Sediment Dynamics Simulation via Cellular Automata Artificial Intelligence - SYNOPSIS 1.0

Dr. Juan M. Restrepo
Matthew D. Jallo
Jason E. Young

A synopsis of Spring 02 undergraduate research conducted under the VIGRE grant

I) Introduction

Our names are Matthew Jallo and Jason Young, and this brief report will attempt to outline and document our experiences as undergraduate research assistants in the Math Center at the University of Arizona under the VIGRE grant. We had the opportunity to work side by side with the math department faculty, especially Dr. Restrepo, to whom we are grateful for his contribution of time and expert guidance.

Scope of Project

This project is compromised of study into two primary interests: sediment dynamics and artificial intelligence.

Sediment Dynamics

Sediment dynamics is the physics of sedimentary beds (i.e. sand, mud, etc.) under external forces, usually viscous sheer of fluids. For the purposes of this project, we have limited our consideration of the scenario of wind blowing over sand, as one might expect to find in the Sahara desert or at the beach in San Diego. The University of Arizona, situated in the heart of the Sonoran desert, has afforded us an appreciation of sand even locally.

Although initially we sought to do broad based research on neural network AI, that would have shifted the project emphasis from mathematics to neurobiology, and was ultimately decided to be too complex an element to address in one semester of work.

After coming to this conclusion we decided to form a solid groundwork of a physical phenomenon well researched by the mathematics community applied to less ambitious forms of artificial intelligence systems.

Thus studying sediment dynamics with cellular automata became the focus of this semester's work, and proved to be quite a task indeed.

Artificial Intelligence

Because this research project is primary motivated by our desire to get involved in the field of artificial intelligence, we shall explain what artificial intelligence is and how our project is an example of its contemporary usage.

Modern artificial intelligence is essentially the utilization of computers to discover patterns quicker and easier than by hand. Thus software based on the artificial intelligence paradigm seeks to take some of the lexical leeway off of the user. Artificial intelligence is well suited for applications such as machine image recognition like in a robot or the OCR (optical character recognition) component of document scanning. Additionally automated recognition of

patterns, be they faces, handwriting samples, voices, words spoken in language, or ultimately even the ability to strategize tasks is achievable through artificial intelligence.

This also provides the opportunity to in effect "work backwards" from the paradigm, and recreate more realistic simulations of real phenomenon. Using artificial intelligence software to create models of physical experience with a high degree of accuracy allows us to stimulate our own intelligences to ascertain the underlying dynamics of many questions and contributes greatly to the opportunity of using technology as part of discovering solutions.

The particular type of artificial intelligence used in our system is called cellular automata. As the name suggests, it is a method by which we can use computers to simulate complex systems by identifying and implementing a simulation based on key interactions between discrete units, or cells. A good example is the human body – although the biology of the human body is difficult to study as a complete system, it is beneficial to consider that it is made up of many cells which have relatively few but proportionally more fundamental roles in the workings of the body as a whole.

Likewise, if one is to study the formation of sand dunes, it is valuable to use a computer to track and simulate the roles of every single particle of sand that composes the dune. Such a thing would be incredibly tedious by hand, but computers run up the electric bill whether they're working hard on a task or not. Thus they are prime candidates for the grunt work, and allow us to free our minds to ponder the more profound implications of the data we find.

Cellular automata is probably the simplest form of artificial intelligence, and its hardware requirements are so low that a simple interactive java script in an internet web browser often times can provide indispensable clues to the operation of a complex system.

In the case of our research with sediment dynamics, the ultimate goal of the simulator was to deduce mathematical models of sediment dynamics in the form of partial differential equations, and compare these findings to existing observations established previously by mathematicians.

The core question in the application of our software is consequently this: What conditions contribute to the formation of the patterns found in sand formations?

Components of Research

To help answer this question, we worked as a team under the expert direction of Dr. Restrepo. Michelle Kidd, a graduate student in Applied Mathematics formed our essential link to the study of sediment dynamics, and spent long nights in the computer room writing MATLAB code that was later ported to C++.

Timeline of work

The first month or so of our work consisted of planning, such as determining what type of software to implement and what types of considerations would be necessary in terms of sediment dynamics research. The first C++ port of the code turned out less than optimal and had some issues with data representation and manipulation. It did however provide a few valuable lessons

which were integrated into the second release of the software. After spending a few months discussing the best ways to implement various cellular interactions, we finally set about completing the software so that the graphs could be analyzed.

II) Introduction to the simulation environment

The simulation environment's creation was based on principles from Newtonian physics, so that the basic interactions of the sand particles could be modeled without reinventing the wheel. According to the cellular automata paradigm, mechanical basics such as two dimensional motion, gravity, momentum, friction, static equilibrium and elasticity, and fluid mechanics can all contribute to the final comprehensiveness of the entire model.

A note should be made that although we attempted to adhere to the basics of Newtonian mechanics, occasionally pseudo arbitrary compensating factors were introduced in order to accomplish our goals in a timely fashion. As a result, it was not uncommon for us to "tweak" a calculation occasionally in order to compensate for the simulation's lack of complete adherence to mechanical principle, and ultimately to provide more stability in the software. These manipulations are however well marked in the code, so anyone desiring to enhance the system for better compliance with mechanical physics can do so without too much trouble.

As stated above, the behavior and functionality of the system is derived from the many interactions of each grain of sand. Although there is an exhaustive list of rule possibilities, part of our task was narrowing down what we thought were the most consequential interactions.

When considering a single grain of sand, as seen in figure 1.1, there are many considerations to take into account. At the outset, the diagram is quite unassuming but considerations of the simple things are what make cellular automata systems powerful. The particle itself is presumed to consist of Silicon Dioxide, or some similar compound. Although the crystal structure of the particle is not necessarily well represented by a rectangular shape, we formed the ideal assumption that shape is of little consequence to the motion of the particle. Thus although the particle is represented as a rectangle graphically in order to emphasize the matrix-like nature of the simulation, the software occasionally makes the ideal assumption that the particle is perfectly spherical for purposes of rolling.

Figure 1.1 – *The basic particle*

Next we assert that this particle is subject to viscous forces and, deductively, furthermore subject to frictional forces. In the software, frictional forces are usually calculated by means of a simple fractional coefficient. Although somewhat arbitrary, this allows the various calculations to remain perfectly related to each other in the environment without causing redundancy or lack of numerical cohesion.

Figure 1.2 – *The particle with frictional considerations*

Without forces acting on the particle it would not move, and needless to say that would not provide a very usable simulation. Indeed if the frictional coefficients are set too high the simulation, albeit predictably, will not budge. All mechanical vector calculations are simplified to 4 primary force vectors acting on the particle as illustrated in figure 1.3.

