Abstract

Digital marketing is a growing industry, examples of which can be seen all around. In the past few years, advances in display technologies have enabled relatively low-cost digital signage to spring up in any store, whether it is small or large. It is becoming increasing difficult to present to consumers displays with fresh media. A display system that resembles a large scale mobile presents a unique solution for advertising in a large enclosed space. Explore the physics of the complex harmonic motion behind this unique mobile.

Background

In our competitive advertising industry, it is becoming more difficult to create an ad that attracts the attention of a person walking on the street. Digital displays and computers have given rise to digital signage and advertising. It is not uncommon to see entire video clips created and shown on huge jumbotron displays to get the name of a product out into the population. While these displays are bright and tend to produce decent images, these large format displays become common, especially in places such as Las Vegas, Nevada. This is the challenge I intend to take on with my video mobile display design. The purpose of this display will be to capture the attention of onlookers by presenting rich media in a unique manner, one that is less common in the marketplace.

My display, will be similar to a model solar system in that three arms will *orbit* a center sphere, but will not be allowed to complete more than one or two orbits before reversing direction of motion. This will solve the problem of getting data and power cables to the displays on the arms while they orbit.

To manufacture such a display presents a challenge in balancing naturally occurring friction forces with a drive system primarily powered by a spring force to produce a natural-looking, fluid-like, harmonic motion in the arms. The device will be designed to allow a user to choose an initial offset angle (theta) from rest for each of the arms. The restoring force of the spring(s) on each arm will cause the system to exhibit harmonic motion. Since the rotation of the arms will need to be dampened to prevent them from moving too quickly the system will mimic a dampened harmonic oscillator. This damping will arise naturally due to friction forces. To counter these forces, small driving motors will assist the motion and counter these damping forces, in order to keep the system in motion.

It is the task of my computer model simulation to calculate these forces, and predict the spring, resistive, and driving forces required to create this system. Since I intend to use this as a display, I would also like to be able to lock the system for a few seconds in order to play media

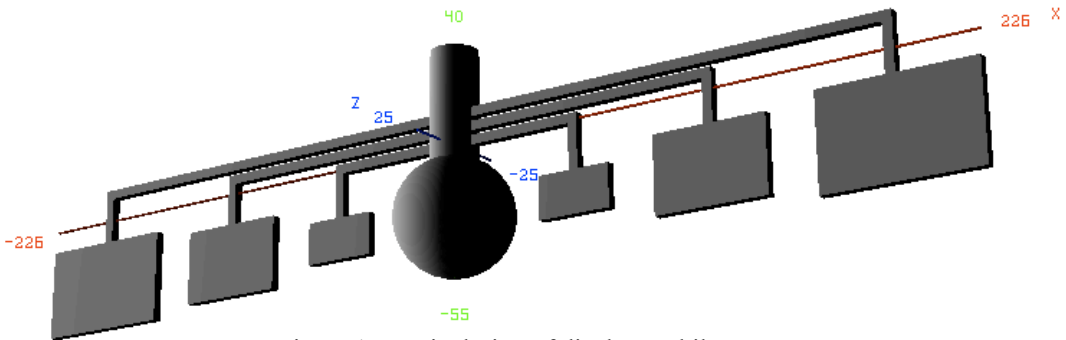

Figure 1 - Basic design of display mobile

that spans all six displays and the center sphere. To do this, I will need to know the positions of the arms in order to trigger the spanned media event when the arms are in alignment. Due to the periodic nature of the motion, I will attempt to calculate the arm positions over time, using my simulated model, rather than build expensive hardware sensors into the mobile.

In the simulator, each of the three arms will have its own equation of motion that will need to be solved simultaneously with the other two arms in order to predict their location over time. When all three arm positions align up at a given position, I can then lock the arm movement and trigger a video playback event.

Methods

To simulate this system, I used ROOT, a data analysis framework developed and maintained by CERN. ROOT enabled me to solve my three equations of motion simultaneously using the fourth-order runge-kutta numerical method of solving differential equations. It also provided a basic three dimensional model of the system.

To begin, I determined my three equations of motion by first determining the moments of inertia for each of the three support bars.

$$I_3 = \frac{1}{12}M_3 * L_3^2 + 2 * (M_3 * R_3^2 + M_3 * (\frac{L_3}{2})^2) + 2 * (\frac{1}{12}M_3(W_3^2 + D_3^2) + M_3(\frac{L_3}{2})^2)$$

$$I_2 = \frac{1}{12}M_2 * L_2^2 + 2 * (M_2 * R_2^2 + M_2 * (\frac{L_2}{2})^2) + 2 * (\frac{1}{12}M_2(W_2^2 + D_2^2) + M_2(\frac{L_2}{2})^2)$$

$$I_1 = \frac{1}{12}M_1 * L_1^2 + 2 * (M_1 * R_1^2 + M_1 * (\frac{L_1}{2})^2) + 2 * (\frac{1}{12}M_1(W_1^2 + D_1^2) + M_1(\frac{L_1}{2})^2)$$

Equations 1-3
Lines 34-46 in code

After deriving the moments of inertia for each bar, I estimated the mass of each bar, as well as the supports and displays on them. The values I ended up are listed below.

Object	Mass (kg)
4.0m top bar	12.50
Large Display	8.25
Large Display Mount	3.00
2.5m middle bar	8.50
Medium Display	6.00
Medium Display Mount	2.00
1.0 m bottom bar	6.00
Small Display	4.00
Small Display Mount	1.00

Table 1 - Object Masses

With two displays and corresponding mounting brackets on each arm, the mass totals for top, middle, and bottom bars were 35.0kg, 24.5kg, and 16.0kg respectively. Using these numbers I found the moments of inertia to be as follows.

I_topBar = 90.0087 kg m^2

I_middleBar = 25.0026 kg m^2

I_bottomBar = 2.5007 kg m^2

With all the basic calculations complete, I moved to ROOT where my first task was to estimate the period of each arm oscillation. I determined that I could use approximately the same spring constant (k) of 0.875 Nm/rad for each arm. Without taking any frictional forces or driving motor acceleration into account I had ROOT estimate the periods based on the following equation. See results section for calculated periods.

$$T = 2\pi\sqrt{\frac{I}{kD}}$$

Equation 4

After calculating the approximate periods of motion I used ROOT, and the adaptive and non-adaptive 4th order runge-kutta methods to simulate the periodic motion of each arm, saving the data points into TTrees. To do this, I modified the DPend.cc class definition to account for the motion of an oscillator with friction forces and driving motor forces, neglecting the effects of gravity since the arms will be locked in a plane perpendicular to the force of gravity. I also modified the TRK4.cc class definition to be compatible with the minor changes that I had made to the DPend.cc class. Using a tau of 0.01s, I found individual plots of the displacement angles in each arm verses time as well as omega verses time for each arm, to double check that I had accurate data.

Results

My ROOT calculations predicted the following periods for the undamped, non-driven version of the mobile system.

