of the laptop mount design

Once the sheet metal was manufactured, a nut and washer was screwed onto the rod. We required that the washer was placed horizontal with the L-bracket. The sheet metal base was then placed onto this foundation and tightly fastened on with screws, nuts, and washers.

iv. Camera and Micro-Controller Mounting

Initially, the Camera and micro-controllers were placed at the bottom of the metal sheet via two rectangular brackets. These metal brackets were custom bent at the edges so that these brackets can be screwed onto the laptop mount. This design securely fastened the camera and micro-controllers, but the design was not modular. Each connection from the micro-controller was soldered on and there is a huge chance of loose of connections. It was also very difficult to take out the AVR chip if there was speed regulation issues. Also, the design lead to common ground issues. The camera mount worked fine but the micro-controller mount needed a complete redesign.

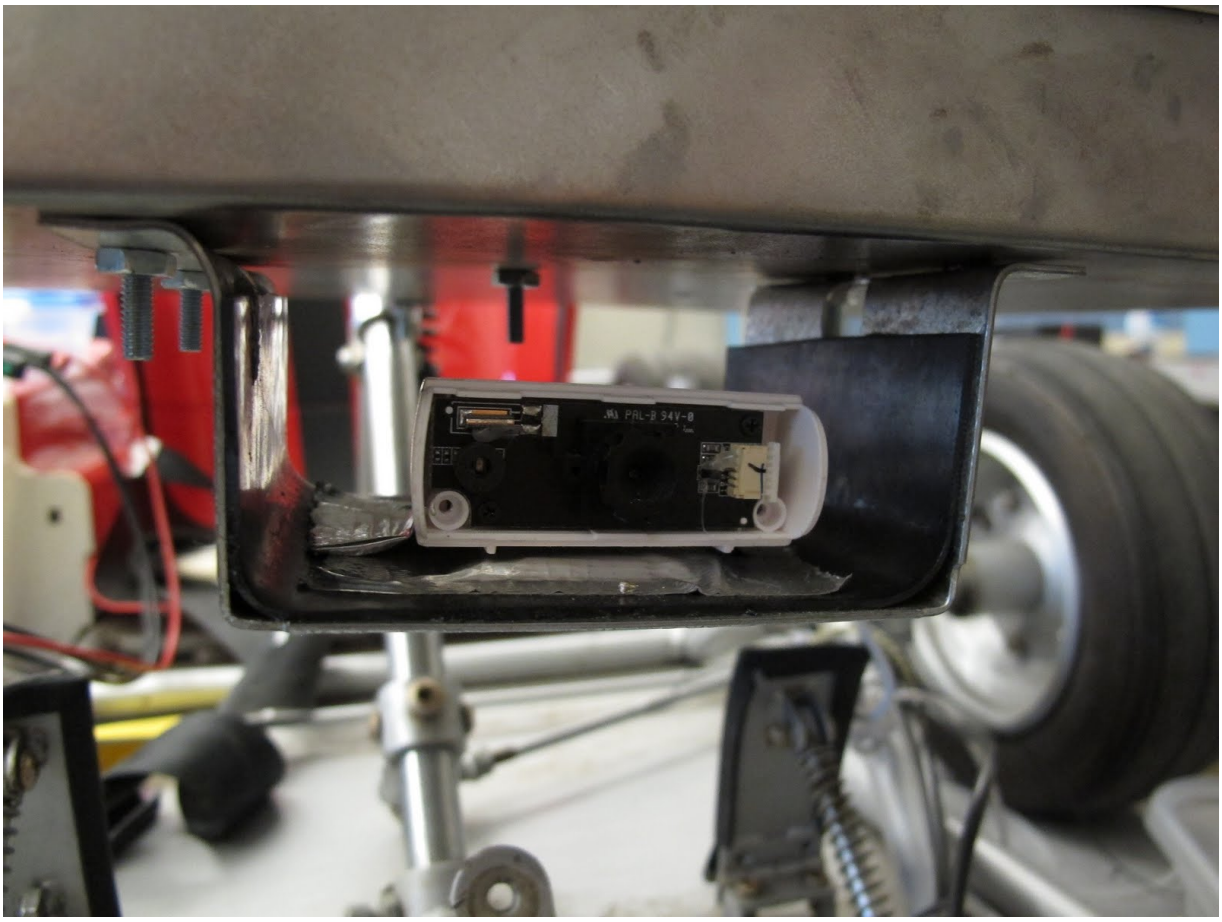


Figure X: The camera counted on the camera mount.

For revision 2, a new design was developed for the micro-controller mounting. Both micro-controllers will now be placed inside glad ware. Glad ware will solve the common ground issue because the container is made of plastic. Banana safety posts were drilled in to the glad ware. There was a total of 6 connections:

- Two Vcc connections
- Two ground connections
- One throttle connection
- One motor controller connection

These connections will be tied to the corresponding connections from the go-kart via alligator clips. Too prevent further ground issues, electrical duct tape was wrapped around the metal casing of the clips. These banana cables were secured to kart via zip ties. Note that there are two vcc and ground connections because the ATmega is being passed in between the throttle and kart controller from the MiniMoto. The glad ware was then velcro-ed onto the plastic side fender of the go-kart. The figures below shows the redesign of the micro-controller mount.

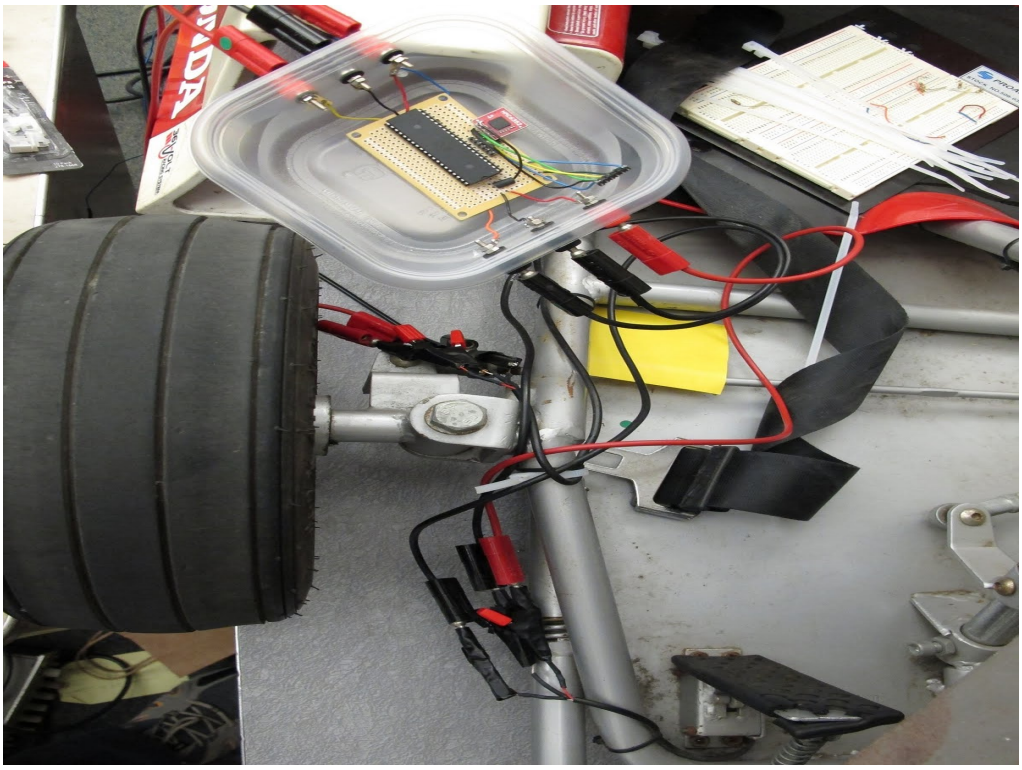


Figure X: Revision 2 Micro-Controller mount

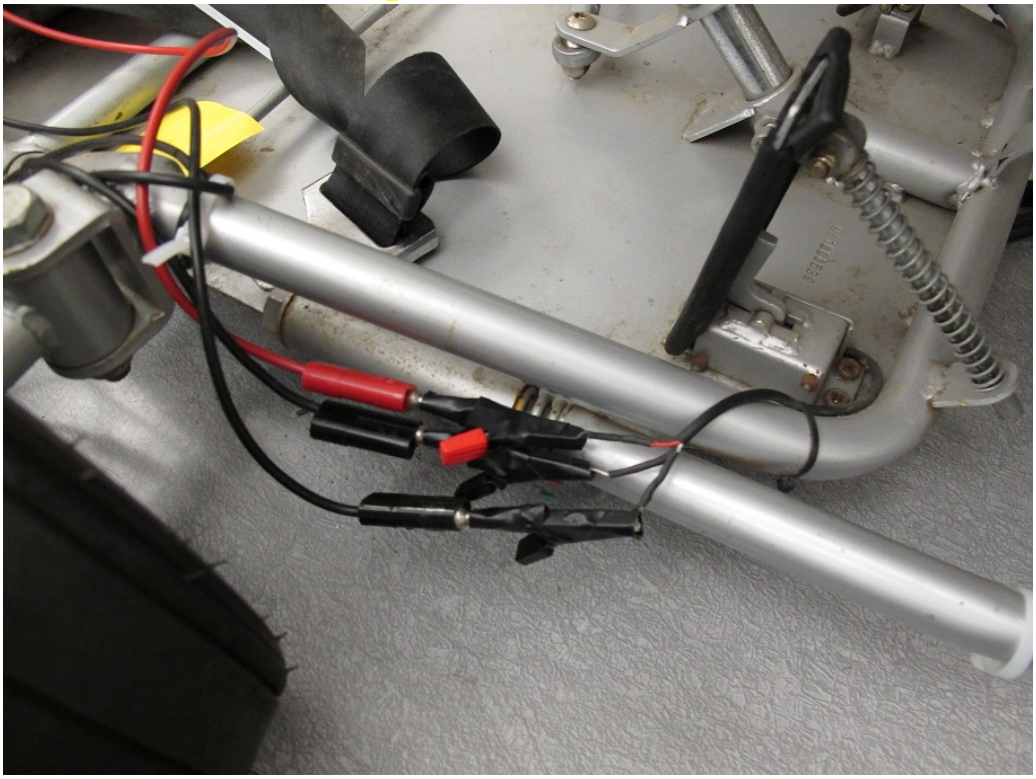


Figure X: Revision 2 Wire Connections

C. Image Overlay

i. OpenGL Setup

Some initialization of opengl is first required before we can show an opengl window. This intialization can be seen in main.cpp with a simple function call to InitializeOpenGL(int argc, char **argv). For further understanding of what this function does refer to the code and read the comments or read the tutorials on opengl in the bibliography section. The two important functions does set opengl to constant animate are

```
glutIdleFunc(glutDisplay);
glutMainLoop();
```

The first function tells OpenGL that the passed in function should be called everytime opengl is idle and the second functions tells OpenGL to loop constantly. The display function is passed so that the animation is running in real time with the race.

ii. Pulling Camera Frames

Before frames can be pulled from the camera, OpenCV has to initialize the camera by making sure there is one available. Once the camera is intialized frames can be pulled from the camera to be displayed onto the screen. The code for initializing and pulling frames from the camera can be seen below.

```
/*initialize camera*/
CVCaptur *capture = cvCaptureFromCAM(cam);
  if(!capture)
  {
    fprintf(stderr, "Unable to initialize webbcam at %d!\n", cam);
  }
/*pull frames*/
IplImage *qFrame = cvQueryFrame(capture);
```

Since the VideoMan library was discarded we had to find another way of displaying the pulled frames from the camera as the background in OpenGL. To set the background as the camera stream we placed a polygon in the background that is textured with the current pulled frame. This can be seen in the hud.cpp class in the function called display. The polygon is offseted at a far distant to prevent any 3D animation from colliding.

iii. Adding HUD Items to Camera Frame

Since OpenCV allows the manipulation of an image's pixels, we were able to place hud items on the existing camera frame by focusing on a small area of the frame and changing the pixels with the hud item's pixel we wanted in that area. Instead of having a seperate layer for the camera frames, the hud layer and camera layer were combined into one layer for simplicity. A function call in the class kartHud.cpp was created to place these hud items onto the current camera frame with some passed in

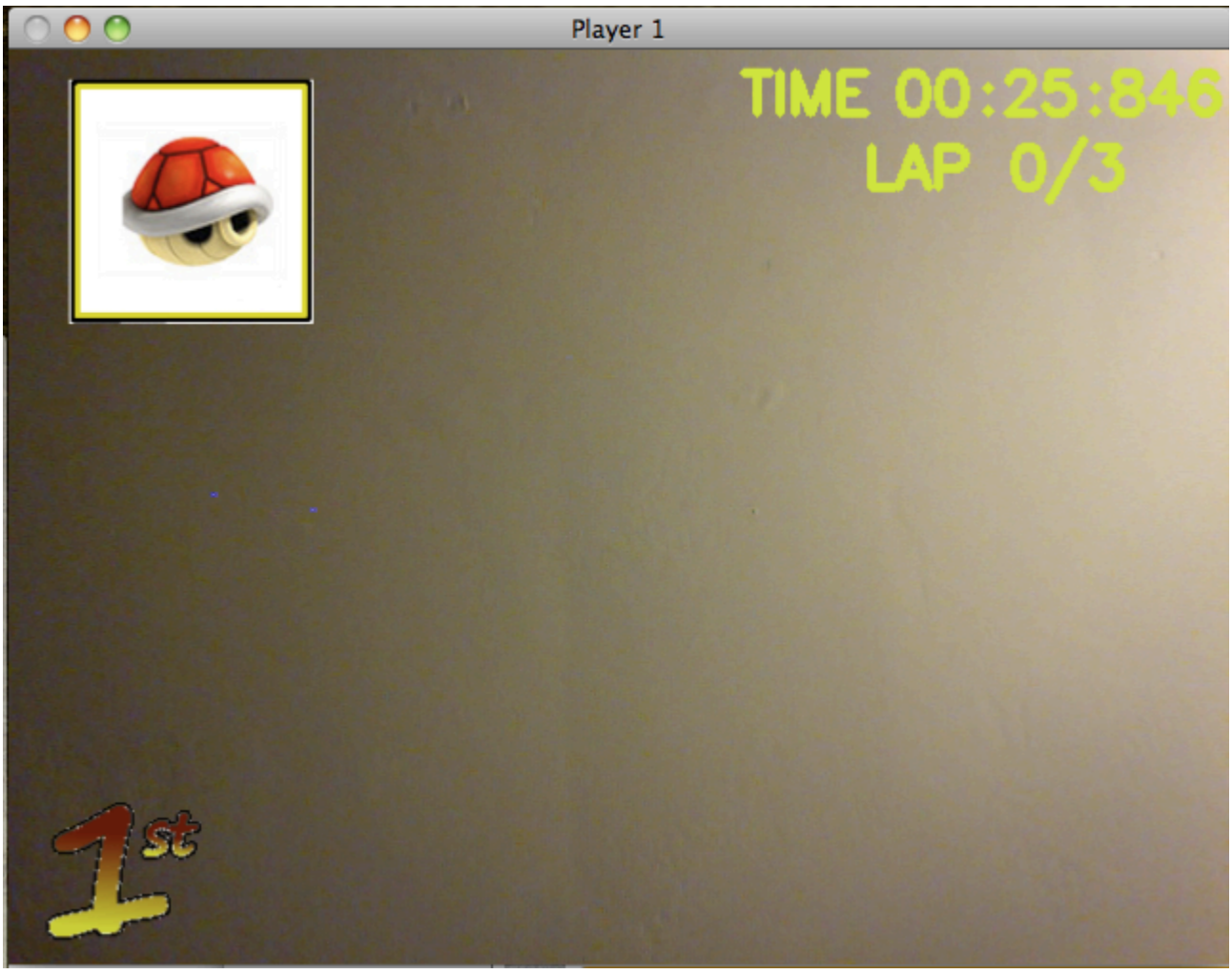
parameters of where the image should be place. The function call and an example of how to use it can be seen below:

```
void kartHud::placeImage(IplImage *background, IplImage *foreground,
      double size, double xoffset, double yoffset)
```

If we wanted to place a red shell hud item on the top left of the current frame the function call would look like the following:

```
original = cvLoadImage("./hudImages/redshell.jpg");
 placeImage(currFrame, original, 20, 5, 70);
```

Here is an example of the red shell hud item being placed on the current frame:



iv. Placing 3D Objects into the View

In order to show some advanced 3D models we first had to find a way to load the vertices of a .obj file into OpenGL and then with that try to recreate the 3D model of the .obj file. Thankfully there was a free open source library that enables us to do this. The library classes that enables can be seen in the appendix section with the names glm.cpp, glm.h, glimg.cpp, and texture.h. With this library I've broken down each 3D object that needs to be display with a draw function that takes in coordinates. The coordinates passed into this function are relative to the kart's current position and game world's coordinates. In order to draw a red shell that is two units in front of us and one units to left we call the function drawRShell(-1, 2). The example display of this 3D object can be seen below.

In order to keep track of all the 3D objects that needed to be displayed on the screen an array of MAXOBJECTS was first initialized in our kartObjLayer.cpp class. This array would hold information of what type of item needed to be displayed and what the coordinates of that item were. An example struct of what the array held can be seen below.

```
typedef struct
  {
  /* x and y coordinates of an object is relative to the
```

```
    * objects position. where an increase in y is the increase
    * in forward distance of the object and increase in x would
    * increase an object to the right of the kart
    */
      double x;
      double y;
      item_values_t type;
   }myCoords;
```

Whenever another class would need to add an object in the view all the class had to do was call the int addObj(double x, double y, item_values_t type) function. This function would return a key which can be used to modify the object's position or value.

D. Speed Regulation

Before displaying the actual source code for this section, I would like to describe each function used. The code was written in AVR Studio and the Arduino software. Equipment used in this section include an Atmel ATmega32 micro-controller and an Arduino UNO. Also, the picture below shows the I/O pinout of the ATmega32 for revision 2.

Rev. 2 ATmega32 I/O Pinouts

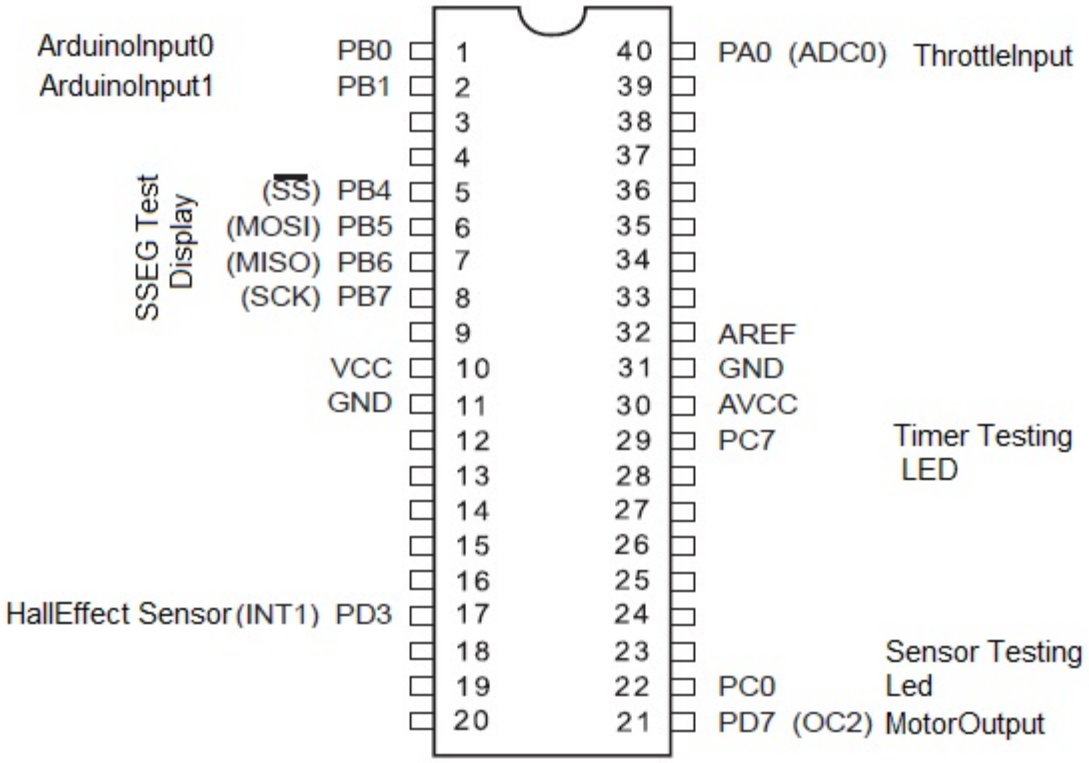


Figure X: ATmega32 pinout for revision 2

i. readADC()

Reads a digital value from the throttle. This value is a value from 0-255.

```
uint8_t readADC()
{
      ADCSRA |= _BV(ADSC); //start AD conversion
      while (!(ADCSRA & (1<<ADIF))) {}; //wait for conversion to
complete

      return(ADCH);
}
```

Figure X: readADC() source code

ii. speedChange(uint8_t aData,uint8_t mode)

Controls the speed of the kart based on the value passed in from the arduino. This function takes in two parameters: *aData* is the ADC value which was read from readADC(), and *mode* corresponds to the specified speed mode. The function returns a newly modified integer.

Basically, the duty cycle of the kart will be set to a certain threshold for partcular speeds. For example, if 75% speed is selected, the max possible duty cycle is “129”. Similarly, If 0% speed is selected, the max possible duty cycle is “0”.

```
uint8_t speedChange(uint8_t aData, uint8_t mode){
      uint8_t newADC;

      switch (mode){
            case 0:
                  newADC = aData;
                  break;
            case 1:
                  if (aData > 129){
                        newADC = 129;
                  }
                  else {
                        newADC= aData;
                  }
                  break;
```

```
            case 2:
                  if (aData >128){
                        newADC = 128;
                  }
                  else {
                        newADC = aData;
                  }
                  break;

            case 3:
                  newADC = 0;
                  break;
            default:
                  newADC = 0;
                  break;
      }
      return (newADC);
}
```

Figure X: speedChange() source code

iii. setPWM(uint8_t duty)

This function changes the output analog voltage. Remember, adjusting the duty cycle of a pulse-width modulated signal adjusts the voltage of a signal. The function takes in a duty cycle and will adjust the PWM signal accordingly.

```
void setPWM(uint8_t duty){
   OCR2= duty;
}
```

Figure X: setPWM() source code

Depending on the PWM duty cycle returned from speedChange(), this function will output a relative “analog” voltage.

iv. Revision 2 Speed Regulation Design

In the second design the wheel encoder code was moved to the ATmega MCU because a control loop was to be implemented for the kart. However, this closed loop system was not completed by the time of the demo.

The control system would basically check for speed errors, and regulate the speed based on the analog signal coming from the throttle. The error gain would adjust the PWM signal until the actual velocity equals that of the throttle value and/or speed threshold (which was set by the server). For testing

purposes, a seven segment display was used to display the velocity of the kart in meters per second.

```
/* wheelEncoder()
 *calculates the velocity of the wheel
 */
void wheelEncoder(){
      float ratio;

      newCount = count;

      countDiff = newCount-oldCount;
      oldCount = newCount;

      ratio = ((float)countDiff/(float)TICKS)*(float)CIRCUMFERENCE;
      velocity = ratio/TIME;
}

/*transfer()
 *Transfers the data to the seven segment Display. The display
 *shows the value in this form [XX.XX] where X is an integer.
 */
void transfer(){
      int tens;
      int ones;
      int tenths;
      int hundredths;

      tens    = ((int)(velocity))/10%10;
      ones    = (int)velocity%10;
      tenths    = ((int)(velocity*10))%10;
      hundredths= ((int)(velocity*100))%10;

      SSEG_Write_digit(1,tens);
      SSEG_Write_digit(2,ones);
      SSEG_Write_digit(3,tenths);
      SSEG_Write_digit(4,hundredths);
      SSEG_Write_Decimal_Point(SSEG_DP_1);
}
```

Figure X: Revision 2 wheelEncoder() source code

SSEG_Write_digit(x,y) is a function called from spi_sseg.h, a driver created in CPE 439 for an SPI-based seven segment display. Note that the Wheel encoder code was the same code used for the IR sensor. However, TICKS was now defined as 8 because there are only 8 magnets on the wheel of the kart.

E. Networking

i. General Information

The networking aspect for the project is fairly seamless, with the user only having to call one connect function per kart playing.

ii. Game Server

When the constructor is called with the certain parameters that allow it to create a new server object, it calls the constructor, which invokes the **server_setup()** method if the correct parameter was entered. NETWORK_SERVER is a macro that is just the number 2, that is used as the correct parameter in the constructor that determines if the server is going to be initialized.

a. server_setup()

This method calls the necessary commands to help set up the game server. It first calls socket(), which returns a socket descriptor. The socket descriptor is used as the main channel of information for how communication is transferred. Because the UDP protocol is being used, SOCK_DGRAM and IPPROTO_UDP gets passed as the 2nd and 3rd input variable, respectively. Also, it changes the socket address info's family to AF_INET to represent the address family for internet sockets. Once that initialization finishes, then the socket gets binded to the address structure, and server_setup() prints out the port at which it is waiting for a kart to connect on.

