simple_sim Operation and Programming Handbook

simple_sim was conceived of and designed over a two-week period prior to the beginning of the MIT 2.12 Introduction to Robotics class. It was initially going to be a simple simulator, but evolved into a more complex application due to changing requirements.

Overall design

- Simple interface, both for the user and the programmer
- Multi-threaded architecture
- Leverages OpenGL for graphics and SDL for OS interfacing, minimizing porting effort
- Uses the Open Dynamics Engine to simulate the physics of the real world

Usage manual

Parameters

The `simple_sim` program takes a few parameters, listed below:

- `--help, -h` - Print out the included help
- `--window-size WxH, -s WxH` - Change the starting window size (default is 640x480, use it like so: -s 1024x768)
- `--walls FILE, -w FILE` - Load the walls from FILE
- `--targets FILE, -t FILE` - Load the target positions from FILE
- `--log FILE, -l FILE` - Write the log to FILE
- `--ucparams PARAMS, -p PARAMS` - Pass parameters to the UserCode
  - To pass a single parameter, PARAMS should be something like "`start_mode=manual`"
  - To pass multiple parameters, PARAMS should be comma delimited KEY=VALUE pairs, example: "`start_mode=wp,wpfile=wps.txt`"

Parameters for the built-in UserCode

The built-in `UserCode` supports a few parameters:

- `start_mode=`*mode*, where *mode* can be one of `manual`, `auto` or `automatic`, or `waypoint` or `wp`; this sets the startup mode of the default `UserCode`. The default is `manual`
- `waypointfile=`*FILE* or `wpfile=`*FILE* - *FILE* is the waypoint file to load for use by the built-in waypoint controller
- `auto_wp=`*enable* or *disable* - Enable or disable auto incrementing waypoint (continue to next waypoint automatically instead of waiting for user input)

Format of the waypoint file

The waypoint file is a text file containing one waypoint per line, with comma-delimited X and Y values. The characters '`%`' and '`#`' begin comments, and cause the rest of the line to be ignored.

The following is an example waypoint file:

```
# this is a comment
% this is another comment

2,2
2,-2
-2,-2
-2,2

# end waypoint file
```

Format of the walls file

The walls file is a text file containing coordinates, size, and color of any walls one wants. The characters '`%`' and '`#`' begin comments, and cause the rest of the line to be ignored. The fields are all comma-delimited double-precision values, in this order:

$$X_1, Y_1, X_2, Y_2, Thickness_{line}, Red, Green, Blue$$

The color values (*Red*, *Green*, and *Blue*) are to be between 0.0 (fully off) and 1.0 (fully on.) The following is an example walls file that sets up a 10 meter by 10 meter box around the origin:

```
# format: x1, y1, x2, y2, thickness, r, g, b
-5, -5, 5, -5, 0.1, 0.5, 0.5, 0.5
5, -5, 5, 5, 0.1, 0.5, 0.5, 0.5
5, 5, -5, 5, 0.1, 0.5, 0.5, 0.5
-5, 5, -5, -5, 0.1, 0.5, 0.5, 0.5
```

Format of the targets file

The targets file is a text file containing coordinates, size, and tint color of all the targets (mines) in the simulation. The characters '%' and '#' begin comments, and cause the rest of the line to be ignored. The fields are all comma-delimited double-precision values, in this order:

$$X_{target}, Y_{target}, Radius_{target}, Red, Green, Blue$$

The color values (*Red*, *Green*, and *Blue*) are to be between 0.0 (fully off) and 1.0 (fully on.) The following is an example targets file that sets up 4 targets in various positions inside the aforementioned 10 meter by 10 meter box:

```
# format: x, y, radius, r, g, b
-1.5,  2.5, 0.2, 1.0, 0.5, 0.5
 1.5, -3.5, 0.2, 0.5, 1.0, 0.5
-3.0, -1.0, 0.2, 0.5, 0.5, 1.0
-1.25,-3.0, 0.2, 0.5, 1.0, 1.0
```