Period3 = 90.122s

Period2 = 60.081s

Period1 = 30.043s

With these estimations, I used my simulator to determine the approximate spring and driving motor forces required to provide the system with my desired periods of motion. The results

showed that the arms do in fact re-align every few minutes allowing the mobile to function as a time-based, advertising display. The system aligns every 123.86 seconds, so just a little more than 2 minutes pass between spanned content events. The actual periods of each arm shifted slightly after accounting for frictional forces and driving forces.

My simulation also predicted that the angular velocity of each of the arms does not have to be very high to produce the desired timing. The fastest movement is in the shortest bar since it has the smallest moment of inertia, but never does it move faster than about 30-40 degrees per second. These values can be adjusted by adjusting spring tension or changing the amount of friction forces in the system. This is important if the mobile is to exist safely in a public space. While the display is designed to be hung from a high ceiling, it still must not rotate so quickly that it could present a risk to an observer viewing it from any angle.

Figures and Tables

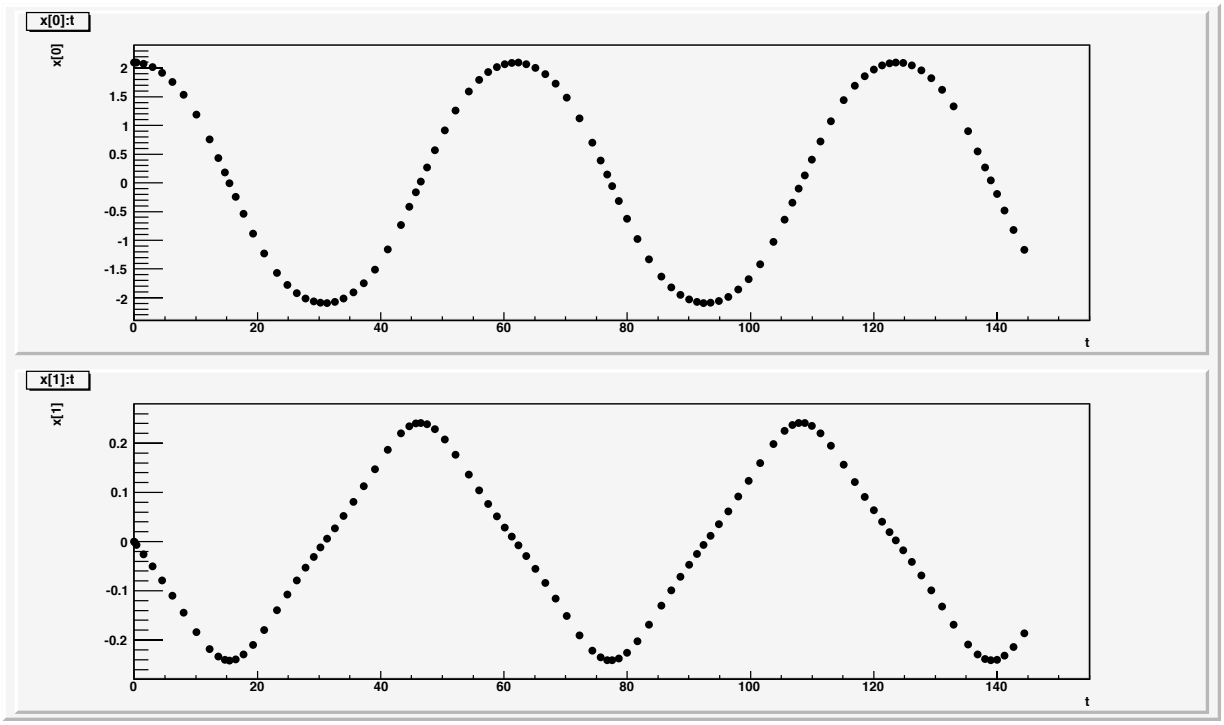

Figure 2 - Top Bar - Displacement vs Time (top) and Omega vs Time (bottom)

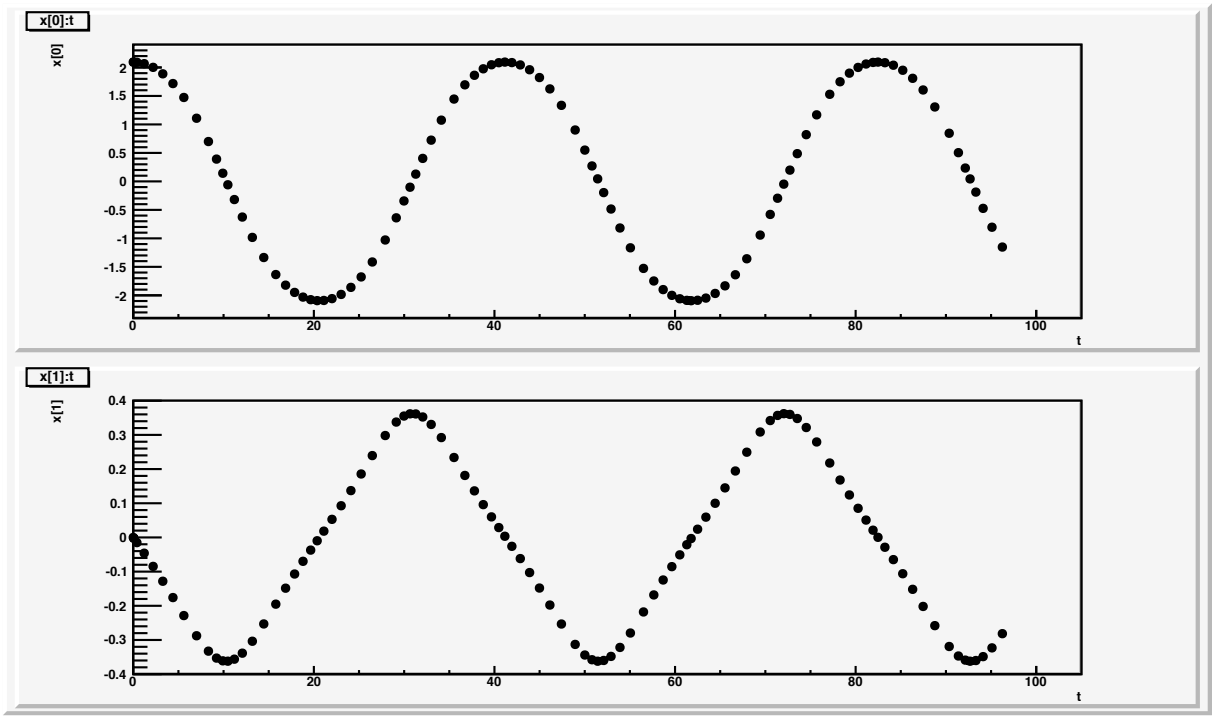

Figure 3 - Middle Bar - Displacement vs Time (top) and Omega vs Time (bottom)