Once the user creates a new networking server object, the **connect()** method should then be called.

b. void connect(void)

This method allocates memory for a new packet using the malloc() function. It then waits on recvfrom() until the specified kart connects to it. Once the kart sends the packet and connects with the server, it prints a message on the screen that a kart has connected, and then sends back a simple string "Connected to Server!" to the kart. After that it just frees the packet and exits.

iii. Kart's Networking Specifications

When the constructor is called with the certain parameters that allow it to create a new kart object, it calls the **kart_setup()** method.

a. kart_setup()

This method is used when initalizing the network communication for each kart object. This method is much shorter than the server_setup because it does not bind the port. It calls socket() to return a socket descriptor. It also uses the object's own socket address information structure. It addresses the same AF_INET protocol, and makes sure that the game is playing on the correct port. One variant that it does that the server does not, is address a specific IP address for the server. This is to ensure that data is being sent to that direct IP, and not any others.

Once that method is called, the kart must send an initial packet to connect with the server

iv. Dual Network Commands

a. int recv_data(char *buffer)

This is the wrapper method to handle seamless receiving of data. It just uses the class information being the socket and the socket address structures, and receives data on the line. Because it does not use the MSG_DONTWAIT parameter, it does hangs on receive of data. This means that if no new data gets transferred, then the game mechanics cannot update. Luckily, since it is running in a separate thread, the game will not crash due to dependencies on other processes.

The message gets sent into a buffer that must already be allocated. If the buffer is not allocated beforehand, then bogus data may come in. It also receives MAX_PACKET_SIZE, which is 150 bytes long. This size is optimal for the implementation because it is not too long that it congests the network, but not short enough to where data gets shortened due to restrictions.

b. int send_data(char *buffer)

This is the wrapper method to handle seamless sending of data. It sends the buffer data on the socket and port that it is respectively connected to. The user must handle all of the buffer sizes, and when sending, the strlen of the buffer gets passed into the parameter that specifies the length of the data to send.

F. Positioning

i. Constructing a Working Positioning System

a. Ground Nodes for Multilateration and Triangulation

Initially, multilateration was considered because unlike triangulation, its lack of moving parts made it easier to construct. We purchased four 434MHz transmitters and receivers in order to create three ground nodes determining the location of a single kart. Figures 4 and 5 depict these devices.

Figure 5. RF Link 2400bps 434MHz Receiver

Figure 6. RF Link 2400bps 434MHz Transmitter

As an initial test, we connected each device to two different Arduinos and separated them at a reasonable distance while still being within line-of-sight. Figures 7 and 8 show the setup for each device.

INSERT FIGURES OF TRANSMITTER AND RECEIVER

I. Setting Up the Transmitter and Receiver

A sample code found through Spark Fun Electronics was used.

```
/* * Simple Transmitter Code
  * This code simply counts up to 255
  * over and over
  * (TX out of Arduino is Digital Pin 1)
  */
 byte counter;
 void setup(){
      //2400 baud for the 434 model
      Serial.begin(2400);
      counter = 0;
 }
 void loop(){
 //send out to transmitter
 Serial.print(counter);
 counter++;
 delay(10);
```

```
}

/* * Simple Receiver Code
 * (TX out of Arduino is Digital Pin 1)
 * (RX into Arduino is Digital Pin 0)
 */
int incomingByte = 0;
void setup(){
     //2400 baud for the 434 model
     Serial.begin(2400);
}
void loop(){
     // read in values, debug to computer
     if (Serial.available() > 0) {
        incomingByte = Serial.read(); Serial.println(incomingByte,
        DEC);
           }
           incomingByte = 0;
        }
```

As seen above, a simple continuous counter is being sent out through the transmitter and picked up by the receiver, which does nothing more than read what the transmitter has sent. After running each code of through the transmitter and receiver, we picked up nothing but noise. Sufficient and helpful could not be obtained. We assumed that the devices were simply too far apart. To fix this, we put them as close as possible and added wires as antennas to increase the likelihood of receiving transmitted data. Unfortunately, this method did not help. Since time was a factor and days were wasted testing and re-testing, we decided to scrap both triangulation and multilateration. We could have bought new parts and tried again but the cost would be more and we may end up with the same problem of receiving only noise.

b. GPS Coordinates For Localization

To avoid this, we moved on to the next method of localizing GPS-coordinates into local coordinates for our outdoor demonstration. The concept seemed doable in a short amount of time and we already had the components, an Arduino board and the EM-406A GPS Module.

Figure 9. EM-406A Receiver with Antenna from Spark Fun Electronics
INCLUDE DEFINITION/EXAMPLES OF NMEA AND DD

I. Converting NMEA to Decimal Degrees (DD)

Right away we encountered a problem from the GPS module: it only outputted in NMEA format. NMEA, or National Marine Electronics Association, format is widely used for real time position information. In this case, it meant a conversion to decimal degrees in order to find distance in meters. To calculate distance from two DD points is as follows:

$$point\ A = (lon1, lat1)$$
$$point\ B = (lon2, lat2)$$
$$radius = 6378137$$
$$t1 = sin^2(\frac{lat1 - lat2}{2})$$
$$t2 = \cos(lon1) * \cos(lon2)$$
$$t3 = sin^2(\frac{lon1 - lon2}{2})$$
$$t4 = t1 + t2 * t3$$
$$distance = 2 * radius * \operatorname{atan}(\frac{\sqrt{t4}}{\sqrt{1 - t4}})$$

The DD points are expressed in radians and radius equaling the radius of the Earth in meters. This formula worked fine and always gave accurate results in relative to the coordinates given from the EM-406A. Unfortunately it was the module itself that gave us problems. Thorough testing of the module revealed that it must in wide-open spaces away from any obstruction to avoid bad data and even so, the data needed constant recalibration in order to stay up-to-date with the several orbiting satellites. We also could not select the satellites we wanted to use to triangulate coordinates with the EM-406A. A module with this capability cost hundreds of dollars, not within our budget. Also, even as a localized coordinate, it could not keep up with the speed of the kart, even when tested at low speeds.
With the issue of the GPS module, it made it impossible for localizing coordinates as well as using D-GPS unless we spent beyond our budget to obtain better GPS data.

As a last-ditch effort to create a working positioning system with parts we already had, we combined odometry with compass headings to create a dead reckoning system.

c. Localization by Odometry

I. Wheel Encoding

Speed was found using odometry, specifically, wheel encoding. Wheel encoders can provide data for odometry and will is the simplest way of obtaining localized coordinates for our track and go-karts. A sensor determines the speed by observing changes specific to that sensor as the wheel of the go-kart rotate. For instance, an optical encoder observes the change in light as the wheel spins. This is accomplished by alternating black and white to maximize and minimize the amount of light the optical sensor catches. Figure 2 shows an example of the alternating color system attached to the wheel itself.

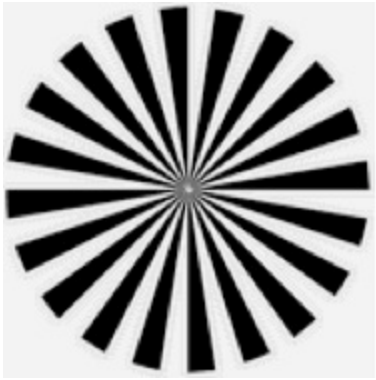


Figure 2. Black and white color wheel

As the wheel rotates, the sensor reads a series of highs and lows (or 0's and 1's) at varying speeds. This is found by using a counter that will keep track of the changes as the sensor toggles from low to high or vice versa. After a specified amount of time, for example one second, a calculation from the counter finds velocity through the simple equation velocity = distance/time. Distance in this case refers to the circumference of the wheel and a specific number of toggles the sensor reads determines if one full rotation has been accomplished.

To create thresholds, the LM339 comparator will be used to output either high or low, depending on what the sensor reads. A sample circuit is shown in Figure 3.

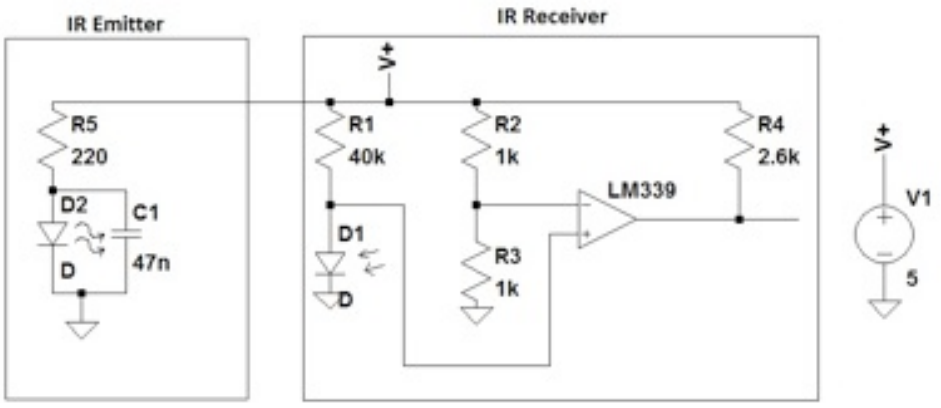


Figure 3. Comparator circuit for encoder
INSERT ACTUAL WHEEL ENCODER CIRCUIT

High and low values for the IR sensor and the eventual hall effect sensor will be read from the wheel as black and white pictures or evenly placed magnets, respectively. Each change in color or pole represents a tick. The formula to obtain velocity from number of ticks is as follows:

$$velocity = \frac{change\ in\ ticks * circumference\ of\ wheel}{number\ of\ ticks * change\ in\ time}$$

II. Determining Orientation

The compass values were found by using its I2C interface on the Arduino Uno.

Figure 8. Compass Module - HMC6352

Heading is important in dead reckoning because it allows us to determine whether an object is going forward, reverse, or turning left or right. Using Honeywell's HMC6352 Compass Module eliminates a lot of design work in actually creating a heading device and allows us to focus on its actual function: obtaining a correct heading. For best results, it must be mounted in such a way that it remains still. This will eliminate errors in heading values and reduces the need for recalibration.

To determine local coordinates for the go-karts using speed and heading, the following equations are used:

$$distance\ travelled = velocity * \frac{time\ elapsed}{1000}$$

$$x = previous\ x + \cos\left(\frac{heading * \pi}{180}\right) * distance\ travelled$$

$$y = previous\ y - \sin\left(\frac{heading * \pi}{180}\right) * distance\ travelled$$

ii. Test Plan

To ensure working functionality of each component of the dead reckoning system, the plan is to test each part individually.

The sensor must first be able output a high and low value—high equaling the supply voltage and low equal to ground. With a working comparator, testing the encoder will require a spinning wheel. Having the go-kart elevated so it does not travel forward as the wheel spins will test that the wheel encoder can both calculate correct speed values as well as do it in real time. The compass values will be found by using its I2C interface on the Arduino Uno and can simply be found by shifting the direction of the compass by hand.

Since both parts do not interact or interfere with each other, integration is simple and the values are only needed for calculation of the coordinates. Both outputs of the compass and encoder will be sent into the Arduino Uno where the coordinates are calculated and sent out to the server.

To fully test the positioning system, they must both be connected to the go-kart and the server will determine if correct positions are found as it traverses through the track.

iii. DR Testing

Each component was tested individually. The optical sensor sent values through a comparator and was inputted into the Arduino's digital pin 4. A counter kept track of the constant toggle between 5V and 0V and vice versa. This was done using the *attachInterrupt()* function with interrupts being made every toggle. Also note that 16 toggles equal to one full rotation of the wheel.

```
/* Event to happen during change from HI - LOW or vice versa */
void encoderEvent()
{
    count++;
}

/* Outputs the velocity of the cart in .1 meters per second */
void wheelEncoder()
{
  newCount = count;
  countDiff = newCount-oldCount;
  oldCount = newCount;

  float ratio = (float)countDiff/(float)TICKS;
  velocity = ratio/(float)TIME*CIRCUMFERENCE;
  if(velocity > THRESHOLD) velocity = 2;
  velocity = velocity * V_SCALE;
}
```

An interrupt service routine (ISR) made the velocity calculation every 500 milliseconds. The speed was determined by how large or small the difference of the counter was every 500 milliseconds. For example, if the counter difference equals 50 and one full rotation of the wheel is defined by 16 toggles, then velocity = (50/16)/(0.5 seconds)*circumference of the wheel in meters.

The compass module has a simple I2C interface that output the heading

```
/* Outputs the heading of the cart in degrees from 0 - 359 */
void compass()
{
      Wire.beginTransmission(slave);
      Wire.send("A");
      Wire.endTransmission();
      delay(10);
      Wire.requestFrom(slave, 2);
      while(Wire.available() && i < 2)
      {
            data[i] = Wire.receive();
```

```
        i++;
    }
    i=0;
    value = data[0]*256 + data[1];
    }
```

“slave” represents the default slave address of the compass. At first, I left this as is, 0x42h. However, incorrect data was being outputted. After some research I found that others using this module had the slave address right-shifted and initialized to 0x21h. After the shift, I finally got useful headings that could be used for dead reckoning.

VI. Integration and Results

A. Integration of Dead Reckoning

Integration was simple because both components did not interact with each other. The only issue was making sure that they both performed as expected within the ISR and on one Arduino board. Surely enough they worked to perfection on the first try. Its output is in format: *(localized x-coordinate, localized y-coordinate, heading, velocity).* Appendix A shows the full source code for this positioning system.

The next issue was attaching the entire system onto the go-kart in preparation for a test run. The compass was to be placed in the front whereas the wheel encoder, or at least the sensor, must be getting values from the back wheel. To properly do this, long wires connected the output of the wheel encoder back to the Arduino, which was placed at the first by the compass.

B. First Run Test Results

Position was successfully found and sent back to the server but it was not accurate enough to get the game mechanics to work with it 100% of the time. There may be several reasons for this. First, the compass may have no been mounted correctly and any bumps or imperfections on the track could throw it off and need recalibration to fix itself into its proper position. Second, an optical sensor was used to read velocity. However, on a sunny day, it may not have been reading the correct high and low values as expected. This meant that velocity was not always accurate and changed the location of the kart. Once again, the disadvantage of dead reckoning is that errors pile up and worsen the system until recalibrated. We added several recalibration stations throughout the course in order to increase the accuracy of the go-kart's position. Lastly, the radius of the coordinate that the item box or weapon may have been too small and made it difficult for the go-karts to easily interact with it.

There was very little problem integrating the game mechanics with the image overlay software. As more objects were added by the server, the imaging handled the increased processing without extra delay. However, a cap was still added to the amount of objects displayed on the kart monitor at a given time (to prevent the data packets from growing too large).

A recommendation to upgrade this system is to fix the mounting of the compass to make it more stable and susceptible to bumps. Another recommendation was to change the optical sensor to a hall effect sensor. This eliminates the lighting issue and magnet sensors would be much more accurate in detecting toggles.

C. Second Run Test Results

A hall effect sensor was implemented and it provided better results than the IR sensor. Maximum velocity for these karts was calculated and read as 14mph through the serial monitor. Below is the circuit design of the wheel encoder.

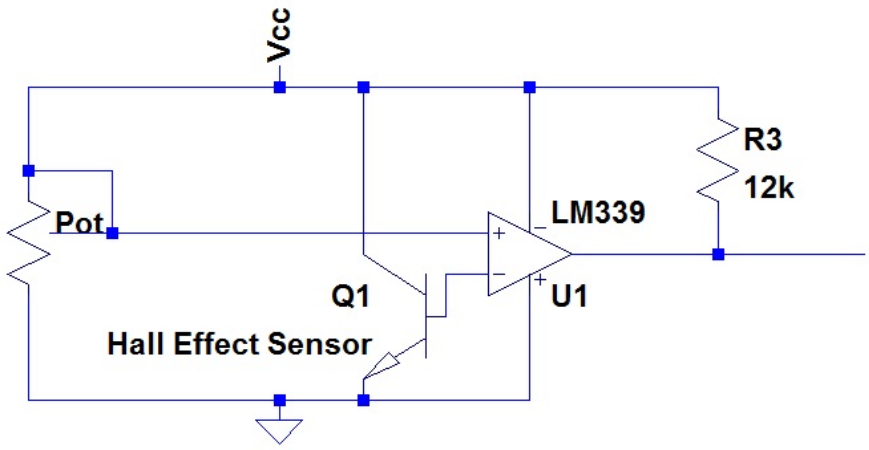


Figure X: Hall effect circuit

The hall effect sensor is placed near a ring of magnets on the wheel. The output of this signal is then fed into a comparator that replaces the small signal as either a high or low voltage that can read into the digital pin of the arduino as an external interrupt. When the arduino senses this interrupt, a velocity is calculated based on the same formula created for the IR wheel encoder. Below is a picture of the final design of the wheel encoder.

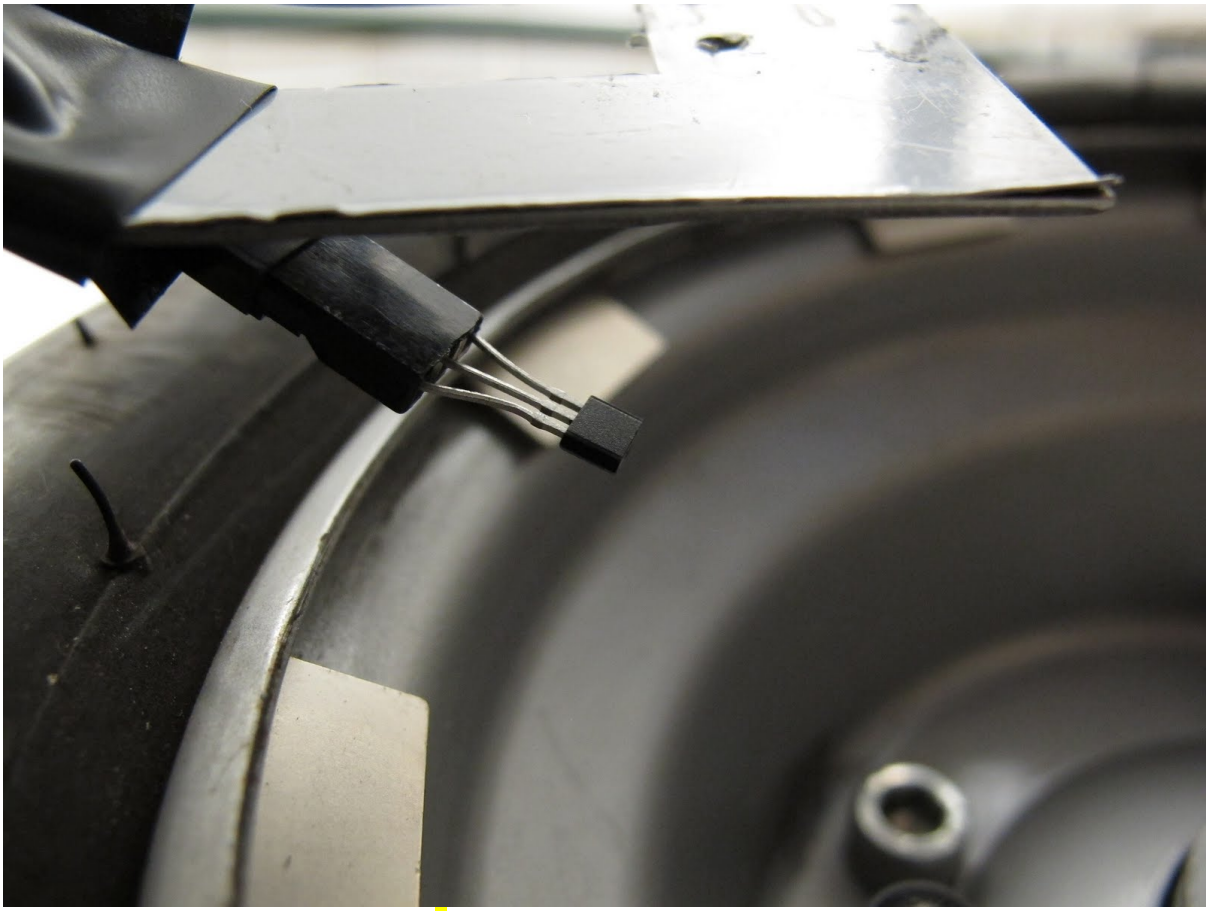


Figure X: Revision 2 Hall-effect Sensor and Magnet Design

Note: The figure does not show the complete wheel encoder system. The metal rectangles on the rim of the wheel are neodymium magnets. An extra metal bar was attached to the rear bumper of the kart. The sensor was connected to the wheel encoder circuit which was mounted near the ATmega box. The sensor was secured using electric tape.

There were alot of hardware issues for the first test run. Alot of the problems included loose wires, and it was a hassle to desolder and resolder all of the wires together. Soldering took a lot of time so it was decided to redo the hardware mounting. The new system required that all electronics were to be encapsulated in plastic cases. The inputs and outputs for the micro-controller were connected via banana-to-banana wires. This allowed for quicker and easier debugging. *See Build: Hardware Mounting.*

With the new positioning system in place, the rest of the game ran much smoother. Driving a kart towards or away from an object appeared more realistic, as the items no longer bounced around due to positioning jitter.

VII. Conclusions and Future Work

A. Game Mechanics

The first change to the game mechanics is a completely redesign of the framework. Many features originally added were never used (object strength, size, a class for each object, etc.), but were left in the code in case they later found a use. In addition, the game mechanics will be designed in anticipation for improvements that may be encountered much later. For example, each object will have 3D coordinates, and until the vertical axis becomes used, all objects are located on the same plane.

Support will be added for pre-loaded maps. Assuming that information about the playing field is already known (grid size, location of obstacles), this information can be represented in a text file and passed to the game during initialization. When the game begins, item boxes will already be created at pre-determined locations. With pre-loaded maps, map-based traps and obstacles can also be added.

In the current build, when an item box is picked up or destroyed, it is removed from the game forever. A new "regenerating" item box will be added that restores itself after being removed for a specified number of seconds.

The controller will be able to support the new networking changes listed in *Future Work: Networking*. Because devices will be allowed to enter at any time, the controller must periodically check the proper networking objects for changes.

Several interesting additions will be added to the game once the core functionality becomes solid. Enhanced drafting or slip streaming can be included if the positioning is accurate enough. If the track layout is known, a simple AI player can also be created that drives around the track interacting with other karts in the virtual world.

B. Hardware Mounting

The major addition in the future for this project is to hide all of the electronics. Currently, every piece of electronic on the kart is placed inside a glad ware. Glad ware will protect the electronics very well, but hiding the electronics inside the fenders will improve it's protection. Also, if the electronics are hidden it will be easier to get in and out of the kart.

Another addition is to include an iPhone mount near the steering wheel. An iPhone app has already been made for this project, and players will be able to see an overview of the map with an iPhone. This app can be used as a mini-map so that players will be able to see where their opponents are at all times.

Currently, all game mechanics requires the player to haunch over the steering wheel to either use an item or calibrate the kart's position. For safety reasons, it will be a good idea to mount buttons on the steering wheel.

Steering servos will be a great addition to this project. Instead of the having steering as mechanical process, the kart can be electronically steered. If the kart is electronically steered a couple of additional features can be made for the kart's. One example is a spin effect. If a kart hits a banana the kart will spin in a circle. This will better emulate the banana effect in the video game.

C. Image Overlay

The 3D objects displayed on the screen seem to float when they are traveling in our augmented reality race world. In order to fix this in the future it would be great to implement a way that the program can detect where the surface of the road is located and anchor the 3D object to the surface. Since OpenCV is already being used to pull frames, that library can be used to do all of that image processing in detecting a road's surface.

D. Speed Regulation

The whole speed regulation system will be redesigned. Although PWM worked, it had a lot of flaws. It takes a couple of seconds for the kart to accelerate at 75% speed. Alot of optimization has been done to try to fix this issue (like changing clock prescaler frequencies), but there was no noticeable difference. It was concluded that this was due to the kart's motor controller. So, in the future, we would like to purchase better hardware. This will include: a new motor, new motor controller, as well as a new go-kart. Better hardware will allow for better manipulation of the speed.

Currently, speed regulation is done by thresholding the duty cycle of the PWM signal. A new closed-loop system will be added. This closed-loop system will check the speed of the kart, calculate its speed error, and adjust the current speed by that error. This error signal will be calculated based on the throttle's A-D value as well as the current regulation that the kart has. Once calulculated, this error will either increase or decrease the PWM signal until the actual speed equals the velocity calculated from the wheel encoder.

Adding this closed-loop system will also add a top speed adjustment. In the future, players will be able to select how much they weight, and this closed-loop system will make adjustments for this weight. This will allow for an even playing field for players with highly diverse weights.

Lastly, a "NOS" feature will be an interesting feature to add. A relay or solenoid can be directly connected to the battery and when a button is pressed on the steering wheel, the kart will get a a huge boost for a couple of seconds, hence, cause a "NOS" boost.