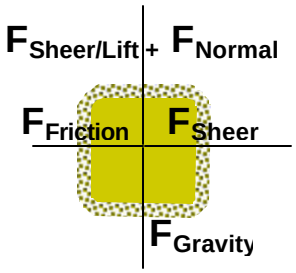


Figure 1.3 – *The frictional particle subjected to forces*

These particles are then simulated in a bed of finite width and height. In the case of figure 1.4, the bed is only 16 discrete particles wide. Even simulations with small beds yield surprisingly interesting results in spite of the fact that a sand dune might actually be many millions or billions of particles wide. In fact, we have made the following observation:

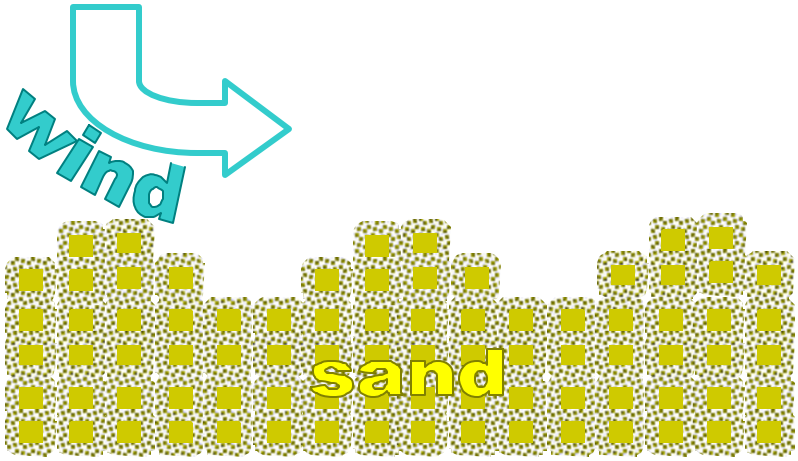


Figure 1.4 – *matrix representation of the sand bed*

a) small numbers of automata yield tend to yield results that are visibly more realistic because of the corresponding "smoothing" effect.
b) Large numbers of automata yield very distinct patterns that are visibly unrealistic, yet very telling of the dynamics included in their forming.

Figure 1.5 is a screenshot of an example sediment bed in the simulator:

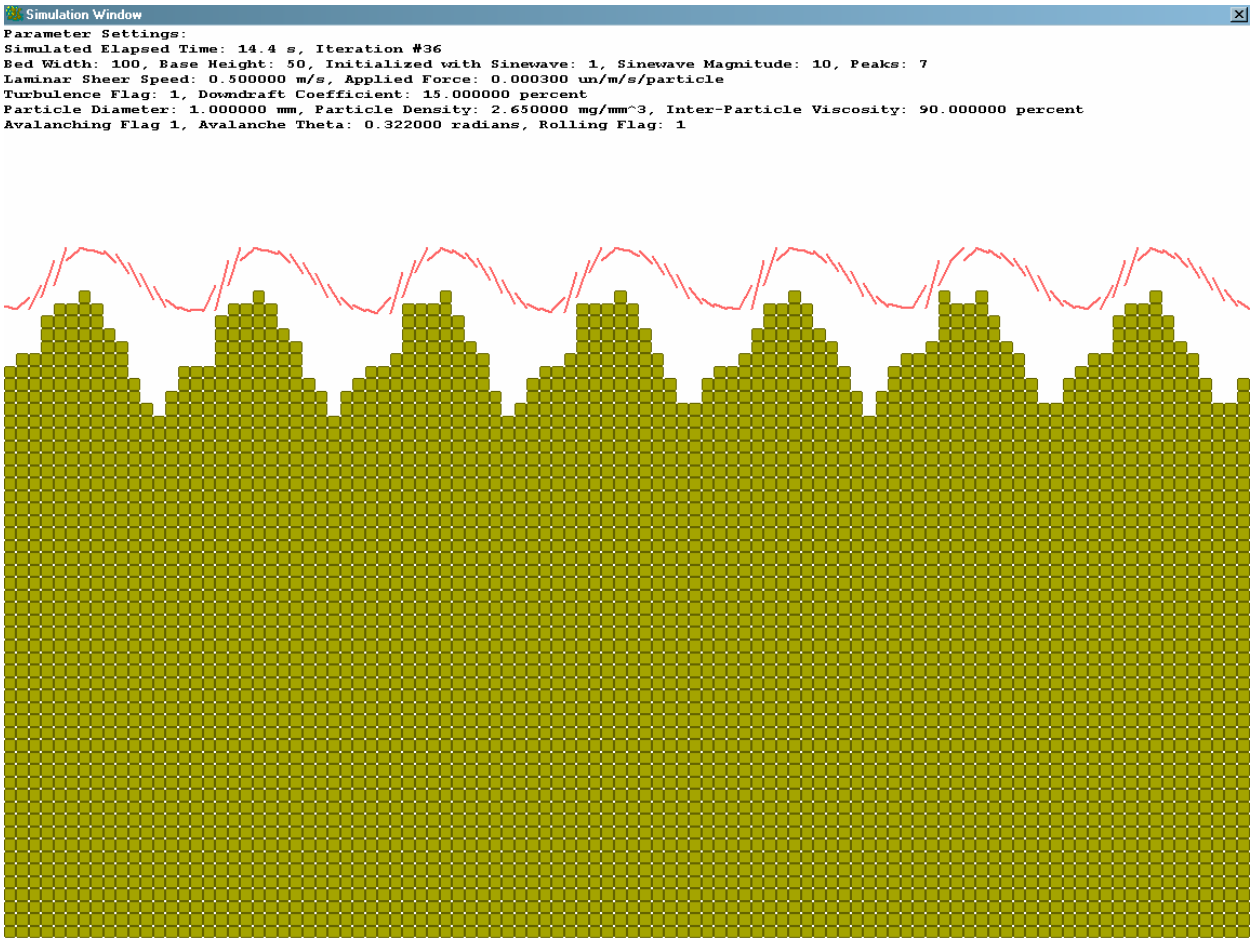


Figure 1.5 – *Screenshot of an actual sand bed in the simulator*

The graph from the simulator contains all the pertinent parameter data at the top, as well as a graphical display of the sand bed as it changes. The red lines indicate the turbulent flow of wind over the bed. Figure 1.6 is a screenshot of the parameter settings screen:

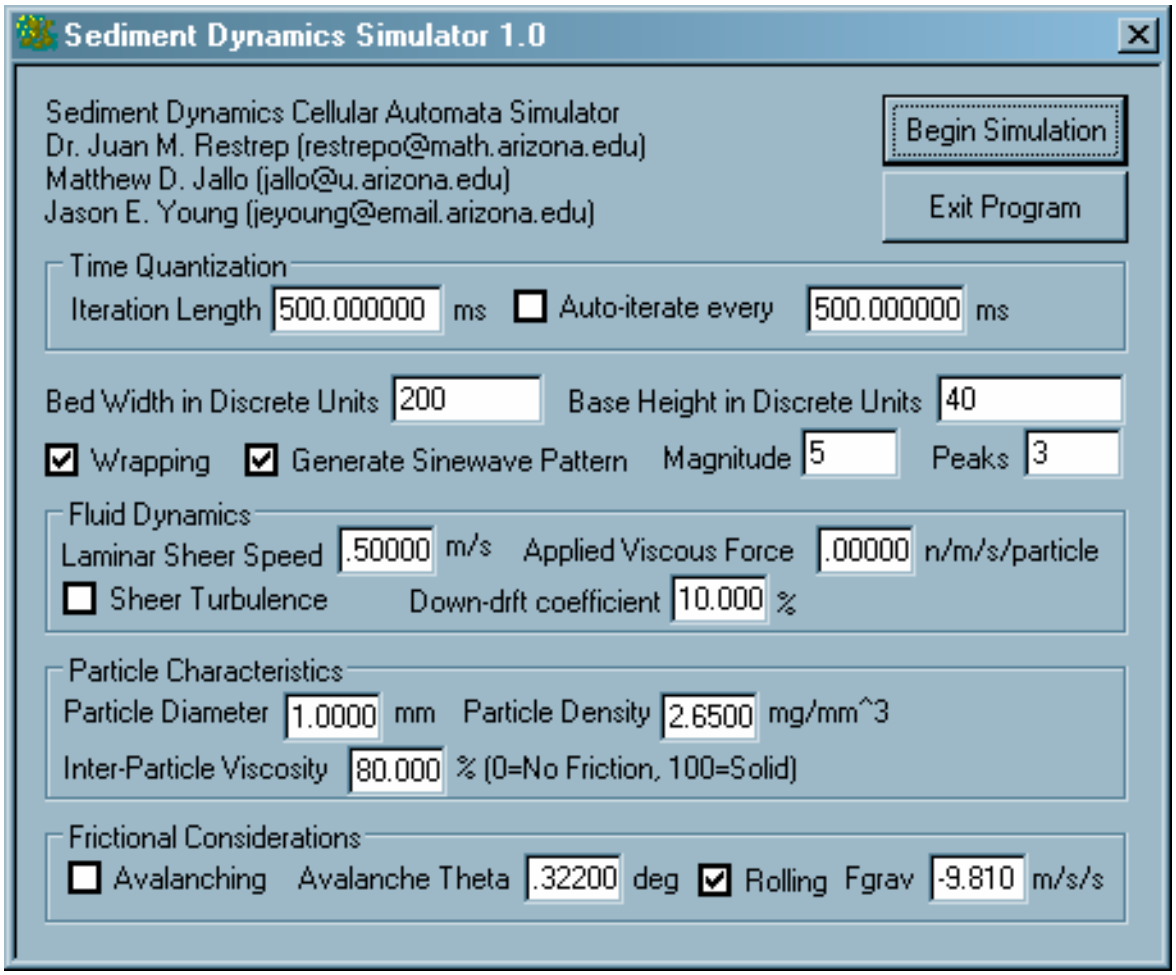


Figure 1.6 – *Screenshot of an actual sand bed in the simulator*

Iteration Length: Because the computer cannot perform calculations in perfect real-time, the simulation time is quantized into the iteration length specified. In this example, each iteration advances half a second in time.

Auto-iterate: When selected, the software will automatically advance through the iterations without the user having to click the mouse. In this example, had auto-iterate been selected the software would automatically iterate every half a second.

Bed Width in Discrete Units: This is the width of the sand bed in particles of sand

Bed Height in Discrete Units: The base height, or depth, of the sand bed.

Wrapping: When selected, particles moving off the right of the matrix are recycled to the left. This is highly recommended.

Generate Sinewave Pattern: When selected, the bed will be initialized with a sine wave pattern that prevents problems with getting the simulation to budge from a perfectly flat configuration.

Magnitude: The magnitude of the sine wave peaks in discrete units

Peaks: The number of sine wave peaks in the bed

Laminar Sheer Speed: The speed of the fluid moving over the sand, in this case wind. Measured in meters per second.

Applied Viscous Force: This is the force applied to each particle as a result of the fluid moving over its top, in newtons per meter per second per particle. By default, $3 \bullet 10^{-10} \frac{n}{m \bullet s^{-1}} \bullet particle^{-1}$

Sheer Turbulence: When selected, the software will duplicate simple turbulent behavior in the fluid.

Down-draft Coefficient: If turbulence is selected, this is essentially a simple coefficient defining the quadratic descent of the fluid pattern.

Particle Diameter: The diameter of each particle, in millimeters. This is used to calculate the weight and perhaps surface area in future versions.

Particle Density: The density of the particulate material in milligrams per cubic millimeter. By default, 2.65 is supplied for the density of Silicon Dioxide.

Inter-Particle Viscosity: This is a coefficient defining the intensity of friction between multiple sand particles.

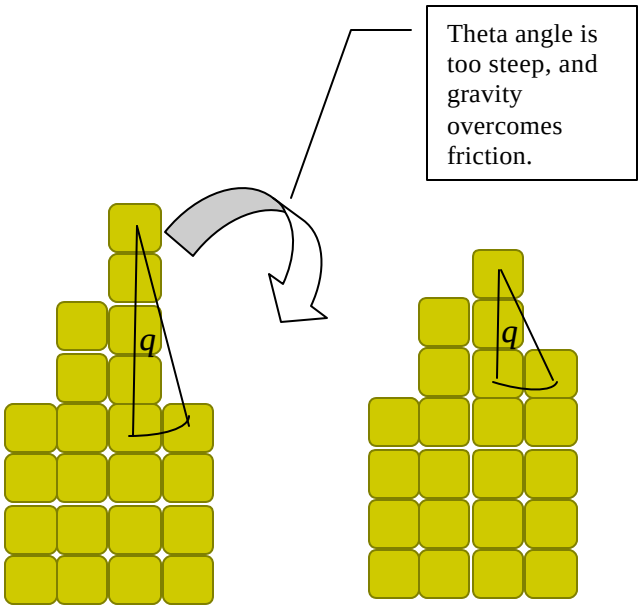


Figure 1.7 - Avalanching

Avalanching: When selected, the software will simulate avalanching, a natural phenomenon caused by gravity and counteracted by friction. Refer to figure 1.7

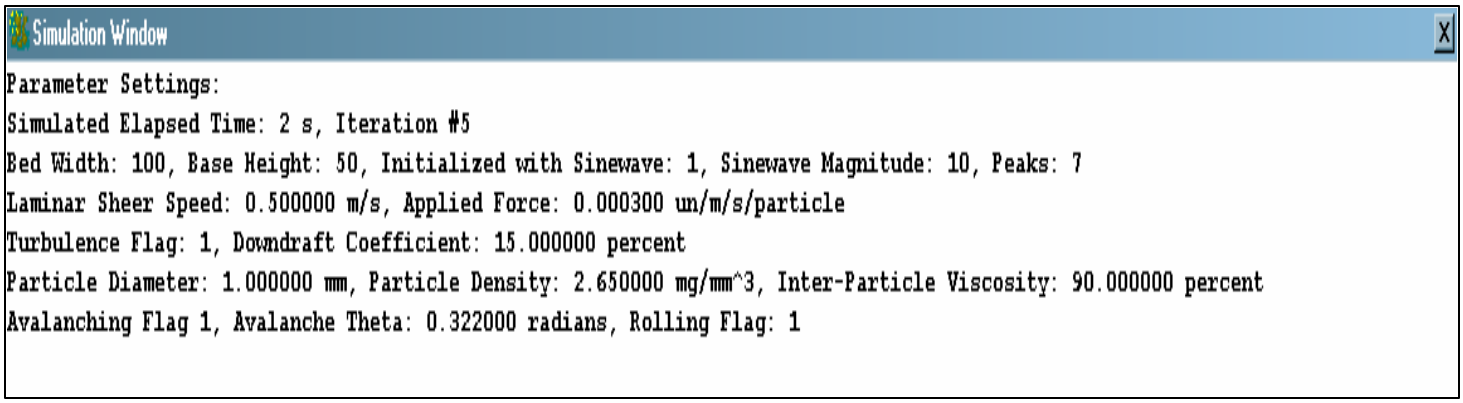
Avalanche Theta: The maximum angle at which particle friction can sustain a column's height before tumbling. Refer to figure 1.7