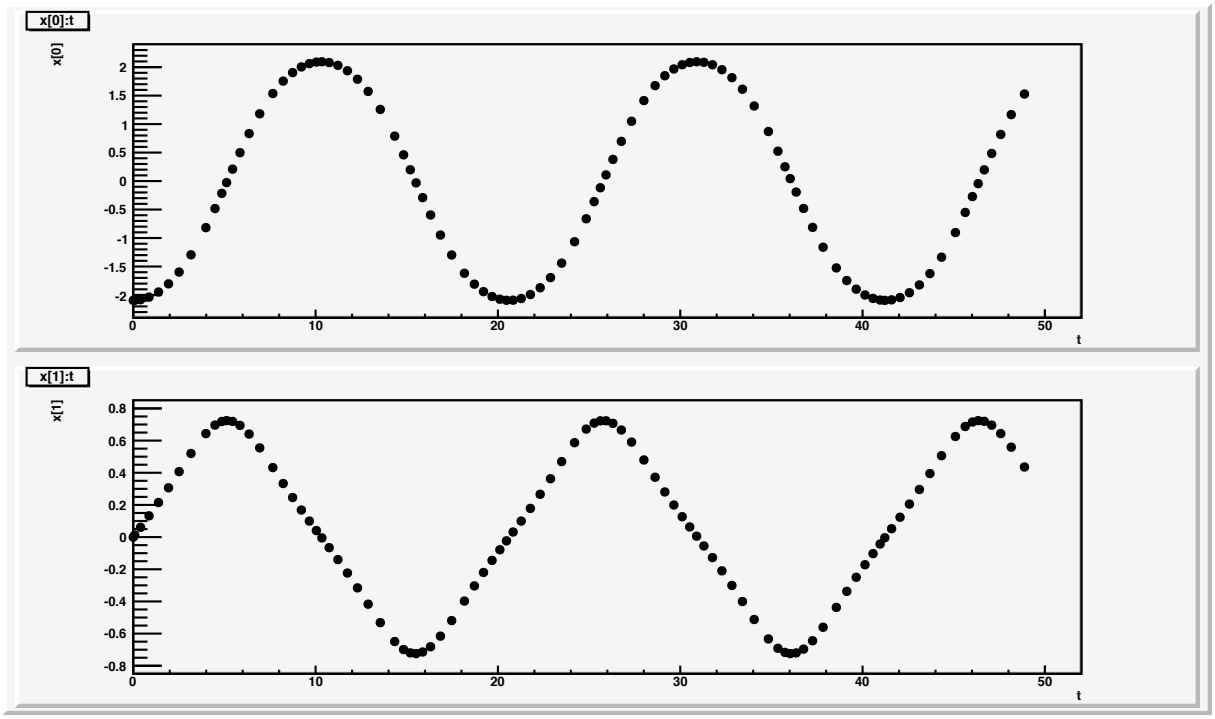

Figure 4 - Bottom Bar - Displacement vs Time (top) and Omega vs Time (bottom)

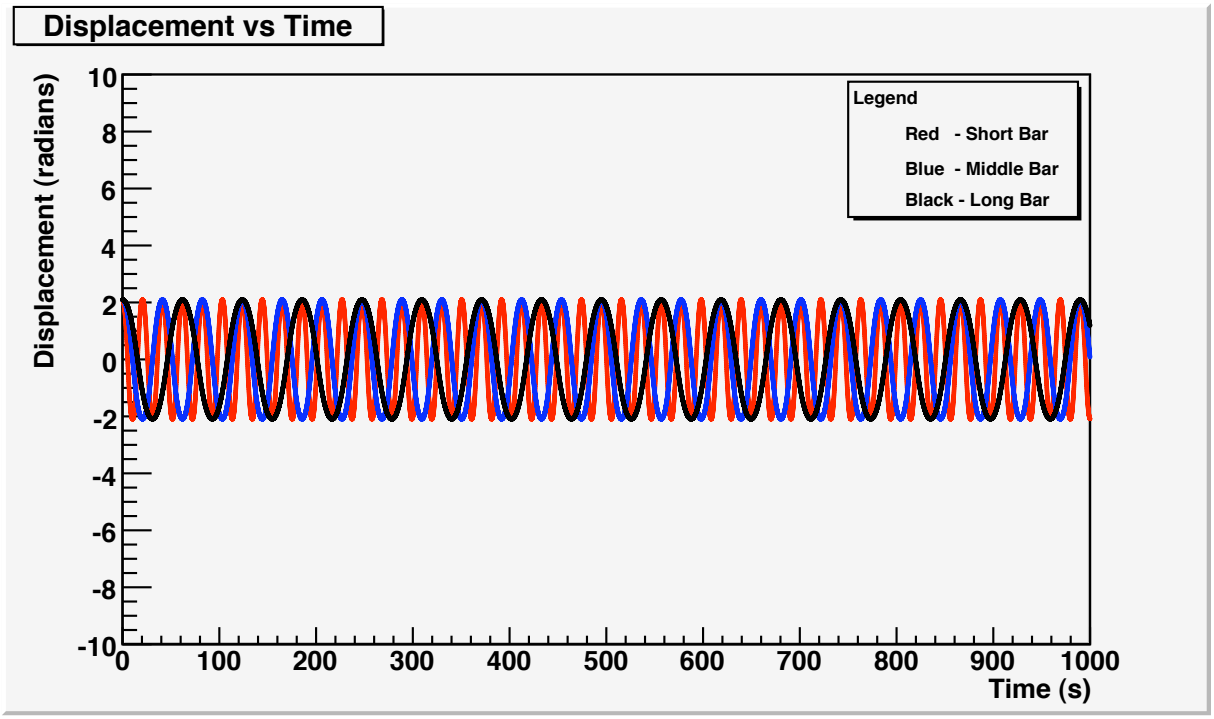

Figure 5 - Complete System - Displacement vs Time

Alignment @ t(s)	Time Between Alignments (in s)
125.57	
249.94	124.37
373.96	124.02
498.84	124.88
621.42	122.58
746.19	124.77
870.25	124.06
992.56	122.31
Average Time Between Alignments	
123.8557143	

Table 2 - Alignment Timing Values

Conclusion

This mobile system exhibits characteristics of a damped, driven harmonic oscillator. While the simulation may take into account, damping forces due to friction, the forces will vary based on environmental conditions and therefore will not always exhibit the exact simulated motion. I will continue to make small corrections to this simulator as I pursue building the mobile. The key piece will be finding coil springs with useable k values that allow the system to move calmly but still have the force to accelerate the arms with high moments of inertia.