E. Networking

As of right now, the user must specify how many players that will be participating in this game, and it only supports two different machines, be it players, "ghost kart's," or Apple's iDevices. In the future, it is planned to scale this up to as many players as possible to fit within the bounds of the course. It is also planned to have a dynamic connection, so any user can join in at any time and the game will

accomodate for it. The fact that the users have to connect at the beginning can become a hindrance in the future in the case that a new player would like to join in the middle of a game, thus, an “infinite” scale of players would highly benefit this project.

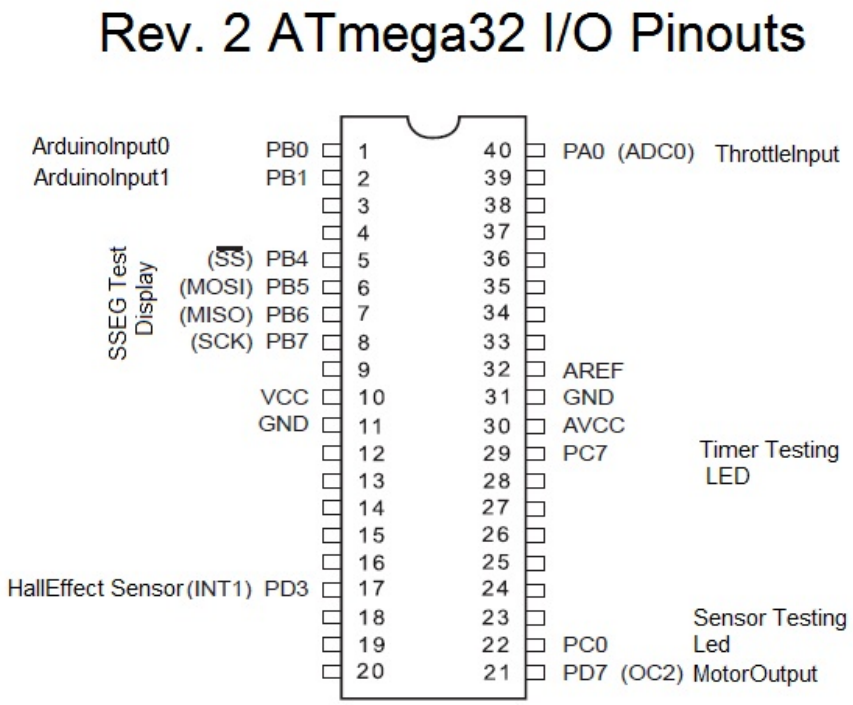
Furthermore, it would be highly convenience if the system had a remote data gathering protocol. This entails the main game server would get data from an HTTPS server, and also allow anonymous internet users to check on the status of a game, enabling them to just go to the domain and live view a game. An HTTPS connection would be necessary to enable security and make sure hackers do not try and manipulate the game.

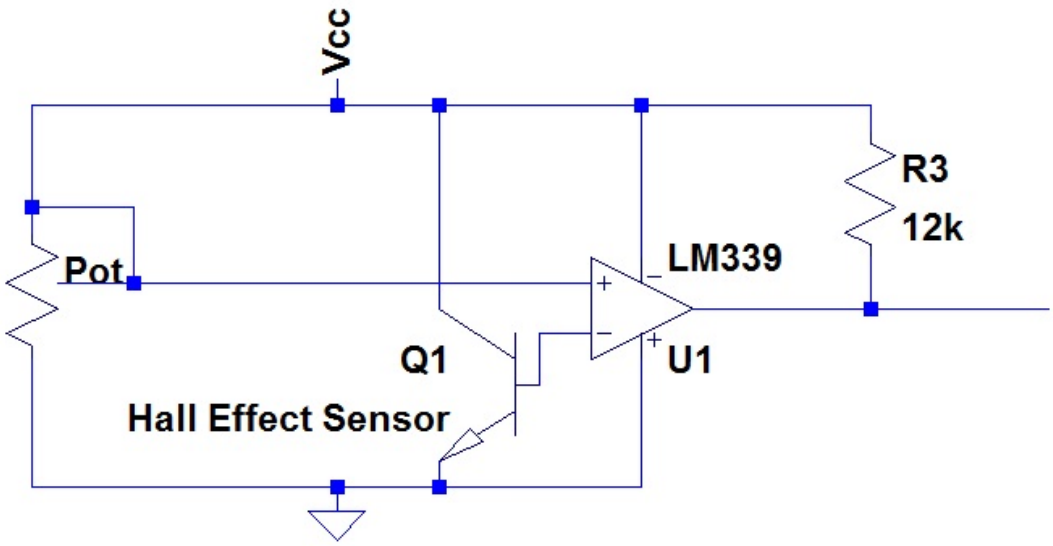
F. Positioning

Although dead reckoning is a simple and effective solution to positioning, a more accurate system is much needed to improve the overall performance of the game. Therefore, research has been conducted to create a hybrid system that incorporates fingerprinting based on RSS that will estimate location of the karts in addition to the dead reckoning system already in use. Fingerprinting can be done indoors but must be calibrated once at the start since signal strength may change. Hopefully an added positioning system will reduce error and allow the game to be played more smoothly.

Appendices

Appendix A: Schematics




Vcc
R3
12k
Pot
LM339
Q1
U1
Hall Effect Sensor

Appendix B: Parts List, Cost, and Time Schedule Allocation

Part	**Cost**
Go-Kart(3)	$500.00
36V Batteries(2)	$180.00
36V Battery Charger	$30.00
GPS EM-406(2)	$120.00
Razor IMU	$125.00
Compass Module - HMC6352(2)	$70.00
Arduino Duemilanove(3)	$90.00
Various Mounting Components	$70.00
Creative ZXXX Webcam(2)	$60.00
Total	$1245.00

Appendix C: Program Listing

NOTE: Due to the very large code base, only the main source code is added in this list. The authors of this document can be contacted if the complete directory is required.