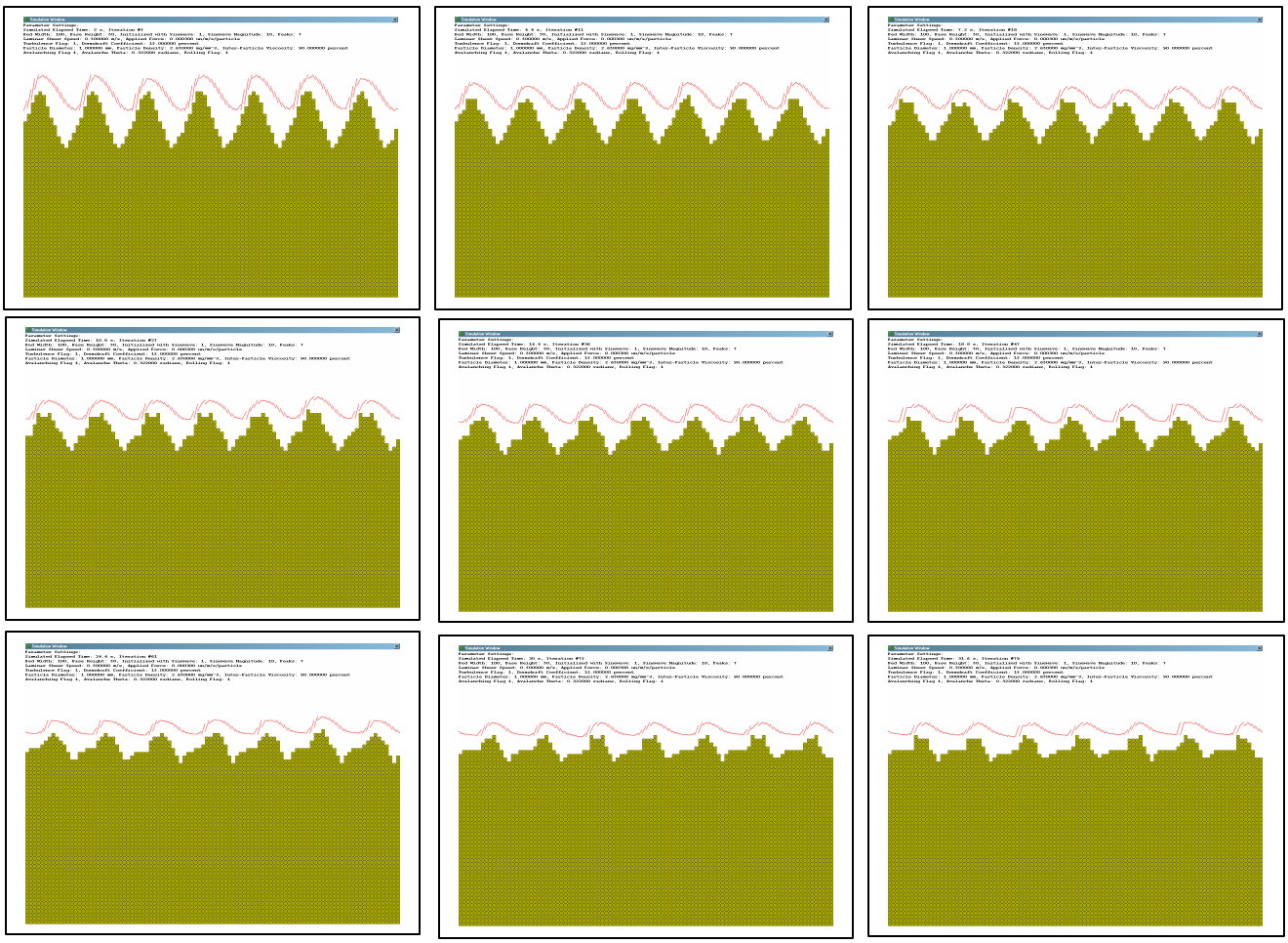
Rolling: When selected, a particle can roll if it has enough force to overcome friction

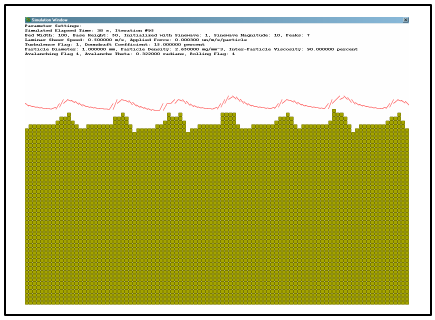
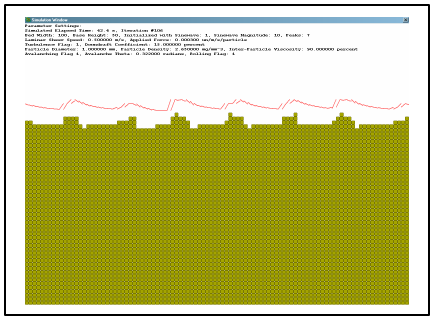
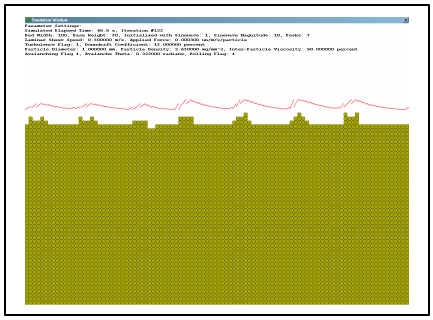
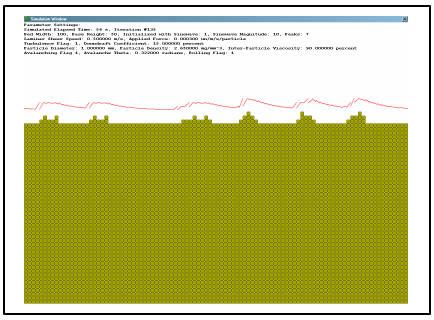
Fgrav: Acceleration of gravity in meters per second per second

III) Selected Results and Conclusions

Comprehensive Simulation:

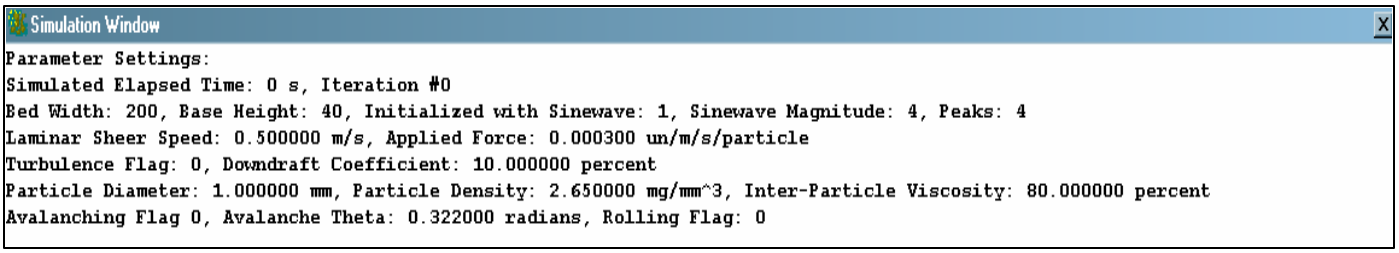


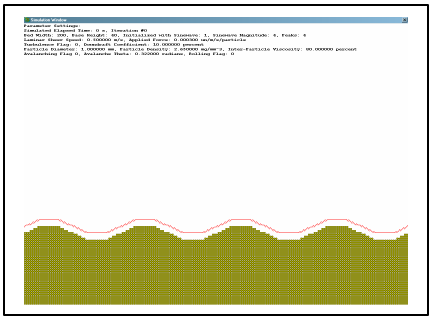
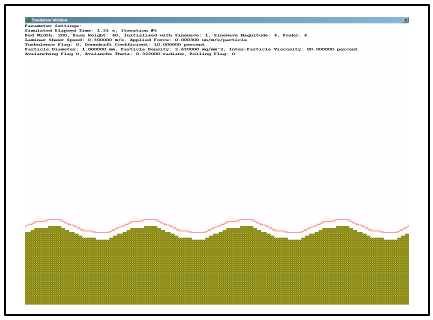
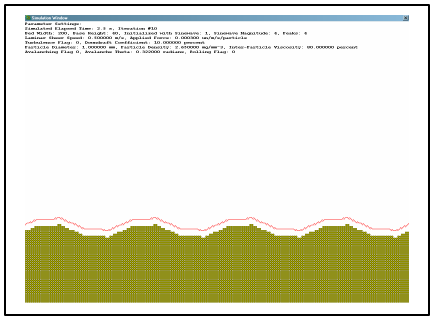
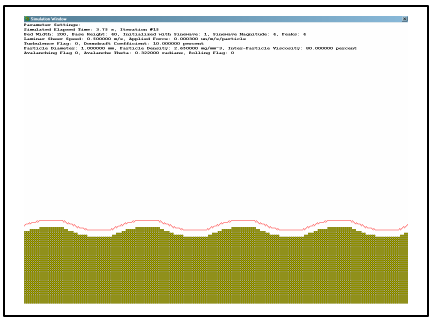
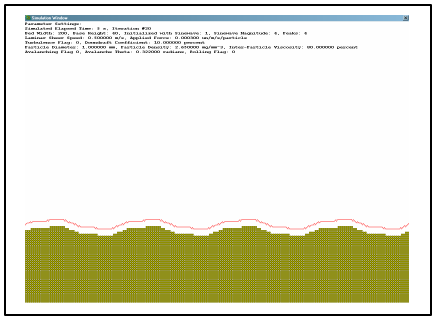
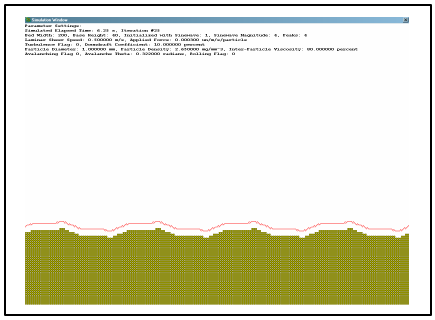


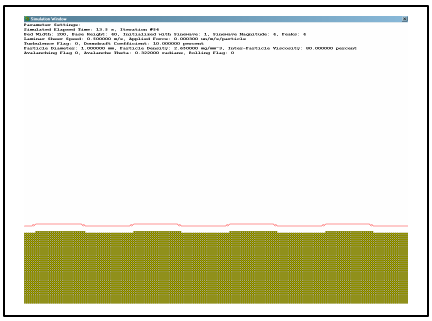





As illustrated from the above comprehensive trials, the strongest tendency exhibited by the model is a flattening effect. Valleys are filled and dunes are sheered. Because the simulation is based on a discrete bed width, integer redundanc ies are modulated even with wrapping.

