A Case Study: A Self-Propagating Continuous Differential System as Limiting Discrete Numerical Construct and Device of Sorting

Thomas Mildorf. TJHSST. Computer Systems Research.

May 31, 2005

1 Abstract

Apology. In a manufacturing environment, it is crucial to establish a high standard of quality control while at the same time maintaining a balanced budget. Robustness of production as well as the minimization of risk are also of concern. Ergo, simple, automated techniques for weeding out defective pieces are desirable. It is the intention of this project to analyze the effectiveness of one such technique.

Purpose. It is not uncommon for constructs to be delivered by conveyor belts as they are processed in a factory. Their continuous motion, when directed over the end of such a surface, induces a certain rotation that accompanies each item during the pursuant fall. Proposed is to sort these items by exploiting variance in this rotation via the precise positioning of one or more slots. It is hoped that this motion will be sufficiently sensitive to deformation as for this procedure to be feasible.

Scope and Procedure. The scope of this project is exploratory in nature; there is no sense in attempting to develop a general method. Thus, we will work with a rather simple subset of possible pieces. Moreover, due to logistic constraints, experimentation will be conducted primarily within a digitally rendered environment. The model will be coded from scratch so as to give me total control of the physics involved, and repeated trials with dependent variance will be employed to discern the efficacy of this sorting technique.

2 Model Synthesis

Derivation of Theoretical Equations. We will work under admittedly simplistic circumstances, assuming that the objects being sorted are approximated by a rectangular block of length, height, and depth l, h, and d respectively. We call the generic block B. We suppose furthermore that there exists a uniform density δ within B, so that its mass M, generally given by

$$M = \iiint_V \delta dV$$

is instead given by $M = hld\delta$. In a real environment, products are typically delivered by conveyor belt. Due to aforementioned logistic constraints, we use the approximation of an inclined plane, which we call P. B will be released from the top of P.

We write θ for the angle beween P and the gravitational equipotential contour. Let μ_s and μ_k denote the static and kinetic coefficients of friction, respectively, between B and P. Under our assumptions of uniform density, the relevant calculations are straightforward. If $\mu_s \geq \tan(\theta)$, then no motion results due to static friction, otherwise B moves under the force of gravity and friction. If $\theta > \frac{\pi}{2} - \tan^{-1}\left(\frac{h}{l}\right)$, then the block tumbles down the incline; we assert that this is not the case. The normal component of the contact force, F_n, is given by $|F_n| = M_B g \cos(\theta)$, and the frictional component, F_k, by $|F_k| = M_B g \mu_k \cos(\theta)$. B then slides down P along path C until enough of it hangs over the edge of P for it to begin to rotate. Let us call this phase of sliding *initiation*. Initiation is governed by Newton's 2nd Law applied in the direction $\hat{x}$, with the positive direction

pointing straight down P:

$$\begin{aligned}
\sum F_{\hat{x}} = F_{g_{\hat{x}}} - F_k &= M_B a_{\hat{x}} \\
M_B g \sin(\theta) - M_B g \mu_k \cos(\theta) &= M_B a_{\hat{x}} \\
a_{\hat{x}} &= g\left(\sin(\theta) - \mu_k \cos(\theta)\right)
\end{aligned}$$

If the plane has length D in the $\hat{x}$ direction, B slides a distance of $D - \frac{1}{2}\left(l + h\tan(\theta)\right)$ straight down the plane, after which it begins to pivot about the edge of the plane. Let us call this edge $\overline{e}$. Calculation of its velocity $\vec{v}$ at this time can be simplified via conservation of energy:

$$\begin{aligned}
\frac{1}{2} M_B |\vec{v}|^2 = KE &= \int_C \vec{F} \cdot \vec{dr} = -\Delta_{PE_g} - W_{F_k} \\
&= M_B g \Delta_H - M_B g \mu_k \left(D - \frac{1}{2}\left(l\cos(\theta) + h\sin(\theta)\right)\right) \\
|\vec{v}| &= \sqrt{2g\left(\Delta_H - \mu_k\left(D - \frac{1}{2}\left(l\cos(\theta) + h\sin(\theta)\right)\right)\right)}
\end{aligned}$$

Thus far, our equations have dealt with constants. In this phase ω (angular velocity) is determined. Accordingly, we call it *glide-rotation*. During glide-rotation, critical variables that govern the movement of B, such as $I_{\overline{e}}$ and $T_{\overline{e}}$, the moment of interia and torque about $\overline{e}$ respectively, are functions of the position of B itself. Let $\overline{e_o}$ denote the axis parallel to $\overline{e}$ that passes through the 3-D center of B. The calculation of the moment of intertia of B about $\overline{e_o}$ is straightforward:

$$\begin{aligned}
I_{\overline{e_o}} = \iiint_V r^2 dm &= \int_{\frac{-l}{2}}^{\frac{l}{2}} \int_{\frac{-h}{2}}^{\frac{h}{2}} \int_0^d \left(x^2 + y^2\right) \delta dz dy dx \\
&= \delta d h l \frac{h^2 + l^2}{12} = M_B \frac{h^2 + l^2}{12}
\end{aligned}$$

The parallel axis theorem yields $I_{\overline{e}} = M_B\left(\frac{h^2+l^2}{12} + r^2\right)$, where r is the distance between $\overline{e}$ and $\overline{e_o}$. Torque is given by $T_{\overline{e}} = M_B g \Delta_x$, where Δ_x is the horizontal displacement of the center of the block past $\overline{e}$. The torque equation then gives us

$$\begin{aligned}
\sum_{\overline{e}} \vec{T} = T_{\overline{e}} &= I_{\overline{e}} \alpha \\
M_B g \Delta_x &= M_B\left(\frac{h^2 + l^2}{12} + r^2\right)\frac{d\omega}{dt} \\
\frac{d\omega}{dt} &= \frac{g\Delta_x}{\frac{h^2+l^2}{12} + r^2}
\end{aligned}$$

Where $\Delta_x = \cos(\beta)\sqrt{r^2 - \frac{h^2}{4}} + \sin(\beta)\frac{h}{2}$ and $\beta = \theta + \int \omega$. A central calculation of α determines the torque due to the contact force between B and P. Since $F_n = M_B g \cos(\beta)$ (so that the block does

not accelerate along the normal direction), this in turn enables us to compute F_k:

$$\begin{aligned}
\sum_{\overline{e_o}} \vec{T} = T_{F_k} + T_{F_n} &= I_{\overline{e_o}}\alpha \\
\frac{F_k h}{2} \pm M_B g \cos(\beta)\sqrt{r^2 - \frac{h^2}{4}} &= M_B \frac{h^2+l^2}{12} \frac{g\Delta_x}{\frac{h^2+l^2}{12} + r^2} \\
F_k &= \frac{2M_B g}{h} \cdot \left(\pm\cos(\beta)\sqrt{r^2 - \frac{h^2}{4}} + \frac{h^2+l^2}{12} \frac{\Delta_x}{\frac{h^2+l^2}{12} + r^2} \right)
\end{aligned}$$

This implicit set of differential equations is much too difficult to resolve by elementary methods, and thus requires a computational model. Furthermore, we assume that contact between B and any slots is by nature a rigid-body collision in which the slots are fixed and infinitely massive. Moreover, we assume a constant elasticity $E \in [0,1]$ for all of the collisions. Let $\vec{r}$, $\vec{p_o}$, $\vec{p}$, ω_i, and $\vec{v_i}$ denote the vector pointing from the center of B to the parcel of B in the collision, a unit vector in the direction of the impulse delivered, the impulse delivered, the initial angular rotation and velocity of B. By our hypotheses, $\left(\frac{1}{2}M_B|\vec{v_i}|^2 + \frac{1}{2}I_o\omega_i^2\right) \cdot E = \frac{1}{2}M_B|\vec{v_f}|^2 + \frac{1}{2}I_o\omega_f^2$. Now,

$$\begin{aligned}
d_\omega &= \text{signedmagnitude}\left(\frac{\vec{r} \times \vec{p_o}}{I_o}\right) \\
\omega_f &= \omega_i + |\vec{p}|d\omega \\
\left(\frac{1}{2}M_B|\vec{v_i}|^2 + \frac{1}{2}I_o\omega_i^2\right) \cdot E &= \frac{1}{2}M_B\left(\left(v_{i_x} + \frac{|\vec{p}|}{M_B}p_{o_x}\right)^2 + \left(v_{i_y} + \frac{|\vec{p}|}{M_B}p_{o_y}\right)^2\right) + \frac{1}{2}I_o\left(\omega_i + |\vec{p}|d_\omega\right)^2 \\
0 &= |\vec{p}|\left(\vec{v_i} \cdot \vec{p_o}\right) + \frac{1}{2}\frac{|\vec{p}|^2}{M_B} + I_o\omega_i d_\omega|\vec{p}| + \frac{1}{2}I_o d_\omega^2|\vec{p}|^2 + (1-E)\cdot\left(\frac{1}{2}M_B|\vec{v_i}|^2 + \frac{1}{2}I_o\omega_i^2\right) \\
\Longrightarrow |\vec{p}| &= \frac{-(\vec{v_i} \cdot \vec{p_o} + I_o\omega_i d_\omega) + \sqrt{(\vec{v_i} \cdot \vec{p_o} + I_o\omega_i d_\omega)^2 - 2\left(\frac{1}{M_B} + I_o d_\omega^2\right)(1-E)E_i}}{\frac{1}{M_B} + I_o d_\omega^2}
\end{aligned}$$

where E_i is the initial energy $\frac{1}{2}M_B|v_i|^2 + \frac{1}{2}I_o\omega_i^2$ and $\vec{v_i} \cdot \vec{p_o} = v_{i_x}p_{o_x} + v_{i_y}p_{o_y}$. We choose $+\sqrt{\cdots}$ because as $E \to 1^-$, the expression with $-\sqrt{\cdots}$ tends to 0. (Recall, for instance, that $\vec{v_i} \cdot \vec{p_o}$ is always negative.) Of course, the fixed elasticity opens the possibility that $|\vec{p}|$ is computed to be imaginary. For such instances, we adopt the convention of perfect elasticity, which is uniquely determined by computing $|\vec{p}|$ with $E = 1$.