i. game.cpp

```
/** @title game.cpp
 *  @brief Main file where all of the high-level game mechanics are located.
 *  @details This file represents the model for a model-view-controller (MVC)
 *      architecture. Public methods are called by the controller when changes
 *      to the game state need to be made. Other query methods are used by the
 *      view to update local game state data displayed on the user interface.
 *  @author David Allender
 *  @author Joryl Calizo
 *  @date March 31, 2011
 */

#include "game.h"

pthread_t temp_speed_thread;
struct speed_thread_param_t {
    game *p_game; // pointer to game object
    unsigned int kart_num; // kart number (1,2,...)
};
void *reset_speed (void*);

/** This constructor creates a new instance of the game. Objects needed to
 *  track the game state are properly created.
 *
 *  @param num_cols Number of columns
 *  @param num_rows Number of rows
 *  @param num_players Number of players
 */
game::game (uint8_t num_cols, uint8_t num_rows, uint8_t num_players) {
    // Save local copies of parameters
    this->num_cols = num_cols;
    this->num_rows = num_rows;
    num_karts = num_players;

    // Create array of pointers to kart objects
    playerKart = new kartObject*[num_karts];
    for (int i=0;i<num_karts;i++) {
        // Give invalid values for now
        playerKart[i] = new kartObject(0,0,0,NO_ITEM);
    }

    // Create grid
    objectGrid = new grid (num_cols, num_rows);

    // Set ready state to false
    kart_ready = new bool[num_karts];
    for (int i=0;i<num_karts;i++) {
        kart_ready[i] = false;
    }

    // Set lap count
    cur_lap = new uint8_t[num_karts];
```

```
    for (int i=0;i<num_karts;i++) {
        cur_lap[i] = 0;
    }

    // Start at normal duty cycle
    max_duty = new uint8_t[num_karts];
    for (int i=0;i<num_karts;i++) {
        max_duty[i] = DUTY_NORM;
    }

    // Start each kart with a unique place
    cur_place = new uint8_t[num_karts];
    for (int i=0;i<num_karts;i++) {
        cur_place[i] = i+1;
    }
}

/** This method checks if a kart is ready to play the game.
 *
 *  @param kart_num Number of kart (1, 2, ...)
 *  @return Ready state
 */
bool game::kart_is_ready (uint8_t kart_num) {
    return kart_ready[kart_num-1];
}

/** This method initializes a kart's position and heading.
 *
 *  NOTE: When a kart wants to join the game, it calls a function that
 *  returns its kart number. That function will call init_kart() using
 *  different values for kart_num until zero is returned.
 *
 *  @param kart_num Number of kart (1, 2, ...)
 *  @param x_coord x-coordinate
 *  @param y_coord y-coordinate
 *  @param heading Compass heading
 *  @return 0 if successful
 *
 *  Error codes:
 *  EOCC: xy-coordinates already occupied
 *  EINI: Kart already initialized
 *  EOOB: xy-coordinates out-of-bounds
 *  EINV: invalid kart number or heading
 */
uint8_t game::init_kart (uint8_t kart_num, uint8_t x_coord, uint8_t y_coord,
                         uint16_t heading) {
    // Kart numbers start at zero
    kart_num--;

    if (kart_num >= num_karts) { // invalid kart number
        return EINV;
    }

    if (heading >= 360) { // invalid heading
        return EINV;
    }

    if (kart_ready[kart_num]) { // already initialized
        return EINI;
    }
```

```
    if (x_coord > num_cols || y_coord > num_rows) { // out of bounds
        return EOOB;
    }

    if (objectGrid->isOccupied(x_coord,y_coord)) { // coordinates occupied
        return EOCC;
    }

    // Initialize kart
    playerKart[kart_num]->setCoord(x_coord,y_coord);
    playerKart[kart_num]->setDirection(heading);

    // Add kart to grid
    objectGrid->addObject(playerKart[kart_num]);

    // Set ready status (subsequent calls to this method with same kart number
    // will always fail)
    kart_ready[kart_num] = true;

    return 0;
}

/** This method displays a text-based representation of the grid. An 'x'
 *  corresponds to a wall, while a '.' corresponds to an unoccupied
 *  location. Any other grid object is labeled by its ID number.
 *
 *  @param scale Scaling factor when printing to cout (0 to 100%)
 */
void game::display_occupancy (uint8_t scale) {
    objectGrid->displayOccupancy (scale);
}

/** This method displays a list of all objects in the grid (does not include
 *  outer walls).
 */
void game::print_list (void) {
    objectGrid->printList ();
}

/** This method manually adds an object to the grid.
 *
 *  @param item Item type (see items.h)
 *  @param x_coord x-coordinate
 *  @param y_coord y-coordinate
 *  @param heading Compass heading
 *  @return 0 if successful
 *
 *  Error codes:
 *  EOCC: xy-coordinates already occupied
 *  EINI: karts not yet initialized
 *  EOOB: xy-coordinates out-of-bounds
 *  EINV: invalid type or heading
 */
uint8_t game::add_object (item_values_t item, uint8_t x_coord, uint8_t y_coord,
                          uint16_t heading) {
    // for red shells
    uint8_t distance = INVALID;
    uint8_t nearest = 0;
    redShellObject* tempRed;
    greenShellObject* tempGreen;
```

```
if (heading >= 360) { // invalid heading
   return EINV;
}

if (x_coord > num_cols || y_coord > num_rows) { // out of bounds
   return EOOB;
}

if (objectGrid->isOccupied(x_coord,y_coord)) { // coordinates occupied
   return EOCC;
}

for (int i=0;i<num_karts;i++) {
   if (!kart_ready[i]) { // karts not initialized
      return EINI;
   }
}

switch (item) {
   case BANANA:
      objectGrid->addObject (new bananaObject (x_coord, y_coord));
      break;
   case REDSHELL:
      tempRed = new redShellObject (x_coord, y_coord,heading, NULL);
      distance = INVALID;
      nearest = 0;
      for (int i=0;i<num_karts;i++) {
         if (tempRed->getDistance(playerKart[i]) < distance) {
            distance = tempRed->getDistance(playerKart[i]);
            nearest = i;
         }
      }
      tempRed->setTargetObject(playerKart[nearest]);
      objectGrid->addObject (tempRed);
      break;
   case GREENSHELL:
      tempGreen = new greenShellObject (x_coord, y_coord,heading);
      distance = INVALID;
      nearest = 0;
      for (int i=0;i<num_karts;i++) {
         if (tempGreen->getDistance(playerKart[i]) < distance) {
            distance = tempGreen->getDistance(playerKart[i]);
            nearest = i;
         }
      }
      tempGreen->setTargetCoord(playerKart[nearest]->getX(),
                        playerKart[nearest]->getY());
      objectGrid->addObject (tempGreen);
      break;
   case ITEMBOX:
      objectGrid->addObject (new itemBoxObject (x_coord, y_coord));
      break;
   // invalid types
   case NO_ITEM:
   case WALL:        // hm, that would be interesting..
   case MUSHROOM:    // could be used as speed boosts?
   case GREENSHIELD3:
   case REDSHIELD3:
   case KART:
   case STAR:        // too much error checking
   default:
```

```
            return EINV;
            break;
    }
    return 0;
}

/** This method returns an array of strings representing all the objects
 *  within a go-kart's viewing angle. The y-axis is parallel to the go-kart's
 *  heading, and the x-axis extends 90 degrees to the right of the driver.
 *
 *  WARNING: Don't forget to free the string after using it. Also, edit once
 *  draw distance is known.
 *
 *  @param kart_num Number of kart (1, 2, ...)
 *  @return Array of strings (check config.h for max), NULL on error
 *
 *  Format: t,x,y
 *  t: type of object (see items.h)
 *  x: x-coordinate (relative to go-kart)
 *  y: y-coordinate
 */
char** game::get_relative_coords (uint8_t kart_num) {
    uint8_t count = 0;  // number of strings
    int16_t x = 0,y = 0; // relative coordinates
    gridObject* cur;
    char** xyArray = new char*[MAX_VIEW + 1];
    char* str;

    // Kart numbers start at zero
    kart_num--;

    if (kart_num >= num_karts) {   // invalid kart number
        return NULL;
    }

    for (int i=0;i<num_karts;i++) {
        if (!kart_ready[i]) {      // karts not initialized
            return NULL;
        }
    }

    for (int i=0;i<objectGrid->listLength();i++) {
        cur = objectGrid->getFromList (i);
        if (!playerKart[kart_num]->sameObject(cur)) {
            y = playerKart[kart_num]->getRelativeY (cur);
            if (y >= 0) {              // in front of kart
                x = playerKart[kart_num]->getRelativeX (cur);
                str = new char[14];        // make room for new string
                if (cur->getType() != KART) {
                    sprintf (str, "%d,%d,%d", cur->getType(), x, y);
                }
                else { // kart
                    switch (cur->getShieldCount()) {
                        case 0: // unarmed
                            sprintf (str, "%d,%d,%d", KART, x, y);
                            break;
                        case 1:
                            if (cur->getShieldType() == REDSHELL) {
                                sprintf (str, "%d,%d,%d", REDSHIELD1, x, y);
                            }
                            else {
```

```
                        sprintf (str, "%d,%d,%d", GREENSHIELD1, x, y);
                    }
                    break;
                case 2:
                    if (cur->getShieldType() == REDSHELL) {
                        sprintf (str, "%d,%d,%d", REDSHIELD2, x, y);
                    }
                    else {
                        sprintf (str, "%d,%d,%d", GREENSHIELD2, x, y);
                    }
                    break;
                case 3:
                    if (cur->getShieldType() == REDSHELL) {
                        sprintf (str, "%d,%d,%d", REDSHIELD3, x, y);
                    }
                    else {
                        sprintf (str, "%d,%d,%d", GREENSHIELD3, x, y);
                    }
                    break;
                case STAR:
                    sprintf (str, "%d,%d,%d", STAR, x, y);
                    break;
                default:
                    break;
            }
        }
        xyArray[count] = str;      // add to string array
        if (++count >= MAX_VIEW) { // can't exceed max
            break;
        }
      }
    }
  }
  for (;count<MAX_VIEW;count++) {
    xyArray[count] = new char[4];
    sprintf (xyArray[count], "0,0,0");
  }
  return xyArray;
}

/** This method loads the kart with a item, if possible.
 *
 *  @param kart_num Numer of kart (1, 2, ...)
 *  @param item New item
 *  @return 0 if successful
 *
 *  Error codes:
 *  EOCC: kart already armed
 *  EINI: karts not yet initialized
 *  EINV: invalid kart number or item
 */
uint8_t game::set_item (uint8_t kart_num, item_values_t item) {
  // Kart numbers start at zero
  kart_num--;

  if (kart_num >= num_karts) { // invalid kart number
    return EINV;
  }

  for (int i=0;i<num_karts;i++) {
    if (!kart_ready[i]) {      // karts not initialized
```

```
            return EINI;
        }
    }

    if (playerKart[kart_num]->getItem() != NO_ITEM) { // already armed
        return EOCC;
    }

    switch (item) {
        case BANANA:
        case REDSHELL:
        case REDSHIELD3:
        case STAR:
        case GREENSHELL:
        case GREENSHIELD3:
        case MUSHROOM:
            playerKart[kart_num]->setItem (item);
            break;
        // invalid types
        //case STAR:
        case KART:
        case ITEMBOX:
        case WALL:
        case NO_ITEM:
        default:
            return EINV;
    }

    return 0;
}

/** This method loads the kart with a random item, if possible.
 *
 *  @param kart_num Numer of kart (1, 2, ...)
 *  @return 0 if successful
 *
 *  Error codes:
 *  EOCC: kart already armed
 *  EINI: karts not yet initialized
 *  EINV: invalid kart number or item
 */
uint8_t game::set_item (uint8_t kart_num) {
    item_values_t item;
    // Kart numbers start at zero
    kart_num--;

    if (kart_num >= num_karts) { // invalid kart number
        return EINV;
    }

    for (int i=0;i<num_karts;i++) {
        if (!kart_ready[i]) {      // karts not initialized
            return EINI;
        }
    }

    if (playerKart[kart_num]->getItem() != NO_ITEM) { // already armed
        return EOCC;
    }

    srand ( time (NULL) );
```

```
    do {
        item = (item_values_t)(rand() % WALL);
        switch (item) {
            case BANANA:
            case REDSHELL:
            case REDSHIELD3:
            case STAR:
            case GREENSHELL:
            case GREENSHIELD3:
            case MUSHROOM:
                playerKart[kart_num]->setItem (item);
                return 0;
                // invalid types
            case KART:
            case ITEMBOX:
            case WALL:
            case NO_ITEM:
            default:
                break;
        }
    }
    while (1);

    return 0;
}

/** This method changes kart positioning data, if possible.
 *
 *  NOTE: No support for speed regulation has been added (yet).
 *
 *  @param kart_num Number of kart (1, 2, ...)
 *  @param x_coord x-coordinate
 *  @param y_coord y-coordinate
 *  @param heading Compass heading
 *  @return 0 if successful
 *
 *  Error codes:
 *  EOCC: xy-coordinates already occupied
 *  EINI: Karts not yet initialized
 *  EOOB: xy-coordinates out-of-bounds
 *  EINV: invalid kart number or heading
 */
uint8_t game::edit_kart (uint8_t kart_num, uint8_t x_coord, uint8_t y_coord,
                         uint16_t heading) {
    gridObject* cur;
    speed_thread_param_t *speed_param;

    // Kart numbers start at zero
    kart_num--;

    if (kart_num >= num_karts) { // invalid kart number
        return EINV;
    }

    if (heading >= 360) { // invalid heading
        return EINV;
    }

    for (int i=0;i<num_karts;i++) {
        if (!kart_ready[i]) {      // karts not initialized
```

```
        return EINI;
    }
}

if (x_coord > num_cols || y_coord > num_rows) { // out of bounds
    return EOOB;
}
if (objectGrid->isOccupied(x_coord,y_coord)) { // occupied
    if (playerKart[kart_num]->                // not moving
        sameObject (objectGrid->get(x_coord,y_coord))) {
        playerKart[kart_num]->setDirection (heading);
        return 0;
    }
    cur = objectGrid->get (x_coord, y_coord);
    switch (cur->getType()) {
        case BANANA:
        case REDSHELL:
        case GREENSHIELD3:
        case REDSHIELD3:
        case GREENSHELL:
            objectGrid->removeObject (cur);
            objectGrid->moveObject (playerKart[kart_num],
                                    x_coord, y_coord);
            playerKart[kart_num]->setDirection(heading);
            if (playerKart[kart_num]->hasShield ()) {
                playerKart[kart_num]->useShield ();
            }
            else {
                max_duty[kart_num] = DUTY_HALT;
                speed_param = new speed_thread_param_t;
                speed_param->p_game = this;
                speed_param->kart_num = kart_num+1;
                pthread_create (&temp_speed_thread, NULL,
                                reset_speed, (void *)speed_param);
            }
            return 0;
        case ITEMBOX:
            set_item (kart_num+1);
            objectGrid->removeObject (cur);
            objectGrid->moveObject (playerKart[kart_num],
                                    x_coord, y_coord);
            playerKart[kart_num]->setDirection(heading);
            return 0;
        case WALL:
        case NO_ITEM:
        case STAR:
        case MUSHROOM:
        case KART:
        default:
            objectGrid->moveObject (playerKart[kart_num],
                                    x_coord, y_coord);
            playerKart[kart_num]->setDirection(heading);
            return 0;
    }
}
else { // not occupied
    objectGrid->moveObject (playerKart[kart_num],
                            x_coord, y_coord);
    playerKart[kart_num]->setDirection(heading);
    return 0;
}
```

```
    find_collisions ();
}

/** This method creates the data packet for the specified kart.
 *
 *  @param kart_num Number of kart (1, 2, ...)
 *  @param data Buffer for packet (150 bytes long)
 */
char* game::generate_packet (uint8_t kart_num, char* data) {
    char* temp;
    char** coord_list;
    uint8_t count;
    item_values_t cur_item, cur_shield;

    // Kart numbers start at zero
    kart_num--;

    if (kart_num >= num_karts) { // invalid kart number
        return NULL;
    }

    for (int i=0;i<num_karts;i++) {
        if (!kart_ready[i]) {      // karts not initialized
            return NULL;
        }
    }

    // Get relative coordinates list
    coord_list = get_relative_coords (kart_num+1);

    memset (data, 0, sizeof(data));

    // Change item labels to match image overlay
    switch (playerKart[kart_num]->getItem()) {
        case REDSHIELD3:
            cur_item = MRSHELL;
            break;
        case GREENSHIELD3:
            cur_item = MGSHELL;
            break;
        default:
            cur_item = playerKart[kart_num]->getItem();
            break;
    }

    switch (playerKart[kart_num]->getShieldType()) {
        case REDSHELL:
            cur_shield = MRSHELL;
            break;
        case GREENSHELL:
            cur_shield = MGSHELL;
            break;
        default:
            cur_shield = playerKart[kart_num]->getShieldType();
            break;
    }


    // Compile HUD info
    temp = new char[25];
    sprintf (temp, "%d,%d,%d,%d,%d,%d,%d,",
```

```
            kart_num+1,
            cur_item,
            cur_shield,
            playerKart[kart_num]->getShieldCount(),
            cur_lap[kart_num],
            cur_place[kart_num],
            max_duty[kart_num]);
    strcpy (data,temp);
    free (temp);

    // Determine number of objects in view
    for (count=0;count<MAX_VIEW;count++) {
        temp = coord_list[count];
        if (temp[0] == ',') { // first empty entry
            break;
        }
    }
    temp = new char[4];
    sprintf (temp, "%d,", count);
    strcat (data,temp);
    free (temp);

    // Add object coordinates
    for (int i=0;i<MAX_VIEW;i++) {
        if (i != MAX_VIEW-1) { // don't add last comma
            temp = new char[13];
            sprintf (temp, "%s,", coord_list[i]);
            strcat (data, temp);
            free (temp);
        }
        else {
            strcat (data, coord_list[i]);
        }
    }

    free (coord_list);

    return data;
}

/** This method checks if any near-collisions have occured and reacts
 *  accordingly. The proximity needed for a "collision" is #define'd in
 *  config.h.
 *
 *  @return Number of collisions detected
 */
uint8_t game::find_collisions (void) {
    uint8_t count = 0; // number of collisions
    uint8_t temp_x, temp_y;
    //uint16_t temp_heading;
    gridObject* obj1;
    gridObject* obj2;

    // remove any objects too close to wall
    for (int i=0;i<objectGrid->listLength();i++) {
        // Get current object
        obj1 = objectGrid->getFromList (i);
        if (obj1->getType () == KART) { // ignore karts
            // do nothing
        }
        else {
```

```
        // Too close to left or right boundaries
        if (obj1->getX() < COLLISION ||
            obj1->getX() > num_cols-COLLISION) {
            if (obj1->getType () == GREENSHELL) {
                if (obj1->getX () < COLLISION) { // left
                    if (obj1->getDirection () > 90 &&
                        obj1->getDirection () < 270) {
                        ((greenShellObject*)obj1)->bounce (90);
                    }
                }
                else { // right
                    if (obj1->getDirection () < 90 ||
                        obj1->getDirection () > 270) {
                        ((greenShellObject*)obj1)->bounce (90);
                    }
                }
            }
            else {
                objectGrid->removeObject (obj1);
                i--;
            }
        }
        // Too close to top or bottom boundaries
        else if (obj1->getY() < COLLISION || obj1->getY() > num_rows-COLLISION) {
            if (obj1->getType () == GREENSHELL) {
                if (obj1->getY () < COLLISION) { // top
                    if (obj1->getDirection () < 180) {
                        ((greenShellObject*)obj1)->bounce (0);
                    }
                }
                else { // bottom
                    if (obj1->getDirection () > 180 &&
                        obj1->getDirection () < 360) {
                        ((greenShellObject*)obj1)->bounce (0);
                    }
                }
            }
            else {
                objectGrid->removeObject (obj1);
                i--;
            }
        }
    }
}

// handle two-object collisions
for (int i=0;i<objectGrid->listLength();i++) {
    for (int j=i+1;j<objectGrid->listLength();j++) {
        // Get current objects
        obj1 = objectGrid->getFromList (i); // move outside of inner loop
        obj2 = objectGrid->getFromList (j);
        if (obj1->getDistance (obj2) <= COLLISION) { // under threshold
            // Both karts (do nothing)
            if (obj1->getType () == KART && obj2->getType () == KART) {
                // do nothing
            }
            // No karts (only preserve walls)
            else if (obj1->getType () != KART && obj2->getType () != KART) {
                // Both walls (do nothing)
                if (obj1->getType () == WALL && obj2->getType () == WALL) {
                    // do nothing
```

```
            }
            // No walls (remove both)
            else if (obj1->getType () != WALL &&
                  obj2->getType () != WALL) {
               objectGrid->removeObject (obj1);
               objectGrid->removeObject (obj2);
               i--;
               count++;
               break;
            }
            // One wall (remove other object IF NOT GREEN SHELL)
            else if (obj1->getType () == WALL) {
               // Check for green shell
               objectGrid->removeObject (obj2);
               j--;
               count++;
            }
            else {
               objectGrid->removeObject (obj1);
               i--;
               count++;
               break;
            }
         }
         // One kart (move kart to obstacle, move back)
         else if (obj1->getType () == KART) { // obj1 is a kart
            for (int k=0;k<num_karts;k++) {
               if (obj1->sameObject (playerKart[k])) {
                  temp_x = obj1->getX ();
                  temp_y = obj1->getY ();
                  edit_kart(k+1,obj2->getX(),obj2->getY(),
                        obj1->getDirection());
                  edit_kart(k+1,temp_x,temp_y,
                        obj1->getDirection());
                  //j--;
                  count++;
                  break;
               }
            }
         }
         else { // obj2 is a kart
            for (int k=0;k<num_karts;k++) {
               if (obj2->sameObject (playerKart[k])) {
                  temp_x = obj2->getX ();
                  temp_y = obj2->getY ();
                  edit_kart(k+1,obj1->getX(),obj1->getY(),
                        obj1->getDirection());
                  edit_kart(k+1,temp_x,temp_y,
                        obj1->getDirection());
                  i--;
                  count++;
                  break;
               }
            }
            break; // exit inner loop in case i = -1
         }
      }
   }
   return count;
}
```

```
/** This method updates all mobile object positions (except karts).
 *  Collisions are handled accordingly.
 *
 *  @param elapsed_ms Time elapsed since last call (in milliseconds)
 *  @return Number of collisions detected
 */
uint8_t game::move_mobile (uint16_t elapsed_ms) {
   uint8_t x,y;
   float distance;
   gridObject* cur;

   // Update objects one at a time
   for (int i=0;i<objectGrid->listLength();i++) {
      cur = objectGrid->getFromList (i);
      distance = (cur->getSpeed () * elapsed_ms) / 1000;
      switch (cur->getType()) {
         case REDSHELL:
            // Update heading before moving
            ((redShellObject*) cur)->updateTrajectory ();
         case GREENSHELL:
            // Get expected x,y
            x = cur->getX() + (distance *
               cos(cur->getDirection()*PI/180));
            y = cur->getY() - (distance *
               sin(cur->getDirection()*PI/180));
//             cout << "Old: (" << cur->getX() << ","
//                << cur->getY() << ")" << endl;
//             cout << "New: (" << x << "," << y << ")" << endl;
//             cout << "Next location: (" << x << "," << y << ")" << endl;
//             cout << "Distance traveled: " << distance << endl;

            // Stop at walls
            if (x < 1) {
               x = 1;
            }
            else if (x > num_cols - 2) {
               x = num_cols - 2;
            }
            if (y < 1) {
               y = 1;
            }
            else if (y > num_rows - 2) {
               y = num_cols - 2;
            }

            // Move shell
            objectGrid->moveObject (cur,x,y);
            break;
         case KART: // may later include support for ghost karts
            break;
         default:
            break;
      }
   }
   return find_collisions();
}

/** This method causes a kart to use its item, if one is available.
 *
 *  @param kart_num Number of kart (1, 2, ...)
```

```
 *  @return 0 if successful
 *
 *  Error codes:
 *  EOCC: xy-coordinates already occupied
 *  EINI: Karts not yet initialized
 *  EOOB: xy-coordinates out-of-bounds
 *  EINV: invalid kart number or item cannot be used
 *  ENOI: kart has no item
 */
uint8_t game::use_item (uint8_t kart_num) {
    gridObject* cur;
    speed_thread_param_t *speed_param;
    item_values_t next_item;
    // for red shells
    uint8_t distance;
    uint8_t nearest;

    // Kart numbers start at zero
    kart_num--;

    for (int i=0;i<num_karts;i++) {
        if (!kart_ready[i]) {      // karts not initialized
            return EINI;
        }
    }

    if (playerKart[kart_num]->hasShield()) {
        next_item = playerKart[kart_num]->getShieldType();
    }
    else {
        next_item = playerKart[kart_num]->getItem();
    }

    switch (next_item) {
        case NO_ITEM:
            // use shield if possible
            if (!playerKart[kart_num]->hasShield()) {
                return ENOI;
            }
        case BANANA:
            cur = playerKart[kart_num]->useItem();
            if (cur == NULL) {
                return EINV;
            }
            objectGrid->addObject (cur);
            break;
        case REDSHELL:
            cur = playerKart[kart_num]->useItem();
            if (cur == NULL) {
                return EINV;
            }
            distance = INVALID;
            for (int i=0;i<num_karts;i++) {
                if (i != kart_num) { // don't target yourself, of course
                    if (cur->getDistance(playerKart[i]) < distance) {
                        distance = cur->getDistance(playerKart[i]);
                        nearest = i;
                    }
                }
            }
            ((redShellObject*)cur)->setTargetObject (playerKart[nearest]);
```

```
            objectGrid->addObject (cur);
            break;
        case GREENSHELL:
            cur = playerKart[kart_num]->useItem();
            if (cur == NULL) {
                return EINV;
            }
//            distance = INVALID;
//            for (int i=0;i<num_karts;i++) {
//                if (i != kart_num) { // don't target yourself, of course
//                    if (cur->getDistance(playerKart[i]) < distance) {
//                        distance = cur->getDistance(playerKart[i]);
//                        nearest = i;
//                    }
//                }
//            }
//            ((greenShellObject*)cur)->setTargetCoord
//                (playerKart[nearest]->getX (), playerKart[nearest]->getY());
            objectGrid->addObject (cur);
            break;
        case GREENSHIELD3:
        case REDSHIELD3:
            playerKart[kart_num]->useItem();
            break;
        case STAR:
            playerKart[kart_num]->useItem();
            if (playerKart[kart_num]->getShieldType() == STAR) {
                max_duty[kart_num] = DUTY_BOOST;
                speed_param = new speed_thread_param_t;
                speed_param->p_game = this;
                speed_param->kart_num = kart_num+1;
                pthread_create (&temp_speed_thread, NULL,
                            reset_speed, (void *)speed_param);
            }
            break;
        case MUSHROOM:
            playerKart[kart_num]->useItem();
            max_duty[kart_num] = DUTY_BOOST;
            speed_param = new speed_thread_param_t;
            speed_param->p_game = this;
            speed_param->kart_num = kart_num+1;
            pthread_create (&temp_speed_thread, NULL,
                        reset_speed, (void *)speed_param);
            break;
        case KART:
        case ITEMBOX:
        case WALL:
        default:
            return EINV;
    }
    return 0;
}

/** This method gets the current lap for the corresponding kart.
 *
 *  @param kart_num Number of kart (1, 2, ...)
 *  @return Current lap (0, 1, 2, ...)
 *
 *  Error codes:
 *  EINI: Karts not yet initialized
 *  EINV: invalid kart number
```

```
 */
uint8_t game::get_lap (uint8_t kart_num) {
    // Kart numbers start at zero
    kart_num--;

    if (kart_num >= num_karts) { // invalid kart number
        return EINV;
    }

    for (int i=0;i<num_karts;i++) {
        if (!kart_ready[i]) {      // karts not initialized
            return EINI;
        }
    }

    return cur_lap[kart_num];
}

/** This method begins the game.
 *
 *  @return 0 if successful
 *
 *  Error codes:
 *  EINI: Karts not yet initialized
 *  EINV: Game already started
 */
uint8_t game::start_game (void) {
    for (int i=0;i<num_karts;i++) {
        if (!kart_ready[i]) {      // karts not initialized
            return EINI;
        }
        if (cur_lap[i]) { // game already started
            return EINV;
        }
    }
    for (int i=0;i<num_karts;i++) { // all lap counts set to one
        cur_lap[i]++;
    }
    return 0;
}

/** This method increments the lap counter for the corresponding kart.
 *
 *  @param kart_num Number of kart (1, 2, ...)
 *  @return Current lap (0, 1, 2, ...)
 *
 *  Error codes:
 *  EINI: Karts not yet initialized
 *  EINV: invalid kart number or game not yet started
 */
uint8_t game::next_lap (uint8_t kart_num) {
    // Kart numbers start at zero
    kart_num--;

    if (kart_num >= num_karts) { // invalid kart number
        return EINV;
    }

    for (int i=0;i<num_karts;i++) {
        if (!kart_ready[i]) {      // karts not initialized
            return EINI;
```

```
        }
    }
    if (!cur_lap[kart_num]) { // game not yet started
        return EINV;
    }
    return ++cur_lap[kart_num];
}

/** This method changes the maximum duty cycle for the corresponding kart.
 *  See duty cycle #define's in config.h.
 *
 *  @param kart_num Number of kart (1, 2, ...)
 *  @param new_duty New duty cycle
 *  @return 0 if successful
 *
 *  Error codes:
 *  EINI: Karts not yet initialized
 *  EINV: Invalid kart number, game not yet started, or invalid duty cycle
 */
uint8_t game::set_max_duty (uint8_t kart_num, uint8_t new_duty) {
    // Kart numbers start at zero
    kart_num--;

    if (kart_num >= num_karts) { // invalid kart number
        return EINV;
    }

    for (int i=0;i<num_karts;i++) {
        if (!kart_ready[i]) {      // karts not initialized
            return EINI;
        }
    }
    if (!cur_lap[kart_num]) { // game not yet started
        return EINV;
    }

    if (new_duty > 100) { // invalid duty cycle
        return EINV;
    }
    max_duty[kart_num] = new_duty;
    return 0;
}

/** This method gets the x-coordinate for the corresponding kart.
 *
 *  @param kart_num Number of kart (1, 2, ...)
 *  @return x-coordinate of kart
 *
 *  NOTE: Will return 0 upon error (invalid kart number, kart not
 *  initialized, etc.
 */
uint8_t game::get_x (uint8_t kart_num) {
    // Kart numbers start at zero
    kart_num--;

    if (kart_num >= num_karts) { // invalid kart number
        return 0;
    }
    for (int i=0;i<num_karts;i++) {
        if (!kart_ready[i]) {      // karts not initialized
            return 0;
```

```
        }
    }
    return playerKart[kart_num]->getX ();
}

/** This method gets the y-coordinate for the corresponding kart.
 *
 *  @param kart_num Number of kart (1, 2, ...)
 *  @return y-coordinate of kart
 *
 *  NOTE: Will return 0 upon error (invalid kart number, kart not
 *  initialized, etc.
 */
uint8_t game::get_y (uint8_t kart_num) {
    // Kart numbers start at zero
    kart_num--;

    if (kart_num >= num_karts) { // invalid kart number
        return 0;
    }
    for (int i=0;i<num_karts;i++) {
        if (!kart_ready[i]) {      // karts not initialized
            return 0;
        }
    }
    return playerKart[kart_num]->getY ();
}

/** This method gets the speed of the corresponding kart.
 *
 *  @param kart_num Number of kart (1, 2, ...)
 *  @return Speed of kart
 *
 *  NOTE: Will return 0 upon error (invalid kart number, kart not
 *  initialized, etc.
 */
uint8_t game::get_speed (uint8_t kart_num) {
    // Kart numbers start at zero
    kart_num--;

    if (kart_num >= num_karts) { // invalid kart number
        return 0;
    }
    for (int i=0;i<num_karts;i++) {
        if (!kart_ready[i]) {      // karts not initialized
            return 0;
        }
    }
    return playerKart[kart_num]->getSpeed ();
}

/** This method gets the heading of the corresponding kart.
 *
 *  @param kart_num Number of kart (1, 2, ...)
 *  @return Heading of kart
 *
 *  NOTE: Will return 0 upon error (invalid kart number, kart not
 *  initialized, etc.
 */
uint16_t game::get_heading (uint8_t kart_num) {
    // Kart numbers start at zero
```

```
    kart_num--;

    if (kart_num >= num_karts) { // invalid kart number
        return 0;
    }
    for (int i=0;i<num_karts;i++) {
        if (!kart_ready[i]) {      // karts not initialized
            return 0;
        }
    }
    return playerKart[kart_num]->getDirection ();
}

/** This method gets the current item of the corresponding kart.
 *
 *  @param kart_num Number of kart (1, 2, ...)
 *  @return Current item
 *
 *  NOTE: Will return NO_ITEM upon error (invalid kart number, kart not
 *  initialized, etc.
 */
item_values_t game::get_item (uint8_t kart_num) {
    // Kart numbers start at zero
    kart_num--;

    if (kart_num >= num_karts) { // invalid kart number
        return NO_ITEM;
    }
    for (int i=0;i<num_karts;i++) {
        if (!kart_ready[i]) {      // karts not initialized
            return NO_ITEM;
        }
    }
    return playerKart[kart_num]->getItem ();
}

/** This method gets the current shield type of the corresponding kart.
 *
 *  @param kart_num Number of kart (1, 2, ...)
 *  @return Current shield
 *
 *  NOTE: Will return NO_ITEM upon error (invalid kart number, kart not
 *  initialized, etc.
 */
item_values_t game::get_shield_type (uint8_t kart_num) {
    // Kart numbers start at zero
    kart_num--;

    if (kart_num >= num_karts) { // invalid kart number
        return NO_ITEM;
    }
    for (int i=0;i<num_karts;i++) {
        if (!kart_ready[i]) {      // karts not initialized
            return NO_ITEM;
        }
    }
    return playerKart[kart_num]->getShieldType ();
}

/** This method gets the current shield count of the corresponding kart.
 *
```

```
 *  @param kart_num Number of kart (1, 2, ...)
 *  @return Current shield count
 */
uint8_t game::get_shield_count (uint8_t kart_num) {
   // Kart numbers start at zero
   kart_num--;

   if (kart_num >= num_karts) { // invalid kart number
      return 0;
   }
   for (int i=0;i<num_karts;i++) {
      if (!kart_ready[i]) {      // karts not initialized
         return 0;
      }
   }
   return playerKart[kart_num]->getShieldCount ();
}

/** This method generates a packet filled with kart information. This data
 *  is used by the ghost karts, but can also be used by the graphics code
 *  to display overhead info on the HUD.
 *
 *  @param kart_num Number of kart (1, 2, ...)
 *  @param data Buffer for packet/string (40 bytes long)
 *
 *  Error codes:
 *  EINI: Karts not yet initialized
 *  EINV: invalid kart number or buffer too small
 */
uint8_t game::generate_ghost_packet (uint8_t kart_num, char* data) {
   // Kart numbers start at zero
   kart_num--;
   if (kart_num >= num_karts) { // invalid kart number
      return EINV;
   }
   for (int i=0;i<num_karts;i++) {
      if (!kart_ready[i]) {      // karts not initialized
         return EINI;
      }
   }
//    if (strlen (data) > MAX_PACKET_GHOST) {
//       return EINV;
//    }

   // Clear buffer
   memset (data, 0, MAX_PACKET_GHOST);

   sprintf (data, "%d,%d,%d,%d,%d,%d,%d,%d",
            playerKart[kart_num]->getX(),
            playerKart[kart_num]->getY(),
            playerKart[kart_num]->getDirection(),
            playerKart[kart_num]->getSpeed(),
            playerKart[kart_num]->getItem(),
            playerKart[kart_num]->getShieldType(),
            playerKart[kart_num]->getShieldCount(),
            cur_lap[kart_num]);
   return 0;
}

/** This method disables any speed boosts (MUSHROOM, STAR) for the
 *  corresponding kart.
```

```
 *
 *  @param kart_num Number of kart (1, 2, ...)
 *  @return 0 if successful
 *
 *  Error codes:
 *  EINI: Karts not yet initialized
 *  EINV: invalid kart number or no boost
 */
uint8_t game::end_boost (uint8_t kart_num) {
   // Kart numbers start at zero
   kart_num--;
   if (kart_num >= num_karts) { // invalid kart number
      return EINV;
   }
   for (int i=0;i<num_karts;i++) {
      if (!kart_ready[i]) {      // karts not initialized
         return EINI;
      }
   }

   if (playerKart[kart_num]->starPowerStop()) {
            max_duty[kart_num] = DUTY_NORM;
      return 0;
   }

   return EINV;
}

/** This method changes a kart's place in the game (1st, 2nd, 3rd, etc.)
 *  without affecting the status of the other karts. Thus, multiple karts
 *  can be in first place. A game grid class is responsible for calling
 *  this method individually for each kart.
 *
 *  @param kart_num Number of kart (1, 2, ...)
 *  @return 0 if successful
 *
 *  Error codes:
 *  EINI: Karts not yet initialized
 *  EINV: Invalid kart number
 */
uint8_t game::set_place (uint8_t kart_num, uint8_t place) {
   // Kart numbers start at zero
   kart_num--;
   if (kart_num >= num_karts) { // invalid kart number
      return EINV;
   }
   for (int i=0;i<num_karts;i++) {
      if (!kart_ready[i]) {      // karts not initialized
         return EINI;
      }
   }
   cur_place[kart_num] = place;
   return 0;
}

/** This method gets a kart's current lap.
 *
 *  @param kart_num Number of kart (1, 2, ...)
 *  @return Kart's lap
 */
uint8_t game::get_place (uint8_t kart_num) {
```

```
    // Kart numbers start at zero
    kart_num--;
    if (kart_num >= num_karts) { // invalid kart number
        return EINV;
    }
    for (int i=0;i<num_karts;i++) {
        if (!kart_ready[i]) {      // karts not initialized
            return EINI;
        }
    }
    return cur_place[kart_num];
}

/** This method creates the data packet for a 2D overhead view.
 *
 *  @param data Buffer for packet
 *
 *  Error codes:
 *  EINI: Karts not yet initialized
 */
uint8_t game::generate_2D_packet (char* data) {
    gridObject* cur;
    item_values_t cur_type;
    uint8_t index;

    for (int i=0;i<num_karts;i++) {
        if (!kart_ready[i]) {      // karts not initialized
            return NULL;
        }
    }

    memset (data, 0, MAX_PACKET_2D);
    index = 0;

    // Start with header
    index += sprintf (data+index, "SO,");

    // Send current item
    index += sprintf (data+index, "%d,%d,", playerKart[0]->getItem(),
                    playerKart[1]->getItem());

    // Select format (only "all" is supported for now)
    index += sprintf (data+index, "A,");

    // Add random element for now
    index += sprintf (data+index, "0,");

    for (int i=0;i<objectGrid->listLength();i++) {
        cur = objectGrid->getFromList (i);
        cur_type = cur->getType();
        if (cur_type == KART) {
            // Replace kart with shield type, if applicable
            if (((kartObject*)cur)->hasShield()) {
                cur_type = ((kartObject*)cur)->getShieldType();
            }
        }
        index += sprintf (data+index, "%d,%d,%d", cur_type, cur->getX(),
                        cur->getY());
        if (i != objectGrid->listLength()-1) {
            index += sprintf (data+index, ",");
        }
```

```
    }

    return 0;
}

/** This method creates the data packet for the RC controller.
 *
 *  @param data Buffer for packet
 *
 *  Error codes:
 *  EINI: Karts not yet initialized
 */
int game::generate_RC_packet (char *data) {
    unsigned int index;
    unsigned int RC_speed;

    for (int i=0;i<num_karts;i++) {
        if (!kart_ready[i]) {      // karts not initialized
            return NULL;
        }
    }

    memset (data, 0, MAX_PACKET_RC);
    index = 0;

    // Start with header
    index += sprintf (data+index, "SRC,");

    // Add random element for now
    index += sprintf (data+index, "0,");

    for (int i=0;i<num_karts;i++) {
        // Kart number
        index += sprintf (data+index, "%d,", i+1);
        // Get speed regulation byte
        switch (max_duty[i]) {
            case DUTY_BOOST:
                RC_speed = RC_FULL;
                break;
            case DUTY_NORM:
            case DUTY_SLOW:
                RC_speed = RC_HALF;
                break;
            case DUTY_HALT:
            default:
                RC_speed = RC_STOP;
                break;
        }
        index += sprintf (data+index, "%d", RC_speed);
        if (i != num_karts-1) { // Not last entry
            index += sprintf (data+index, ",");
        }
    }

    return 0;
}

/** This method returns a pointer to the grid. The controller can use this
 *  method for direct access to grid information.
 *
 *  @return Pointer to grid class object
```

```
 */
grid *game::get_grid (void) {
   return objectGrid;
}

/** This thread waits for a set amount of time before restoring a kart's
 *  normal speed.
 *
 *  @param arg Pointer to struct speed_thread_param_t
 */
void *reset_speed (void *arg) {
   speed_thread_param_t *game_info = (speed_thread_param_t*)arg;
   game *game_model = game_info->p_game;
   unsigned int kart_num = game_info->kart_num;

   struct timespec *delay = new timespec;
   if (game_model->get_shield_type (kart_num) == STAR) {
      delay->tv_sec = STAR_MS / 1000;
      delay->tv_nsec = STAR_MS % 1000000000;
   }
   else {
      delay->tv_sec = MUSHROOM_MS / 1000;
      delay->tv_nsec = MUSHROOM_MS % 1000000000;
   }
   nanosleep (delay,NULL);
   game_model->set_max_duty (kart_num, DUTY_NORM);
   game_model->end_boost (kart_num);
   return NULL;
}

/** This method creates the data packet for the iOS 2D overhead view.
 *
 *  @param data Buffer for packet
 *
 *  Error codes:
 *  EINI: Karts not yet initialized
 */
uint8_t game::generate_2D_iOS_packet (char* data) {
   gridObject* cur;
   item_values_t cur_type;
   uint8_t index;

   for (int i=0;i<num_karts;i++) {
      if (!kart_ready[i]) {      // karts not initialized
         return NULL;
      }
   }

   memset (data, 0, MAX_PACKET_2D);
   index = 0;

   // Start with header
   index += sprintf (data+index, "S2D,");

   // Send current item
   index += sprintf (data+index, "%d,%d,", playerKart[0]->getItem(),
                     playerKart[1]->getItem());

   // Send dimensions
   index += sprintf (data+index, "%d,%d,", num_cols, num_rows);
```

```
    // Send items (max 10)
    for (int i=0;i<min((int)objectGrid->listLength(),10);i++) {
        cur = objectGrid->getFromList (i);
        cur_type = cur->getType();
        if (cur_type == KART) {
            // Replace kart with shield type, if applicable
            if (((kartObject*)cur)->hasShield()) {
                cur_type = ((kartObject*)cur)->getShieldType();
            }
        }
        index += sprintf (data+index, "%d,%d,%d", cur_type, cur->getX(),
                          cur->getY());
        if (i != objectGrid->listLength()-1) {
            index += sprintf (data+index, ",");
        }
    }

    return 0;
}
```