Basic Simulation



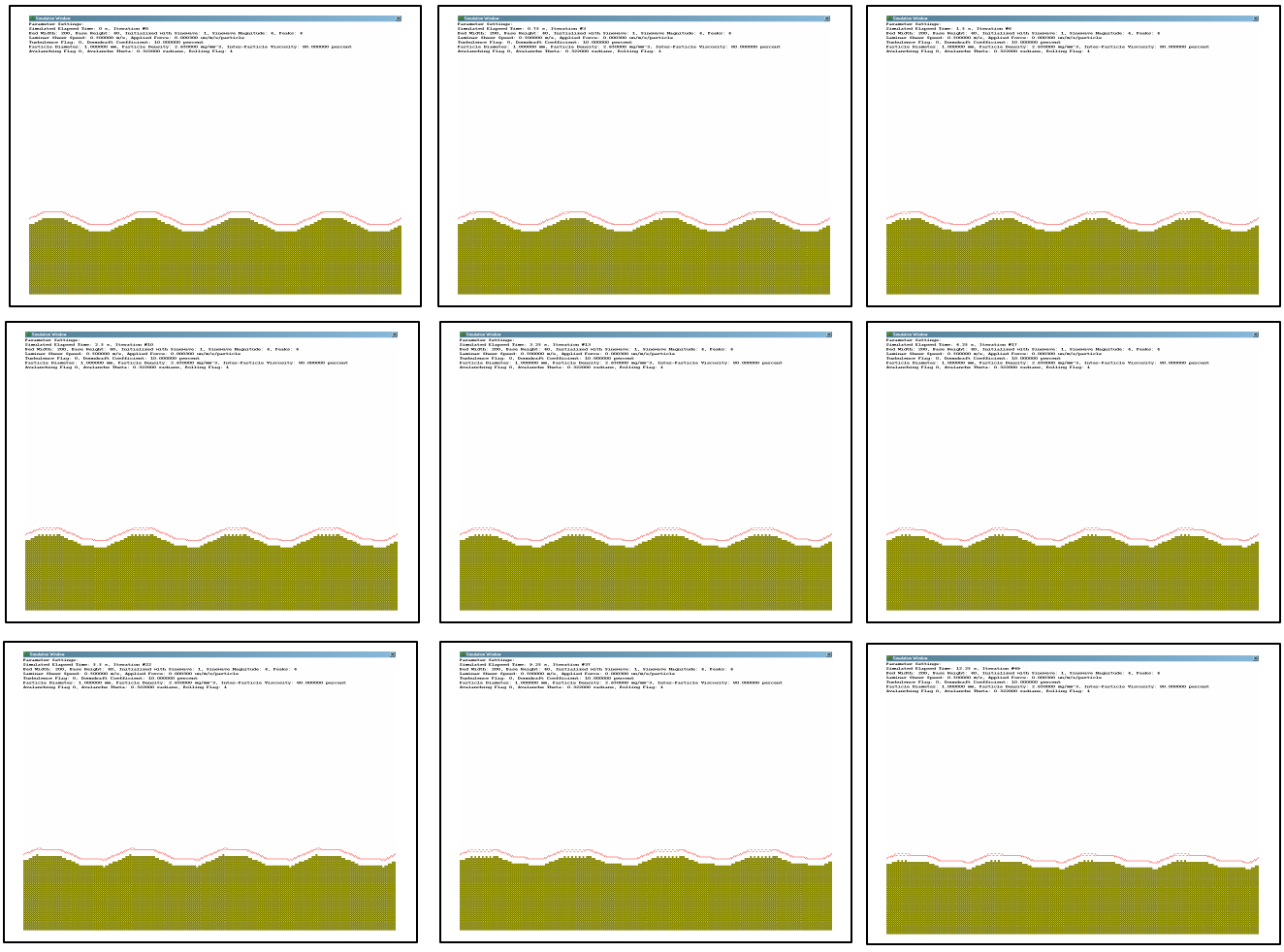


The flattening effect described earlier can clearly be seen in the above simulation, which restricts cellular roles to viscous sheer alone.

Rolling Simulation:

Simulation Window

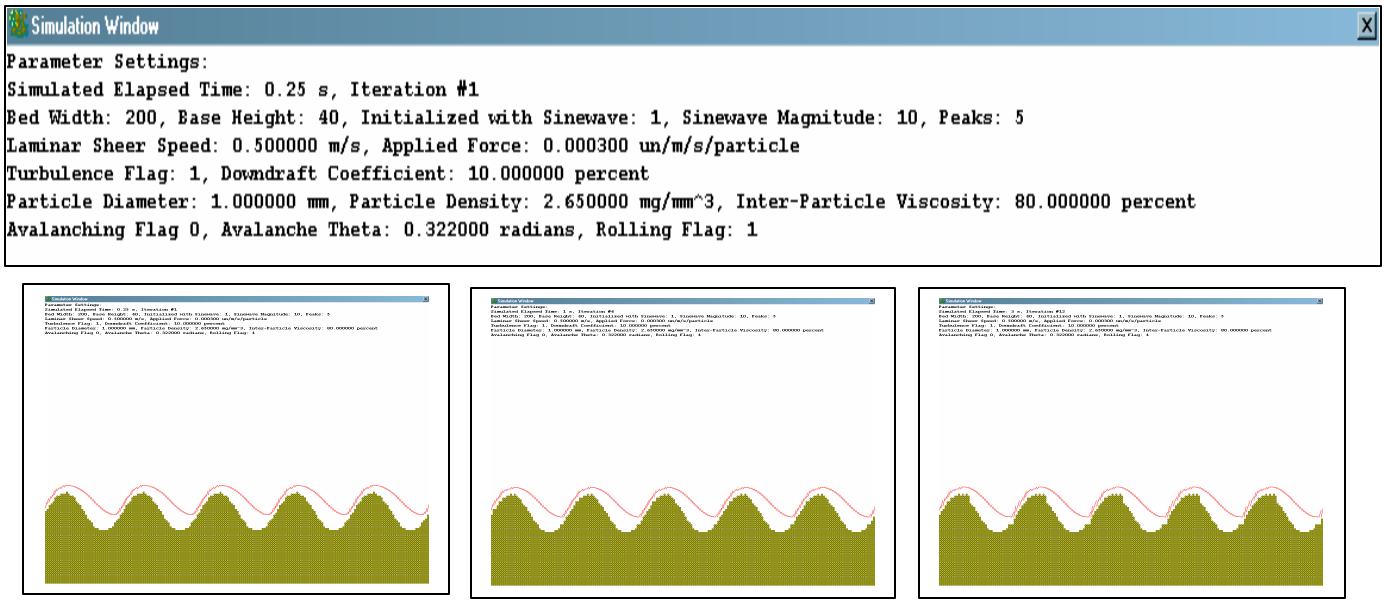
```
Parameter Settings:
Simulated Elapsed Time: 0 s, Iteration #0
Bed Width: 200, Base Height: 40, Initialized with Sinewave: 1, Sinewave Magnitude: 4, Peaks: 4
Laminar Sheer Speed: 0.500000 m/s, Applied Force: 0.000300 un/m/s/particle
Turbulence Flag: 0, Downdraft Coefficient: 10.000000 percent
Particle Diameter: 1.000000 mm, Particle Density: 2.650000 mg/mm^3, Inter-Particle Viscosity: 80.000000 percent
Avalanching Flag 0, Avalanche Theta: 0.322000 radians, Rolling Flag: 1
```