The crux of my project is to create and refine this model to the degree that I can predict the ω that will result when B is released from the top of P. Hopefully, there will be a high enough degree of sensitivity to the dimensions of B that I will be able to sort *good* and *bad* pieces based on the variation in ω.

Computational Modeling In order to separate the model itself from the input and graphics implementation, the source has been partitioned into three files:

1. `PhysModel.cpp` - The body that contains most of the physics equations and graphics routines to render the set up.

2. `Parse.cpp` - The file which takes the command line input, defaulting unassigned variables to the previous run via an intermediary storage file.

3. `Polygon.cpp` - The source that defines general graphics functions, the structs `Polygon` and `Vertex`, and several polygonal intersection routines.

The fundamental hypothesis of this project is the assumption that the complicated motion of the block can be modeled as a discrete set of equations repeatedly propagated through a small time step. The goal, then, is to apply such modeling techniques to a system that is hopefully sufficiently complicated as to exhibit a high degree of sensitivity to independent variables such as height and length.

Implementation begins with calls to several initialization routines. The first call interprets the command line input. From the command line, independent variables can be assigned by appending the word "*VariableName=Value" to the end of the command line call. Graphics and a number of debugging flags can be assigned via the word "-flags". (Graphics, for example, is the flag "g", which then propagates as `GFX=1`.) Then the `Sine`, `Cosine`, `Tangent`, and `Sqrt` look-up tables are calculated. By precomputing the values of sine, cosine, tangent, and square root at millions of points, we can effectively negate the cost-expensiveness of computationally expense Taylor series calculations without intolerable loss of precision. Indeed, a brief scan of direct comparison indicates accuracy to roughly 6 correct decimal places.

Implementation continues with a call to `display`, which serves to manage thousands of repeated calls to `Model`, in which the independent variables of length, height, and the position of the rightmost slot-block are minimally altered. The values returned are tabulated in the file specified by the ofstream `DAT`. The variable `ANOMALOUS` is a global that keeps track of the validity of the computed outcome: the block being either or rejected by the slot. The potential loss of validity is a function of the theoretical assumptions that we have made which are not necessarily valid. Some potential errors and mitigating devices I have used to address them include

- Error induced by Eulerian time discretization. Because of the complex nature of the system, it is difficult to determine precisely how this affects accuracy. The runs I have conducted on a typical PC use what appears to be a small time step: between one and three hundredths of a second.

- Perfectly rigid collision. As we shall see, there is a small tolerance built into impulse function which governs this collision.

- Look-up table error; as mentioned, calculated sufficiently many times, this error can be reduced to on the order of 1 part in 1,000,000.

Implementation continues to the model itself. A few qualitative comparisons are conducted. Static friction must meet or exceed kinetic friction, but must not be so great as to prevent any motion at all. Moreover, the assumption that block-plane contact remains face-to-face throughout initiation asserts a simple trigonometric relation. Finally, if the block is simply too large to fit through the slot, the rejection is categorically recorded as a rejection with zero anomaly.

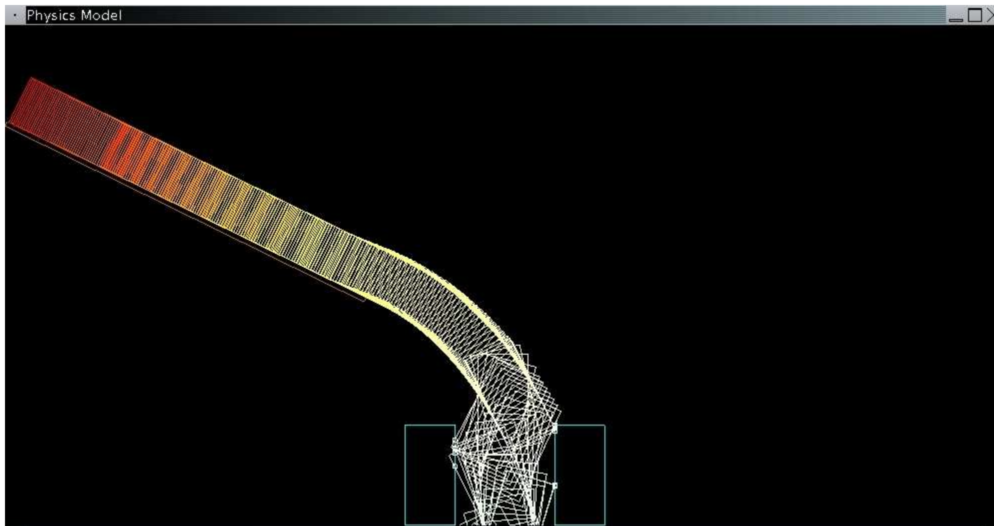
After the said preconditioning, the model simulates the triphasic experiment in three consecutive loops. The variable `IPS` is employed in conjunction with `SDL_DELAY` to add sufficient artificial delay so as to obtain the desired number of *i*terations *p*er *s*econd. The delay function itself takes only integer arguments; thus, all values of `IPS` in excess of 1000 are equivalent. Otherwise, the loops are

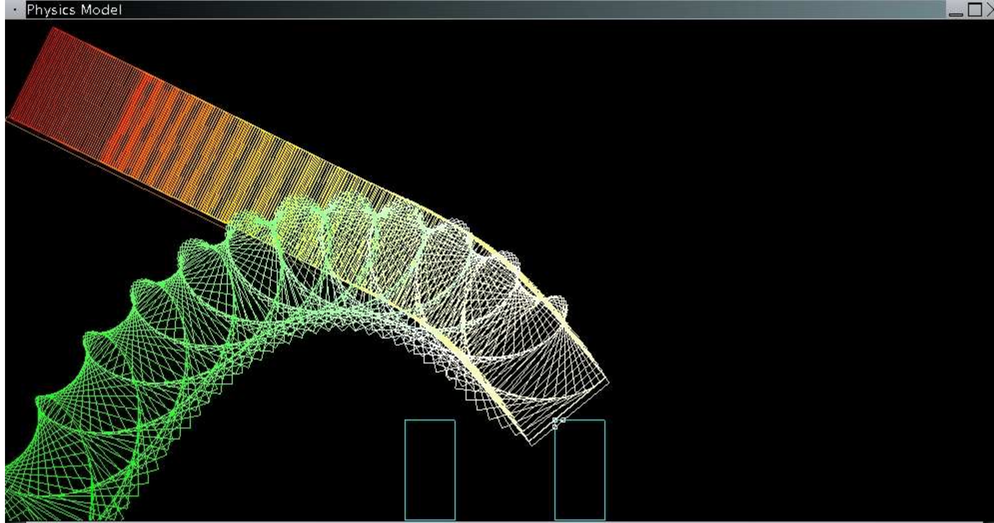
governed by a discrete version of the physics described in detail above. The `Collision` function returns whether or not the block overlaps with any of the slot polygons, assigning to every edge of each shape the number of other edges it intersects with. Upon return to the base loop making the call, the program calls `impulse` to handle the relevant impulse transfer. `impulse` runs time forwards and backwards with progressively smaller time steps until it has precisely determined when the said collision occurred. Again, we have employed a calculational technique to lower the number of computations necessary while maintaining high precision. The function then computes under the standard two-dimensional Newtonian dichotomy:

1. Either a corner of the block has protruded over an edge of a slot-block, or

2. A corner of a slot-block has protruded over an edge of the block.

That is, it assumes that we do not have the scenario where two corners mutually protrude over one another. The above physics equations are then applied. The direction of the frictional component is determined via a sequence of vector calculations. The normal component is then scaled according to `COLFRICOF`, the fixed coefficient of friction for these impacts, and impulse is delivered. The potential error is of the block rotating partially into the slot-block due to its rigidity. A small fraction of the collisions result in this anomalous motion; hence, the resultant motion is checked by another loop governed by the `Collision` function. If at least one iteration is spent with such positioning, the current iteration is flagged with an anomaly value of 2. If too many such iterations occur, a value of 4 is used instead. Finally, the acceptance / rejection of the block is determined by checking its center with a set of horizontal and vertical thresholds. `Model` returns 0 for a rejection and a 1 for acceptance.





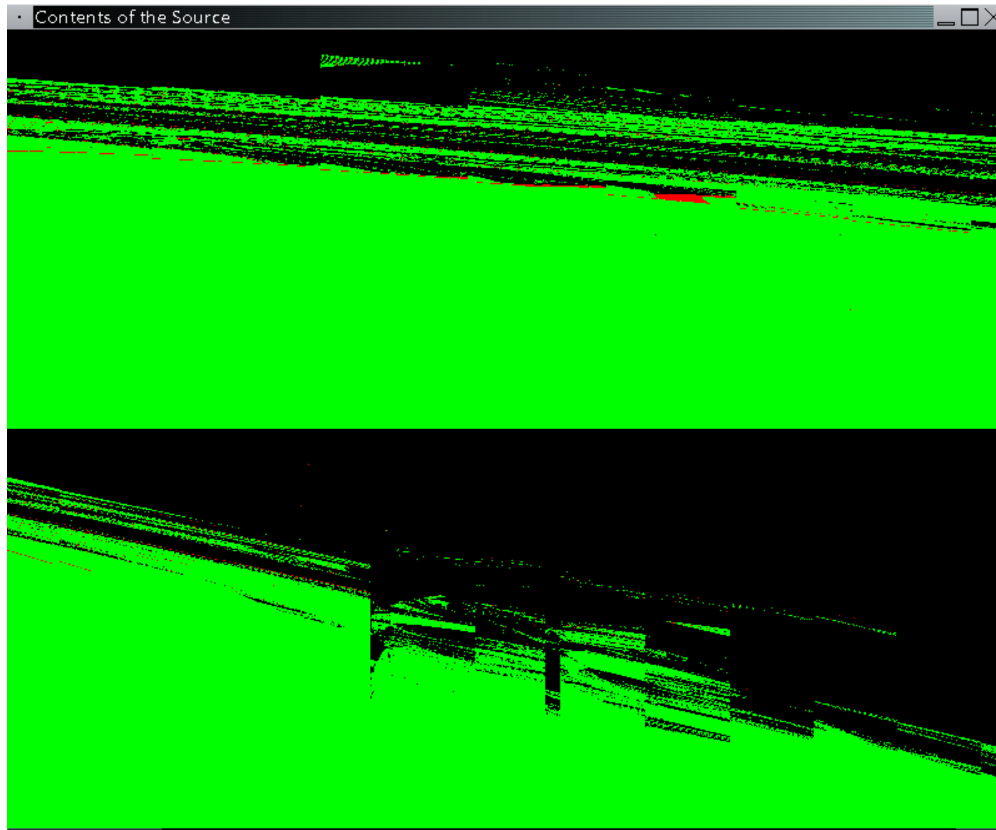
3 Conclusion

We convert the records of the trials into images for a cursory and qualitative analysis. Green pixels are plotted for trials resulting in acceptance, and black is plotted for rejections. Anomalous trials, with various computational errors previously treated, are those appearing in red.