ii. controller.cpp

```
/*  @file controller.cpp
 *  @details This class contains the functionality needed for the controller
 *      of a model-view-controller (MVC) architecture. Public methods decode
 *      packets from both physical and virtual (keyboard-controlled) karts
 *      and call the correct methods of the game class to synchronize the game
 *      state with the information local to each kart.
 *  @author David Allender
 *  @author Joryl Calizo
 *  @date April 11, 2011
 */

#include "controller.h"

#ifdef USE_GUI
struct mouse_input_t {
    bool waiting;
    int x;
    int y;
};
using namespace cv;
void mouse_handler (int, int, int, int, void*);
#endif

/** This constructor creates a controller object.
 *
 *  @param p_game Pointer to game model
 */
controller::controller (game *p_game) {
    // Store local pointer to game model
    game_model = p_game;
    game_grid = game_model->get_grid ();
}

/** This method initializes the game by getting settings from the user via
 *  keyboard inputs.
 *
 *  @param game_type Code for game type (see below)
 *  @return Number of karts
 *
```

```
 *  Game type codes:
 *  GAME_KART: using real karts
 *  GAME_RC:   using RC cars
 */
unsigned int controller::init_game (unsigned int game_type) {
   // Greeting
   cout << "Hello!" << endl;

   // Determine number of karts
   cout << "How many karts will be playing? ";
   cin >> num_karts;
   num_karts -= '0'; // convert to int
   while (num_karts != 2) { // support for two karts only
      cout << "Sorry, invalid input. Please enter a number between 2 "
           "and 2. ";
      cin >> num_karts;
      num_karts -= '0';
   }

   this->game_type = game_type;

   if (game_type == GAME_KART) {
   #ifndef NO_NETWORK
      // Determine number of physical karts
      cout << "How many are physical karts? ";
      cin >> num_real_karts;
      num_real_karts -= '0';
      while (num_real_karts > 2) {
         cout << "Sorry, invalid input. Please enter a number between 0 "
              "and 2. ";
         cin >> num_real_karts;
         num_real_karts -= '0';
      }
   #else
      num_real_karts = 0;
   #endif
      num_ghost_karts = 2-num_real_karts;

      // Create networking objects
   #ifndef NO_NETWORK
      kart = new networking*[num_karts]; // always create two karts
      for (int i=0;i<num_karts;i++) {
         kart[i] = new networking (NETWORK_SERVER,i+1);
         kart[i]->connect();
      }
   #endif

      // Why output EOOB?
      cout << game_model->init_kart (1,20,20,0) << endl;
      game_model->init_kart (2,game_grid->get_cols()-20,
                             game_grid->get_rows()-20,180);
   }
   else if (game_type == GAME_RC) {
      num_RC = num_karts;
   #ifndef NO_NETWORK
      cout << "Connecting to Overhead View..." << endl;
      overhead = new networking (NETWORK_SERVER,1); // first slot
      overhead->connect ();
   #ifdef USE_RC_CTRL
      cout << "Connecting to RC controller..." << endl;
      RC_controller = new networking (NETWORK_SERVER,2);
```

```
        RC_controller->connect ();
    #endif
    #endif
        game_model->init_kart (1,20,20,0);
        game_model->init_kart (2,game_grid->get_cols()-20,
                               game_grid->get_rows()-20,180);
    }

#ifdef USE_GUI
    view.create (600,600,CV_8UC3);
    view_label = "Server View";
    namedWindow (view_label, CV_GUI_EXPANDED);
#endif
    return num_karts;
}

/** This function receives from the networking stack
 *  and calls "edit_kart" to change the game state
 */
void controller::update_game(void){
#ifdef USE_GUI
    gui_occupancy ();
#endif
    if (game_type == GAME_KART) {
        uint32_t x, y;
        uint16_t heading;
        uint32_t speed;
        uint8_t item_used = 0;
        int received = 0;
        char *buffer = (char*) malloc (MAX_PACKET_GHOST);
        char *pch;
        char *tok;
        int tokcount = 0;
        uint8_t num_commas = 0;

        memset (buffer, 0 , MAX_PACKET_GHOST);

        for (int i=0;i<num_real_karts;i++) {
            item_used = 0;
            received = kart[i]->recv_data(buffer);
    #ifdef DEBUG
            cout << "Bytes received: " << received << endl;
            cout << "Packet from kart " << i+1 << ": " << buffer << endl;
    #endif
            // Count number of commas
            num_commas = 0;
            pch = strchr (buffer,',');
            while (pch != NULL) {
                num_commas++;
                pch = strchr(pch+1,',');
            }
            if (num_commas == 4)
            {
                tokcount = 0;
                tok = strtok(buffer, ",");
                while(tok)
                {
                    if(tokcount == 0)
                    {
                        x = atoi(tok);
                    }else if(tokcount == 1)
```

```
                {
                    y = atoi(tok);
                }else if(tokcount == 2)
                {
                    heading = atoi(tok);
                }else if(tokcount == 3)
                {
                    speed = atoi(tok);
                }else if(tokcount == 4)
                {
                    item_used = atoi(tok);
                }
                tok = strtok(NULL, ",");
                tokcount++;
            }
            game_model->edit_kart(i+1, x, y, heading);
            if (item_used) {
                game_model->use_item (i+1);
            }
            received = 0;


        }
        else {
#ifdef DEBUG
            cout << "Bad packet." << endl;
#endif
        }
        memset (buffer, 0 , MAX_PACKET_GHOST);
    }
    free (buffer);
}
else if (game_type == GAME_RC) {
    char *cur_tok;
    char *buffer = new char[MAX_PACKET_OVERHEAD];
    unsigned int *x, *y, *speed, *heading, num_RC, cur_RC, count;
    bool *item_used;
    unsigned int tok_count = 0;

    count = 0;
    memset (buffer, 0, MAX_PACKET_OVERHEAD);

    // Get packet from 2D program
#ifndef NO_NETWORK
    count = overhead->recv_data(buffer);
#endif
    //count = sprintf(buffer, "O,2,0,1,123,102,197,0,1,2,54,67,349,0,1");
#ifdef DEBUG
    cout << "Bytes received: " << count << endl;
    cout << "Packet from 2D: " << buffer << endl;
#endif

    // Get first token
    cur_tok = strtok (buffer, ",");

    while (cur_tok) {
        if (tok_count == 0) { // header
            if (cur_tok[0] != 'O') {
                return;
            }
        }
```

```
else if (tok_count == 1) {
    num_RC = atoi(cur_tok); // should be two
    x = new unsigned int[num_RC];
    y = new unsigned int[num_RC];
    speed = new unsigned int[num_RC];
    heading = new unsigned int[num_RC];
    item_used = new bool[num_RC];
}
else if (tok_count == 2) { // unused (for now)

}
else if (tok_count > 2) { // karts
    // Kart number (starts at one)
    cur_RC = atoi(cur_tok)-1;

    cur_tok = strtok (NULL, ",");
    tok_count++;
    if (!cur_tok) {
        return;
    }

    // x-coordinate
    x[cur_RC] = atoi (cur_tok);

    cur_tok = strtok (NULL, ",");
    tok_count++;
    if (!cur_tok) {
        return;
    }

    // y-coordinate
    y[cur_RC] = atoi (cur_tok);

    cur_tok = strtok (NULL, ",");
    tok_count++;
    if (!cur_tok) {
        return;
    }

    // heading
    heading[cur_RC] = atoi (cur_tok);

    cur_tok = strtok (NULL, ",");
    tok_count++;
    if (!cur_tok) {
        return;
    }

    // speed
    speed[cur_RC] = atoi (cur_tok);

    cur_tok = strtok (NULL, ",");
    tok_count++;
    if (!cur_tok) {
        return;
    }

    // item used
    item_used[cur_RC] = atoi (cur_tok);
}
// Get next token
```

```
            cur_tok = strtok (NULL, ",");
            tok_count++;
        }

        // Execute
        for (unsigned int i=0;i<num_RC;i++) {
            game_model->edit_kart (i+1,x[i],y[i],heading[i]);
            if (item_used[i]) {
                game_model->use_item (i+1);
            }
        }
        free (buffer);
    }
}

/* This function calls generate_packet(for player)
 * and generate_ghost_packet(for ghosts)
 * to generate packets.  Then the user sends that to the kart
 * where it will be parsed by Joseph Joshua Abad
 */
void controller::update_karts(void){
    if (game_type == GAME_KART) {
        char packet[MAX_PACKET_SIZE];

        for (int i=0;i<num_real_karts;i++) {
            memset (packet,0,MAX_PACKET_SIZE);
            game_model->generate_packet (i+1, packet);
    #ifdef DEBUG
            cout << "packet: " << packet << endl;
    #endif
            kart[i]->send_data (packet);
        }
//GHOST KART IS NOW IOS DEVICE
        for (int i=0;i<num_ghost_karts;i++) {
    #ifndef NO_NETWORK
            // Generate pseudo-ghost info
            memset (packet,0,MAX_PACKET_SIZE);
            game_model->generate_packet (i+1+num_real_karts, packet);
            kart[1]->send_data(packet);
            game_model->generate_2D_iOS_packet (packet);
            kart[1]->send_data(packet);
    #ifdef DEBUG
            cout << "packet: " << packet << endl;
    #endif
            //kart[i+num_real_karts]->send_data (packet);
    #endif
        }
    }
    else if (game_type == GAME_RC) {
        char packet[MAX_PACKET_2D];

        memset (packet,0,MAX_PACKET_2D);

        game_model->generate_2D_packet (packet);

    #ifdef DEBUG
        cout << "Packet for 2D view: " << packet << endl;
    #endif
    #ifndef NO_NETWORK
        overhead->send_data (packet);
    #endif
```

```
        memset (packet,0,MAX_PACKET_2D);

        game_model->generate_RC_packet (packet);
    #ifdef DEBUG
        cout << "Packet for RC controller: " << packet << endl;
    #endif
    #ifndef NO_NETWORK
    #ifdef USE_RC_CTRL
        RC_controller->send_data (packet);
    #endif
    #endif
    }
}

/** This method displays an overhead, text-based representation of the
 *  playing grid.
 *
 *  @param scale Scaling factor (0 to 100%)
 */
void controller::display_occupancy (uint8_t scale) {
    game_model->display_occupancy (scale);
}

/** This method prints a list of all the objects currently in the grid.
 */
void controller::print_list (void) {
    game_model->print_list ();
}

/** This method begins the game.
 *
 *  @return 0 if successful
 *
 *  Error codes:
 *  EINI: Karts not yet initialized
 *  EINV: Game already started
 */
uint8_t controller::start_game (void) {
    return game_model->start_game ();
}

/** This method creates a text-based interface for manually changing the
 *  game state.
 */
void controller::main_ui (void) {
    char string[MAX_PACKET_2D];
    uint8_t temp;
    char input;
    cout << "Enter a command. 'H' prints out a list of commands." << endl;
    while (true) {
        memset (string, 0, MAX_PACKET_2D);
#ifndef USE_GUI
        cout << "> ";
        cin >> input;
#else
        do {
            imshow (view_label, view);
            input = waitKey (500);
        } while (input == -1);
#endif
```

```
      switch (input) {
         case 'H':
         case 'h':
//             cout << "Press 'G' to control ghost kart." << endl;
            cout << "Press 'K' to make changes to a kart." << endl;
            cout << "Press 'L' to display list of grid objects." << endl;
            cout << "Press 'O' to display an occupancy grid." << endl;
            cout << "Press 'A' to add new objects to the grid." << endl;
            cout << "Press 'S' to turn off STAR power." << endl;
            cout << "Press 'P' to change kart places." << endl;
            cout << "Press 'U' to generate a 2D overhead view packet." << endl;
            cout << "Press 'I' to generate an iOS 2D overhead view packet." << endl;
            cout << "Press '1 or 2' to generate a HUD packet." << endl;
            break;
         case 'K':
         case 'k':
               cout << "Which kart would you like to control?" << endl;
            do {
#ifndef USE_GUI
               cout << "> ";
               cin >> input;
#else
               do {
                  imshow (view_label, view);
                  input = waitKey (500);
               } while (input == -1);
#endif
            }
            while ((input - '0') > num_karts || !(input - '0'));
            control_kart (input - '0');
            break;
         case 'L':
         case 'l':
            print_list ();
            break;
         case 'O':
         case 'o':
            display_occupancy (25);
            break;
         case 'S':
         case 's':
            game_model->end_boost(1);
            game_model->end_boost(2);
            break;
         case 'A':
         case 'a':
            add_object();
            break;
         case 'P':
         case 'p':
            for (int i=1;i<=num_karts;i++) {
               temp = game_model->get_place(i);
               if (temp == num_karts) {
                  game_model->set_place(i,1);
               }
               else {
                  game_model->set_place(i,temp+1);
               }
            }
            break;
         case 'U':
```

```
            case 'u':
                game_model->generate_2D_packet(string);
                cout << string << endl;
                break;
            case 'I':
            case 'i':
                game_model->generate_2D_iOS_packet(string);
                cout << string << endl;
                break;
            case '1':
            case '2':
                game_model->generate_packet(input-'0',string);
                cout << string << endl;
                break;
            default:
                cout << "Invalid or unsupported command. Sorry." << endl;
                break;
        }
    }
}

/** This method allows a user to change kart settings.
 *
 *  @param kart_num Number of kart (1, 2, ...)
 */
void controller::control_kart (uint8_t kart_num) {
    char input;
    uint8_t result;
#ifdef USE_GUI
    mouse_input_t *mouse_click = new mouse_input_t;
    mouse_input_t *mouse_unclick = new mouse_input_t;
    signed int x,y,heading;
#endif

    // Kart numbers start at zero
    kart_num--;
    if (kart_num >= num_karts) {
        return;
    }

    cout << "Controlling kart " << kart_num + 1 << "." << endl;
    while (true) {
#ifndef USE_GUI
        cout << "> ";
        cin >> input;
#else
        do {
            imshow (view_label, view);
            input = waitKey(500);
        } while (input == -1);
#endif
        switch (input) {
            case 'H':
            case 'h':
                cout << "Press 'S' to move kart." << endl;
                cout << "Press 'D' to change compass direction." << endl;
                cout << "Press 'L' to move on to the next lap." << endl;
                cout << "Press '+' to remove speed regulation." << endl;
                cout << "Press '-' to limit speed." << endl;
                cout << "Press 'X' to stop kart." << endl;
                cout << "Press '*' to get a star." << endl;
```

```
    cout << "Press 'B' to get a banana." << endl;
    cout << "Press 'R' to get a red shell." << endl;
    cout << "Press 'G' to get a green shell." << endl;
    cout << "Press '3' to get a triple red shell." << endl;
    cout << "Press '4' to get a triple green shell." << endl;
    cout << "Press 'M' to get a mushroom." << endl;
    cout << "Press '?' to get a random item." << endl;
    cout << "Press 'U' to use item." << endl;
    cout << "Press 'C' to exit." << endl;
    break;
  case 'D':
  case 'd':
    // Create drag algorithm
    mouse_click->waiting = true;
    setMouseCallback (view_label, mouse_handler, mouse_click);
    cout << "Drag cursor in direction desired." << endl;
    do {
      imshow (view_label, view);
      waitKey (500);
    } while (mouse_click->waiting);
    setMouseCallback (view_label, mouse_handler, mouse_unclick);
    mouse_unclick->waiting = true;
    do {
      imshow (view_label, view);
      waitKey (500);
    } while (mouse_unclick->waiting);
    setMouseCallback (view_label, NULL, NULL);
    heading = atan2 (mouse_click->y - mouse_unclick->y,
                mouse_unclick->x - mouse_click->x)
                * 180 / PI;
    if (heading < 0) {
      heading += 360;
    }
    cout << "New heading: " << heading << endl;
    game_model->edit_kart (kart_num+1, get_x (kart_num+1),
                        get_y (kart_num+1), heading);
    break;
  case 'S':
  case 's':
    mouse_click->waiting = true;
    setMouseCallback (view_label, mouse_handler, mouse_click);
    cout << "Click on new coordinates." << endl;
    do {
      imshow (view_label, view);
      waitKey (500);
    } while (mouse_click->waiting);
    setMouseCallback (view_label, NULL, NULL);
    x = mouse_click->x * game_grid->get_cols () / 600;
    y = mouse_click->y * game_grid->get_rows () / 600;
    game_model->edit_kart (kart_num+1,x,y,get_heading(kart_num+1));
    break;
  case 'L':
  case 'l':
    game_model->next_lap (kart_num+1);
    cout << "Kart " << kart_num + 1 << "'s lap incremented."
        << endl;
    break;
  case '+':
    game_model->set_max_duty (kart_num+1, DUTY_BOOST);
    cout << "Kart " << kart_num + 1 << " set to max duty cycle."
        << endl;
```

```
    break;
  case '-':
    game_model->set_max_duty (kart_num+1, DUTY_NORM);
    cout << "Kart " << kart_num + 1 << " set to normal duty cycle."
         << endl;
    break;
  case 'X':
  case 'x':
    game_model->set_max_duty (kart_num+1, DUTY_HALT);
    cout << "Kart " << kart_num + 1 << " set to zero duty cycle."
         << endl;
    break;
  case '*':
    result = game_model->set_item (kart_num+1, STAR);
    switch (result) {
      case 0:
        cout << "Kart " << kart_num + 1 << " received a star."
             << endl;
        break;
      case EOCC:
        cout << "Kart " << kart_num + 1 << " already armed."
             << endl;
        break;
      default:
        break;
    }
    break;
  case 'B':
  case 'b':
    result = game_model->set_item (kart_num+1, BANANA);
    switch (result) {
      case 0:
        cout << "Kart " << kart_num + 1 << " received a "
             << "banana." << endl;
        break;
      case EOCC:
        cout << "Kart " << kart_num + 1 << " already armed."
             << endl;
        break;
      default:
        break;
    }
    break;
  case 'G':
  case 'g':
    game_model->set_item (kart_num+1, GREENSHELL);
    break;
  case 'R':
  case 'r':
    result = game_model->set_item (kart_num+1, REDSHELL);
    switch (result) {
      case 0:
        cout << "Kart " << kart_num + 1 << " received a "
             << "red shell." << endl;
        break;
      case EOCC:
        cout << "Kart " << kart_num + 1 << " already armed."
             << endl;
        break;
      default:
        break;
```

```
    }
    break;
case '3':
    result = game_model->set_item (kart_num+1, REDSHIELD3);
    switch (result) {
        case 0:
            cout << "Kart " << kart_num + 1 << " received "
                << "red shell x3." << endl;
            break;
        case EOCC:
            cout << "Kart " << kart_num + 1 << " already armed."
                << endl;
            break;
        default:
            break;
    }
    break;
case '4':
    result = game_model->set_item (kart_num+1, GREENSHIELD3);
    switch (result) {
        case 0:
            cout << "Kart " << kart_num + 1 << " received "
                << "green shell x3." << endl;
            break;
        case EOCC:
            cout << "Kart " << kart_num + 1 << " already armed."
            << endl;
            break;
        default:
            break;
    }
    break;
case 'M':
case 'm':
    result = game_model->set_item (kart_num+1, MUSHROOM);
    switch (result) {
        case 0:
            cout << "Kart " << kart_num + 1 << " received "
            << "a mushroom." << endl;
            break;
        case EOCC:
            cout << "Kart " << kart_num + 1 << " already armed."
            << endl;
            break;
        default:
            break;
    }
    break;
case '?':
    result = game_model->set_item (kart_num+1);
    switch (result) {
        case 0:
            cout << "Kart " << kart_num + 1 << " received a "
                << "random item." << endl;
            break;
        case EOCC:
            cout << "Kart " << kart_num + 1 << " already armed."
                << endl;
            break;
        default:
            break;
```

```
            }
            break;
        case 'U':
        case 'u':
            game_model->use_item (kart_num+1);
            break;
        case 'C':
        case 'c':
            cout << "Returning to main menu." << endl;
            return;
        default:
            break;
      }
   }
}

/** This method allows a user to manually add an object to the grid.
 */
void controller::add_object (void) {
   char input;
#ifndef USE_GUI
   char temp[4];
#endif
   uint8_t x,y;
   uint8_t result;
   item_values_t new_item;
#ifdef USE_GUI
   mouse_input_t *mouse_click = new mouse_input_t;
#endif

   cout << "Adding objects." << endl;
   while (true) {
#ifndef USE_GUI
      cout << "> ";
      cin >> input;
#else
      do {
         imshow (view_label, view);
         input = waitKey (500);
      } while (input == -1);
#endif
      switch (input) {
         case 'H':
         case 'h':
            cout << "Press 'B' to add a banana." << endl;
            cout << "Press 'R' to add a red shell." << endl;
            cout << "Press 'G' to add a green shell." << endl;
            cout << "Press 'I' to add an item box." << endl;
            cout << "Press 'C' to exit." << endl;
            continue;
         case 'C':
         case 'c':
            cout << "Returning to main menu." << endl;
            return;
         case 'B':
         case 'b':
         case 'R':
         case 'r':
         case 'G':
         case 'g':
         case 'I':
```

```
            case 'i':
#ifdef USE_GUI
                mouse_click->waiting = true;
                setMouseCallback (view_label, mouse_handler, mouse_click);
                cout << "Click to drop item." << endl;
                do {
                    imshow (view_label, view);
                    waitKey(500);
                } while (mouse_click->waiting);
                x = mouse_click->x * game_grid->get_cols () / 600;
                y = mouse_click->y * game_grid->get_rows () / 600;
                setMouseCallback (view_label, NULL, NULL);
#else
                cout << "X: ";
                cin >> temp;
                x = atoi (temp);
                cout << "Y: ";
                cin >> temp;
                y = atoi (temp);
#endif
                break;
            default:
                cout << "Invalid character, try again." << endl;
                continue;
        }
        switch (input) {
            case 'B':
            case 'b':
                new_item = BANANA;
                break;
            case 'R':
            case 'r':
                new_item = REDSHELL;
                break;
            case 'G':
            case 'g':
                new_item = GREENSHELL;
                break;
            case 'I':
            case 'i':
                new_item = ITEMBOX;
                break;
            default:
                continue;
        }
        result = game_model->add_object (new_item,x,y,0);
        switch (result) {
            case 0:
                cout << "Item added to (" << (int)x << "," << (int)y
                    << ")" << endl;
                break;
            case EOCC:
                cout << "(" << (int)x << ","
                    << (int)y << ") already occupied." << endl;
                break;
            case EOOB:
                cout << "Out of bounds." << endl;
                break;
            default:
                break;
        }
```

```
    }
}

/** This method gets the x-coordinate for the corresponding kart.
 *
 *  @param kart_num Number of kart (1, 2, ...)
 *  @return x-coordinate of kart
 *
 *  NOTE: Will return 0 upon error (invalid kart number, kart not
 *  initialized, etc.
 */
uint8_t controller::get_x (uint8_t kart_num) {
    return game_model->get_x (kart_num);
}

/** This method gets the y-coordinate for the corresponding kart.
 *
 *  @param kart_num Number of kart (1, 2, ...)
 *  @return y-coordinate of kart
 *
 *  NOTE: Will return 0 upon error (invalid kart number, kart not
 *  initialized, etc.
 */
uint8_t controller::get_y (uint8_t kart_num) {
    return game_model->get_y (kart_num);
}

/** This method gets the speed of the corresponding kart.
 *
 *  @param kart_num Number of kart (1, 2, ...)
 *  @return Speed of kart
 *
 *  NOTE: Will return 0 upon error (invalid kart number, kart not
 *  initialized, etc.
 */
uint8_t controller::get_speed (uint8_t kart_num) {
    return game_model->get_speed (kart_num);
}

/** This method gets the heading of the corresponding kart.
 *
 *  @param kart_num Number of kart (1, 2, ...)
 *  @return Heading of kart
 *
 *  NOTE: Will return 0 upon error (invalid kart number, kart not
 *  initialized, etc.
 */
uint16_t controller::get_heading (uint8_t kart_num) {
    return game_model->get_heading (kart_num);
}

/** This method gets the current item of the corresponding kart.
 *
 *  @param kart_num Number of kart (1, 2, ...)
 *  @return Current item
 *
 *  NOTE: Will return NO_ITEM upon error (invalid kart number, kart not
 *  initialized, etc.
 */
item_values_t controller::get_item (uint8_t kart_num) {
    return game_model->get_item (kart_num);
```

```
}

/** This method gets the current shield type of the corresponding kart.
 *
 *  @param kart_num Number of kart (1, 2, ...)
 *  @return Current shield
 *
 *  NOTE: Will return NO_ITEM upon error (invalid kart number, kart not
 *  initialized, etc.
 */
item_values_t controller::get_shield_type (uint8_t kart_num) {
   return game_model->get_shield_type (kart_num);
}

/** This method gets the current shield count of the corresponding kart.
 *
 *  @param kart_num Number of kart (1, 2, ...)
 *  @return Current shield count
 */
uint8_t controller::get_shield_count (uint8_t kart_num) {
   return game_model->get_shield_count (kart_num);
}

/** This method gets the current lap number for the corresponding kart.
 *
 *  @param kart_num Number of kart (1, 2, ...)
 *  @return Current lap
 */
uint8_t controller::get_cur_lap (uint8_t kart_num) {
   return game_model->get_lap (kart_num);
}

#ifdef USE_GUI
/** This method prints the occupancy grid onto the GUI.
 */
void controller::gui_occupancy (void) {
   Mat overlay, resized, *new_view;
   Vec3b *over_row, *main_row;
   char kart_label[2];
   int i,x,y,x_start,y_start;
   gridObject *cur_item;

   // Clear display

   overlay = imread ("./hudImages/star.jpg");
   new_view = new Mat (600,600,CV_8UC3);
   //resize (overlay, *new_view, Size (600,600), 0, 0);
   *new_view = Scalar (255,255,255);

//    view = Mat::ones (600, 600, CV_8UC3);

   // Label each kart
   for (i=0;i<num_karts;i++) {
      x = get_x (i+1) * 600 / game_grid->get_cols ();
      y = get_y (i+1) * 600 / game_grid->get_rows ();
      kart_label[0] = i+1+'0';
//       putText (view, kart_label, Point (x,y), FONT_HERSHEY_PLAIN, 2,
//                Scalar (255,255,255),3);
      putText (*new_view, kart_label, Point (x,y), FONT_HERSHEY_PLAIN, 2,
               Scalar (0,0,0),3);
   }
```

```
    // Add each item
    for (i=0;i<game_grid->listLength ();i++) {
        cur_item = game_grid->getFromList (i);
        switch (cur_item->getType ()) {
            case REDSHELL:
                overlay = imread ("./hudImages/redshell.jpg");
                break;
            case BANANA:
                overlay = imread ("./hudImages/banana.jpg");
                break;
            case GREENSHELL:
                overlay = imread ("./hudImages/greenshell.jpg");
                break;
            case STAR:
                overlay = imread ("./hudImages/star.jpg");
                break;
            case REDSHIELD3:
            case REDSHIELD2:
            case REDSHIELD1:
            case MRSHELL:
                overlay = imread ("./hudImages/multipleredshell.jpg");
                break;
            case ITEMBOX:
                overlay = imread ("./hudImages/itembox.jpg");
                break;
            case NO_ITEM:
            default:
                continue;
        }
        resize (overlay, resized, Size(), 0.05, 0.05);
        overlay.release ();
        x_start = cur_item->getX () * 600 / game_grid->get_cols ();
        y_start = cur_item->getY () * 600 / game_grid->get_rows ();
        for (y=0;y<resized.rows;y++) {
            if (y+y_start < new_view->rows) {
                over_row = resized.ptr<Vec3b>(y);
                main_row = new_view->ptr<Vec3b>(y+y_start);
                for (x=0;x<resized.cols;x++) {
                    if (x+x_start < new_view->cols &&
                        (over_row[x][0] != 255 || over_row[x][1] != 255 ||
                         over_row[x][2] != 255)) {
                            main_row[x+x_start] = over_row[x];
                        }
                }
            }
        }
        resized.release ();
    }
    view = *new_view;
}

/** This function is used to determine the location of mouse clicks on the
 *  GUI window.
 *
 *  @param event Mouse event, one of CV_EVENT_*
 *  @param x x-coordinate of mouse pointer
 *  @param y y-coordinate of mouse pointer
 *  @param flags Combination of CV_EVENT_FLAG_*
 *  @param param Parameter for callback function
 */
```

```
void mouse_handler (int event, int x, int y, int flags, void *param) {
    if (event == CV_EVENT_LBUTTONDOWN ||
        event == CV_EVENT_LBUTTONUP) {
        mouse_input_t *input = (mouse_input_t *)param;
        input->x = x;
        input->y = y;
        input->waiting = false;
    }
}

#endif
```