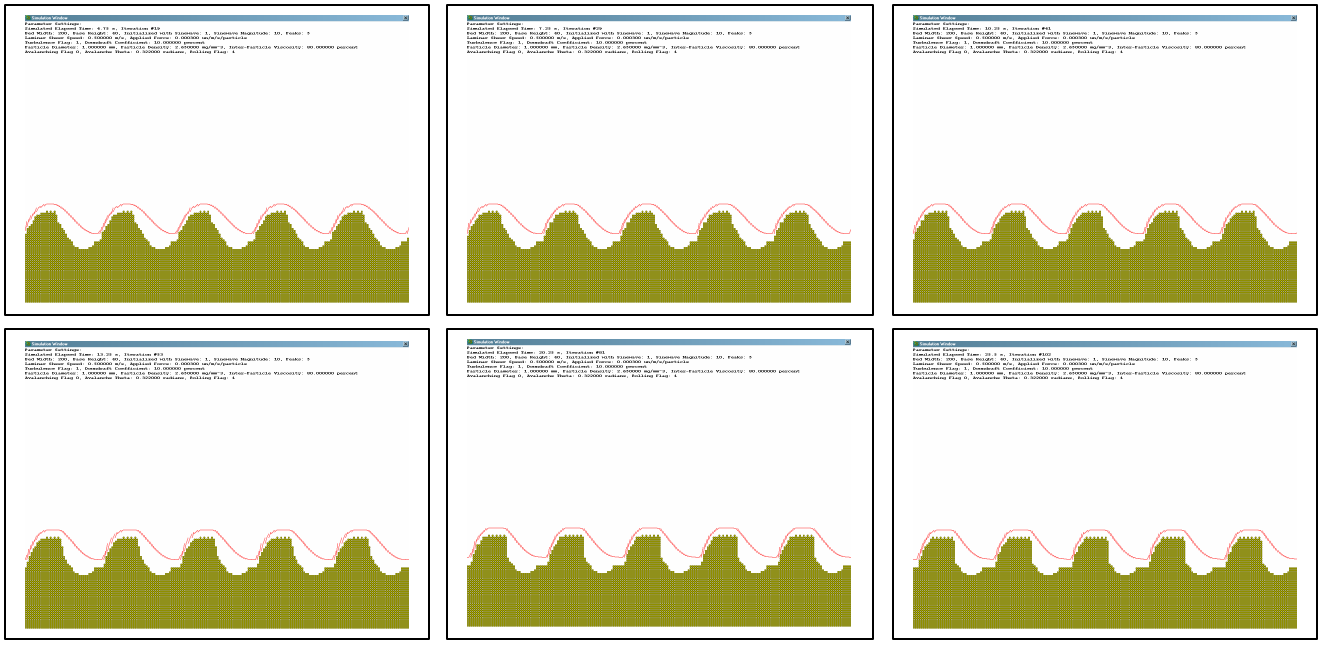


The above simulation demonstrates the effects of adding rolling into the simulation parameters. The flattening effect of the sheer is still quite evident, perhaps more so.

Expanded Rolling Simulation with Turbulence

The following simulation adds turbulence to the basic sheer force and rolling:

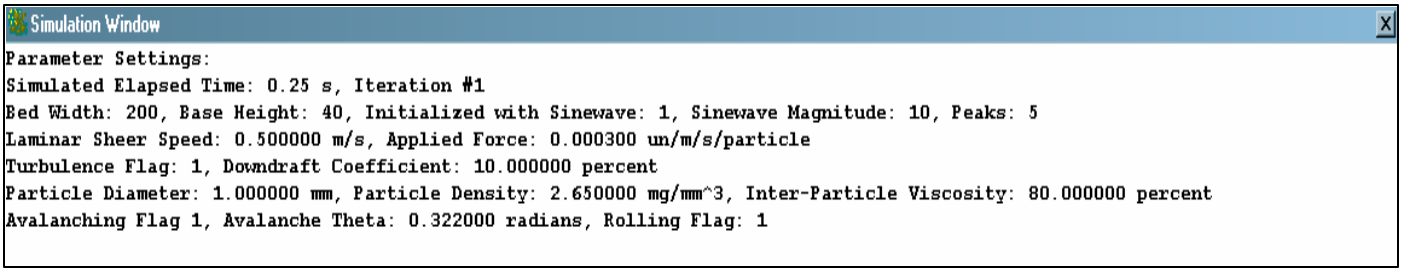


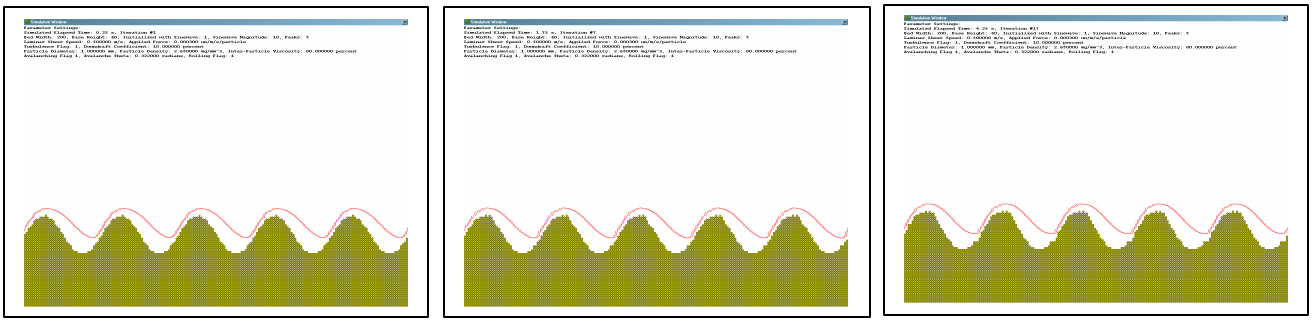


The resulting sphinx-like structure of the dunes is primarily a result of the wind turbulence, and seems to be what prevents complete flattening from happening. Incidentally the frequent recurrence of pyramidal, obelisk, and sphinx-like shapes throughout our research have resounded well with the monuments of the ancient Egyptians, no strangers to wind and sand.

Avalanching

Now we will add avalanching to the simulation:





The above simulation clearly shows the additional flattening effect brought by avalanching. In theory, because the force of gravity is always pulling normal to the sand bed it is

the ideal flattening rule. In our Sahara desert simulation, the sheering effect is much more pronounced than avalanching but this would not be the case in situations such as quicksand, where gravity and lack of inter-particle friction are the dominant roles.

The simulations above and many others not included for purposes of brevity allow us to come to the following conclusions:

a) viscous sheer and gravity on an ideally flat plane will not cause any changes to bed structure
b) perturbations of the sand column heights be they discrete or even microscopic cause some amount of turbulence which combines with other factors such as rolling to create periodic patterns in the sand bed.

Potential Future Development

In addition to modifying the software for better compliance with mechanics, there are plenty of rules that are still to be tried. Further development might take into consideration the crystal structures of the particles, surface area, enhanced quadratic simulation of gravity and turbulence, or varying particle sizes. An option to turn on varying levels of pseudo-random deviation from selected parameters might also yield interesting results.

Although time constraints rendered us unable to analyze the graphs with respect to their correspondence to known partial differential equations for various dynamics in proper detail, clearly this is a good next step as well.

To combine the two, a symbolic manipulation system such as Jason Young and Matt Jallo's "Aristotle CAS" could be used to automatically analyze a wide range of parameters, develop accurate polynomial representations of them, and then compare their deviation from a Taylor Series or Fourier transformation of known equations.