Here we refer to the parameters of the subsequent image. For this particular image, the horizontal scale is one pixel equal to one millimeter in block variance, with the base value being plotted in the leftmost column and increased with each pixel to right. Vertically, one pixel is a half of a centimeter in slotwidth flux, with the initial value at the top and increases in slotwidth with each pixel down. More precisely, the leftmost block was held fixed - the changes here were in the position of the right half of the slot. The base block is one meter long and half a meter in height. The image itself actually consists of two sensitivity tests - the top half corresponds to flux in length, with the bottom depicting sensitivity to increases in height. We base our conclusion primarily on this image, which by itself raises some interesting points.

First, the complicated nature of the image suggests the presence of a high degree of sensitivity not unlike that which was hoped for at the outset. Although nearly all blocks become accepted for suitably large slots, only a thin band are accepted prior to the threshold; hence, with blocks suitably tailored, this exploratory experiment seems to indicate that a single ramp-slot can be manipulated to exact sensitivities corresponding to a few pixels - or roughly one part in one hundred. Moreover, there is nothing to confine the application of this methodology to simply one slot. It is clear that the use multiple ramps and slots would enable us to select the intersection of superimposed images.

Second, there are a number of columns which across which there is a drastic change in acceptance patterns. These are the result of bifurcation, or qualitative change in computed outcome. In this experiment, such bifurcation occurs when the collisions rendered between the blocks change from block overlapping slot and vice versa. Our treatment of this topic has been rather crude. We have held the first block fixed throughout computations yet this block plays a large role in separatrix demarcation.



4 Appendix A - Code

For completeness, the three source files `PhysModel.cpp`, `Parse.cpp`, and `Polygon.cpp` are reproduced here. The user need only compile and run `PhysModel`, the others are included. Various flags and assignments are available as defined in `Parse`, a brief explanation is provided upon executing the program with the help flag, '-h'. The other file, `ImageBuilder.cpp`, accepts the contents of a data file, "Default.txt", as created and formatted by the `PhysModel`, displaying an image like the one previously discussed in the conclusion.