iii. grid.cpp

```
/** @title grid.cpp
 *  @description A grid to track objects for the game. I suck at descriptions.
 *  @author David Allender
 *  @author Joryl Calizo
 *  @date February 8, 2011
 */

#include "grid.h"

/** This constructor creates a new grid to store grid objects. The grid
 *  begins as a 2D pointer array with all boundary coordinates set to walls.
 *
 *  @param cols Number of columns
 *  @param rows Number of rows
 */
grid::grid (uint8_t cols, uint8_t rows) {
    // Initialize dimensions
    num_cols = cols;
    num_rows = rows;
    num_objs = num_cols*num_rows;

    // Create array
    objectGrid = new gridObject*[num_objs];

    // Begin with empty linked list
    objectList = new linked_list();

    // Begin with NULL pointers
    for (int i=0;i<num_objs;i++) {
        objectGrid[i] = NULL;
    }

    // Add walls at outermost coordinates
    for (int i=0;i<num_rows;i++) {
        addObject (new wallObject (0,i));
        addObject (new wallObject (num_cols-1,i));
    }
    for (int j=0;j<num_cols;j++) {
        addObject (new wallObject (j,0));
        addObject (new wallObject (j,num_rows-1));
    }

    // Reset object list (outer walls do not belong in list)
//    objectList->clear_list ();
    objectList = new linked_list();
}
```

```
/** This method checks if the coordinates inputted by the user are valid. This
 *  method is used prior to accessing the grid array to prevent segfaults and
 *  indexing incorrect addresses.
 *
 *  @param x x-coordinate
 *  @param y y-coordinate
 *  @return False if coordinates are out-of-bounds
 */
bool grid::is_valid (uint8_t x, uint8_t y) {
    if (x > num_cols) {
        return false;
    }
    if (y > num_rows) {
        return false;
    }
    return true;
}

/** This method adds a new object to the grid, if possible.
 *
 *  @param obj New object to be added to grid
 *  @return False if object cannot be added or coordinates are already occupied
 */
bool grid::addObject (gridObject* obj) {
    // Return false if object is NULL
    if (!obj) {
        return false;
    }

    // Get coordinates
    uint8_t x = obj->getX();
    uint8_t y = obj->getY();

    // Return false if object is out-of-bounds
    if (!is_valid(x,y)) {
        return false;
    }

    // Return false if coordinates are already occupied
    if (objectGrid[(num_cols * y + x)] != NULL) {
        return false;
    }

    // Assign object to location
    objectGrid[(num_cols * y + x)] = obj;

    // Add object to linked list
    objectList->add_to_tail (obj);
    return true;
}

/** This method moves an object to a new location on the grid, if possible.
 *
 *  @param obj Object to be moved
 *  @param x New x-coordinate
 *  @param y New y-coordinate
 *  @return False if object does not exist, is a wall, or is trying to move
 *      to occupied coordinates
 */
bool grid::moveObject (gridObject* obj, uint8_t x, uint8_t y) {
    // Return false if object is NULL
```

```
    if (!obj) {
        return false;
    }

    // Get coordinates
    uint8_t x_old = obj->getX();
    uint8_t y_old = obj->getY();

    // Return false if coordinates are the same
    if (x_old == x && y_old == y) {
        return false;
    }

    // Return false if old or new coordinates are out-of-bounds
    if (!is_valid(x,y) || !is_valid(x_old,y_old)) {
        return false;
    }

    // Return false if coordinates are already occupied
    if (objectGrid[(num_cols * y + x)] != NULL) {
        return false;
    }

    // Assign object to new location
    obj->setCoord (x,y);
    objectGrid[(num_cols * y + x)] = obj;
    objectGrid[(num_cols * y_old + x_old)] = NULL;

    return true;
}

/** This method allows access to a one-dimensional array using both an x and
 *  y-coordinate.
 *
 *  @param x x-coordinate
 *  @param y y-coordinate
 *  @return Object at given coordinates
 */
gridObject* grid::get (uint8_t x, uint8_t y) {
    return objectGrid[(num_cols * y + x)];
}

/** This method gets a grid object from the linked list.
 *
 *  @param index Index of object in linked list
 *  @return Object at index
 */
gridObject* grid::getFromList (uint8_t index) {
    if (index >= objectList->get_length()) { // invalid index
        return NULL;
    }
    return objectList->get(index);
}

/** This method removes an object pointer from the grid, if possible.
 *
 *  @param obj Object to be removed from grid
 *  @return False if object does not exist (or is a wall)
 */
bool grid::removeObject (gridObject* obj) {
    if (!obj) {
```

```
        return false;
    }
    // Get coordinates
    uint8_t x = obj->getX();
    uint8_t y = obj->getY();

    // Return false if coordinates are out-of-bounds
    if (!is_valid(x,y)) {
        return false;
    }

    if (obj->sameObject(objectGrid[(num_cols * y + x)])) {
//        delete objectGrid[(num_cols * y + x)];
        objectGrid[(num_cols * y + x)] = NULL;
        objectList->remove_node (obj);
        return true;
    }

    cout << obj->getTypeString() << " cannot be removed." << endl;
    cout << "Located at (" << x << "," << y << ")" << endl;
    if (isOccupied (x,y)) {
        cout << "(" << x << "," << y << ") is not occupied." << endl;
    }
    return false;
}

/** This method displays a list of all objects in the linked list.
 */
void grid::printList (void) {
    for (int i=0;i<objectList->get_length();i++) {
        cout << objectList->get(i) << endl;
    }
}

/** This method checks if a location is occupied.
 *
 *  @param x_coord x-coordinate
 *  @param y_coord y-coordinate
 *  @return True if location is occupied
 */
bool grid::isOccupied (uint8_t x, uint8_t y) {
    if (objectGrid[(num_cols * y + x)] != NULL) {
        return true;
    }
    return false;
}

/** This method returns the number of objects in the linked list.
 *
 *  @return Number of objects in list
 */
uint8_t grid::listLength (void) {
    return objectList->get_length();
}


/** This method displays a text-based representation of the grid. An 'x'
 *  corresponds to a wall, while a '.' corresponds to an unoccupied
 *  location. Any other grid object is labeled by its ID number.
 */
void grid::displayOccupancy (void) {
```

```
    gridObject* cur;
    for (int i=0;i<num_rows;i++) {
        for (int j=0;j<num_cols;j++) {
            cur = objectGrid[(num_cols * i + j)];
            if (dynamic_cast<wallObject*> (cur)) {
                cout << setw(4) << "x";
            }
            else if (cur == NULL) {
                cout << setw(4) << ".";
            }
            else {
                cout << setw(4) << (int)cur->getID();
            }
        }
        cout << endl;
    }
}

/** This method displays a text-based representation of the grid. An 'x'
 *  corresponds to a wall, while a '.' corresponds to an unoccupied
 *  location. Any other grid object is labeled by its ID number.
 *
 *  @param scale Scaling factor used when printing to cout (0 to 100%)
 */
void grid::displayOccupancy (uint8_t scale) {
    gridObject* cur;
    uint8_t dx, dy;
    bool found_first;
    if (scale >= 100) { // saturate at 100%, call normal method
        displayOccupancy();
    }
    else if (scale){
        dx = num_cols / scale;
        dy = num_rows / scale;
        for (int i=0;i<num_rows-dy;i+=dy) {
            for (int j=0;j<num_cols-dx;j+=dx) {
                found_first = false;
                for (int k=i;k<i+dy;k++) {
                    for (int l=j;l<j+dx;l++) {
                        if (objectGrid[num_cols * k + l] != NULL &&
                            !found_first) {
                            cur = objectGrid[num_cols * k + l];
                            found_first = true;
                            break;
                        }
                    }
                    if (found_first) {
                        break;
                    }
                }
                if (!found_first) {
                    cout << setw(4) << ".";
                }
                else if (dynamic_cast<wallObject*> (cur)) {
                    cout << setw(4) << "x";
                }
                else {
                    cout << setw(4) << (int)cur->getID();
                }
            }
            cout << setw(4) << "x" << endl;
```

```
        }
        for (int i=0;i<num_rows;i+=dy) {
            cout << setw(4) << "x";
        }
    }
    cout << endl;
}

/** This method returns the number of columns.
 *
 *  @return Number of columns
 */
unsigned int grid::get_cols (void) {
    return num_cols;
}

/** This method returns the number of rows.
 *
 *  @return Number of rows
 */
unsigned int grid::get_rows (void) {
    return num_rows;
}

ostream& operator<<(ostream& out, gridObject* my_obj) {
    if (!my_obj) {
        return out;
    }
    // Print generic grid object stuff
    out << "Object " << my_obj->getID() << ": " << my_obj->getTypeString()
    << endl;
    out << "Coordinates: (" << my_obj->getX() << ", "
    << my_obj->getY() << ")" << endl;
    out << "Size: " << my_obj->getSize() << endl;
    out << "Strength: " << my_obj->getStrength() << endl;
    // Print mobile grid object stuff
    if (dynamic_cast<mobileGridObject*> (my_obj)) {
        out << "Speed: " << my_obj->getSpeed() << endl;
        out << "Heading: " << my_obj->getDirection() << endl;
    }
    // Print kart object stuff
    if (dynamic_cast<kartObject*> (my_obj)) {
        out << "Item: " << my_obj->getItemString() << endl;
        out << "Shield: ";
        if (!my_obj->hasShield()) {
            out << "None." << endl;
        }
        else {
            out << my_obj->getShieldType();
            if (my_obj->getShieldType() != STAR) {
                out << " x" << my_obj->getShieldCount() << endl;
            }
        }
    }
    out << endl;
    return out;
}

ostream& operator<<(ostream& out, uint8_t x) {
    out << (int)x;
    return out;
```

```
}

ostream& operator<<(ostream& out, int8_t x) {
    out << (int)x;
    return out;
}

ostream& operator<<(ostream& out, item_values_t item) {
    switch (item) {
        case NO_ITEM:
            out << "No Item";
            break;
        case KART:
            out << "Kart";
            break;
        case GREENSHELL:
            out << "Green Shell";
            break;
        case REDSHELL:
            out << "Red Shell";
            break;
        case GREENSHIELD3:
            out << "Green Shell x3";
            break;
        case GREENSHIELD2:
            out << "Green Shell x2";
            break;
        case GREENSHIELD1:
            out << "Green Shell x1";
            break;
        case REDSHIELD3:
            out << "Red Shell x3";
            break;
        case REDSHIELD2:
            out << "Red Shell x2";
            break;
        case REDSHIELD1:
            out << "Red Shell x1";
            break;
        case BANANA:
            out << "Banana";
            break;
        case MUSHROOM:
            out << "Mushroom";
            break;
        case STAR:
            out << "Star";
            break;
        case ITEMBOX:
            out << "Item Box";
            break;
        case WALL:
            out << "Wall";
            break;
        default:
            out << "Oh shit (" << (int)item << ").";
            break;
    }
    return out;
}
```

iv. gridObject.cpp

```
/** @title gridObject.cpp
 *  @description Object to be located on a grid. Obviously.
 *  @author David Allender
 *  @author Joryl Calizo
 *  @date February 1, 2011
 */

#include "gridObject.h"

/** This constructor creates a new generic grid object. The size of the object
 *  is represented as a radius and centered at the specified x,y-coordinates.
 *
 *  @param x_coord x-coordinate
 *  @param y_coord y-coordinate
 *  @param radius Size of object
 *  @param strength Strength of object
 *  @param type Object type
 */
gridObject::gridObject (uint8_t x_coord, uint8_t y_coord, uint8_t radius,
                        uint8_t strength, item_values_t type) {
    static uint8_t count = 0;   // number of unique grid objects
    x = x_coord;
    y = y_coord;
    size = radius;
    this->strength = strength;
    ID = ++count;         // generates a new ID everytime
    this->type = type;
}

/** This method returns the x and y coordinates of the object as a combined
 *  16-bit integer. This can be used directly in the data packet sent to each
 *  go-kart. The higher-order bits represent the x-coordinate, while the lower-
 *  order bits represent the y-coordinate.
 *
 *  @return Object coordinates
 */
uint16_t gridObject::getCoords(void) {
    return (x << 8) + y;
}

/** This method returns the x-coordinate only.
 *
 *  @return x-coordinate
 */
uint8_t gridObject::getX(void) {
    return x;
}

/** This method returns the y-coordinate only.
 *
 *  @return y-coordinate
 */
uint8_t gridObject::getY(void) {
    return y;
}

/** This method returns the size of the object.
 *
```

```
 *  @return Object size
 */
uint8_t gridObject::getSize(void) {
    return size;
}

/** This method returns the strength of the object.
 *
 *  @return Object strength
 */
uint8_t gridObject::getStrength(void) {
    return strength;
}

/** This method returns the unique ID of the object.
 *
 *  @return Object ID
 */
uint16_t gridObject::getID(void) {
    return ID;
}

/** This method returns the object type.
 *
 *  @return Object type
 */
item_values_t gridObject::getType(void) {
    return type;
}

/** This method returns the string representation of the object type.
 *
 *  @return String representation of object type
 */
const char* gridObject::getTypeString(void) {
    switch (type) {
        case NO_ITEM:
            return "No Item";
        case KART:
            return "Kart";
        case GREENSHELL:
            return "Green Shell";
        case REDSHELL:
            return "Red Shell";
        case GREENSHIELD3:
            return "Green Shell x3";
        case GREENSHIELD2:
            return "Green Shell x2";
        case GREENSHIELD1:
            return "Green Shell x1";
        case REDSHIELD3:
            return "Red Shell x3";
        case REDSHIELD2:
            return "Red Shell x2";
        case REDSHIELD1:
            return "Red Shell x1";
        case BANANA:
            return "Banana";
        case MUSHROOM:
            return "Mushroom";
        case STAR:
```

```
            return "Star";
        case ITEMBOX:
            return "Item Box";
        case WALL:
            return "Wall";
        default:
            return "Oh shit.";
    }
}

/** This method changes the object's coordinates. Currently, there is no reason
 *  to return "false," but I'm sure we'll think of something.
 *
 *  @param x_coord New x-coordinate
 *  @param y_coord New y-coordinate
 *  @return False if coordinates cannot be changed
 */
bool gridObject::setCoord(uint8_t x_coord, uint8_t y_coord) {
    x = x_coord;
    y = y_coord;
    return true;
}

/** This method changes the object's size. I guess this can be used for
 *  animation purposes? Always returns true (for now).
 *
 *  @param size New object size
 *  @return False if size cannot be changed
 */
bool gridObject::setSize(uint8_t size) {
    this->size = size;
    return true;
}

/** This method changes the object's strength. Always returns true (for now).
 *
 *  @param strength New object strength
 *  @return False if strength cannot be changed
 */
bool gridObject::setStrength(uint8_t strength) {
    this->strength = strength;
    return true;
}

/** This method changes the object ID. Always returns true (for now).
 *
 *  @param id New object ID
 *  @return False if ID cannot be changed
 */
bool gridObject::setID(uint16_t ID) {
    this->ID = ID;
    return true;
}

/** This method returns the distance between this object and another object
 *  on the grid.
 *
 *  @param other Pointer to second object
 *  @return Distance from second object
 */
uint16_t gridObject::getDistance(gridObject* other) {
```

```
    int16_t x_diff = other->x - this->x;
    int16_t y_diff = this->y - other->y;
    return sqrt(x_diff*x_diff + y_diff*y_diff);
}

/** This method returns the angle towards a second object on the grid.
 *
 *  @param other Pointer to second object
 *  @return Angle to second object
 */
uint16_t gridObject::getAngle(gridObject* other) {
    int16_t heading = atan2 (this->y-other->y,other->x-this->x) * 180 / PI;
    if (heading < 0) {
        heading = heading + 360;
    }
    return (uint16_t)heading;
}

/** This method checks to see if the second object is actually the same (to
 *  avoid false collisions).
 *
 *  @param object Pointer to second object
 *  @return True if objects are the same
 */
bool gridObject::sameObject(gridObject* other) {
    return other == this;
}

/* Virtual methods */
uint8_t gridObject::getSpeed(void) {
    return 0;
}

uint16_t gridObject::getDirection(void) {
    return 0;
}

bool gridObject::setSpeed(uint8_t speed) {
    return false;
}

bool gridObject::setDirection(uint16_t heading) {
    return false;
}

item_values_t gridObject::getItem(void) {
    return NO_ITEM;
}

const char* gridObject::getItemString(void) {
    return "I'm not a kart.";
}

gridObject* gridObject::useItem() {
    return NULL;
}

bool gridObject::setItem(item_values_t item) {
    return false;
}
```

```
bool gridObject::starPowerStart(void) {
   return false;
}

bool gridObject::starPowerStop(void) {
   return false;
}

bool gridObject::setTargetCoord (uint8_t x_coord, uint8_t y_coord) {
   return false;
}

bool gridObject::setTargetObject(gridObject* target) {
   return false;
}

bool gridObject::hasShield (void) {
   return false;
}

item_values_t gridObject::getShieldType(void) {
   return NO_ITEM;
}

uint8_t gridObject::getShieldCount(void) {
   return 0;
}

bool gridObject::setShield(item_values_t shield_type) {
   return false;
}

bool gridObject::useShield(void) {
   return false;
}

//uint16_t gridObject::getRelativeAngle (gridObject* other) {
//    return 0;
//}

int16_t gridObject::getRelativeX (gridObject* other) {
   return 0;
}

int16_t gridObject::getRelativeY (gridObject* other) {
   return 0;
}
```

v. kartObject.cpp

```
/** @title kartObject.cpp
 *  @description A go-kart grid object. Descendant of the mobileGridObject
 *      class.
 *  @author David Allender
 *  @author Joryl Calizo
 *  @date February 7, 2011
 */

#include "kartObject.h"
```

```
/** This constructor creates a new go-kart object. Aside from the standard
 *  mobile object parameters, the go-kart can also start with an item.
 *
 *  @param x_coord x-coordinate
 *  @param y_coord y-coordinate
 *  @param heading Compass direction of go-kart
 *  @param item Item held (0x00 for no item)
 */
kartObject::kartObject (uint8_t x_coord, uint8_t y_coord, uint16_t heading,
                        item_values_t item)
:mobileGridObject(x_coord, y_coord, KART_SIZE, KART_STRENGTH, 0, heading,
                  KART) {
   this->item = item;
   shield.item = NO_ITEM;
   shield.num = 0;
}

/** This method returns the value of the current item held by the go-kart. See
 *  items.h for item values.
 *
 *  @return Item held
 */
item_values_t kartObject::getItem(void) {
   return item;
}

/** This method returns the string represenation of the current item held by
 *  the go-kart.
 *
 *  @return String represenation of item held
 */
const char* kartObject::getItemString(void) {
   switch (item) {
      case NO_ITEM:
         return "No Item";
      case KART:
         return "Kart";
      case GREENSHELL:
         return "Green Shell";
      case REDSHELL:
         return "Red Shell";
      case GREENSHIELD3:
         return "Green Shell x3";
      case GREENSHIELD2:
         return "Green Shell x2";
      case GREENSHIELD1:
         return "Green Shell x1";
      case REDSHIELD3:
         return "Red Shell x3";
      case REDSHIELD2:
         return "Red Shell x2";
      case REDSHIELD1:
         return "Red Shell x1";
      case BANANA:
         return "Banana";
      case MUSHROOM:
         return "Mushroom";
      case STAR:
         return "Star";
      default:
         return "Oh shit.";
```

```
    }
}

/** This method "uses" the item held by the go-kart. After calling this method,
 *  the go-kart is unarmed. See items.h for item values.
 *
 *  @return Pointer to new grid object if one is created
 */
gridObject* kartObject::useItem() {
    gridObject* p_new;  // Create pointer to new object
    // Determine location of new object
    int16_t x_new;
    int16_t y_new;

    if (hasShield()) { // Kart has shield, use first
        switch (shield.item) {
            case GREENSHELL:
            case REDSHELL:
                x_new = round(x+DX*cos(heading*PI/180));
                y_new = round(y-DY*sin(heading*PI/180));

                if (x_new < 0) {    // Boundary conditions
                    x_new = 0;
                }
                if (y_new < 0) {
                    y_new = 0;
                }
                if (shield.item == GREENSHELL) {
                    p_new = new greenShellObject (x_new,y_new,heading);
                }
                else if (shield.item == REDSHELL) {
                    p_new = new redShellObject (x_new,y_new,heading,NULL);
                }
                useShield();
                return p_new;
            case STAR: // Continue if current item is not shield
                switch (item) {
                    case GREENSHIELD3:
                    case REDSHIELD3:
                    case STAR:
                        return NULL;
                    default:
                        break;
                }
                break; // if this point is reached, item is ok to use
            default: // invalid shield, how'd you get here?
                return NULL;
        }
    }

    if (item == NO_ITEM) { // Kart has no item
        return NULL;
    }

    switch (item) {
        case GREENSHELL:
        case REDSHELL:
            x_new = round(x+DX*cos(heading*PI/180));
            y_new = round(y-DY*sin(heading*PI/180));

            if (x_new < 0) {    // Boundary conditions
```

```
            x_new = 0;
        }
        if (y_new < 0) {
            y_new = 0;
        }
        if (item == GREENSHELL) {
            p_new = new greenShellObject (x_new,y_new,heading);
        }
        else if (item == REDSHELL) {
            p_new = new redShellObject (x_new,y_new,heading,NULL);
        }
        item = NO_ITEM;
        return p_new;
    case BANANA:
        x_new = round(x-DX*cos(heading*PI/180));
        y_new = round(y+DY*sin(heading*PI/180));
        if (x_new < 0) {    // Boundary conditions
            x_new = 0;
        }
        if (y_new < 0) {
            y_new = 0;
        }
        p_new = new bananaObject (x_new,y_new);
        item = NO_ITEM;
        return p_new;
    case GREENSHIELD3:
        setShield(GREENSHELL);
        item = NO_ITEM;
        return NULL;
    case REDSHIELD3:
        setShield(REDSHELL);
        item = NO_ITEM;
        return NULL;
    case MUSHROOM:
        item = NO_ITEM;
        return NULL;
    case STAR:
        starPowerStart();
        return NULL;
    default: // Invalid value
        return NULL;
  }
}

/** This method gives the go-kart a new item, if possible. See items.h for item
 *  values.
 *
 *  @param item New item
 *  @return False if go-kart already has an item
 */
bool kartObject::setItem(item_values_t item) {
  if (this->item) {
    return false;
  }
  this->item = item;
  return true;
}

/** This method tells the go-kart to take steroids, raising its strength.
 *  After calling this method, the go-kart is unarmed. See items.h for item
 *  values.
```

```
 *  NOTE: A thread must be created to track length of star power.
 *
 *  @return False if go-kart does not have a star
 */
bool kartObject::starPowerStart(void) {
    if (item != STAR) {
        return false;
    }
    item = NO_ITEM;
    strength = STAR_STRENGTH;
    shield.item = STAR;
    shield.num = STAR;
    return true;
}

/** This method tells the go-kart that steroids are unhealthy. The go-kart is
 *  returned to base strength.
 *  NOTE: Will be called by a terminating thread when star power is exhausted.
 *
 *  @return False if go-kart already stopped taking steroids
 */
bool kartObject::starPowerStop(void) {
    if (shield.item != STAR) {
        return false;
    }
    strength = KART_STRENGTH;
    shield.item = NO_ITEM;
    shield.num = 0;
    return true;
}

/** This method checks whether or not the go-kart has a shield.
 *
 *  @return True if go-kart has a shield
 */
bool kartObject::hasShield(void) {
    if (shield.num) {
        return true;
    }
    return false;
}

/** This method returns the shield type. If the go-kart has no shield, this
 *  method returns NO_ITEM.
 *
 *  @return Shield type
 */
item_values_t kartObject::getShieldType(void) {
    return shield.item;
}

/** This method returns the number of shields protecting the go-kart.
 *
 *  @return Number of shields
 */
uint8_t kartObject::getShieldCount(void) {
    return shield.num;
}

/** This method gives the go-kart a shield if it does not currently have one.
 *
```

```
 *  @param shield_type Shield type
 *  @return False if already has a shield or shield type is invalid
 */
bool kartObject::setShield(item_values_t shield_type) {
    if (hasShield()) {
        return false;
    }
    switch (shield_type) {
        case STAR:
            shield.item = STAR;
            shield.num = STAR;
            break;
        case GREENSHELL:
        case GREENSHIELD3: // just in case
            shield.item = GREENSHELL;
            shield.num = 3;
            break;
        case REDSHELL:
        case REDSHIELD3:
            shield.item = REDSHELL;
            shield.num = 3;
            break;
        default: // invalid type
            return false;
    }
    return true;
}

/** This method uses one of the go-kart's shields. Go-karts with stars remain
 *  unchanged.
 *
 *  @return False if go-kart does not have a shield
 */
bool kartObject::useShield(void) {
    switch (shield.num) {
        case 1: // no more shield
            shield.item = NO_ITEM;
        case 2:
        case 3:
            shield.num--;
        case STAR: // do nothing for stars
            return true;
        default:
            return false;
    }
}

/** This method returns the angle towards a second object on the grid
 *  relative to the kart's position and heading.
 *
 *  @param other Pointer to second object
 *  @return Relative angle to second object
 */
uint16_t kartObject::getRelativeAngle (gridObject* other) {
    if (sameObject(other)) {
        return 0;
    }
    int16_t angle = 90 - (heading - getAngle (other));
    if (angle < 0) {
        angle += 360;
    }
```

```
    if (angle >= 360) {
        angle -= 360;
    }
    return (uint16_t)angle;
}

/** This method returns the x-coordinate of a second object on the grid
 *  relative to the kart's position and heading.
 *
 *  @param other Pointer to second object
 *  @return Relative x-coordinate to second object
 */
int16_t kartObject::getRelativeX (gridObject* other) {
    if (sameObject(other)) {
        return 0;
    }
    return round(getDistance(other) * cos(getRelativeAngle(other)*PI/180));
}

/** This method returns the y-coordinate of a second object on the grid
 *  relative to the kart's position and heading.
 *
 *  @param other Pointer to second object
 *  @return Relative y-coordinate to second object
 */
int16_t kartObject::getRelativeY (gridObject* other) {
    if (sameObject(other)) {
        return 0;
    }
    return round(getDistance(other) * sin(getRelativeAngle(other)*PI/180));
}
```