Format of the output log file

The output log file contains a space-delimited list of columns that may be loaded into numerous programs including `MATLAB` and `octave`. The columns are double-precision values; there are 5 built-in columns logged, and the `UserCode` provided by default logs an additional 5 values, for a total of 10 columns. They are in this order:

$$Time_{unix}, X_{sim}, Y_{sim}, Z_{sim}, \theta_{sim}, X_{integrated}, Y_{integrated}, \theta_{integrated}, V_{translational}, V_{angular}$$

This file may be loaded directly into both `MATLAB` and `octave` with the following command:

```
logfile = load('log_file_path.txt');
```

To plot the simulator position versus the integrated position, the following commands can be used:

```
clearplot;
hold on;
plot(logfile(:,2), logfile(:,3));
plot(logfile(:,6), logfile(:,7));
```

Using the simple_sim user interface

There are numerous keys that control the simulation.

- `-` - zooms out
- `=` - zooms in

- `Tab` - Cycles the camera mode through the following:
  - 2D, following robot translation (default)
  - 2D, fixed camera (movable with arrow keys)
  - 3D, tracking robot X position
  - 3D, tracking robot Y position
  - 3D, behind and above robot position
  - 2D, following robot translation and rotation
- `Arrow keys` - in fixed camera mode, moved the camera around in 2D
- `d` - Dumps the ODE simulation world to `world.dump`
- `t` - Toggles the blue robot trail on and off
- `p` - Pauses and resumes the simulation
- `w` - Reduces the time multiplier by 0.1x (down to 0.1x)
- `e` - Increases the time multiplier by 0.1x (up to 10x)
- `r` - Resets the time multiplier to 1.0x

With the built-in `UserCode`, there are more keys that can be used for control.

- `a` - In manual mode, increases translational velocity
- `z` - In manual mode, decreases translational velocity
- `m` - In manual mode, increases rotational velocity (turn to the right)
- `n` - In manual mode, decreases rotational velocity (turn to the left)
- `,` - In manual mode, set rotational velocity to 0
- `Space` - In manual mode, set rotational and translational velocity to 0
- `o` - Cycle through the modes available
  - `manual` - allows the end-user to drive the robot by hand
  - `automatic` - currently does nothing but act as a placeholder for code
  - `waypoint` - does simple waypoint following from a loaded list of waypoints
- `x` - In waypoint mode, set the desired waypoint to the next waypoint in the list, or exit waypoint mode if it's reached the end of the list
- `c` - In waypoint mode, toggle whether the UserCode automatically advances to the next waypoint when the current one is reached

UserCode interface

The `UserCode` is a C++ singleton class where one may put their own control code. All calls to UserCode are from a single thread of execution, so one does not have to worry about re-entrancy.

List of functions prototypes and enums

- These need to be implemented at the minimum:

```
UserCode::UserCode(RobotIF &robot);
UserCode::~UserCode();
void UserCode::setup();
void UserCode::handle_param(string key, string val);
void UserCode::cycle();
void UserCode::key_press(char k);
void UserCode::mouse_click(int x, int y, double worldx,
                           double worldy, int button);
void UserCode::collision(int object_type);
void UserCode::get_log_data(double[]);
int UserCode::num_log_vars();

enum {
    OBJECT_ROBOT,
    OBJECT_WALL,
    OBJECT_GROUND,
    OBJECT_TARGET,
};
```

- These are related to the built-in user code:

```
void UserCode::toggle_mode();

DifferentialDriveIntegrator integrator;

void DifferentialDriveIntegrator::SetAxleLength(double axl);
void DifferentialDriveIntegrator::SetWheelDiameter(double diam);
void DifferentialDriveIntegrator::SetInputConversionFactor(double icf);
void DifferentialDriveIntegrator::ZeroPosition();
void DifferentialDriveIntegrator::StepSystem(double rtime, double lmovement, dou
double DifferentialDriveIntegrator::X();
double DifferentialDriveIntegrator::Y();
double DifferentialDriveIntegrator::T();
```