PhysModel.cpp

```
#include <stdlib.h>
#include <SDL/SDL.h>
#include <GL/glut.h>
#include <iostream>
#include <math.h>
#include <fstream>
#include "Polygon.cpp"
#include "Parse.cpp"
```

```
#define TOLERANCE 0.5 //the number of pixels of movement.
#define VECSCALE 25.0
#define COLSTOP 10
#define DISPLAYFRAC 1 //1 out of DISPLAYFRAC frames.
#define FOO 0
#define ELASTICITY 1.00
#define SLOTFLUX 320
#define SLOTINITIAL 50
#define DIMFLUX 800
#define MAXCOLLISIONS 15
#define SPEEDBYPASS 10
#define SLOTRESOLUTION .5
#define DIMRESOLUTION .1
#define PASSDEBUG 1000000
//SLOTRESOLUTION and DIMRESOLUTION are the numbers of centimeters of adjustment
//SPEEDBYPASS will be the number of iterations gone without Collision(1).
//ELASTICITY is the fraction of energy preserved during the collision.

double ForceProj;
double ProjX;
double NULLMOMENT = MASS * (HEIGHT*HEIGHT + LENGTH*LENGTH) / 12.0;
double SIGN;
int ININTERVALHIST=0;
int ININTERVAL=0;
int COLLIDED=0;//Counts the number of collisions during a certain pass.
double FALLZERO;
int EDGE;
int SLOTHIT;
int ANOMALOUS=0;
int SPECIALDELIVERY=0;
int IMPULSEDEPTH=0;
int SLOTWIDTH=0;
int WHERE=0;

ofstream DAT("Default.txt");

int QUESTIONININTERVAL()
{
ININTERVALHIST=ININTERVAL;
if (Box.Center.X >= Slots[0].Vertices[1].X-75
&& Box.Center.X <= Slots[1].Vertices[2].X+25)
ININTERVAL=1;
else
ININTERVAL=0;
return(ININTERVAL);
```

```
}

void updateDimensions(int SW,int DC,int dim)
{//slot width set to SW+SLOTINITIAL
Slots[1].Vertices[0].X = 500+SW*SLOTRESOLUTION+SLOTINITIAL;//starts at x=500+SW+SLOTINITIAL
Slots[1].Vertices[1].X = 500+SW*SLOTRESOLUTION+SLOTINITIAL;
Slots[1].Vertices[2].X = 550+SW*SLOTRESOLUTION+SLOTINITIAL;
Slots[1].Vertices[3].X = 550+SW*SLOTRESOLUTION+SLOTINITIAL;
Slots[1].Vertices[0].Y = 0;//the other slot ends at 500.
Slots[1].Vertices[1].Y = 100;
Slots[1].Vertices[2].Y = 100;
Slots[1].Vertices[3].Y = 0;
if (dim==1)
LENGTH=1.0+0.01*DC*DIMRESOLUTION;
else
LENGTH=1.0;
if (dim==2)
HEIGHT=0.5+0.01*DC*DIMRESOLUTION;
else
HEIGHT=0.5;
}

void DrawPic()
{
//scale 1 meter = 100 pixels
int i;
if (GFX == 1)
{
glClearColor(0.0,0.0,0.0,1.0);
glClear( GL_COLOR_BUFFER_BIT );
glColor3f(0.8, 0.4, 0.0);
Line(0,400,-5*Sine(theta),400+5*Cosine(theta));
Line(0,400,100*DISTANCE*Cosine(theta), 400+100*DISTANCE*Sine(theta));
Line(100*DISTANCE*Cosine(theta),400+100*DISTANCE*Sine(theta),
100*DISTANCE*Cosine(theta)-5*Sine(theta),400+100*DISTANCE*Sine(theta)+5*Cosine(theta));
Line(100*DISTANCE*Cosine(theta)-5*Sine(theta),400+100*DISTANCE*Sine(theta)+5*Cosine(theta),
-5*Sine(theta),400+5*Cosine(theta));
i = 0;
while (i < NUMBLOCKS)
{
DrawPolygon(Slots[i]);
i++;
}
glFlush();
glFlush();
```

```
}
}

void PlotData()
{
double XCOORD;
ITMIN = 175000;
ITMAX = 200000;
if (GFX == 1)
{
//Variables: VELOCITY(RED), FRICTION(YELLOW), TORQUE(GREEN),
//MOMENT(CYAN), ANGLE(BLUE), ANGULAR(PURPLE).
if (ITERATION < ITMIN —— ITERATION > ITMAX) return;
XCOORD = (ITERATION-ITMIN)*SWIDTH/(ITMAX-ITMIN);
glColor3f(1.0,0.0,0.0);
Point(XCOORD,VELOCITY/MAXSPEED*SHEIGHT);
glColor3f(1.0,1.0,0.0);
Point(XCOORD,(FRICTION+10)/MAXFRICTION*SHEIGHT);
glColor3f(0.0,1.0,0.0);
Point(XCOORD,TORQUE/MAXTORQUE*SHEIGHT);
glColor3f(0.0,1.0,1.0);
Point(XCOORD,MOMENT/MAXMOMENT*SHEIGHT);
glColor3f(0.0,0.0,1.0);
Point(XCOORD,mod(ANGLE,PI)/MAXANGLE*SHEIGHT);
glColor3f(1.0,0.0,1.0);
Point(XCOORD,ANGULAR/MAXANGULAR*SHEIGHT);
}
}

int Collision(int draw)
{
int i = 0, j = 0;
for (i = 0; i < NUMBLOCKS; i++)
for (j = 0; j < Slots[i].Size; j++) Slots[j].EdgeIntersect[i] = 0;
for (i = 0; i < Box.Size; i++) Box.EdgeIntersect[i] = 0;
i = 0; j = 0;
while (i < NUMBLOCKS)
{
if (PolyIntersect(Box,Slots[i],draw,1.0,1.0,1.0))
{
j = 1;
}
i++;
}
if (j != 0)//there must be some overlap. EDGE is left as -1 if the collision is V-V.
```

```
//SLOTHIT is assigned to -10 if an edge of the block is intersected twice.
//EDGE is assigned to the edge of slot block SLOTHIT otherwise.
{
EDGE=-1;
SLOTHIT=-1;
for (i = 0; i < NUMBLOCKS; i++)
for (j = 0; j < Slots[i].Size; j++)
if (Slots[j].EdgeIntersect[i]>=2) { SLOTHIT=j; EDGE=i; }
for (i = 0; i < Box.Size; i++)
if (Box.EdgeIntersect[i]>=2) { SLOTHIT=-10; EDGE=i; }
j = 1;
}
return(j);
}

int impulse()
{
int LOOPCOUNT;
int i = 0, j = 0, siz, badness = 3;
double x1, x2, y1, y2;
double I1, I2, J1, J2;
double nx = 0, ny = 0, mag = 0;
double px = 0, py = 0;
double cx = 0, cy = 0, dx = 0, dy = 0;
double dot = 0;
double fric = 1.0, fricx = 0, fricy = 0;
double p0x = 0, p0y = 0, domega = 0;
double a=0, b=0, c=0;
COLLIDED++;
IMPULSEDEPTH++;
while(Collision(0))//Going backwards! GAMMA4=10GAMMA3=100GAMMA2=1000GAMMA
{
FALL -= VY * GAMMA4;
Box.Center.Y -= 100.0 * VY * GAMMA4;
Box.Center.X -= 100.0 * VX * GAMMA4;
VELOCITY = Sqrt(VX*VX+VY*VY);
VY += GRAVITY*GAMMA4;
ANGLE -= ANGULAR * GAMMA4;
I1 = 50.0 * LENGTH * Cosine(ANGLE);
I2 = -50.0 * HEIGHT * Sine(ANGLE);
J1 = 50.0 * LENGTH * Sine(ANGLE);
J2 = 50.0 * HEIGHT * Cosine(ANGLE);
Box.Vertices[0].X = Box.Center.X + I1 + I2;
Box.Vertices[0].Y = Box.Center.Y + J1 + J2;
Box.Vertices[1].X = Box.Center.X - I1 + I2;
```

```
Box.Vertices[1].Y = Box.Center.Y - J1 + J2;
Box.Vertices[2].X = Box.Center.X - I1 - I2;
Box.Vertices[2].Y = Box.Center.Y - J1 - J2;
Box.Vertices[3].X = Box.Center.X + I1 - I2;
Box.Vertices[3].Y = Box.Center.Y + J1 - J2;
if (EXPERIMENT && DEBUG —— WHERE == PASSDEBUG)
{ SDL_Delay(200); ITERATION+=10; DrawBox();}
}
LOOPCOUNT = 0;
while (!Collision(0))
{
ANGLE += ANGULAR * GAMMA3;
VY -= GRAVITY*GAMMA3;
VELOCITY = sqrt(VX*VX+VY*VY);
Box.Center.X += 100.0 * VX * GAMMA3;
Box.Center.Y += 100.0 * VY * GAMMA3;
FALL += VY*GAMMA3;
I1 = 50.0 * LENGTH * Cosine(ANGLE);
I2 = -50.0 * HEIGHT * Sine(ANGLE);
J1 = 50.0 * LENGTH * Sine(ANGLE);
J2 = 50.0 * HEIGHT * Cosine(ANGLE);
Box.Vertices[0].X = Box.Center.X + I1 + I2;
Box.Vertices[0].Y = Box.Center.Y + J1 + J2;
Box.Vertices[1].X = Box.Center.X - I1 + I2;
Box.Vertices[1].Y = Box.Center.Y - J1 + J2;
Box.Vertices[2].X = Box.Center.X - I1 - I2;
Box.Vertices[2].Y = Box.Center.Y - J1 - J2;
Box.Vertices[3].X = Box.Center.X + I1 - I2;
Box.Vertices[3].Y = Box.Center.Y + J1 - J2;
if (EXPERIMENT && DEBUG —— WHERE == PASSDEBUG)
{ SDL_Delay(200); ITERATION+=10; DrawBox();}
LOOPCOUNT++;
if (LOOPCOUNT > 12) return(0);//Possible only due to
//algorithmically induced error. 12 and not 10 is used
//for tolerance purposes.
}
while(Collision(0))
{
FALL -= VY * GAMMA2;
Box.Center.Y -= 100.0 * VY * GAMMA2;
Box.Center.X -= 100.0 * VX * GAMMA2;
VELOCITY = Sqrt(VX*VX+VY*VY);
VY += GRAVITY*GAMMA2;
ANGLE -= ANGULAR * GAMMA2;
I1 = 50.0 * LENGTH * Cosine(ANGLE);
```

```
I2 = -50.0 * HEIGHT * Sine(ANGLE);
J1 = 50.0 * LENGTH * Sine(ANGLE);
J2 = 50.0 * HEIGHT * Cosine(ANGLE);
Box.Vertices[0].X = Box.Center.X + I1 + I2;
Box.Vertices[0].Y = Box.Center.Y + J1 + J2;
Box.Vertices[1].X = Box.Center.X - I1 + I2;
Box.Vertices[1].Y = Box.Center.Y - J1 + J2;
Box.Vertices[2].X = Box.Center.X - I1 - I2;
Box.Vertices[2].Y = Box.Center.Y - J1 - J2;
Box.Vertices[3].X = Box.Center.X + I1 - I2;
Box.Vertices[3].Y = Box.Center.Y + J1 - J2;
if (EXPERIMENT && DEBUG —— WHERE == PASSDEBUG)
{ SDL_Delay(200); ITERATION+=10; DrawBox();}
}
LOOPCOUNT = 0;
while (!Collision(0))
{
ANGLE += ANGULAR * GAMMA;
VY -= GRAVITY*GAMMA;
VELOCITY = sqrt(VX*VX+VY*VY);
Box.Center.X += 100.0 * VX * GAMMA;
Box.Center.Y += 100.0 * VY * GAMMA;
FALL += VY*GAMMA;
I1 = 50.0 * LENGTH * Cosine(ANGLE);
I2 = -50.0 * HEIGHT * Sine(ANGLE);
J1 = 50.0 * LENGTH * Sine(ANGLE);
J2 = 50.0 * HEIGHT * Cosine(ANGLE);
Box.Vertices[0].X = Box.Center.X + I1 + I2;
Box.Vertices[0].Y = Box.Center.Y + J1 + J2;
Box.Vertices[1].X = Box.Center.X - I1 + I2;
Box.Vertices[1].Y = Box.Center.Y - J1 + J2;
Box.Vertices[2].X = Box.Center.X - I1 - I2;
Box.Vertices[2].Y = Box.Center.Y - J1 - J2;
Box.Vertices[3].X = Box.Center.X + I1 - I2;
Box.Vertices[3].Y = Box.Center.Y + J1 - J2;
if (EXPERIMENT && DEBUG —— WHERE == PASSDEBUG)
{ SDL_Delay(200); ITERATION+=10; DrawBox();}
LOOPCOUNT++;
if (LOOPCOUNT > 12) return(0);
}
NCOL++;
if (SLOTHIT>=0)
{
x1 = Slots[SLOTHIT].Vertices[EDGE].X;
x2 = Slots[SLOTHIT].Vertices[(EDGE+1)y1 = Slots[SLOTHIT].Vertices[EDGE].Y;
```

```
y2 = Slots[SLOTHIT].Vertices[(EDGE+1)if (GFX == 1)
{
glColor3f(1.0,1.0,1.0);
Line(x1,y1,x2,y2);
}
}
else if (EDGE == -1)
{
ny = -Sine((ANGLE-PI)/2.0); nx = Cosine((ANGLE-PI)/2.0);
x1 = XINTERSECT+100.0*nx;
x2 = XINTERSECT-100.0*nx;
y1 = YINTERSECT+100.0*ny;
y2 = YINTERSECT-100.0*ny;
if (GFX == 1)
{
glColor3f(0.0,0.0,0.8);
Line(x1,y1,x2,y2);
}
}
else if (SLOTHIT == -10)
{
x1 = Box.Vertices[EDGE].X;
x2 = Box.Vertices[(EDGE+1)y1 = Box.Vertices[EDGE].Y;
y2 = Box.Vertices[(EDGE+1)}
if (GFX == 1)
{
SmallBox(XINTERSECT,YINTERSECT,1.0,1.0,1.0);
SmallBox(x1,y1,1.0,0.0,0.0);
SmallBox(x2,y2,0.0,1.0,0.0);
}
nx = y2 - y1;
ny = x1 - x2;
mag = sqrt(nx*nx + ny*ny);
nx /= mag;
ny /= mag;//Unit vector normal to surface.
if (nx * (Box.Center.X - XINTERSECT) + ny * (Box.Center.Y - YINTERSECT) < 0)
{
//Makes sure normal direction pointing towards Box.Center.
nx *= -1;
ny *= -1;
}
dx = (XINTERSECT - Box.Center.X)/100;//r = <dx, dy>
dy = (YINTERSECT - Box.Center.Y)/100;
cx = VX - ANGULAR * dy;
cy = VY + ANGULAR * dx;//velocity of the piece of B in contact, in meters per sec.
```

```
fric = nx*cy-ny*cx;//n cross vparcel
fric = fabs(fric)/fric;//gives the sign, indicating direction of friction.
fricx = fric * COLFRICOF * ny;
fricy = fric * COLFRICOF * -nx;
p0x = nx + fricx;
p0y = ny + fricy;
mag = Sqrt(p0x*p0x + p0y*p0y);
p0x /= mag;//unit vector in direction of impulse delivered.
p0y /= mag;
domega = (dx * p0y - dy * p0x) / MOMENT; //<dx,dy> cross <p0x,p0y> / Io
dot = VX * p0x + VY * p0y;//recall the equation a|p|^2 + b|p| + c = 0;