When selling ad space on the display, I am able to tell buyers approximately how many times per hour they will see their high priced, full spread ads on the mobile. My simulation predicts that the three arms of the model will align approximately once every two minutes. Using these results I am able to begin constructing the physical model according to my calculated specifications. More importantly, I have a tool that can serve as a software simulator to estimate when the screens are lined up and que a playback event with content spanned across all six displays.

```
1: // *********************************
2: //        Final Project Code
3: //          Nathan Schulz
4: //        nschulz@gmail.com
5: //        Revision 5.6.1111
6: // *********************************
7:
8: // *********************************
9: //      SET UP GLOBAL CONSTANTS
10: // *********************************
11:
12: // Masses of Displays
13: double M_display3 = 8.25; // Mass of outer display in kg
14: double M_display2 = 6.0; // Mass of middle display in kg
15: double M_display1 = 4.0; // Mass of inner display in kg
16:
17: // Widths and Depths of Displays
18: double W_display3 = 0.5; // width of display 3 in m
19: double W_display2 = 0.3; // width of display 2 in m
20: double W_display1 = 0.2; // width of display 1 in m
21:
22: double D_display3 = 0.0254; // depth of display 3 in m
23: double D_display2 = 0.0254; // depth of display 2 in m
24: double D_display1 = 0.0254; // depth of display 1 in m
25:
26: // Masses of Bars
27: double M_topBar     = 12.5; // Mass of top bar in kg
28: double M_middleBar  = 8.5; // Mass of middle bar in kg
29: double M_bottomBar  = 6.0; // Mass of bottom bar in kg
30:
31: double M_smallSupportBar  = 1.0; // Mass of small display support bar
32: double M_mediumSupportBar = 2.0; // Mass of medium display support bar
33: double M_largeSupportBar  = 3.0; // Mass of large display support bar
34:
35: // Lengths of Bars
36: double L_topBar     = 4.0; // Length of top bar in m
37: double L_middleBar  = 2.5; // Length of middle bar in m
38: double L_bottomBar  = 1.0; // Length of bottom bar in m
39:
40: // Moments of Inertia
41: double I_topBar     = (1/12)*M_topBar*L_topBar*L_topBar; // Moment of inertia for top bar alone about
    its center
42: double I_middleBar  = (1/12)*M_middleBar*L_middleBar*L_middleBar; // Moment of inertia for middle bar
    alone about its center
43: double I_bottomBar  = (1/12)*M_bottomBar*L_bottomBar*L_bottomBar; // Moment of inertia for bottom bar
    alone about its center
44:
45: // Add moments of inertia for display supports
46: I_topBar    += 2*(M_largeSupportBar*0.0381*0.0381 +
    M_largeSupportBar*(0.5*L_topBar)*(0.5*L_topBar)); // Add moment of inertia of 2 large support bars
    with radius 1.5in or 3.81cm using parallel axis theorem distance 1/2 bar length
47: I_middleBar += 2*(M_mediumSupportBar*0.0254*0.0254 +
    M_mediumSupportBar*(0.5*L_middleBar)*(0.5*L_middleBar)); // Add moment of inertia of 2 medium support
    bars with radius 1.0in or 2.54cm using parallel axis theorem distance 1/2 bar length
48: I_bottomBar += 2*(M_smallSupportBar*0.01905*0.01905 +
    M_smallSupportBar*(0.5*L_bottomBar)*(0.5*L_bottomBar)); // Add moment of inertia of 2 small support
    bars with radius 0.75in or 1.905cm using parallel axis theorem distance 1/2 bar length
49:
50: // Add moments of inertia for displays
51: I_topBar    += 2*((1/12)*M_display3*(W_display3*W_display3+D_display3*D_display3) +
    M_display3*(0.5*L_topBar)*(0.5*L_topBar)); // Add moments of inertia for 2 large displays of width W
    and depth D at distance 1/2 topBarLength
52: I_middleBar += 2*((1/12)*M_display2*(W_display2*W_display2+D_display2*D_display2) +
    M_display2*(0.5*L_middleBar)*(0.5*L_middleBar)); // Add moments of inertia for 2 medium displays of
    width W and depth D at distance 1/2 middleBarLength
53: I_bottomBar += 2*((1/12)*M_display1*(W_display1*W_display1+D_display1*D_display1) +
    M_display1*(0.5*L_bottomBar)*(0.5*L_bottomBar)); // Add moments of inertia for 2 small displays of
    width W and depth D at distance 1/2 bottomBarLength
```

```
54:
55: double Mtot_topBar      = M_topBar + 2*M_smallSupportBar + 2*M_display3;
56: double Mtot_middleBar   = M_middleBar + 2*M_mediumSupportBar + 2*M_display2;
57: double Mtot_bottomBar   = M_bottomBar + 2*M_smallSupportBar + 2*M_display1;
58:
59: const double pi = 3.141592654;
60:
61: int nStep = 1000;               // number of steps to simulate (steps*tau = s)
62: double k = 0.875;               // spring constant in N*m/rad
63: double tau = 0.02;
64:
65: TEveGeoTopNode* top_eve = 0; // Initialize top_eve for model animation
66:
67:
68: // *********************************
69: //          DEFAULT METHOD
70: // *********************************
71:
72: void schulz_final()
73: {
74:     // Set up basics and draw button bar
75:     cout.precision(20);
76:     TControlBar *bar = new TControlBar("vertical", "Final Project Controls");
77:     bar->AddButton("Show Moments of Inertia", "listMomentsOfInertia()", "Output moments of inertia");
78:     bar->AddButton("Show Plot Long Bar", "runPlot3(1)", "calculate and draw plot for long bar");
79:     bar->AddButton("Show Plot Middle Bar", "runPlot2(1)", "calculate and draw plot for middle bar");
80:     bar->AddButton("Show Plot Short Bar", "runPlot1(1)", "calculate and draw plot for short bar");
81:     bar->AddButton("Run All Plots\nWithout Subplots", "runAllPlots(0)", "calculate and draw plot for
    All bars");
82:     bar->AddButton("Run All Plots\nWith Subplots", "runAllPlots(1)", "calculate and draw plot for All
    bars");
83:
84:
85:     bar->AddButton("Show Model", "renderModel()", "Build and show basic mobile model");
86:     bar->AddButton("Animate Model", "animateModel()", "Animate mobile model");
87:     bar->AddButton("Show Periods", "calcPeriods()", "Calculate and Display estimated periods");
88:     bar->AddButton("Reset", "gROOT->Reset()", "Reset ROOT environment");