vi. networks.cpp

```
/** @title networks.cpp
 *  @brief This is the networks file, where all of the inter-device communication happens
 *  @details This file represents the model for a model-view-controller (MVC)
 *      architecture. Public methods are called by the controller when changes
 *      to the game state need to be made. Other query methods are used by the
 *      view to update local game state data displayed on the user interface.
 *  @author David Allender
 *  @author Joryl Calizo
 *  @date April 10, 2011
 */

#include "networks.h"

using namespace std;

/** This constructor creates a new networking object.
 *
 *  @param type The type of network object(1 for KART, 2 for SERVER)
 *  @param player This is the player number(port changes according to this number)
 */
networking::networking (int type, int player) {
    socket_addrinfo = (struct sockaddr_in *)malloc(sizeof(struct sockaddr_in));
    len = sizeof(*socket_addrinfo);
    if (type == NETWORK_KART) {
        kart_setup(player);
    }
    else if (type == NETWORK_SERVER){
```

```
        server_setup(player);
        //connect();
    }
    else{
        cerr << "ERROR: Wrong type for network object(1 for Kart, 2 for Server" << endl;
    }
}

/** This is the networking setup code for the server
 *
 *  @param player This is the player number you wish to connect to
 */
void networking::server_setup(int player){

    if((sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0){
        perror("Socket Call");
        exit(-1);
    }

    memset(socket_addrinfo, 0, sizeof(*socket_addrinfo));
    //set up the sock
    socket_addrinfo->sin_family = AF_INET;
    socket_addrinfo->sin_addr.s_addr = htonl(INADDR_ANY);
    //choose what port to use
    socket_addrinfo->sin_port = htons(PORTNUMBER + (player - 1));
    //bind the name (address) to a port
    if(bind(sock,(struct sockaddr*)socket_addrinfo, sizeof(*socket_addrinfo)) < 0){
        perror("bind call");
        exit(-1);
    }

    if(getsockname(sock,(struct sockaddr*)socket_addrinfo,&len) < 0){
        perror("setsockname call");
        exit(-1);
    }
    printf("Player %d has port %d \n", player, ntohs(socket_addrinfo->sin_port));
}

/** This is the networking setup code for the kart
 *
 *  @param player This is the Player Number
 */
void networking::kart_setup(int kartnum){
    struct hostent *hp;

    if((sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0){
        perror("Socket Call");
        exit(-1);
    }

    socket_addrinfo->sin_family = AF_INET;
    //What port do you want to connect to?

    socket_addrinfo->sin_port = htons(PORTNUMBER + (kartnum - 1));

    //s_addrinfo.sin_addr.s_addr = htonl(INADDR_ANY);
    //The address below is the IP address of the SERVER, the
    //kart uses this to connect
    hp = gethostbyname("192.168.1.101");
    //memcpy(&s_addrinfo.sin_addr, hp.h_addr, hp.h_length);
    bcopy(hp->h_addr, &(socket_addrinfo->sin_addr.s_addr), hp->h_length);
```

```
}

/** This method sends data, be it from kart to server or server to kart, it doesn't matter
 *
 *  @param player This is the buffer if information to send
 *  @return the number of bytes actually sent(used for debug)
 */
int networking::send_data(char *buffer){
    return sendto(sock, buffer, strlen(buffer), 0,
            (struct sockaddr *)socket_addrinfo, sizeof(*socket_addrinfo));
}

/** This method sends data, be it from kart to server or server to kart, it doesn't matter
 *
 *  @param player This is the buffer if information to recieve from(max recv length is 150)
 *  @return the number of bytes actually received(used for debug)
 */
int networking::recv_data(char *buffer){
    return recvfrom(sock, buffer, MAX_PACKET_SIZE, 0,
            (struct sockaddr *)socket_addrinfo, &len);

}

/**
 * This method gets called on the server end, and waits for a packet to arrive from the kart
 */
void networking::connect(){
    char* inpacket = (char*)malloc(50);
    recvfrom(sock, inpacket, 50, 0,
            (struct sockaddr *)socket_addrinfo, &len);
    cout << "Kart connected: " << endl;
    memset(inpacket, 0, 50);
    strcpy(inpacket, "Connected to server!");
    send_data (inpacket);
    free(inpacket);
}
```

vii. kartHud.cpp

```
#include "kartHud.h"
#include "items.h"

/*************public functions***************/
/*
 * constructor
 */
void kartHud::init(int pos, int totalLap, int cam)
{
    hud::init(cam);
    this->pos = pos;
            wep = NO_ITEM;
    currLap = 0;
    this->totalLap = totalLap;
    raceTime = 0;
}


int kartHud::getWep()
{
    return wep;
```

```
}
int kartHud::getPos()
{
    return pos;
}
int kartHud::getLap()
{
    return currLap;
}


/*
 *  function used to change huds status
 * pos - pos is change if not -1
 * currlap - changed if not -1
 * wep - huds weapon is set to this weapon if it is not -1
 * totallap - huds totallap is changed if it is not -1;
 * elapsedtime - huds elapsed time is change if it not set to -1
 */
void kartHud::changeHud(int pos, int currLap, int wep, int totalLap)
{

    if(pos!= -1)
    {
        this->pos = pos;
    }

    if(currLap != -1)
    {
        this->currLap = currLap;
    }

    if(wep != -1)
    {
        this->wep = wep;
        //wepBuff = wep;
    }
    if(totalLap != -1)
    {
        this->totalLap = totalLap;
    }


}

void kartHud::setRaceTime(int time)
{
    raceTime = time;
}

/*************private functions**************/

void kartHud::updateFrame()
{
    placePos(currFrame, pos);

    placeWep();
    placeTime((raceTime/60000)%60, (raceTime/1000) %60,(raceTime%1000));

    placeLap(currFrame, currLap, totalLap);
}
```

```
/*
 * shows what weapon a person has on the HUD
 */
bool kartHud::placeWep()
{
    IplImage *original = NULL;

    if(wep == MRSHELL)
    {
        original = cvLoadImage("./hudImages/multipleredshell.jpg");
    }else if(wep == REDSHELL)
    {
        original = cvLoadImage("./hudImages/redshell.jpg");
    }else if(wep == BANANA)
    {
        original = cvLoadImage("./hudImages/banana.jpg");
    }else if(wep == MUSHROOM)
    {
        original = cvLoadImage("./hudImages/mushroom.jpg");
    }else if(wep == STAR)
    {
        original = cvLoadImage("./hudImages/star.jpg");
    }else if(wep == GREENSHELL)
    {
        original = cvLoadImage("./hudImages/greenshell.jpg");
    }else if(wep == MGSHELL)
    {
        original = cvLoadImage("./hudImages/multiplegreenshell.jpg");
    }

    if(!original)
    {
        return false;
    }

    cvFlip(original);
    placeImage(currFrame, original, 20, 5, 70);

    cvReleaseImage(&original);
    return true;
}

/* function is to place two images on top of each other
 * the background image is the anchor image and if the two resolutions
 * are not the same then the foreground will be set to the background
 * image resolution.
 *Args: background- background image
 *      foreground - foreground image
 *      shrink - percent of size the foreground image should change too
 *      xoffset - percent of offset in x the foreground image should be moved too
 *      yoffset - percent of offset in y the foreground image should be moved too
 */
void kartHud::placeImage(IplImage *background, IplImage *foreground,
        double size, double xoffset, double yoffset)
{
    IplImage * newsize;
    int i, j;
    int offseti, offsetj;
    CvScalar pixel;

    offseti = (int)((background->height * yoffset)/100);
```

```
    offsetj = (int)((background->width * xoffset)/100);


    newsize = cvCreateImage(cvSize((int)((background->width*size)/100), (int)((background->width*size)/100)),
            background->depth, foreground->nChannels);
    cvResize(foreground, newsize);

    for(i = 0; i < newsize->height; i++)
    {
        for(j=0; j< newsize->width; j++)
        {
            if(i+offseti < background->height &&
                    j+offsetj < background->width)
            {
                pixel = cvGet2D(newsize, i, j);
                cvSet2D(background, i+offseti, j+offsetj, pixel);
            }

        }
    }

    cvReleaseImage(&newsize);
}



/*
 * function the places a person position on the HUD
 */
void kartHud::placePos(IplImage *frame, int pos)
{
    int i, j;
    IplImage *original;
    IplImage * place;
    if(pos == 1)
    {
        original = cvLoadImage("./hudImages/1st.jpg");
    }else
    {
        original = cvLoadImage("./hudImages/2nd.jpg");
    }

    if(!original)
    {
        return;
    }
    place = cvCreateImage(cvSize((int)(frame->width *.20), (int)(frame->height*.20)), frame->depth, original->nChannels);
    cvResize(original, place);
    cvFlip(place, place, 0);

    for(i = (place->height - 1); i >= 0; i--)
    {
        for(j=0; j< place->width; j++)
        {
            CvScalar v = cvGet2D(place, i, j);
            if(v.val[0] < 200)
            {
                cvSet2D(frame, i,j, v);
            }
        }
```

```
    }

    cvReleaseImage(&place);
    cvReleaseImage(&original);

}


void kartHud::placeTime(int minutes, int seconds, int milliseconds)
{



    string time = "TIME";
    string mins;
    string secs;
    string millis;
    stringstream convInt;

    CvScalar color;
    color = cvScalar(40,238,204);

    CvFont font;
    double hScale=1.0;
    double vScale=1.0;
    int posx, posy;
    int    lineWidth= 3;



    /*convert ints to string*/
    convInt.str("");
    convInt << minutes;
    mins = convInt.str();

    convInt.str("");
    convInt << seconds;
    secs= convInt.str();;

    convInt.str("");
    convInt << milliseconds;
    millis= convInt.str();

    if(milliseconds < 100)
    {
        millis = "0" + millis;
        if(milliseconds < 10)
        {
            millis = "0" + millis;
        }

    }

    if(seconds < 10)
    {
        secs = "0" + secs;

    }
```

```
    if(minutes < 10)
    {
        mins = "0" + mins;

    }



    posx = currFrame->width;
    posx = (int)posx *.6;
    posy = currFrame->height;
    posy = (int)(posy * .07);



    /*font initialization */
    cvInitFont(&font,CV_FONT_HERSHEY_SIMPLEX|CV_FONT_ITALIC, hScale,vScale,0,lineWidth);

    cvFlip(currFrame, 0);

    //Time
    cvPutText (currFrame, time.c_str(),cvPoint(posx,posy), &font, color);
    posx = posx + (int)(currFrame->width*.125);
    // 00:
    cvPutText (currFrame, mins.c_str(),cvPoint(posx,posy), &font, color);
    posx = posx + (int)(currFrame->width*.07);
    cvPutText (currFrame, ":",cvPoint(posx,posy), &font, color);
    // 00:
    posx = posx + (int)(currFrame->width*.02);
    cvPutText (currFrame, secs.c_str(),cvPoint(posx,posy), &font, color);
    posx = posx + (int)(currFrame->width*.07);
    cvPutText (currFrame, ":",cvPoint(posx,posy), &font, color);
    // 000
    posx = posx + (int)(currFrame->width*.02);
    cvPutText (currFrame, millis.c_str(),cvPoint(posx,posy), &font, color);

    cvFlip(currFrame, 0);

}

void kartHud::placeLap(IplImage *frame, int currlap, int totallap)
{



    string lap = "LAP";
    string curr;
    string total;
    stringstream convInt;

    CvScalar color;
    color = cvScalar(40,238,204);

    CvFont font;
    double hScale=1.0;
    double vScale=1.0;
    int posx, posy;
    int    lineWidth= 3;
```

```
    convInt << currlap;
    curr = convInt.str();
    convInt.str("");
    convInt << totallap;
    total = convInt.str();




    posx = frame->width;
    posx = (int)posx *.7;
    posy = frame->height;
    posy = (int)(posy * .15);



    /*font initialization */
    cvInitFont(&font,CV_FONT_HERSHEY_SIMPLEX|CV_FONT_ITALIC, hScale,vScale,0,lineWidth);

    cvFlip(frame, 0);

    //lap
    cvPutText (frame, lap.c_str(),cvPoint(posx,posy), &font, color);
    posx = posx + (int)(frame->width*.12);

    // 00:
    cvPutText (frame, curr.c_str(),cvPoint(posx,posy), &font, color);
    posx = posx + (int)(frame->width*.03);
    cvPutText (frame, "/",cvPoint(posx,posy), &font, color);
    // 00:
    posx = posx + (int)(frame->width*.035);
    cvPutText (frame, total.c_str(),cvPoint(posx,posy), &font, color);

    cvFlip(frame, 0);

}
```

viii. kartNetworks.cpp

```
/**
 *  CPE461 Winter 2011
 *  @author: David Allender
 *  @author: Joseph Abad
 *
 *  This is to be used on the actual Go-Kart
 *
 *  It's responsibilities include:
 *   *Getting raw coordinates from the Arduino
 *   *Sending the raw coordinates back to the main computer
 *   *Receiving the data and passing it to Joseph Abad
 */

#include "kartNetworks.h"


using namespace std;
```

```
void kartNetworks::init()
{
    char *buffer;
    buffer = (char *) malloc(sizeof(char *));
    send_data(buffer);


}

void kartNetworks::step()
{
    char *recPacket;

    recv_data(recPacket);


}

void parsePacket(char *inpacket)
{
    //cout << "sending initial packet" << endl;
    char *tok;
    int tokCount = 0;
    int wep;
    int shield;
    int lap;
    int place;
    int type;
    int speed;
    int key = 0;
    double x;
    double y;
    bool hasShield = false;
    pthread_t gwepThread;
    tok = strtok(inpacket, " ,");

    while(tok)
    {
        //printf("parsing packet (%d): %s\n", tokCount, tok);
        if(tokCount == 1)
        {
            //currentWeapon being held
            wep = atoi(tok);
            if(wep == NOWEP  )
            {
                changeHud(-1, -1, NOWEP, -1, -1);
            }else if(getWep() != wep)
            {
                setWeapon(wep);
                pthread_create(&gwepThread, NULL, getWeapon, (void *) &wep);
            }


        }else if(tokCount == 2)
        {
            //current shield
            hasShield = false;
            shield = atoi(tok);
            if(shield == STAR)
            {
```

```
        setStar(true);

    }else if(shield == RSHELL)
    {
        hasShield = true;
    }else
    {
        setStar(false);
    }

}else if(tokCount == 3)
{
    //shield count
    if(hasShield)
    {
        setStar(false);
        setShield(atoi(tok));
    }else
    {
        setShield(0);
    }

}else if(tokCount == 4)
{
    //lap No.
    lap = atoi(tok);
    if(getLap() != lap)
    {
        changeHud(-1, atoi(tok), -1, -1, -1);
    }
    if(!raceStarted && lap == 1)
    {
        raceStarted = true;
        startRace();
    }

}else if(tokCount == 5)
{
    //place
    place = atoi(tok);
    if(getPos() != place)
    {
        changeHud(atoi(tok), -1, -1, -1, -1);
    }
}else if (tokCount == 6)
{
    speed = atoi(tok);
    if(speed == 100 && currSpeed != 100)
    {
        setSpeed("d");
        setSpeed("d");
    }else if(speed == 75 && currSpeed != 75)
    {
        setSpeed("K");
        setSpeed("K");
    }else if(speed == 50 && currSpeed != 50)
    {
        setSpeed("2");
        setSpeed("2");
    }else if(currSpeed != 0 && speed == 0)
    {
```

```
                setSpeed("0");
                setSpeed("0");
                        pthread_create(&vibThread, NULL, vibrate, (void *)1);

            }
                currSpeed = speed;


        }
        else if(tokCount > 7)
        {
            type = atoi(tok);
            tok = strtok(NULL, " ,");
            tokCount++;
        //printf("parsing packet (%d) in packet x: %s\n", tokCount, tok);
            x = (double)atoi(tok);
            tok = strtok(NULL, " ,");
            tokCount++;
        //printf("parsing packet (%d) in packet y: %s\n", tokCount, tok);
            y = (double)atoi(tok);
            if(type == NOWEP)
            {
                removeObj(key);
            }else
            {
                updateObj(key, x/(10.0), y/(10.0), type);
            }
            key++;

        }

        tok = strtok(NULL, " ,");
        tokCount++;
    }



}

void fireButt()
{
    wepUsed = true;

}

void *sendPackets(void *arg)
{
    //char buffer[20] = "30,30,10,20,1";
    //char buffer[40];
    char *buffer = (char *) malloc(40);

    while(1)
    {
        readData(buffer);
        //buffer[strlen(buffer)-1] = '\0';
        if(strlen(buffer) > 2)
        {
            if(wepUsed)
            {
                strcat(buffer, ",1");
                wepUsed = false;
```

```
            }else
            {
                strcat(buffer, ",0");
            }
            cout << strlen(buffer) << endl;
            cout << "buffer: " << buffer << endl;
            sendto(sock, buffer, 40, 0, (struct sockaddr *)&remote, sizeof(remote));
            memset(buffer, 0, 40);
        }
    }
}
void receivePackets()
{
    char *inpacket = (char*)malloc(150);
    int received = 0;
    uint32_t len = sizeof(remote);

    while(1)
    {
        //Receive packet in 'inpacket' buffer
        received = recvfrom(sock, inpacket, 150, 0, (struct sockaddr *)&remote, &len);
        //Decode packets here
        //cout << "Received: " << received << endl;
        //cout << "Packet: " << inpacket << endl;
        parsePacket(inpacket);
        memset(inpacket, 0, 150);
        //call necessary functions that use packet's data
    }
}
```