a = 0.5 * (1/MASS + MOMENT*domega*domega);
b = dot + MOMENT*ANGULAR*domega;
c = (1.0 - ELASTICITY) * (0.5) * (MASS*(VX*VX+VY*VY)+MOMENT*ANGULAR*ANGULAR);
if (b*b-4*a*c > 0)
{
mag = (-b + Sqrt(b*b-4.0*a*c))/(2.0 * a);
if (DEBUG)
{
cout<<"a="<<a<<"; b="<<b<<"; c="<<c<<endl;
cout<<"Using "<<100.0*ELASTICITY
<<"}
}
else
{
mag = -b/a;
cout<<"Unable to deliver fixed elasticity, using perfect conservation of energy."<<endl;
SPECIALDELIVERY=1;
}
px = mag * p0x;
py = mag * p0y;//momentum impulse delivered by slot.
VX += px/MASS;
VY += py/MASS;
ANGULAR += mag*domega;
Box.Center.X += nx * TOLERANCE;
Box.Center.Y += ny * TOLERANCE;
if (FOO == 666 && GFX == 1)
{
glColor3f(1.0,1.0,0.0);
Line(XINTERSECT,YINTERSECT,XINTERSECT+100*nx,YINTERSECT+100*ny);
glColor3f(1.0,0.0,1.0);
Line(Box.Center.X,Box.Center.Y,Box.Center.X+dx*100,Box.Center.Y+dy*100);
glColor3f(0.0,1.0,1.0);
Line(XINTERSECT,YINTERSECT,XINTERSECT+cx*100,YINTERSECT+cy*100);
glColor3f(1.0,0.0,0.0);
```

```
Line(XINTERSECT,YINTERSECT,XINTERSECT+100*fricx,YINTERSECT+100*fricy);
glColor3f(0.0,0.0,1.0);
Line(XINTERSECT,YINTERSECT,XINTERSECT+100*p0x,YINTERSECT+100*p0y);
glColor3f(1.0,1.0,1.0);
Line(XINTERSECT+2,YINTERSECT,XINTERSECT+2+100*px,YINTERSECT+100*py);
cout<<”Impulse Physics:”<<endl
<<”(Dimensionless.) <nx,ny> = <”<<nx<<”,”<<ny<<”>; drawn in yellow.”<<endl
<<”(In meters.) Vec{r} = <dx,dy> = <”<<dx<<”,”<<dy<<”>; drawn in purple.”<<endl
<<”(In meters.) Vec{c}= <cx,cy> = <”<<cx<<”,”<<cy<<”>; drawn in cyan.”<<endl
<<”(Dimensionless.) <fricx,fricy> = <”<<fricx<<”,”<<fricy<<”; drawn in red.”<<endl
<<”(Dimensionless.) <p0x,p0y> = <”<<p0x<<”,”<<p0y<<”>; drawn in blue.”<<endl
<<”(In inverse meter-kilograms.) domega = ”<<domega<<endl
<<”(In meter kilograms per second.) —Vec{p}— = mag = ”<<mag<<endl
<<”(In inverse seconds.) omega_f = ANGULAR = ”<<ANGULAR<<endl
<<”(In kilograms times meters over time.) <px,py> = <”<<px<<”,”
<<py<<”>; drawn in white.”<<endl;
}
DrawBox();
while (Collision(0) && FALL > 0)// falling in air
{
ANOMALOUS=2;
badness–;//Badness starts as a tolerance of anomaly.
SDL_Delay((int) (1000.0/IPS));
ANGLE += ANGULAR * GAMMA2;
VY -= GRAVITY*GAMMA2;
VELOCITY = sqrt(VX*VX+VY*VY);
Box.Center.X += 100.0 * VX * GAMMA2;
Box.Center.Y += 100.0 * VY * GAMMA2;
DIST_TRAVELLED -= VY*GAMMA2;
if (!QUESTIONININTERVAL() && ININTERVALHIST)
{ return(666); }//Rejected!
if (EXPERIMENT && (ITERATION if (GRAPH) PlotData();
if (badness < 0)
{
cout<<”Terribly anomalous case. ”<<endl;
ANOMALOUS=4;
return(666);
}
}
return(0);
}

int Model()
{
double Hold = 0;
```

```
SPECIALDELIVERY=0;
ANOMALOUS=0;
ITERATION=0;
DIST_TRAVELLED=0.0;
ININTERVAL=0;
COLLIDED=0;
ANGULAR=0.0;
VELOCITY=0.0;
VX=0.0;
VY=0.0;
SIGN = -1;
MAXSPEED = 10.0;
MAXFRICTION = GRAVITY*MuK*MASS*5;
MAXTORQUE = (LENGTH+WIDTH)*GRAVITY*MASS;
MAXMOMENT = MASS*(LENGTH*LENGTH+HEIGHT*HEIGHT)/3.0;
MAXANGLE = PI;
MAXANGULAR = 3.0;
theta = -THETA / 180.0 * PI;
ANGLE = theta;
if (MuS < MuK)
{ cout<<"Invalid Friction: MuS < MuK."<<endl; exit(0); }
if (90.0 - THETA < 180.0 / PI * atan(HEIGHT / LENGTH))
{ cout<<"Box tumbles, decrease THETA or HEIGHT / LENGTH"<<endl; return(10); }
if (MuS >= fabs(Tangent(theta)))
{
cout<<"No motion results. Increase angle."<<endl;
cout<<"theta="<<theta<<"; Tangent("<<theta<<")="<<Tangent(theta)<<"; tan("<<theta<<")="<<Tan
exit(0); }
if (ITMIN >= ITMAX)
{ cout<<"ITMIN = "<<ITMIN<<"; ITMAX = "<<ITMAX<<endl; return(-1); }
if (SLOTWIDTH + SLOTINITIAL < 100.0 * min(HEIGHT,LENGTH))
{ cout<<"Auto-reject; Box too big."<<endl; return(0); }
if (EXPERIMENT && GFX == 1) { DrawPic(); }
DISTANCE2 = DISTANCE - LENGTH/2.0 + Tangent(theta)*HEIGHT/2.0;
strcpy(Box.Name, "TheBlock");
Box.Center.X = -5.0*Sine(theta) + (-HEIGHT * Sine(theta) + LENGTH * Cosine(theta)) * 50.0;
Box.Center.Y = 400+5.0 * Cosine(theta) + (HEIGHT * Cosine(theta) + LENGTH * Sine(theta))
* 50.0;
Box.Size = 4;
MOMENT = MASS*(LENGTH*LENGTH+HEIGHT*HEIGHT)/12.0;
DIST_TRAVELLED = 0;
FALL = FALLZERO;
while (DIST_TRAVELLED < DISTANCE2)//Still sliding.
{
ITERATION++;
```

```
SDL_Delay((int) (1000.0/IPS));
FRICTION = MASS * GRAVITY * Cosine(theta) * MuK;
ACCELERATION = -GRAVITY * Sine(theta) - FRICTION / MASS;
VELOCITY += ACCELERATION * DT;
DIST_TRAVELLED += VELOCITY * DT;
Box.Center.X += 100.0 * VELOCITY * DT * Cosine(theta);
Box.Center.Y += 100.0 * VELOCITY * DT * Sine(theta);
if (EXPERIMENT && (ITERATION if (GRAPH) PlotData();
if (STATS) {
cout<<"Iteration #"<<ITERATION
<<". Phase 1. Acceleration="<<ACCELERATION
<<" Velocity="<<VELOCITY
<<" Distance Travelled="<<DIST_TRAVELLED
<<" Box Center (x,y)=("<<Box.Center.X<<","<<Box.Center.Y
<<") Friction="<<FRICTION<<endl; }
}
ANGULAR = 0;
PX = 100*DISTANCE*Cosine(theta)-5*Sine(theta);
PY = 400+100*DISTANCE*Sine(theta)+5*Cosine(theta);
Hold = Sqrt((Box.Center.X - PX)*(Box.Center.X - PX) +
(Box.Center.Y - PY)*(Box.Center.Y - PY)) / 100.0;//H in meters.
VX = VELOCITY*Cosine(ANGLE);
VY = VELOCITY*Sine(ANGLE);
while (DIST_TRAVELLED < DISTANCE)// falling off edge
{
DX = Box.Center.X - PX;
DY = Box.Center.Y - PY;//<DX,DY> is vector from <PX,PY> to <BCX,BCY>.
H = Sqrt(DX*DX + DY*DY) / 100.0;//H in meters after dividing.
ITERATION++;
SDL_Delay((int) (1000.0/IPS));
MOMENT = MASS * ((HEIGHT*HEIGHT + LENGTH*LENGTH) / 12.0 + H*H);
TORQUE = DX/100.0*MASS*GRAVITY;
ANGULAR -= TORQUE / MOMENT * DT;
ANGLE += ANGULAR * DT;
ProjX = DISTANCE - (LENGTH/2.0) - DIST_TRAVELLED;
NORMALFORCE = MASS * GRAVITY * Cosine(ANGLE);
FRICTION = (-2.0/HEIGHT)*((DX/100.0)*MASS*GRAVITY*NULLMOMENT/MOMENT
-NORMALFORCE * (ProjX));
ForceProj = -MASS * GRAVITY * Sine(ANGLE)-FRICTION;
VX += Cosine(ANGLE)*(DT*ForceProj/MASS);
VY += Sine(ANGLE)*(DT*ForceProj/MASS);
Box.Center.X += VX * DT * 100.0;
Box.Center.Y += VY * DT * 100.0;
VELOCITY = Sqrt(VX*VX+VY*VY);
DIST_TRAVELLED += VELOCITY*DT*Cosine(ANGLE-theta);
```

```
if (EXPERIMENT && (ITERATION if (GRAPH) PlotData();
if (STATS —— VELOCITY < 0) {
cout<<”Iteration #”<<ITERATION
<<”. Phase 2. Lever Arm=”<<H
<<” Moment of Inertia=”<<MOMENT
<<” Mass=”<<MASS
<<” Torque=”<<TORQUE
<<” Vx=”<<VX
<<” Vy=”<<VY
<<” Box Center (x,y)=(”<<Box.Center.X<<”,”<<Box.Center.Y
<<”) Angular Velocity=”<<ANGULAR
<<” Angle=”<<ANGLE
<<” Acceleration=”<<ACCELERATION
<<” Velocity=”<<VELOCITY
<<” Friction=”<<FRICTION<<endl; }
}
TORQUE = 0.0;
FRICTION = 0.0;
MOMENT = MASS*(LENGTH*LENGTH+HEIGHT*HEIGHT)/12.0;
ININTERVALHIST = 0;
while (FALL > 0)// falling in air
{ //fall adjusted to be just at bottom of slot.
ITERATION++;
SDL_Delay((int) (1000.0/IPS));
IMPULSEDEPTH=0;
ANGLE += ANGULAR * DT;
VY -= GRAVITY*DT;
VELOCITY = Sqrt(VX*VX+VY*VY);
Box.Center.X += 100.0 * VX * DT;
Box.Center.Y += 100.0 * VY * DT;
FALL += VY * DT;
if (!QUESTIONININTERVAL() && ININTERVALHIST)
{cout<<”Rejected by interval history.”<<endl; return(0);} //rejected.
if (Collision(1)) if(impulse()==666)
{
if (ANOMALOUS != 4)
cout<<”Rejected by impulse bounding.”<<endl;
else
cout<<”Rejected anomalous.”<<endl;
return(0); }//rejected
if (EXPERIMENT && (ITERATION if (GRAPH) PlotData();
if (STATS) {
cout<<”Iteration #”<<ITERATION
<<”. Phase 3. Moment of Inertia=”<<MOMENT
<<” Vx=”<<VX
```

```
<<" Vy="<<VY
<<" Box Center (x,y)=("<<Box.Center.X<<","<<Box.Center.Y
<<") Angular Velocity="<<ANGULAR
<<" Angle="<<ANGLE
<<" Acceleration="<<ACCELERATION
<<" Velocity="<<VELOCITY
<<" Distance Travelled="<<DIST_TRAVELLED<<endl; }
if (COLLIDED > MAXCOLLISIONS)
{ cout<<"Accepted due to bouncing."<<endl;
return(1); //bouncing in slot.
}
}
cout<<"ACCEPTED!"<<endl;
return(1);//accepted.
}

void display(void)
{ /* clear all pixels */
int DIMCHANGE, value;
FALLZERO = FALL;
cout<<"Running primary simulation loop."<<endl;
DAT<<"ELASTICITY "<<ELASTICITY<<endl;
DAT<<"SLOTFLUX "<<SLOTFLUX<<endl;
DAT<<"SLOTRESOLUTION "<<SLOTRESOLUTION<<endl;
DAT<<"SLOTINITIAL "<<SLOTINITIAL<<endl;
DAT<<"DIMFLUX "<<DIMFLUX<<endl;
DAT<<"DIMRESOLUTION "<<DIMRESOLUTION<<endl;
for (SLOTWIDTH=0; SLOTWIDTH<SLOTFLUX; SLOTWIDTH++)
{
for (DIMCHANGE=0; DIMCHANGE<DIMFLUX; DIMCHANGE++)
{
updateDimensions(SLOTWIDTH,DIMCHANGE,1);
if (GFX == 1)
{
glClear(GL_COLOR_BUFFER_BIT);
glFlush();
glFlush();
}
cout<<"Pass "<<WHERE<<" of "<<2*SLOTFLUX*DIMFLUX
<<" ("<<100.0*WHERE/(2.0*SLOTFLUX*DIMFLUX)<<"value=Model();
if (GFX == 1)
{
glFlush(); //Force writing all pending
//OpenGL actions to the screen.
glFlush();
```

```
}
WHERE++;
DAT<<(value+ANOMALOUS+10*SPECIALDELIVERY);//0 mod 2 rejected; 1 mod 2 accepted.
//ANOMALOUS = 0 if good, 2 if roughly good, 4 if BAD.
if (DIMCHANGE<DIMFLUX-1) DAT<<" ";
}
DAT<<endl;
}
DAT<<endl;
for (SLOTWIDTH=0; SLOTWIDTH<SLOTFLUX; SLOTWIDTH++)
{
for (DIMCHANGE=0; DIMCHANGE<DIMFLUX; DIMCHANGE++)
{
updateDimensions(SLOTWIDTH,DIMCHANGE,2);
if (GFX == 1)
{
glClear(GL_COLOR_BUFFER_BIT);
glFlush();
glFlush();
}
cout<<"Pass "<<WHERE<<" of "<<2*SLOTFLUX*DIMFLUX
<<" ("<<100.0*WHERE/(2.0*SLOTFLUX*DIMFLUX)<<"value=Model();
if (GFX == 1)
{
glFlush();
glFlush();
}
WHERE++;
DAT<<(value+ANOMALOUS+10*SPECIALDELIVERY);
if (DIMCHANGE<DIMFLUX-1) DAT<<" ";
}
DAT<<endl;
}
exit(0);
}

int InitializeGFX(int argc, char ** argv)
{
if (GFX)
{
glutInit(&argc, argv);
glutInitDisplayMode( GLUT_SINGLE — GLUT_RGB );
glutInitWindowSize( (int) SWIDTH, (int) SHEIGHT );
glutInitWindowPosition( SXCOORD, SYCOORD );
glutCreateWindow ("Physics Model");
```

```
glClearColor(0.0, 0.0, 0.0, 0.0);
glMatrixMode( GL_PROJECTION );
glLoadIdentity();
glOrtho(0,SWIDTH,0,SHEIGHT, -1.0, 1.0);
glutDisplayFunc( display );
return(0);
}
else
display();
}

int main(int argc, char ** argv)
{
if (parse(argc, argv))
return(0);
SQRTINIT();
TRIGINIT();
InitializeGFX(argc, argv);
glutMainLoop();
cout<<"End of Simulation."<<endl;
getchar();
return 0;
}
```