- These are related to UserCode's control over the robot:

```
RobotIF &UserCode::robot;

void RobotIF::SetLeftMotorVel(double revs_per_sec);
void RobotIF::SetRightMotorVel(double revs_per_sec);
long RobotIF::GetLeftEncoder();
long RobotIF::GetRightEncoder();
double RobotIF::GetLeftCommandedVel();
double RobotIF::GetRightCommandedVel();
double RobotIF::GetLeftActualVel();
double RobotIF::GetRightActualVel();
double RobotIF::GetAxleLength();
double RobotIF::GetWheelRadius();
long RobotIF::GetTicksPerRevolution();
void RobotIF::SetWindowTitleExtra(const char *s);
double RobotIF::GetSimulatorXPosition();
double RobotIF::GetSimulatorYPosition();
double RobotIF::GetSimulatorTPosition();
```

UserCode functions definitions

`UserCode::UserCode(RobotIF &robot);`
: The constructor for the class. You may want to create objects, allocate memory here, etc. This function should at a minimum set the class member `robot` to the argument `robot` (“`this->robot = robot;`”)

`UserCode::~UserCode();`
: The destructor for the class. You may want to delete objects, close files, free `malloc`'d memory, etc. here.

`void UserCode::setup();`
: This function is called before the simulation loop begins. You may want to put setup here. This function is called after all of the paramaters are loaded via `handle_param`.

`void UserCode::handle_param(string key, string val);`
: This function is called once for each `KEY=VALUE` pair passed on the commandline via `-p` or `--ucparams`.

`void UserCode::cycle();`
: This function is called at 10Hz while the simulation is running.

`void UserCode::key_press(char k);`
: This function is called whenever a key not handled by the simulation core is pressed. (These keys are `=`, `-`, Tab, Arrow keys, `d`, `t`, `p`, `w`)

`void UserCode::mouse_click(int x, int y, double worldx, double worldy, int button);`

This function is called whenever a mouse click is performed while in either the fixed 2D camera mode or the 2D camera-follows-robot-translation mode.

`void UserCode::collision(int object_type);`

This function is called whenever a collision is detected between the robot and an object in the environment. `object_type` may be any of the enums listed above in the definition section.

`int UserCode::num_log_vars();`

This function should return a number saying how many variables you want to log.

`void UserCode::get_log_data(double vars[]);`

This function is called whenever a `cycle()` completes. `vars` is an array of doubles of a size determined by the result of `num_log_vars()`. For example: if `num_log_vars()` returned 5, `vars` would be an array of five doubles, accessible from `vars[0]` to `vars[4]`. These variables should be set inside this function.

`void UserCode::toggle_mode();`

This function is called internally by the default `UserCode` to handle switching between the `manual`, `automatic`, and `waypoint` modes. It shouldn't need to be changed.

`DifferentialDriveIntegrator UserCode::integrator;`

This object is used to provide an integrated position of the vehicle based soley on the pseudo-encoders attached to the wheels.

`void DifferentialDriveIntegrator::SetAxleLength(double axl);`

Sets the axle length used in calculations of the integrator to `axl`.

`void DifferentialDriveIntegrator::SetWheelDiameter(double diam);`

Sets the wheel diameter used in calculations of the integrator to `diam`.

`void DifferentialDriveIntegrator::SetInputConversionFactor(double icf);`

Sets the conversion factor used inside the integrator to go from encoder counts to meters ($1/TicksPerMeter$)

`void DifferentialDriveIntegrator::ZeroPosition();`

Resets the integrator's idea of where it is to (0,0,0).

`void DifferentialDriveIntegrator::StepSystem(double rtime, double lmovement, double rmovement);`

Steps the integrator system; rtime is the reading time, lmovement is the difference in the left encoder value, rmovement is the difference in the right encoder value. StepSystem expects both values to increase when moving forwards.

`double DifferentialDriveIntegrator::X();`

`double DifferentialDriveIntegrator::Y();`

`double DifferentialDriveIntegrator::T();`

Returns the X, Y, or T position of the robot as determined by the integrator.

`RobotIF &UserCode::robot;`

This object contains the functions one may call to get and set status of the robot.

`void RobotIF::SetLeftMotorVel(double revs_per_sec);`

`void RobotIF::SetRightMotorVel(double revs_per_sec);`

These functions allows one to set the motor velocities. (Note: positive values for the left motor are *backwards*, while positive values for the right motor are forwards.)