ix. kartObjLayer.cpp

```
#include "kartObjLayer.h"



/***************public functions***********/
//kartObjLayer::kartObjLayer():objLayer()
void kartObjLayer::init()
{
    objLayer::init();
    lightState = REDLIGHT;
    showLight = false;
    initObjects();
}

void kartObjLayer::setShield(int count, item_values_t type)
{
    objects[MAXOBJECTS].x = 0;
    objects[MAXOBJECTS].y = 0;
    if(type == REDSHELL)
    {
        if(count == 3)
        {
            objects[MAXOBJECTS].type = REDSHIELD3;
        }else if(count == 2)
        {
            objects[MAXOBJECTS].type = REDSHIELD2;
        }else
        {
            objects[MAXOBJECTS].type = REDSHIELD1;
```

```
        }
    }else if(type == GREENSHELL)
    {
        if(count == 3)
        {
            objects[MAXOBJECTS].type = GREENSHIELD3;
        }else if(count == 2)
        {
            objects[MAXOBJECTS].type = GREENSHIELD2;
        }else
        {
            objects[MAXOBJECTS].type = GREENSHIELD1;
        }
    }else
    {
        objects[MAXOBJECTS].type = NO_ITEM;
    }

}


/************private functions************/


void kartObjLayer::updateFrame()
{
    glPushMatrix();
    if(showLight)
    {
        drawTrafficLight();
    }

    for(int i = 0; i < (MAXOBJECTS +1); i++)
    {
        if(objects[i].type == ITEMBOX)
        {
            drawWepBox(objects[i].x, objects[i].y);
        }else if(objects[i].type == REDSHELL)
        {
            drawRShell(objects[i].x, objects[i].y);
        }else if(objects[i].type == BANANA)
        {
            drawBanana(objects[i].x, objects[i].y);
        }else if(objects[i].type == GREENSHELL)
        {
            drawGShell(objects[i].x, objects[i].y);
        }else if(objects[i].type == GREENSHIELD3)
        {
            drawShield(objects[i].x, objects[i].y, 3, GREENSHELL);
        }else if(objects[i].type == GREENSHIELD2)
        {
            drawShield(objects[i].x, objects[i].y, 2, GREENSHELL);
        }else if(objects[i].type == GREENSHIELD1)
        {
            drawShield(objects[i].x, objects[i].y, 1, GREENSHELL);
        }else if(objects[i].type == REDSHIELD3)
        {
            drawShield(objects[i].x, objects[i].y, 3, REDSHELL);
        }else if(objects[i].type == REDSHIELD2)
        {
            drawShield(objects[i].x, objects[i].y, 2, REDSHELL);
```

```
        }else if(objects[i].type == REDSHIELD1)
        {
            drawShield(objects[i].x, objects[i].y, 1, REDSHELL);
        }

    }
    glPopMatrix();

}
/*
 * initializes the objects to be empty so no 3d objects
 * will be drawn on the screen
 */
void kartObjLayer::initObjects()
{
    int i;
    for(i = 0; i < MAXOBJECTS; i++)
    {
        objects[i].x = 0;
        objects[i].y = 0;
        objects[i].type = NO_ITEM;
    }
    initItems();
}

void kartObjLayer::initItems()
{
    wepBox1 = glmReadOBJ("./layerObjects/wepBox1.obj");
    if(!wepBox1)
    {
        printf("unable to load wepBox1.obj\n");
    }
    rshell = glmReadOBJ("./layerObjects/SHELL.obj");
    if(!rshell)
    {
        printf("unable to load SHELL.obj\n");
    }
            gshell = glmReadOBJ("./layerObjects/GSHELL.obj");
    if(!gshell)
    {
        printf("unable to load GSHELL.obj\n");
    }
    banana = glmReadOBJ("./layerObjects/banana.obj");
    if(!banana)
    {
        printf("unable to load banana.obj\n");
    }
}


void kartObjLayer::drawShield(double x, double y, int count, item_values_t type)
{

    static int shieldRotate = 0;

    if(shieldRotate>= 360)
    {
                shieldRotate = 0;
    }
    shieldRotate += 5;
```

```
    glPushMatrix();
    glTranslatef(x, y, 0);

    glRotatef(shieldRotate, 0,0, 1);
    glPushMatrix();
       glTranslatef(0, -1.5, 0);
       if(count>2)
       {
          if(type == REDSHELL)
          {
             drawRShell(0, 0);
          }else
          {
             drawGShell(0, 0);
          }
       }
    glPopMatrix();

    glPushMatrix();
       glTranslatef(-1.5, 1.5, 0);
       if(count > 1)
       {
          if(type == REDSHELL)
          {
             drawRShell(0, 0);
          }else
          {
             drawGShell(0, 0);
          }
       }
    glPopMatrix();


    glPushMatrix();
       glTranslatef(1.5, 1.5, 0);
       if(type == REDSHELL)
       {
          drawRShell(0, 0);
       }else
       {
          drawGShell(0, 0);
       }
    glPopMatrix();

    glPopMatrix();


}

void kartObjLayer::drawGShell(double x, double y)
{
    glPushMatrix();
    //glutSolidCube(1);
    glTranslatef(x-.3, y-.3, -.5);
    glRotatef(-110, 0, 0, 1);
    glRotatef(90, 1, 0, 0);
    glScalef(.5, .5, .5);
    glScalef(.5, .5, .5);
    glmDraw(gshell, GLM_COLOR);
    glPopMatrix();
```

```
}
void kartObjLayer::drawRShell(double x, double y)
{
   glPushMatrix();
   //glutSolidCube(1);
   glTranslatef(x-.3, y-.3, -.5);
   glRotatef(-110, 0, 0, 1);
   glRotatef(90, 1, 0, 0);
   glScalef(.5, .5, .5);
   glScalef(.5, .5, .5);
   glmDraw(rshell, GLM_COLOR);
   glPopMatrix();

}
void kartObjLayer::drawBanana(double x, double y)
{


   glPushMatrix();
   //glutSolidCube(1);
   glTranslatef(x, y-.4, -.7);
   glRotatef(160, 0, 0, 1);
   glRotatef(90, 1, 0, 0);
   glScalef(.2, .2, .2);
   glmDraw(banana, GLM_COLOR);

   glPopMatrix();
}

void kartObjLayer::drawWepBox(double x, double y)
{


   glPushMatrix();
   //glutSolidCube(1);
   glTranslatef(x, y+.3, -.5);
   glRotatef(45, 0, 0, 1);
   glRotatef(90, 1, 0, 0);
   glScalef(.5, .5, .5);
   glScalef(.35, .35, .35);
   glmDraw(wepBox1, GLM_COLOR);

   glPopMatrix();
}




void kartObjLayer::drawSphere(float xSize, float ySize, float zSize)
{
  // save the transformation state
  glPushMatrix();

  // locate it in the scene
  glMatrixMode(GL_MODELVIEW);
  // note that center of cube is at origin
  glTranslatef(0, 0, 0);
  glScalef(xSize, ySize, zSize);

  // draw the cube - the parameter is the length of the sides
  //glutSolidCube(2.0);
```

```
  glutSolidSphere(1, 15, 15);

  // recover the transform state
  glPopMatrix();

  return;
}

void kartObjLayer::drawTrafficLight()
{

  GLfloat defaultColor[4];
  GLfloat redLightColor[4];
  redLightColor[1] = 0; redLightColor[2] = 0; redLightColor[3] = 1.0;
  GLfloat greenLightColor[4];
  greenLightColor[0] = 0; greenLightColor[2] = 0;greenLightColor[3] = 1.0;
  GLfloat yellowLightColor[4];
  yellowLightColor[0] = 0; yellowLightColor[2] = 0;yellowLightColor[3] = 1.0;
  GLfloat dontWalkLightColor[4];
  dontWalkLightColor[2] = 0;dontWalkLightColor[3] = 1.0;

   GLfloat lightDir[] = {0,-1, 1, 0.0};
   GLfloat diffuseComp[] = {1.0, 1.0, 1.0, 1.0};


   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_NORMALIZE);
   glEnable(GL_COLOR_MATERIAL);

   glLightfv(GL_LIGHT0, GL_POSITION, lightDir);
   glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseComp);


  if (lightState == REDLIGHT) {
    redLightColor[0] = 1;
    greenLightColor[1] = 0.4;
    yellowLightColor[0] = .4;
    yellowLightColor[1] = .4;
  }
  else if (lightState == GREENLIGHT ) {
    redLightColor[0] = 0.4;
    greenLightColor[1] = 1.0;
    yellowLightColor[0] = .4;
    yellowLightColor[1] = .4;
  }else if (lightState == YELLOWLIGHT)
  {
    redLightColor[0] = 0.4;
    greenLightColor[1] = .4;
    yellowLightColor[0] = 1;
    yellowLightColor[1] = 1;

  }


  glPushMatrix();
    // draw red light on top - just a red box
    glColor3fv(redLightColor);
    //glMaterialfv(GL_FRONT, GL_DIFFUSE, redLightColor);
    glTranslatef(-2, 10, 0);
    drawSphere(1, 0.1, 1);
```

```
    // draw yellow light below red light
    //glMaterialfv(GL_FRONT, GL_DIFFUSE, greenLightColor);
    glColor3fv(yellowLightColor);
    glTranslatef(2, 0, 0);
    drawSphere(1, 0.1, 1);

    // draw green light below red light
    //glMaterialfv(GL_FRONT, GL_DIFFUSE, greenLightColor);
    glColor3fv(greenLightColor);
    glTranslatef(2, 0, 0);
    drawSphere(1, 0.1, 1);

  defaultColor[2] = 1;
  defaultColor[3] = 1.0;
  defaultColor[0]= 1.0;
  defaultColor[1] = 1.0;

   glColor3fv(defaultColor);

  glPopMatrix();

}


/*
void *moveShell(void *arg)
{
   while(shellTrans< 10)
   {
      shellTrans= shellTrans+.5;
      usleep(50000);
   }
   shellShot = false;
}

void shootShell()
{

   shellShot = true;
   shellTrans = 0;
   pthread_create(&shellThread, NULL, moveShell, (void *) 1);

}

*/
```

x. kartViewController.cpp

```
#include "kartViewController.h"
#define  SHELLSPEED 3
#define  STEADYSPEED .5
#define NETWORK
#define ARDUINO
#define PI 3.14159

/*public functions*/
void kartViewController::init(int player)
{
   /*3d inits*/
   myLayer.init();
```

```
    fps = 0;

    /*hud inits*/
    myHud.init(1, 3, 1);
    raceTimer= 0;
    wepBuffer = NO_ITEM;

    raceStart = false;
    scrollWep = false;
    vibrateScreen = false;
    runStar = false;
    fireWep = false;

    /*sound init*/
    bgMusic.OpenFromFile("./sounds/rainbowroad.ogg");

    itemreel.LoadFromFile("./sounds/itemreel.wav");
    gotitem.LoadFromFile("./sounds/gotitem.wav");
    starpower.LoadFromFile("./sounds/starpower.wav");
    startup.LoadFromFile("./sounds/startup.wav");
    countdown.LoadFromFile("./sounds/countdown.wav");
    gothit.LoadFromFile("./sounds/gothit.wav");
    cputhrow.LoadFromFile("./sounds/cputhrow.wav");
    itemdrop.LoadFromFile("./sounds/itemdrop.wav");
    woohoo.LoadFromFile("./sounds/woohoo.wav");
    boost.LoadFromFile("./sounds/boost.wav");

    starSound.SetBuffer(starpower);
    lightSound.SetBuffer(startup);
    hitSound.SetBuffer(gothit);
    throwSound.SetBuffer(cputhrow);
    dropSound.SetBuffer(itemdrop);
    shieldSound.SetBuffer(woohoo);
    mushroomSound.SetBuffer(boost);

    starSound.SetLoop(true);
    bgMusic.SetLoop(true);
    bgMusic.SetVolume(30);

    lightSound.Play();


    /*networks init*/
#ifdef NETWORK
    char *buffer;
    buffer = (char *) malloc(sizeof(buffer));
    myNetwork.init(1, player);
    myNetwork.send_data(buffer);
    free(buffer);
#endif

#ifdef ARDUINO
    myArduino.init("/dev/tty.usbserial-A700fiFR", 0);
#endif

    for(int i = 0; i < MAXOBJECTS; i++)
    {
        objBuffer[i].type = NO_ITEM;
    }
//    myLayer.addObj(0, 0, REDSHELL);
 //  objBuffer[0].type = REDSHELL;
```

```
 //  objBuffer[0].x = 0;
 //  objBuffer[0].y = 10;

}

void kartViewController::start()
{

   if(!raceStart && !myLayer.showLight)
   {
      lightSound.SetBuffer(countdown);
      lightSound.Play();
      myLayer.lightState = REDLIGHT;
      myLayer.showLight = true;
   }
}

void kartViewController::pickUpWep(item_values_t wep)
{
   if(wepBuffer == NO_ITEM && wep != NO_ITEM)
   {
      wepBuffer = wep;
      scrollWep = true;
      scrollWepSound.SetBuffer(itemreel);
      scrollWepSound.Play();
   }

}

void kartViewController::useWep()
{
   if(scrollWep)
   {
      scrollWep = false;
      scrollWepSound.Stop();
      scrollWepSound.SetBuffer(gotitem);
      scrollWepSound.Play();
   }
#ifndef NETWORK
   else if(wepBuffer != NO_ITEM)
   {
      if(wepBuffer == STAR)
      {
         runStar = true;
         starSound.Play();
      }else if(wepBuffer == REDSHELL || wepBuffer == GREENSHELL)
      {
         throwSound.Play();
      }else if(wepBuffer == MUSHROOM)
      {
         mushroomSound.Play();
      }else if(wepBuffer == BANANA)
      {
         dropSound.Play();
      }else if(wepBuffer == MRSHELL || wepBuffer == MGSHELL)
      {
         shieldSound.Play();
      }
      wepBuffer = NO_ITEM;
   }
#else
```

```
    else
    {
        fireWep = true;
    }
#endif
}

void kartViewController::gotHit()
{
    vibrateScreen = true;
    hitSound.Play();
}

void kartViewController::step()
{
   //cout << "starting controller step" << endl;
    calcFPS(getTickCount());

#ifdef NETWORK
    char inPacket[150];
    if(myNetwork.recv_data(inPacket) > 20)
    {
        parsePacket(inPacket);
    }
    sendPacket();
    makeObjectTravel();
#endif

    detLight();
   //cout << "done detlight" << endl;
    placeRaceTime(getTickCount());
   //cout << "done placeracetie" << endl;
    placeWep();
   //cout << "done placewep" << endl;
    modifyScreen();
   //cout << "done modifyscreen" << endl;
    myLayer.step();
   //cout << "done layerstep" << endl;
    myHud.step();
   //cout << "done hudstep" << endl;
}

void kartViewController::testButton(char *buff)
{

    myArduino.overWrite(buff);
}


/*private functions*/

unsigned kartViewController::getTickCount()
{
    struct timeval tv;
    if(gettimeofday(&tv, NULL) != 0)
        return 0;
    return (tv.tv_sec * 1000) + (tv.tv_usec / 1000);
}

void kartViewController::calcFPS(int currTime)
{
```

```
    static int lastTFrame = currTime;
    static int frameCounter = 0;

    if(currTime - lastTFrame >= 1000)
    {
        fps = frameCounter;
        frameCounter = 0;
        lastTFrame = getTickCount();
        //cout << fps << endl;

    }else
    {
        frameCounter++;
    }

}
void kartViewController::placeRaceTime(int currTime)
{
    static int startTime = -1;
    if(raceStart && startTime == -1)
    {
        startTime = currTime;
    }

    if(startTime != -1)
    {
        raceTimer = currTime - startTime;
    }else
    {
        raceTimer= 0;
    }
    myHud.setRaceTime(raceTimer);

}

void kartViewController::placeWep()
{
    static int currSetWep = 0;
    static int scrollTime = 0;
    if(scrollWep)
    {
        if(getTickCount()-scrollTime >= 100)
        {
            scrollTime = getTickCount();
            currSetWep++;
        }
        myHud.changeHud(-1, -1, currSetWep%NUMWEP +1, -1);
        if(currSetWep>= NUMWEP*5)
        {
            scrollWep = false;
            scrollWepSound.Stop();
            scrollWepSound.SetBuffer(gotitem);
            scrollWepSound.Play();
        }

    }else
    {
        myHud.changeHud(-1, -1, wepBuffer, -1);
        currSetWep = 0;
    }
```

```
}

void kartViewController::modifyScreen()
{
   static int screen = 0;
   static int vibTime = 0;
   static int color = -1;
   static int colorTime = 0;

   if(vibrateScreen)
   {
      if(getTickCount()-vibTime >= 10)
      {
         screen++;
         vibTime = getTickCount();
      }
      myHud.setScreenOffset( (screen%2*10) -5 , 0, 0);
      if(screen >= 6)
      {
         vibrateScreen = false;
      }
   }else
   {
      myHud.setScreenOffset(0, 0, 0);
      screen = 0;
   }

   if(runStar)
   {
      if(getTickCount()-vibTime >= 10)
      {
         color++;
      }

      if(color%3 == 0)
      {
         myHud.setScreenColor(1, .4, .4);
      }else if(color%3 == 1)
      {
         myHud.setScreenColor(1, 1, 0);
      }else
      {
         myHud.setScreenColor(1, 1, 1);
      }

   }else
   {
      color = 0;
      myHud.setScreenColor(1, 1, 1);
   }


}
void kartViewController::detLight()
{
   static int lightTimer = -1;
   if(myLayer.showLight)
   {
      if(lightTimer == -1)
      {
         lightTimer = getTickCount();
```

```
        }
        if(getTickCount() - lightTimer >= 3000)
        {
            myLayer.showLight= false;
            raceStart = true;
            lightTimer = -1;
            bgMusic.Play();
        }else if((getTickCount() - lightTimer) >= 2000)
        {
            myLayer.lightState = GREENLIGHT;

        }else if((getTickCount() - lightTimer) >= 1000)
        {
            myLayer.lightState = YELLOWLIGHT;
        }
    }

}

void kartViewController::makeObjectTravel()
{
    static double lastTFrame = -1;
    double dTime;
    if(lastTFrame == -1)
    {
        lastTFrame = getTickCount();
    }
    dTime = getTickCount() - lastTFrame;

    for(int i = 0; i < MAXOBJECTS; i++)
    {
        if(objBuffer[i].type != NO_ITEM
            && myLayer.objects[i].type == objBuffer[i].type)
        {

           if(objBuffer[i].y > myLayer.objects[i].y &&
            objBuffer[i].y - myLayer.objects[i].y > .3)
            {
                myLayer.objects[i].y = SHELLSPEED * (dTime/1000) + myLayer.objects[i].y;
            }else if(objBuffer[i].y < myLayer.objects[i].y &&
                    myLayer.objects[i].y - objBuffer[i].y > .3)
            {
                myLayer.objects[i].y = -SHELLSPEED * (dTime/1000) + myLayer.objects[i].y;
            }

           if(objBuffer[i].x > myLayer.objects[i].x &&
            objBuffer[i].x - myLayer.objects[i].x > .3)
            {
                myLayer.objects[i].x = SHELLSPEED * (dTime/1000) + myLayer.objects[i].y;
            }else if(objBuffer[i].x < myLayer.objects[i].y &&
                    myLayer.objects[i].x - objBuffer[i].y > .3)
            {
                myLayer.objects[i].x = -SHELLSPEED * (dTime/1000) + myLayer.objects[i].y;
            }



        }
    }
    lastTFrame = getTickCount();
```

```
}

void kartViewController::parsePacket(char *inpacket)
{
    char *tok;
    item_values_t currItem = NO_ITEM;
    int tokCount = 0;
    int speed;
    int key = 0;
    double x;
    double y;
    static item_values_t currShield = NO_ITEM;
    static int currSpeed = -1;
    cout << inpacket << endl;
    tok = strtok(inpacket, " ,");
    while(tok)
    {
        //cout << "tokenizing" << endl;
        if(tokCount == 1)
        {
            currItem = (item_values_t) atoi(tok);
            if(wepBuffer != currItem)
            {
                if(currItem == NO_ITEM)
                {
                    if(wepBuffer == REDSHELL || wepBuffer == GREENSHELL)
                    {
                        throwSound.Play();
                    }else if(wepBuffer == MUSHROOM)
                    {
                        mushroomSound.Play();
                    }else if(wepBuffer == BANANA)
                    {
                        dropSound.Play();
                    }else if(wepBuffer == MRSHELL || wepBuffer == MGSHELL)
                    {
                        shieldSound.Play();
                    }

                    wepBuffer = NO_ITEM;

                }else
                {
                    pickUpWep(currItem);
                }
            }

        }else if(tokCount == 2)
        {

            currItem = (item_values_t)atoi(tok);
            tok = strtok(NULL, " ,");
            tokCount++;

            if(currItem == STAR && currShield != STAR)
            {
                runStar = true;
                starSound.Play();
            }else if(currItem  == MRSHELL || currItem == REDSHELL)
```

```
            {
                myLayer.setShield(atoi(tok), REDSHELL);
            }else if(currItem == MGSHELL || currItem == GREENSHELL)
            {
                myLayer.setShield(atoi(tok), GREENSHELL);
            }else
            {
                if(currShield == STAR && currItem != STAR)
                {
                    cout << "stopping star" << endl;
                    runStar = false;
                    starSound.Stop();
                }else
                {
                    myLayer.setShield(0, NO_ITEM);
                }
            }
            currShield = currItem;


        }else if(tokCount == 4)
        {
            //lap No.
            if(!raceStart &&  atoi(tok) != 0)
            {
                start();
            }
            myHud.changeHud(-1, atoi(tok), -1, -1);
        }else if(tokCount == 5)
        {
            //place
            myHud.changeHud(atoi(tok), -1, -1, -1);
        }else if (tokCount == 6)
        {

#ifdef ARDUINO
            myArduino.setSpeed(atoi(tok));
#endif
            if(currSpeed != 0 && atoi(tok) == 0)
            {
                gotHit();
            }
            currSpeed = atoi(tok);
        }
        else if(tokCount > 7)
        {
            //type
            currItem = (item_values_t) atoi(tok);
            tok = strtok(NULL, " ,");
            tokCount++;
            //x position
            x = (double)atoi(tok);
            tok = strtok(NULL, " ,");
            tokCount++;

            //yposition
            y = (double)atoi(tok);
            if(currItem == NO_ITEM)
            {
                myLayer.removeObj(key);
                objBuffer[key].type = NO_ITEM;
```

```
            objBuffer[key].x = 0;
            objBuffer[key].y = 0;

        }else if(currItem != myLayer.objects[key].type)
        {
            myLayer.updateObj(key, x/(10.0), y/(10.0), currItem);
            objBuffer[key].type = currItem;
            objBuffer[key].x = x/(10.0);
            objBuffer[key].y = y/(10.0);
        }else
        {
            myLayer.objects[key].type = currItem;
            objBuffer[key].type = currItem;
            objBuffer[key].x = x/(10.0);
            objBuffer[key].y = y/(10.0);

        }
        key++;

      }

      tok = strtok(NULL, " ,");
      tokCount++;
   }
}

void kartViewController::sendPacket()
{
   static int lastTFrame = 0;
   static char outPacket[40];
#ifdef ARDUINO
   if(strlen(outPacket) < 3 && getTickCount()- lastTFrame >= 250)
   {
      myArduino.readData(outPacket);
            cout << "packet length " << strlen(outPacket) << endl;
   }
#else
   strcpy(outPacket, "0,0,0,0");
#endif

if(getTickCount() -lastTFrame >= 250 && strlen(outPacket) > 3)
   {
      if(fireWep)
      {
         strcat(outPacket, ",1");
         fireWep = false;
      }else
      {
         strcat(outPacket, ",0");
      }

      cout << "sending " <<  outPacket <<endl;
      myNetwork.send_data(outPacket);
      lastTFrame = getTickCount();
      memset(outPacket, 0, 40);
   }


}
```

xi. main.cpp (Laptop)

```
// mastergraphics.cpp : main project file.

#include "kartViewController.h"
#include "main.h"


//hud* myHud = new hud();
//kartObjLayer *myLayer = new kartObjLayer();
int testx = 0;
int testy = 0;
int testz = 0;
int testWep = 0;

bool fullScreened;

void setUpView()
{
   glLoadIdentity();
   gluLookAt(0, -10, 0, 0.0, 0.0, 0.0, 0.0, 0, 1.0);
   glTranslatef(0.0f,-10, 0.0f);


   return;
}
void reshapeCallback(int w, int h)
{

   printf("width %d, height %d\n", w, h);
   glViewport(0,0,w,h);

   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(60, 1.0, 0.1, 200.0);

   glMatrixMode(GL_MODELVIEW);
}


void clear()
{

}



void glutKeyboard(unsigned char key, int x, int y)
{

   switch (key)
   {
      case 27:
         {
            clear();
            exit(0);
         }
      case '1':
         {
            myController.start();
```

```
            break;
        }
      case '2':
        {
            myController.pickUpWep((item_values_t)testWep);
            break;
        }
      case 'W':
        {
            myController.useWep();
            break;
        }
      case 'e':
        {
            myController.gotHit();
            break;
        }
      case 'r':
        {
            break;
        }
      case 'b':
        {
            myController.useWep();
            break;
        }
      case 'p':
        {
            testx++;
            break;
        }
      case 'P':
        {
            testx--;
            break;
        }
      case 'o':
        {
            testy++;
            break;
        }
      case 'O':
        {
            testy--;
            break;
        }
      case 'i':
        {
            testz++;
            break;
        }
      case 'I':
        {
            testz--;
            break;
        }
case 'a':
{
   myController.testButton("a");
   break;
}
```

```
case 's':
{
   myController.testButton("s");
   break;
}
case 'z':
{
   myController.testButton("z");
   break;
}
case 'x':
{
   myController.testButton("x");
   break;
}
case 'q':
{
   myController.testButton("q");
   break;
}
case 'w':
{
   myController.testButton("w");
   break;
}

   }
}


void glutSpecialKeyboard(int value, int x, int y)
{
   switch (value)
   {
      case GLUT_KEY_F1:
         {
            testWep++;
            if(testWep > NUMWEP)
            {
               testWep = 0;
            }
         }
      case GLUT_KEY_F2:
         {

            if ( !fullScreened )
            {
               glutFullScreen();
            }else
            {
               glutPositionWindow( 0, 20 );
               glutReshapeWindow( 640, 480 );
            }
            fullScreened = !fullScreened;
            break;
         }


   }
}
```

```
void glutDisplay(void)
{

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glEnable(GL_DEPTH_TEST);
    setUpView();


    glTranslatef(testx, testy, testz);
    myController.step();
    //myLayer->step();

    glutSwapBuffers();

}

void glutMouseFunc( int button, int state, int x, int y )
{
#ifdef DEBUG
    if ( state ==  GLUT_DOWN && button == GLUT_LEFT_BUTTON )
    {
        printf("screen coords %d, %d\n", x, y);
    }
#endif

}



void InitializeOpenGL(int argc, char **argv)
{
    string windowName = "Player ";
    glutInitDisplayMode( GLUT_DOUBLE | GLUT_DEPTH | GLUT_RGBA | GLUT_MULTISAMPLE );
    glutInitWindowPosition( 0, 0 );
    glutInitWindowSize( 640, 480 );
    glutInit( &argc, argv );
    windowName.append(argv[1]);
    glutCreateWindow(windowName.c_str());
    glutReshapeFunc(reshapeCallback);
    glutDisplayFunc(glutDisplay);
    glutKeyboardFunc(glutKeyboard);
    glutSpecialFunc(glutSpecialKeyboard);
    glutIdleFunc(glutDisplay);
    glutMouseFunc(glutMouseFunc);

    glutFullScreen();
    fullScreened = true;

    glutMainLoop();
}
int main(int argc, char** argv)
{
    int self;

    if(argc != 2)
    {
        fprintf(stderr, "Usage: %s kartnumber\n", argv[0]);
        return -1;
```

```
    }

    self = atoi(argv[1]);
    if(self == 0)
    {
        fprintf(stderr, "Invalid number given: %d\n", self);
        return -1;
    }

    //
    myController.init(self);
    InitializeOpenGL(argc, argv);
    clear();

    return 0;
}
```