Parse.cpp

```
ifstream infy("Blocks.txt");
ifstream inf;
ofstream outf;

char * File;
char LINE[100];

void clr() { int i; for (i = 0; i < 100; i++) LINE[i] = 0; }

double mod(double x,double y) { //returns x mod y, -1 if y = 0;
double a = x, b = y;
if (b == 0) return (-1);
if (b < 0)
b = -b;
while (a < 0) a += b;
while (a > b) a -= b;
return(a);
}

double decimate(char * S)
{
double D = 0; int n = 0; int m = 0; double p = 0;
if (DEBUG) cout<<"Decimate received: "<<S<<endl;
while (S[n]) {
if (S[n] == '.') { m = 1; break; }
else if (S[n] == 'e') return(D * pow(10,decimate(S+n+1)));
D = 10 * D + (int) (S[n] - '0');
n++;
}
n++;
if (m) {
while (S[n]) {
p++; D += (double) (((int) (S[n] - '0')) * (pow(10.0,-p))); n++;
}
} return D; }

int parse(int argc, char ** argv)
{
int i = 0,j = 0;
int k, l;
inf.open("Input.txt");
while (!inf.eof())
{
```

```
clr();
inf>>LINE;
j = 0;
while (LINE[j])
{
j++;
if (LINE[j] == 'H' && LINE[j+1] == 'E' && LINE[j+2] == 'I')
HEIGHT = decimate(LINE+8);
if (LINE[j] == 'M' && LINE[j+1] == 'u' && LINE[j+2] == 'S')
MuS = decimate(LINE+5);
if (LINE[j] == 'M' && LINE[j+1] == 'u' && LINE[j+2] == 'K')
MuK = decimate(LINE+5);
if (LINE[j] == 'M' && LINE[j+1] == 'A' && LINE[j+2] == 'S')
MASS = decimate(LINE+6);
if (LINE[j] == 'D' && LINE[j+1] == 'I')
DISTANCE = decimate(LINE+10);
if (LINE[j] == 'L' && LINE[j+1] == 'E' && LINE[j+2] == 'N')
LENGTH = decimate(LINE+8);
if (LINE[j] == 'D' && LINE[j+1] == 'E' && LINE[j+2] == 'P')
DEPTH = decimate(LINE+7);
if (LINE[j] == 'W' && LINE[j+1] == 'I')
WIDTH = decimate(LINE+7);
if (LINE[j] == 'F' && LINE[j+1] == 'A')
FALL = decimate(LINE+6);
if (LINE[j] == 'D' && LINE[j+1] == 'E' && LINE[j+2] == 'N')
DENSITY = decimate(LINE+9);
if (LINE[j] == 'G' && LINE[j+1] == 'R')
GRAVITY = decimate(LINE+9);
if (LINE[j] == 'T' && LINE[j+1] == 'H' && LINE[j+2] == 'E')
THETA = decimate(LINE+7);
if (LINE[j] == 'M' && LINE[j+1] == 'O')
MOMENT = decimate(LINE+8);
if (LINE[j] == 'V' && LINE[j+1] == 'E')
VELOCITY = decimate(LINE+10);
if (LINE[j] == 'A' && LINE[j+1] == 'C')
ACCELERATION = decimate(LINE+14);
if (LINE[j] == 'I' && LINE[j+1] == 'T' && LINE[j+2] == 'M' && LINE[j+3] == 'A')
ITMAX = decimate(LINE+7);
if (LINE[j] == 'I' && LINE[j+1] == 'T' && LINE[j+2] == 'M' && LINE[j+3] == 'I')
ITMIN = decimate(LINE+7);
}
}
inf.close();
if (DEBUG)
{
```

```
cout<<"WARNING: DEBUG = 1"<<endl;
cout<<endl<<"**********"<<endl<<"ARGC: "<<argc<<endl;
}
EXPERIMENT = 1;
while (i < argc)
{
j = 0;
if (argv[i][j] == '-')
while (argv[i][j])
{
j++;
if (argv[i][j] == 'h')
HELP = 1;
if (argv[i][j] == 'g')
{
GFX = 1;
}
if (argv[i][j] == 's')
STATS = 1;
if (argv[i][j] == 'p')
PRINT_VARS = 1;
if (argv[i][j] == 'd')
DEBUG = 1;
if (argv[i][j] == 'e') {
GFX = 1;
GRAPH = 1; }
if (argv[i][j] == 'f') {
infy>>NUMBLOCKS;
if (NUMBLOCKS > MAXBLOCKS)
{ cout<<"ERROR: Expects too many blocks."<<endl;
exit(0);
}
k = 0;
while (k < NUMBLOCKS)
{
if (DEBUG) cout<<"Obtaining info for slot block "<<k<<endl;
//l is the size.
infy>>Slots[k].Size;
infy>>Slots[k].Red;
infy>>Slots[k].Green;
infy>>Slots[k].Blue;
l = Slots[k].Size;
while (l > 0)
{
l--;
```

```
infy>>Slots[k].Vertices[l].X;
infy>>Slots[k].Vertices[l].Y;
}
k++;
}
infy.close();
}
}
j = 0;
if (argv[i][j] == '*')
while (argv[i][j])
{
j++;
if (argv[i][j] == 'H' && argv[i][j+1] == 'E' && argv[i][j+2] == 'I')
HEIGHT = decimate(argv[i]+8);
if (argv[i][j] == 'M' && argv[i][j+1] == 'u' && argv[i][j+2] == 'S')
MuS = decimate(argv[i]+5);
if (argv[i][j] == 'M' && argv[i][j+1] == 'u' && argv[i][j+2] == 'K')
MuK = decimate(argv[i]+5);
if (argv[i][j] == 'M' && argv[i][j+1] == 'A' && argv[i][j+2] == 'S')
MASS = decimate(argv[i]+6);
if (argv[i][j] == 'D' && argv[i][j+1] == 'I')
DISTANCE = decimate(argv[i]+10);
if (argv[i][j] == 'L' && argv[i][j+1] == 'E' && argv[i][j+2] == 'N')
LENGTH = decimate(argv[i]+8);
if (argv[i][j] == 'D' && argv[i][j+1] == 'E' && argv[i][j+2] == 'P')
DEPTH = decimate(argv[i]+7);
if (argv[i][j] == 'W' && argv[i][j+1] == 'I')
WIDTH = decimate(argv[i]+7);
if (argv[i][j] == 'F' && argv[i][j+1] == 'A')
FALL = decimate(argv[i]+6);
if (argv[i][j] == 'D' && argv[i][j+1] == 'E' && argv[i][j+2] == 'N')
DENSITY = decimate(argv[i]+9);
if (argv[i][j] == 'G' && argv[i][j+1] == 'R')
GRAVITY = decimate(argv[i]+9);
if (argv[i][j] == 'T' && argv[i][j+1] == 'H' && argv[i][j+2] == 'E')
THETA = decimate(argv[i]+7);
if (argv[i][j] == 'M' && argv[i][j+1] == 'O')
MOMENT = decimate(argv[i]+8);
if (argv[i][j] == 'V' && argv[i][j+1] == 'E')
VELOCITY = decimate(argv[i]+10);
if (argv[i][j] == 'A' && argv[i][j+1] == 'C')
ACCELERATION = decimate(argv[i]+14);
if (argv[i][j] == 'I' && argv[i][j+1] == 'P')
IPS = decimate(argv[i]+5);
```

```
if (argv[i][j] == 'D' && argv[i][j+1] == 'T' && argv[i][j+2] == '=')
DT = decimate(argv[i]+4);
if (argv[i][j] == 'I' && argv[i][j+1] == 'T' && argv[i][j+2] == 'M' && argv[i][j+3] == 'A')
ITMAX = decimate(argv[i]+7);
if (argv[i][j] == 'I' && argv[i][j+1] == 'T' && argv[i][j+2] == 'M' && argv[i][j+3] == 'I')
ITMIN = decimate(argv[i]+7);
}
j = 0;
i++;
}
theta = THETA / 180.0 * PI;
ANGLE = theta;
if (PRINT_VARS)
{
cout<<endl<<"**********"<<endl<<"Physical Variables:"<<endl<<endl;
cout<<"MuS="<<MuS<<endl
<<"MuK="<<MuK<<endl
<<"MASS="<<MASS<<endl
<<"DISTANCE="<<DISTANCE<<endl
<<"LENGTH="<<LENGTH<<endl
<<"DEPTH="<<DEPTH<<endl
<<"WIDTH="<<WIDTH<<endl
<<"FALL="<<FALL<<endl
<<"DENSITY="<<DENSITY<<endl;
cout<<"GRAVITY="<<GRAVITY<<endl
<<"THETA="<<THETA<<endl
<<"MOMENT="<<MOMENT<<endl
<<"VELOCITY="<<VELOCITY<<endl
<<"ACCELERATION="<<ACCELERATION<<endl;
cout<<"DT="<<DT<<endl
<<"HEIGHT="<<HEIGHT<<endl
<<"IPS="<<IPS<<endl<<endl;
cout<<"Display Variables:"<<endl<<endl
<<"ITMIN="<<ITMIN<<endl
<<"ITMAX="<<ITMAX<<endl;
cout<<"**********"<<endl<<endl;
exit(0);
}
outf.open("Input.txt");
outf<<"*MuS="<<MuS<<endl
<<"*MuK="<<MuK<<endl
<<"*MASS="<<MASS<<endl
<<"*DISTANCE="<<DISTANCE<<endl
<<"*LENGTH="<<LENGTH<<endl
<<"*DEPTH="<<DEPTH<<endl
```

```
<<"*WIDTH="<<WIDTH<<endl
<<"*FALL="<<FALL<<endl
<<"*DENSITY="<<DENSITY<<endl;
outf<<"*GRAVITY="<<GRAVITY<<endl
<<"*THETA="<<THETA<<endl
<<"*MOMENT="<<MOMENT<<endl
<<"*VELOCITY="<<VELOCITY<<endl
<<"*ACCELERATION="<<ACCELERATION<<endl;
outf<<"*DT="<<DT<<endl
<<"*HEIGHT="<<HEIGHT<<endl
<<"*ITMIN="<<ITMIN<<endl
<<"*ITMAX="<<ITMAX<<endl;
outf.close();
if (HELP)
{
cout<<endl<<"**********"<<endl<<"Help v1.1 (Last Updated 12/2/04)"<<endl
<<"——-"<<endl<<"Compile: './Lg++ <file>"'
<<endl<<"Run: './<file> -<parameters> *<assignments>"'
<<endl<<"——-"<<endl;
cout<<"h - Print this help."<<endl
<<"g - Display the Experimental Setup"<<endl
<<"e - Display the Physical Quantities"<<endl
<<"g - Use general graphics"<<endl
<<"f - Input slots from 'Blocks.txt"'<<endl
<<"p - Print model variables."<<endl<<"s - Print live statistics."
<<endl<<"d - Debug Mode"<<endl<<"**********"<<endl<<endl;
exit(0);
}
if (GRAPH && EXPERIMENT)
cout<<"WARNING - Overlapping of experiment and data."<<endl;
return(0);
}
```