89:     bar->AddButton("Quit", ".q", "Quit ROOT Demonstration");
90:     bar->Show();
91: }
92:
93: void calcPeriods(){
94:     cout.precision(20);
95:     double sqrtNum = I_bottomBar/((k/Mtot_bottomBar)*Mtot_bottomBar*L_bottomBar/2);
96:     double period = 4*pi*pow(sqrtNum, 0.5);
97:     std::cout<<"Period1: "<<period<<"s"<<std::endl;
98:     double sqrtNum = I_middleBar/((k/Mtot_middleBar)*Mtot_middleBar*L_middleBar/2);
99:     double period = 4*pi*pow(sqrtNum, 0.5);
100:    std::cout<<"Period2: "<<period<<"s"<<std::endl;
101:    double sqrtNum = I_topBar/((k/Mtot_topBar)*Mtot_topBar*L_topBar/2);
102:    double period = 4*pi*pow(sqrtNum, 0.5);
103:    std::cout<<"Period3: "<<period<<"s"<<std::endl;
104: }
105: void runAllPlots(int showPlot) {
106:     cout.precision(20);
107:
108:     std::cout << "Running All Plots" << std::endl;
109:     std::cout << "Beginning Plot 1" << std::endl;
110:
111:     double sqrtNum = I_bottomBar/((k/Mtot_bottomBar)*Mtot_bottomBar*L_bottomBar/2);
112:     double period = 4*pi*pow(sqrtNum, 0.5);
113:     std::cout<<"Period1: "<<period<<"s"<<std::endl;
114:     double sqrtNum = I_middleBar/((k/Mtot_middleBar)*Mtot_middleBar*L_middleBar/2);
115:     double period = 4*pi*pow(sqrtNum, 0.5);
116:     std::cout<<"Period2: "<<period<<"s"<<std::endl;
117:     double sqrtNum = I_topBar/((k/Mtot_topBar)*Mtot_topBar*L_topBar/2);
118:     double period = 4*pi*pow(sqrtNum, 0.5);
119:     std::cout<<"Period3: "<<period<<"s"<<std::endl;
```

```
120:
121:     gROOT->ProcessLine(".L D.cc+");
122:     gROOT->ProcessLine(".L DPend.cc+");
123:     gROOT->ProcessLine(".L TRK4.cc+");
124:
125:     std::cout << "Starting clock..." << std::endl;
126:
127:     double stime = gROOT->ProcessLine(".! perl -e 'use Time::HiRes qw(time); print time'");
128:     std::cout<<" "<<std::endl;
129:
130:     double theta0 = 120.0;          // Initial displacement angle in degrees
131:     double theta = theta0*pi/180;   // Convert angle to radians
132:     double omega = 0.0;             // Set the initial velocity in rads/s
133:     double alpha = -0.5;            // Set initial acceleration in rads/s/s
134:
135:     //* Set the physical constants and other variables
136:     //double k = 0.695;                 // spring constant in N*m/rad
137:     double time = 0.0;              // Initial time
138:     double time_old;                // Time of previous reversal
139:
140:     int iStep;
141:     double p[4] = {Mtot_bottomBar, L_bottomBar/2, I_bottomBar, k}; // params that get passed to DPend
     {mass, distanceFromAxis, MOI, k(spring const)}
142:     double x0[3] = {theta,omega,alpha};
143:
144:     std::cout<<"Calculations completed at:"<<std::endl;
145:     double htime = gROOT->ProcessLine(".! perl -e 'use Time::HiRes qw(time); print time'");
146:     std::cout<<" "<<std::endl;
147:
148:
149:     DPend *g = new DPend(4,4,p);
150:     TRK4 *t1 = new TRK4("trk4","trk4", true, nStep, tau, x0, g);
151:
152:     if (showPlot){
153:         runPlot1(1);
154:     }
155:     std::cout<<"Complete Time:"<<std::endl;
156:     double etime = gROOT->ProcessLine(".! perl -e 'use Time::HiRes qw(time); print time'");
157:     std::cout<<" "<<std::endl;
158:
159:
160:     std::cout << "Beginning Plot 2" << std::endl;
161:
162:     std::cout << "Starting clock..." << std::endl;
163:
164:     stime = gROOT->ProcessLine(".! perl -e 'use Time::HiRes qw(time); print time'");
165:     std::cout<<" "<<std::endl;
166:
167:      theta0 = 120.0;            // Initial displacement angle in degrees
168:      theta = theta0*pi/180;   // Convert angle to radians
169:      omega = 0.0;               // Set the initial velocity in rads/s
170:      alpha = -0.5;          // Set initial acceleration in rads/s/s
171:
172:     //* Set the physical constants and other variables
173:      //k = 0.695;               // spring constant in N*m/rad
174:      time = 0.0;              // Initial time
175:      //tau = 0.01;
176:
177:      double p2[4] = {Mtot_middleBar, L_middleBar/2, I_middleBar, k}; // params that get passed to
     DPend {mass, distanceFromAxis, MOI, k(spring const)}
178:      double x0_2[3] = {theta,omega,alpha};
179:
180:     std::cout<<"Calculations completed at:"<<std::endl;
181:      htime = gROOT->ProcessLine(".! perl -e 'use Time::HiRes qw(time); print time'");
182:     std::cout<<" "<<std::endl;
183:
184:
185:     DPend *g2 = new DPend(4,4,p2);
```

```
186:     TRK4 *t2 = new TRK4("trk4","trk4", true, nStep, tau, x0_2, g2);
187:
188:     if (showPlot){
189:         runPlot2(1);
190:     }
191:
192:     std::cout<<"Complete Time:"<<std::endl;
193:      etime = gROOT->ProcessLine(".! perl -e 'use Time::HiRes qw(time); print time'");
194:     std::cout<<" "<<std::endl;
195:
196:
197:     std::cout << "Beginning Plot 3" << std::endl;
198:
199:     std::cout << "Starting clock..." << std::endl;
200:
201:      stime = gROOT->ProcessLine(".! perl -e 'use Time::HiRes qw(time); print time'");
202:     std::cout<<" "<<std::endl;
203:
204:      theta0 = 120.0;            // Initial displacement angle in degrees
205:      theta = theta0*pi/180;     // Convert angle to radians
206:      omega = 0.0;               // Set the initial velocity in rads/s
207:      alpha = -0.5;              // Set initial acceleration in rads/s/s
208:
209:     //* Set the physical constants and other variables
210:      time = 0.0;                // Initial time
211:      time_old;                  // Time of previous reversal
212:      //tau = 0.01;
213:
214:      double p3[4] = {M_topBar, L_topBar/2, I_topBar, k}; // params that get passed to DPend {mass,
    distanceFromAxis, MOI, k(spring const)}
215:      double x0_3[3] = {theta,omega,alpha};
216:
217:     std::cout<<"Calculations completed at:"<<std::endl;
218:      htime = gROOT->ProcessLine(".! perl -e 'use Time::HiRes qw(time); print time'");
219:     std::cout<<" "<<std::endl;
220:
221:
222:     DPend *g3 = new DPend(4,4,p3);
223:     TRK4 *t3 = new TRK4("trk4","trk4", true, nStep, tau, x0_3, g3);
224:
225:     if (showPlot) {
226:         runPlot3(1);
227:     }
228:     std::cout<<"Complete Time:"<<std::endl;
229:      etime = gROOT->ProcessLine(".! perl -e 'use Time::HiRes qw(time); print time'");
230:     std::cout<<" "<<std::endl;
231:
232:     // DRAW data on plot
233:     TCanvas *c = new TCanvas("c","Model Simulation", 1400,850);
234:     TH2D *h = new TH2D("h","Displacement vs Time", 1, 0, nStep/10, 1, -10, 10);
235:     h->GetXaxis()->SetTitle("Time (s)");
236:     h->GetYaxis()->SetTitle("Displacement (radians)");
237:     h->Draw();
238:
239:
240:     // Save out TTree data sets
241:
242:     //gROOT->ProcessLine("t1->fT->Scan('*'); >t1_TTree.log");
243:     //gROOT->ProcessLine("t2->fT->Scan('*'); >t2_TTree.log");
244:     //gROOT->ProcessLine("t3->fT->Scan('*'); >t3_TTree.log");
245:
246:     //t1->fT->StartViewer();
247:     //t2->fT->StartViewer();
248:     //t3->fT->StartViewer();
249:
250:     //gROOT->ProcessLine("tv__tree->SetScanField(0);");
251:     //gROOT->ProcessLine("tv__tree->Scan('*','','', 9000, 0); >tree1.data");
252:
```

```
253:
254:
255:     t1->fT->SetMarkerColor(kRed);
256:     t1->fT->SetMarkerSize(0.4);
257:     t1->fT->Draw("x[0]:t","", "same");              // Add Short Bar data
258:
259:     t2->fT->SetMarkerColor(kBlue);
260:     t2->fT->SetMarkerSize(0.4);
261:     t2->fT->Draw("x[0]:t","", "same");              // Add Medium bar data
262:
263:     t3->fT->SetMarkerSize(0.4);
264:     t3->fT->Draw("x[0]:t","", "same");              // Add Long Bar Data
265:
266:     gStyle->SetOptStat("");                         // Remove default histogram legend
267:
268:     legend = new TLegend(0.7,0.7,0.89,0.89);                // Draw Legend
269:     legend->AddEntry(t1->fT, "Red   - Short Bar", "1");
270:     legend->AddEntry(t2->fT, "Blue  - Middle Bar", "2");
271:     legend->AddEntry(t3->fT, "Black - Long Bar", "3");
272:     legend->SetHeader("Legend");
273:     legend->Draw();
274:
275: }
276: // Period of 30
277: void runPlot1(int showPlot)
278: {
279:     cout.precision(20);
280:     std::cout << "Beginning Plot 1" << std::endl;
281:     gROOT->ProcessLine(".L D.cc+");
282:     gROOT->ProcessLine(".L DPend.cc+");
283:     gROOT->ProcessLine(".L TRK4.cc+");
284:
285:     std::cout << "Starting clock..." << std::endl;
286:
287:     double stime = gROOT->ProcessLine(".! perl -e 'use Time::HiRes qw(time); print time'");
288:     std::cout<<" "<<std::endl;
289:
290:     double theta0 = -120.0;         // Initial displacement angle in degrees
291:     double theta = theta0*pi/180;   // Convert angle to radians
292:     double omega = 0.0;             // Set the initial velocity in rads/s
293:     double alpha = -0.5;            // Set initial acceleration in rads/s/s
294:
295:     //* Set the physical constants and other variables
296:     //double k = 0.695;                 // spring constant in N*m/rad
297:     double time = 0.0;              // Initial time
298:     double time_old;                // Time of previous reversal
299:
300:     int iStep;
301:     double p[4] = {Mtot_bottomBar, L_bottomBar/2, I_bottomBar, k}; // params that get passed to DPend
    {mass, distanceFromAxis, MOI, k(spring const)}
302:     double x0[3] = {theta,omega,alpha};
303:
304:     std::cout<<"Calculations completed at:"<<std::endl;
305:     double htime = gROOT->ProcessLine(".! perl -e 'use Time::HiRes qw(time); print time'");
306:     std::cout<<" "<<std::endl;
307:
308:
309:     DPend *g = new DPend(4,4,p);
310:     TRK4 *t1 = new TRK4("trk4","trk4", true, nStep, tau, x0, g);
311:
312:     if (showPlot){
313:         TCanvas *c1 = new TCanvas("c1","Model Simulation for Bottom Bar", 1400,850);
314:         c1->Divide(1,2);
315:         c1->cd(1);
316:         t1->fT->Draw("x[0]:t");
317:         c1->cd(2);
318:         t1->fT->Draw("x[1]:t");
319:     }
```

```
320:     std::cout<<"Complete Time:"<<std::endl;
321:     double etime = gROOT->ProcessLine(".! perl -e 'use Time::HiRes qw(time); print time'");
322:     std::cout<<" "<<std::endl;
323:
324:     t1->fT->StartViewer();
325:
326: }
327:
328:
329: // Period of 60s
330: void runPlot2(int showPlot)
331: {
332:     cout.precision(20);
333:     std::cout << "Beginning Plot 2" << std::endl;
334:     gROOT->ProcessLine(".L D.cc+");
335:     gROOT->ProcessLine(".L DPend.cc+");
336:     gROOT->ProcessLine(".L TRK4.cc+");
337:
338:     std::cout << "Starting clock..." << std::endl;
339:
340:     double stime = gROOT->ProcessLine(".! perl -e 'use Time::HiRes qw(time); print time'");
341:     std::cout<<" "<<std::endl;
342:
343:     double theta0 = 120.0;          // Initial displacement angle in degrees
344:     double theta = theta0*pi/180;   // Convert angle to radians
345:     double omega = 0.0;             // Set the initial velocity in rads/s
346:     double alpha = -0.5;            // Set initial acceleration in rads/s/s
347:
348:     int iStep;
349:     double p[4] = {Mtot_middleBar, L_middleBar/2, I_middleBar, k}; // params that get passed to DPend
    {mass, distanceFromAxis, MOI, k(spring const)}
350:     double x0[3] = {theta,omega,alpha};
351:
352:     std::cout<<"Calculations completed at:"<<std::endl;
353:     double htime = gROOT->ProcessLine(".! perl -e 'use Time::HiRes qw(time); print time'");
354:     std::cout<<" "<<std::endl;
355:
356:
357:     DPend *g = new DPend(4,4,p);
358:     TRK4 *t2 = new TRK4("trk4","trk4", true, nStep, tau, x0, g);
359:
360:     if (showPlot){
361:         TCanvas *c2 = new TCanvas("c2","Model Simulation for Middle Bar", 1400,850);
362:         c2->Divide(1,2);
363:         c2->cd(1);
364:         t2->fT->Draw("x[0]:t");
365:         c2->cd(2);
366:         t2->fT->Draw("x[1]:t");
367:     }
368:
369:     std::cout<<"Complete Time:"<<std::endl;
370:     double etime = gROOT->ProcessLine(".! perl -e 'use Time::HiRes qw(time); print time'");
371:     std::cout<<" "<<std::endl;
372:
373:     t2->fT->StartViewer();
374: }
375:
376:
377: // Period of 90s
378: void runPlot3(int showPlot)
379: {