`long RobotIF::GetLeftEncoder();`

`long RobotIF::GetRightEncoder();`

These functions returns the current encoder position on the vehicle (does not handle wrap for you.) Remember that the left wheel encoder values increase as it is spun backwards.

`double RobotIF::GetLeftCommandedVel();`

`double RobotIF::GetRightCommandedVel();`

These functions returns the commanded velocity (in revolutions per second) of the motors.

`double RobotIF::GetLeftActualVel();`

`double RobotIF::GetRightActualVel();`

These functions returns the actual velocity (in revolutions per second) of the motors.

`double RobotIF::GetAxleLength();`

This function returns the length of the axle (distance from one wheel to the other wheel.)

`double RobotIF::GetWheelRadius();`

This function returns the radius of the drive wheels.

`long RobotIF::GetTicksPerRevolution();`

This function returns the number of encoder ticks there are per revolution.

`void RobotIF::SetWindowTitleExtra(const char *s);`

This function allows the UserCode to set part of the window title for any small amount of information.

`double RobotIF::GetSimulatorXPosition();`

`double RobotIF::GetSimulatorYPosition();`

`double RobotIF::GetSimulatorTPosition();`

These functions return the position of the robot according to the simulator core.

Description of the built-in UserCode

Programming with simple_sim

Obtaining and building simple_sim under Linux or other UNIX-compatible systems

`simple_sim` has a few dependancies:

- SDL-1.2.7 or later (available from `http://www.libsdl.org/`)
- ODE-0.5.0 or later (available from `http://www.ode.org/`)

In addition to these libraries, you need to make sure that you have OpenGL support to be able to use the viewer into the world. Simply run "`glxgears`"; if OpenGL support is installed and working, a window will open containing 3 colored gears that spin.

`simple_sim` is available from the COE public CVS server. To a checkout of the latest available code, run this command:

```
cvs -d:pserver:cvs@oe.mit.edu:/raid/cvs-server/REPOSITORY co simple_sim
```

This will create a directory named `simple_sim` in the directory in which you run the `cvs` command.

If all of the dependancies are installed, building `simple_sim` is very easy; just type `make` inside the `simple_sim` directory.

Obtaining and installing dependancies

Many UNIX-compatible systems provide automatic facilities for installing packages. Procedures for installing SDL and ODE are listed below.

- SDL
  - Debian: apt-get install libsdl1.2-dev
  - Gentoo: emerge libsdl
  - Fedora: yum install SDL (may not install development headers, which necessitates installing from source)
  - From source:
    * Download `http://www.libsdl.org/release/SDL-1.2.9.tar.gz`
    * Unpack with "`tar zxvf SDL-1.2.9.tar.gz`"
    * Enter the directory ("`cd SDL-1.2.9`")
    * Run the configuration ("`./configure`")
    * Build SDL (run "`make`")
    * Become root ("`su`") and install the library with `make install`
- ODE

- Debian: doesn't have a package, install from source
- Gentoo: emerge ode
- Fedora: doesn't have a package, install from source
- From source:
  * Download `http://easynews.dl.sourceforge.net/sourceforge/opende/ode-0.5.tgz`
  * Unpack with "`tar zxvf ode-0.5.tgz`"
  * Enter the directory ("`cd ode-0.5`")
  * Edit config/user-settings, change `PLATFORM` if necessary
  * Configure ODE (run "`make configure`")
  * Build ODE (run "`make ode-lib`")
  * Become root ("`su`") and install the library and include files:
    - `cp lib/libode.a /usr/local/lib/`
    - `mkdir /usr/local/include`
    - `cp -r include/ode /usr/local/include/`
    - `ranlib /usr/local/lib/libode.a`

Obtaining and building simple_sim under Windows

This should be possible, but I haven't done it, so I'm not sure how... CYGWIN may work, Dev-C++ may work (preferable).

Miscellaneous

- If someone/something drives the robot into a wall too hard/face on, ODE may explode (the physics engine is unable to come up with a suitable solution for a situation with multiple forces that directly complement each other hitting against restrictions. It has a lot to do with the way that ODE implements contacts as points.)