Polygon.cpp

```
using namespace std;

#define MAXVERTICES 20
#define MAXNAMELENGTH 50

struct Vertex
{
double X;
double Y;
};

struct Polygon
{
int Size;
double Red;
double Green;
double Blue;
Vertex Center;
Vertex Vertices[MAXVERTICES];
int EdgeIntersect[MAXVERTICES];
char Name[MAXNAMELENGTH];
};

Polygon Box;

#define BIGARRAY 5000000
#define GAMMA 0.00001
#define GAMMA2 0.0001
#define GAMMA3 0.001
#define GAMMA4 0.01
#define SWIDTH 1000.0
#define SHEIGHT 500.0
#define SXCOORD 10
#define SYCOORD 10
#define MAXBLOCKS 10
#define MILDORF 0.8

int DEBUG = 0;
int HELP = 0;
int GFX = 0;
int STATS = 0;
int ITERATION = 0;
int PRINT_VARS = 0;
```

```
int EXPERIMENT = 0;
int GRAPH = 0;

int NUMBLOCKS = 2;
int NCOL = 0;
Polygon Slots[MAXBLOCKS];
double ITMIN = 0.0;
double ITMAX = SWIDTH;

//glColor3f(R,G,B) 0 0 0 = BLACK, 1 1 1 = WHITE
//Keep distances in meters, time in seconds, masses in kilograms

long double PI = 3.14159265358979323846264338 3;
long double TWOPI = 6.283185307;
double MuS = 1.0;
double MuK = 0.9;
double MASS = 1.0;
double DISTANCE = 1.0;
double DISTANCE2 = 1.0;
double LENGTH = 0.5;
double DEPTH = 0.5;
double WIDTH = 0.5;
double FALL = 1.0;
double DENSITY = 1.0;
double GRAVITY = 9.81;
double THETA = 60;//in degrees; theta = THETA * PI / 180.0
double MOMENT = 0;
double VELOCITY = 0;
double ACCELERATION = 0;
double DIST_TRAVELLED = 0;
double IPS = 99999999;
double FRICTION = 0;
double NORMALFORCE = 0;
double DT = .03;
double HEIGHT = 1.0;
double theta = -THETA*PI/180.0;
double VX = 0;
double VY = 0;
double CX = 0;
double CY = 0;
double PX = 0;
double PY = 0;
double H = 0;
double DX = 0;
double DY = 0;
```

```
double blah = 0;
double NewDX = 0;
double NewDY = 0;
double ForceX = 0;
double ForceY = 0;
double TORQUE = 0;
double ANGLE = theta;
double ANGULAR = 0;
double MAXSPEED = 2;
double MAXFRICTION = GRAVITY*MuK*MASS*5;
double MAXTORQUE = (LENGTH+WIDTH)*GRAVITY*MASS;
double MAXMOMENT = MASS*(LENGTH*LENGTH+HEIGHT*HEIGHT)/3.0;
double MAXANGLE = PI;
double MAXANGULAR = 1.0;
double XINTERSECT = 0;
double YINTERSECT = 0;
double MOMENTIMPULSE = 0;
double COLFRICOF = 0.2;
double COS[BIGARRAY];
double SIN[BIGARRAY];
double TAN[BIGARRAY];
double SQRT[BIGARRAY];
double SLICE = (double) (2.0 * PI) / ((double) BIGARRAY);
double SLICE2 = (double) (25.0) / ((double) BIGARRAY);

void SQRTINIT()
{
int iterator;
for (iterator=0; iterator < BIGARRAY; iterator++)
{
SQRT[iterator] = sqrt(SLICE2 * (double) iterator);
}
cout<<"Square root look up table complete."<<endl;
}

double Sqrt(double x)
{
if (x > 25)
return(sqrt(x));
int y = ((int) (x / SLICE2))while (y < 0)
y += BIGARRAY;
return(SQRT[y]);
}

void TRIGINIT()
```

```
{
double count;
double iterator;
for (iterator=0; iterator < BIGARRAY; iterator++)
{
COS[(int) iterator] = cos(SLICE * (iterator+0.5));
SIN[(int) iterator] = sin(SLICE * (iterator+0.5));
TAN[(int) iterator] = tan(SLICE * (iterator+0.5));
}
cout<<”Trig look up table complete.”<<endl;
}

double Cosine(double x)//BIGARRAY is the size of the look up tables.
{
return(COS[(int) ((x-TWOPI*floor(x/TWOPI))/SLICE)]);
}

double Tangent(double x)
{
return(TAN[(int) ((x-TWOPI*floor(x/TWOPI))/SLICE)]);
}

double Sine(double x)
{
return(SIN[(int) ((x-TWOPI*floor(x/TWOPI))/SLICE)]);
}

void swap(double &x, double&y)
{
double temp = x;
x = y;
y = temp;
}

int ordered(double a, double b, double c){ return((a <= b && b <= c) —— (a >= b && b
>= c)); }

int SegIntersect(double x1, double y1, double x2, double y2, double x3, double y3, double x4,
double y4) {
//returns 1 if the segments intersect, 0 otherwise.
//assigns (XINTERSECT,YINTERSECT) to the coordinates of the intersection.
double m1, m2, b1, b2;
double xintersect = 0, yintersect = 0;
if (x1 == x2 && x3 == x4) return(0);
else if (x1 == x2 —— x3 == x4)
```

```
{
if (x1 == x2) { swap(x1,x3); swap(x2,x4); swap(y1,y3); swap(y2,y4); }
xintersect = x3;
m1 = (y2 - y1)/(x2 - x1);
b1 = y1 - x1 * m1;
yintersect = b1 + m1 * xintersect;
if (ordered(y1, yintersect, y2) && ordered(y3, yintersect, y4) && ordered(x1, xintersect, x2))
{
XINTERSECT = xintersect;
YINTERSECT = yintersect;
return(1);
}
else
return(0);
}
else
{
m1 = (y2 - y1)/(x2 - x1);
b1 = y1 - x1 * m1;
m2 = (y3 - y4)/(x3 - x4);
b2 = y3 - x3*m2;
xintersect = (b2 - b1)/(m1 - m2);
yintersect = (m1*b2 - m2*b1)/(m1 - m2);
if (ordered(x1, xintersect, x2) && ordered(y1, yintersect, y2) && ordered(x3, xintersect, x4) &&
ordered(y3, yintersect, y4))
{
XINTERSECT = xintersect;
YINTERSECT = yintersect;
return(1);
}
else return(0);
}
}

void Point(double x, double y)
{
if (GFX == 1)
{
glBegin( GL_POINTS );
glVertex3f(x, y, 0.0);
glEnd();
glFlush();
glFlush();
}
}
```

```
void Line(double x1, double y1, double x2, double y2)
{
if (GFX == 1)
{
glBegin( GL_LINES );
glVertex3f(x1, y1, 0.0);
glVertex3f(x2, y2, 0.0);
glEnd();
glFlush();
glFlush();
}
}

void SmallBox(double x,double y,double r, double g, double b)
{
if (GFX == 1)
{
glColor3f(r,g,b);
Line(x-2,y-2,x+2,y-2);
Line(x+2,y-2,x+2,y+2);
Line(x+2,y+2,x-2,y+2);
Line(x-2,y+2,x-2,y-2);
}
}

int PolyIntersect(Polygon &P1, Polygon &P2,double draw, double r, double g, double b)//returns
1 if P1 and P2 intersect, 0 otherwise.
{
int i = 0,j, k = 0;
int P1S = P1.Size, P2S = P2.Size;
while (i < P1S)
{
j = 0;
while (j < P2S)
{
if (SegIntersect(P1.Vertices[i].X,P1.Vertices[i].Y,
P1.Vertices[(i+1)P2.Vertices[j].X,P2.Vertices[j].Y,
P2.Vertices[(j+1){
k = 1;
P1.EdgeIntersect[i]++;
P2.EdgeIntersect[j]++;
if (draw) SmallBox(XINTERSECT,YINTERSECT,r,g,b);
}
j++;
```

```
}
i++;
}
return(k);
}

void DrawPolygon(Polygon Par)
{
int i = 0;
if (GFX == 1)
{
glColor3f(Par.Red,Par.Green,Par.Blue);
while (i < Par.Size - 1)
{
Line(Par.Vertices[i].X,Par.Vertices[i].Y,
Par.Vertices[i+1].X,Par.Vertices[i+1].Y);
i++;
}
Line(Par.Vertices[i].X,Par.Vertices[i].Y,
Par.Vertices[0].X,Par.Vertices[0].Y);
}
}

void SetColor(Polygon &Par, double R, double G, double B)
{
Par.Red = R;
Par.Green = G;
Par.Blue = B;
}

void DrawBox()//RGB Valus from 0 to 1, black to white.
{
double x1 = 50.0 * LENGTH * Cosine(ANGLE);
double x2 = -50.0 * HEIGHT * Sine(ANGLE);
double y1 = 50.0 * LENGTH * Sine(ANGLE);
double y2 = 50.0 * HEIGHT * Cosine(ANGLE);
Box.Vertices[0].X = Box.Center.X + x1 + x2;
Box.Vertices[0].Y = Box.Center.Y + y1 + y2;
Box.Vertices[1].X = Box.Center.X - x1 + x2;
Box.Vertices[1].Y = Box.Center.Y - y1 + y2;
Box.Vertices[2].X = Box.Center.X - x1 - x2;
Box.Vertices[2].Y = Box.Center.Y - y1 - y2;
Box.Vertices[3].X = Box.Center.X + x1 - x2;
Box.Vertices[3].Y = Box.Center.Y + y1 - y2;
```

```
if (GFX == 1)
{

int IT = ITERATION
if (IT <= 100)
SetColor(Box, 1.0, IT/100.0, 0.0);//red to yellow
else if (IT <= 200)
SetColor(Box, 1.0, 1.0,(IT-100)/100.0);//yellow to white
else if (IT <= 300)
SetColor(Box, (300-IT)/100.0,1.0,(300-IT)/100.0);//white to green
else if (IT <= 400)
SetColor(Box, 0.0,1.0,(IT-300)/400.0);//green to cyan
else if (IT <= 500)
SetColor(Box, 0.0,(500-IT)/100.0,1.0);//cyan to blue
else if (IT <= 600)
SetColor(Box, (IT-500)/100.0,0.0,1.0);//blue to purple
else
SetColor(Box, 1.0,0.0,(700-IT)/100.0);//purple to red
DrawPolygon(Box);
}
}
```