380:     cout.precision(20);
381:     std::cout << "Beginning Plot 3" << std::endl;
382:     gROOT->ProcessLine(".L D.cc+");
383:     gROOT->ProcessLine(".L DPend.cc+");
384:     gROOT->ProcessLine(".L TRK4.cc+");
385:
386:     std::cout << "Starting clock..." << std::endl;
```

```
387:
388:     double stime = gROOT->ProcessLine(".! perl -e 'use Time::HiRes qw(time); print time'");
389:     std::cout<<" "<<std::endl;
390:
391:     double theta0 = 120.0;          // Initial displacement angle in degrees
392:     double theta = theta0*pi/180;   // Convert angle to radians
393:     double omega = 0.0;             // Set the initial velocity in rads/s
394:     double alpha = -0.5;            // Set initial acceleration in rads/s/s
395:
396:     //* Set the physical constants and other variables
397:     //double k = 0.695;                 // spring constant in N*m/rad
398:     double time = 0.0;              // Initial time
399:     double time_old;                // Time of previous reversal
400:
401:     int iStep;
402:     double p[4] = {M_topBar, L_topBar/2, I_topBar, k}; // params that get passed to DPend {mass,
    distanceFromAxis, MOI, k(spring const)}
403:     double x0[3] = {theta,omega,alpha};
404:
405:     std::cout<<"Calculations completed at:"<<std::endl;
406:     double htime = gROOT->ProcessLine(".! perl -e 'use Time::HiRes qw(time); print time'");
407:     std::cout<<" "<<std::endl;
408:
409:
410:     DPend *g = new DPend(4,4,p);
411:     TRK4 *t3 = new TRK4("trk4","trk4", true, nStep, tau, x0, g);
412:
413:     if (showPlot) {
414:         TCanvas *c3 = new TCanvas("c3","Model Simulation for Long Bar", 1400,850);
415:         c3->Divide(1,2);
416:         c3->cd(1);
417:         t3->fT->Draw("x[0]:t");
418:         c3->cd(2);
419:         t3->fT->Draw("x[1]:t");
420:     }
421:     std::cout<<"Complete Time:"<<std::endl;
422:     double etime = gROOT->ProcessLine(".! perl -e 'use Time::HiRes qw(time); print time'");
423:     std::cout<<" "<<std::endl;
424:
425:     t3->fT->StartViewer();
426: }
427:
428: // Method for calculating and returning moments of inertia globally
429: double momentOfInertia(int forBar){
430:
431:     switch(forBar){
432:         case 3:
433:             return I_topBar;
434:             break;
435:         case 2:
436:             return I_middleBar;
437:             break;
438:         case 1:
439:             return I_bottomBar;
440:             break;
441:         default:
442:             return 0;
443:             break;
444:     }
445:
446: }
447: void listMomentsOfInertia()
448: {
449:
450:     std::cout << "Moments of Inertia:" << std::endl;
451:     std::cout << "I_topBar: " << momentOfInertia(3) << " kg m^2" << std::endl;
452:     std::cout << "I_middleBar: " << momentOfInertia(2) << " kg m^2" << std::endl;
453:     std::cout << "I_bottomBar: " << momentOfInertia(1) << " kg m^2" << std::endl;
```

```
454: }
455:
456:
457: // Macro to create and render a mobile design in using ROOT and OpenGL
458: // Code inspired by ROOT tutorial $ROOYSYS/tutorials/geom/rootgeom.C by Andrei Gheata
459:
460: // Mobile design and code by Nate Schulz (nschulz@gmail.com)
461: // Animation assistance provided by Rene Brun (CERN), Andrei Gheata (CERN), Matevz on ROOT Talk
     Forums
462: // and Prof. Dominguez (UNL). Additional information can be found at:
463: // http://root.cern.ch/phpBB2/viewtopic.php?p=27387#27387
464:
465: void renderModel()
466: {
467:
468:         gStyle->SetCanvasPreferGL(true);
469:         gSystem->Load("libGeom");
470:         TGeoManager *geom = new TGeoManager("simple1", "Simple geometry");
471:
472:         //--- define some materials
473:         TGeoMaterial *matVacuum = new TGeoMaterial("Vacuum", 0,0,0);
474:         TGeoMaterial *matAl = new TGeoMaterial("Al", 26.98,13,2.7);
475:         //   //--- define some media
476:         TGeoMedium *Vacuum = new TGeoMedium("Vacuum",1, matVacuum);
477:         TGeoMedium *Al = new TGeoMedium("Root Material",2, matAl);
478:
479:         // Create 3D Space
480:         TGeoVolume *top = geom->MakeBox("TOP", Vacuum, 270., 270., 120.);
481:         geom->SetTopVolume(top);
482:
483:         TGeoVolume *model = geom->MakeBox("ROOT", Vacuum, 110., 50., 5.);
484:         model->SetVisibility(kFALSE);
485:
486:         TGeoVolume *mainShaft = geom->MakeTube("Main Shaft", Al, 5, 10, 40);
487:         mainShaft->SetLineColor(kWhite);
488:         model->AddNode(mainShaft, 1, new TGeoRotation("rot5",0,0,90,180,90,270));
489:
490:     // Shortest Bar with displays
491:
492:         TGeoVolume *lowShortBar = geom->MakeBox("lowShortBar", Al, 60, 2., 2.);
493:         lowShortBar->SetLineColor(kWhite);
494:         model->AddNode(lowShortBar, 1, new TGeoCombiTrans());
495:
496:         TGeoVolume *bar1LeftDrop = geom->MakeBox("bar1LeftDrop", Al, 2, 16, 2);
497:         bar1LeftDrop->SetLineColor(kWhite);
498:         lowShortBar->AddNode(bar1LeftDrop, 2, new TGeoTranslation(-58, -14., 0.));
499:
500:         TGeoVolume *bar1LeftDisplay = geom->MakeBox("bar1LeftDisplay", Al, 16, 9, 2);
501:         bar1LeftDisplay->SetLineColor(kWhite);
502:         lowShortBar->AddNode(bar1LeftDisplay, 2, new TGeoTranslation(-58, -30., 0.));
503:
504:         TGeoVolume *bar1RightDrop = geom->MakeBox("bar1RightDrop", Al, 2, 16, 2);
505:         bar1RightDrop->SetLineColor(kWhite);
506:         lowShortBar->AddNode(bar1RightDrop, 2, new TGeoTranslation(58, -14., 0.));
507:
508:         TGeoVolume *bar1RightDisplay = geom->MakeBox("bar1RightDisplay", Al, 16, 9, 2);
509:         bar1RightDisplay->SetLineColor(kWhite);
510:         lowShortBar->AddNode(bar1RightDisplay, 2, new TGeoTranslation(58, -30., 0.));
511:
512:
513:     // Medium Bar with displays
514:         TGeoVolume *middleMediumBar = geom->MakeBox("middleMediumBar", Al, 120, 2., 2.);
515:         middleMediumBar->SetLineColor(kWhite);
516:         TGeoCombiTrans *middle = new TGeoCombiTrans();
517:         middle->SetTranslation(0, 6, 0);
518:         model->AddNode(middleMediumBar, 1, middle);