ImageBuilder.cpp

```
#include <stdlib.h>
#include <SDL/SDL.h>
#include <GL/glut.h>
#include <iostream>
#include <math.h>
#include <fstream>
#include "Polygon.cpp"
#include "Parse.cpp"

char line[250];
int numrows;
int numperrow;
double IMPULSEADJUST=0.0;
int VALUE;

ifstream file("Default.txt");

void display(void)
{
glClear( GL_COLOR_BUFFER_BIT );
glFlush();
glFlush();
int i=0,j=0,k=0;
cout<<"In display."<<endl;
if (0)
{
glColor3f(1.0,1.0,1.0);
Line(0,0,300,400);
glColor3f(1.0,0.0,1.0);
Line(0,0,300,-400);
glColor3f(0.0,1.0,1.0);
Line(0,0,-300,400);
glColor3f(0.0,0.0,1.0);
Line(0,0,-300,-400);
}
while (!file.eof())
{
file>>VALUE;
if (VALUE > 5)
{
VALUE -= 10;
IMPULSEADJUST = 1.0;
}
```

```
else
{
IMPULSEADJUST = 0.0;
}
if (VALUE == 0)
{ glColor3f(0.0,0.0,IMPULSEADJUST); }//RGB values. Non-anomalous, reject.
else
if (VALUE == 1)//NA Accept.
{ glColor3f(0.0,1.0,IMPULSEADJUST); }
else
if (VALUE == 2)//Sketchy reject.
{ glColor3f(0.5,0.0,IMPULSEADJUST); }
else
if (VALUE == 3)//Sketchy accept.
{ glColor3f(0.5,1.0,IMPULSEADJUST); }
else
if (VALUE == 4)//Terrible reject.
{ glColor3f(1.0,0.0,IMPULSEADJUST); }
else
//Terrible accept.
{ glColor3f(1.0,0.0,IMPULSEADJUST); }
Point(i,2*numrows-j);
//cout<<"Tried to plot a value "<<VALUE<<" at ("<<i<<","<<2*numrows-j<<"<<)."<<endl;
i++;
if (i }
getchar();
exit(0);
}

int InitializeGFX(int argc, char ** argv)
{
glutInit(&argc, argv);
glutInitDisplayMode( GLUT_SINGLE — GLUT_RGB );
glutInitWindowSize( (int) numperrow, (int) 2*numrows );
glutInitWindowPosition( 5, 5 );
glutCreateWindow ("Contents of the Source");
glClearColor(0.0, 0.0, 0.0, 0.0);
glMatrixMode( GL_PROJECTION );
glLoadIdentity();
glOrtho(0,numperrow,0,2*numrows, -1.0, 1.0);
glutDisplayFunc( display );
cout<<"Submitted window of width "<<numperrow<<" and height "<<2*numrows<<"."<<endl;
GFX = 1;
cout<<"Finished graphics initializer."<<endl;
return(0);
```

```
}

int parsefile()
{
file>>line;
cout<<line<<endl;
file>>line;
cout<<line<<endl;
file>>line;
cout<<line<<endl;
file>>numrows;
cout<<"numrows="<<numrows<<endl;
file>>line;
cout<<line<<endl;
file>>line;
cout<<line<<endl;
file>>line;
cout<<line<<endl;
file>>line;
cout<<line<<endl;
file>>line;
cout<<line<<endl;
file>>numperrow;
cout<<"numperrow="<<numperrow<<endl;
file>>line;
cout<<line<<endl;
file>>line;
cout<<line<<endl;
}

int main(int argc, char ** argv)
{
if (!parsefile()) { cout<<"Error reading file."<<endl; exit(0); }
InitializeGFX(argc, argv);
glutMainLoop();
cout<<"End of Program. Hit any key to exit."<<endl;
getchar();
return 0;
}
```

5 Appendix B - Credits

Tipler Physics.