519:
520:         TGeoVolume *bar2LeftDrop = geom->MakeBox("bar2LeftDrop", Al, 2, 25, 2);
```

```
521:          bar2LeftDrop->SetLineColor(kWhite);
522:          middleMediumBar->AddNode(bar2LeftDrop, 2, new TGeoTranslation(-118, -18., 0.));
523:
524:          TGeoVolume *bar2LeftDisplay = geom->MakeBox("bar2LeftDisplay", Al, 24, 16, 2);
525:          bar2LeftDisplay->SetLineColor(kWhite);
526:          middleMediumBar->AddNode(bar2LeftDisplay, 2, new TGeoTranslation(-118, -30., 0.));
527:
528:          TGeoVolume *bar2RightDrop = geom->MakeBox("bar2RightDrop", Al, 2, 25, 2);
529:          bar2RightDrop->SetLineColor(kWhite);
530:          middleMediumBar->AddNode(bar2RightDrop, 2, new TGeoTranslation(118, -18., 0.));
531:
532:          TGeoVolume *bar2RightDisplay = geom->MakeBox("bar2RightDisplay", Al, 24, 16, 2);
533:          bar2RightDisplay->SetLineColor(kWhite);
534:          middleMediumBar->AddNode(bar2RightDisplay, 2, new TGeoTranslation(118, -30., 0.));
535:
536:      // Long Bar with displays
537:          TGeoVolume *topLongBar = geom->MakeBox("topLongBar", Al, 194, 2., 2.);
538:          topLongBar->SetLineColor(kWhite);
539:          TGeoCombiTrans *topbar = new TGeoCombiTrans();
540:          topbar->SetTranslation(0, 12, 0);
541:          model->AddNode(topLongBar, 1, topbar);
542:
543:          TGeoVolume *bar3LeftDrop = geom->MakeBox("bar3LeftDrop", Al, 2, 24, 2);
544:          bar3LeftDrop->SetLineColor(kWhite);
545:          topLongBar->AddNode(bar3LeftDrop, 2, new TGeoTranslation(-192, -14., 0.));
546:
547:          TGeoVolume *bar3LeftDisplay = geom->MakeBox("bar3LeftDisplay", Al, 32, 20, 2);
548:          bar3LeftDisplay->SetLineColor(kWhite);
549:          topLongBar->AddNode(bar3LeftDisplay, 2, new TGeoTranslation(-194, -30., 0.));
550:
551:          TGeoVolume *bar3RightDrop = geom->MakeBox("bar3RightDrop", Al, 2, 24, 2);
552:          bar3RightDrop->SetLineColor(kWhite);
553:          topLongBar->AddNode(bar3RightDrop, 2, new TGeoTranslation(192, -14., 0.));
554:
555:          TGeoVolume *bar3RightDisplay = geom->MakeBox("bar3RightDisplay", Al, 32, 20, 2);
556:          bar3RightDisplay->SetLineColor(kWhite);
557:          topLongBar->AddNode(bar3RightDisplay, 2, new TGeoTranslation(194, -30., 0.));
558:
559:
560:      // Video Sphere
561:          TGeoVolume *sphere = geom->MakeSphere("sphere", Al, 0, 25.0, 0, 180.0, 0, 360.0);
562:          sphere->SetLineColor(kWhite);
563:          gGeoManager->GetVolume("sphere")->SetTransparency(20);
564:          model->AddNode(sphere, 1, new TGeoTranslation(0, -30., 0.));
565:
566:
567:
568:          top->AddNode(model, 1, new TGeoTranslation(0, 0, 0));
569:
570:          //--- close the geometry
571:          geom->CloseGeometry();
572:
573:          // top->Draw("ogl");
574:
575:          TEveManager::Create();
576:          top_eve = new TEveGeoTopNode(geom, geom->GetTopNode());
577:          top_eve->SetVisOption(0);
578:          gEve->AddGlobalElement(top_eve);
579:          gEve->Redraw3D(kTRUE);
580:
581:          // For some strange reason the visibility option is not
582:          // honoured on first draw ... force immediate redraw.
583:          gEve->DoRedraw3D();
584:          gEve->FullRedraw3D();
585: }
586:
587:
588:
```

```
589: void animateModel()
590: {
591:     std::cout<<"Loading Animation Data... Please Wait..."<<std::endl;
592:     ifstream in;
593:     in.open("animationData.dat");
594:     double x1[10000],x2[10000],x3[10000],aligned[10000];
595:     for (int aniStep = 0; aniStep<9995; aniStep++) {
596:       in >> x1[aniStep] >> x2[aniStep] >> x3[aniStep] >> aligned[aniStep];      // Load animation
    data with pre-determined alignment points
597:
598:         // Calculate alignment points
599: //      if (x1[aniStep] <= 0.009 && x1[aniStep] >= -0.009 && x2[aniStep] <= 0.009 && x2[aniStep] >=
    -0.009 && x3[aniStep] <= 0.009 && x3[aniStep] >= -0.009){
600: //          aligned[aniStep] = 1;
601: //          std::cout<<"Alignment at step:"<< aniStep << std::endl;
602: //      }
603: //      else aligned[aniStep] = 0;
604:     }
605:     in.close();
606:
607:   for (Int_t i=0; i<9995; ++i)
608:   {
609:     // Sum multiple steps to increase animation speed at 60fps
610:     i++;
611:     i++;
612:     i++;
613:     i++;
614:     i++;
615:
616:     gGeoManager->cd("/TOP_1/ROOT_1/lowShortBar_1");
617:     gGeoManager->GetCurrentNode()->GetMatrix()-
    >RotateY(x1[i-5]+x1[i-4]+x1[i-3]+x1[i-2]+x1[i-1]+x1[i]);
618:
619:     gGeoManager->cd("/TOP_1/ROOT_1/middleMediumBar_1");
620:     gGeoManager->GetCurrentNode()->GetMatrix()-
    >RotateY(x2[i-5]+x2[i-4]+x2[i-3]+x2[i-2]+x2[i-1]+x2[i]);
621:
622:     gGeoManager->cd("/TOP_1/ROOT_1/topLongBar_1");
623:     gGeoManager->GetCurrentNode()->GetMatrix()-
    >RotateY(x3[i-5]+x3[i-4]+x3[i-3]+x3[i-2]+x3[i-1]+x3[i]);
624:
625:     top_eve->ElementChanged();
626:     gEve->FullRedraw3D();
627:
628:     // If screens are in alignment, pause for 1 second to highlight the fact that they're aligned
629:     if (aligned[i] == 1||aligned[i-1] == 1||aligned[i-2] == 1||aligned[i-3] == 1||aligned[i-4] == 1||
    aligned[i-5] == 1) gSystem->Sleep(1000);
630:     gSystem->Sleep(1);
631:   }
632: }
633:
634:
635: void reloadAndDraw()
636: {
637:     // Basic method to reload and redraw 3D model
638:     // Used for development purposes
639:     gROOT->ProcessLine(".x schulz_final.C");
640:     gROOT->ProcessLine("showModel()");
641: }
```

References

Brun, R. The ROOT Users Guide 5.14. Dec 2006.

Fowles, G. and Cassiday, G. Analytical Mechanics 6th ed.. Orlando: Harcourt Brace & Company, 1999.

Garcia, A. Numerical Methods for Physics 2nd ed. Upper Saddle River: Prentice-Hall, Inc, 2000.

Root Talk: Discussion Forums. CERN. 5 May. 2008 <http://root.cern.ch/phpBB2/viewtopic.php?p=27387#27387>

ROOT Reference Guide. CERN. 5 May. 2008 <http://root.cern.ch/root/Reference.html>

Young, H. and Freedman, R. University Physics Volume 1 12th ed. San Francisco: Pearson Addison-Wesley, 2008.