Source Code

```
//SedSim.cpp : implementation file

//updated may 02 - Matt Jallo
//updated march 29 - Matt Jallo
//updated march 24 - Jason Young


#include "stdafx.h"
#include "sediment.h"
#include "SedSim.h"
#include "math.h"

// CSedSim

IMPLEMENT_DYNAMIC(CSedSim, CWnd)
CSedSim::CSedSim()
: bedHeight(100)
, bedWidth(200)
, bedDepth(40)
, Wrapping(true)
```

```
, generateSinewave(true)
, SinewaveMagnitude(5)
, SinewavePeaks(3)
, iterationLength(500)
, autoIterate(false)
, sheerSpeed(.5)
, appliedViscousForce(.0000000003)
, sheerTurbulence(false)
, downDraftCoefficient(10)
, autoIterateLength(500)
, particleDiameter(1)
, particleDensity(2.65)
, interParticleViscosity(80)
, iterationNumber(0)
, avalanching(false)
, avalancheTheta(.322)
, rolling(true)
, forceGravity(-9.81)
{
      infoFont.CreateFont(14, 10, 0, 0, FW_BOLD, false, false, false,
ANSI_CHARSET, OUT_TT_PRECIS, CLIP_DEFAULT_PRECIS, ANTIALIASED_QUALITY,
FF_DONTCARE, "Courier New");
}

CSedSim::~CSedSim()
{
}

BEGIN_MESSAGE_MAP(CSedSim, CWnd)
      ON_WM_PAINT()
      ON_WM_ERASEBKGND()
      ON_WM_LBUTTONDOWN()
      ON_WM_TIMER()
END_MESSAGE_MAP()

// initializes the simulation, and allocates memory required for current
parameters
void CSedSim::InitSimulation(void)
{
      //reset iteration counter
      iterationNumber=0;

      //Clear the primary sandbed vector
      sandbed.clear();

      //Allocate the sandbed vector
      sandbed.resize(bedWidth);

      //"Fill" the sandbed with bedDepth grains if there is to be no sinewave
      if(!generateSinewave)
      {
            for(int i=0; i<bedWidth; i++)
            {
                  sandbed[i].columnHeight=bedDepth;
                  sandbed[i].sheerHeight=bedDepth+1;
                  sandbed[i].particleVelocity=0;
            }
```

```
	}

	//Generate Sinewave pattern if requested
	if(generateSinewave)
	{
		//Find period step with relation to bedwidth
		double pi=3.1415926535;
		double grainStep=(pi*double(SinewavePeaks*2))/double(bedWidth);

		//Adjust CHARVECTORs for sinewave magnitude
		for(int i=0; i<bedWidth; i++)
		{
			int columnHeight=bedDepth;
			columnHeight+=sin(grainStep*i)* SinewaveMagnitude;
			sandbed[i].columnHeight=columnHeight;
			sandbed[i].sheerHeight=columnHeight+1;
			sandbed[i].particleVelocity=0;
		}
	}
}

void CSedSim::OnPaint()
{
	CPaintDC dc(this); // device context for painting
	/*Draw the sandbed vectors onto the display context*/

	//Find window dimensions
	CRect winSize;
	this->GetClientRect(winSize);

	//double buffer blitting
	CDC memDC;
	memDC.CreateCompatibleDC(&dc);
	CBitmap memBmp;
	memBmp.CreateCompatibleBitmap(&dc, winSize.Width(), winSize.Height());
	memDC.SelectObject(memBmp);
	memDC.FillRect(winSize, &CBrush(0x00FFFFFF));

	//Create GDI objects
	CPen grainOutline(PS_SOLID, 1, 0x00006464);
	CPen fluidLine(PS_SOLID, 2, 0x006464FF);
	CBrush grainFill(0x0000A4A4);

	//Select the objects into the GDI
	CPen *oldPen = static_cast<CPen*>(memDC.SelectObject(grainOutline));
	CBrush *oldBrush = static_cast<CBrush*>(memDC.SelectObject(grainFill));
	CFont *oldFont = static_cast<CFont*>(memDC.SelectObject(infoFont));

	//Create Simulator Settings String
	char simInfo[4096];
	sprintf(simInfo,
		"Parameter Settings:\nSimulated Elapsed Time: %g s, Iteration
#%d\nBed Width: %d, Base Height: %d, Initialized with Sinewave: %d, Sinewave
Magnitude: %d, Peaks: %d\nLaminar Sheer Speed: %f m/s, Applied Force: %f
un/m/s/particle\nTurbulence Flag: %d, Downdraft Coefficient: %f
percent\nParticle Diameter: %f mm, Particle Density: %f mg/mm^3, Inter-
```

```
Particle Viscosity: %f percent\nAvalanching Flag %d, Avalanche Theta: %f
radians, Rolling Flag: %d"
            ,iterationNumber*double(iterationLength)/1000, iterationNumber,
            bedWidth, bedDepth, generateSinewave, SinewaveMagnitude,
SinewavePeaks,
            sheerSpeed, appliedViscousForce*1000*1000,
            sheerTurbulence, downDraftCoefficient,
            particleDiameter, particleDensity,  interParticleViscosity,
            avalanching, avalancheTheta, rolling);

      //Draw the text
      memDC.DrawText(simInfo, strlen(simInfo), winSize, DT_LEFT | DT_TOP);
      //Declare variables used by the looping
      CRect grainRect; /* will keep track of where to draw each grain */
      CPoint roundingPoint; /*indicates the amount of rounding on the grains
*/
      CPoint startFluidPoint; /*the beginning point of the fluid line*/
      CPoint endFluidPoint; /*the end point of the fluid line*/
      float grainSize=float(winSize.Width())/float(bedWidth); /* tracks the
width and height of the grain */
      int xStep=0, yStep=0;

      //Start looping
      for(xStep=0; xStep<bedWidth; xStep++)
      {
            for(yStep=0; yStep<bedHeight; yStep++)
            {
                  //Does this coordinate contain a sand grain?
                  if(yStep>=sandbed[xStep].columnHeight) continue;
                  //calculate this grain's dimensions and position
            grainRect.left=float(xStep * grainSize);
                  grainRect.top=winSize.Height()-(yStep*grainSize) +
grainSize;
                  grainRect.right=(xStep * grainSize)+grainSize;
                  grainRect.bottom=(winSize.Height()-(yStep*grainSize) +
grainSize)+grainSize;
                  roundingPoint.x=grainSize/3+1;
                  roundingPoint.y=grainSize/3+1;

                  //Draw the grain
                  memDC.SelectObject(grainOutline);
                  memDC.RoundRect(grainRect, roundingPoint);
            }

            //Draw the fluid pattern
            startFluidPoint.x=xStep*grainSize;
            if(xStep>0)
                  startFluidPoint.y=winSize.Height()-(sandbed[xStep-
1].sheerHeight*grainSize + grainSize/2);
            else
            {
                  if(Wrapping)
                startFluidPoint.y=winSize.Height()-(sandbed[bedWidth-
1].sheerHeight*grainSize + grainSize/2);
                  else
                        startFluidPoint.y=winSize.Height()-
(sandbed[xStep].sheerHeight*grainSize + grainSize/2);
```

```
            }

            endFluidPoint.x=xStep*grainSize+grainSize;
            if(xStep!=bedWidth-1)
                  endFluidPoint.y=winSize.Height()-
(sandbed[xStep+1].sheerHeight*grainSize + grainSize/2);
            else
            {
                  if(Wrapping)
                        endFluidPoint.y=winSize.Height()-
(sandbed[0].sheerHeight*grainSize + grainSize/2);
                  else
                        endFluidPoint.y=winSize.Height()-
(sandbed[xStep].sheerHeight*grainSize + grainSize/2);
            }

            memDC.SelectObject(fluidLine);
            memDC.MoveTo(startFluidPoint);
            memDC.LineTo(endFluidPoint);
      }


      //blit the buffer to the screen DC
      dc.BitBlt(0, 0, winSize.Width(), winSize.Height(), &memDC, 0, 0,
SRCCOPY);

      //Restore GDI objects to previous state
      //dc.SelectObject(oldPen);
      //dc.SelectObject(oldBrush);
}

BOOL CSedSim::OnEraseBkgnd(CDC* pDC)
{
      return CWnd::OnEraseBkgnd(pDC);
}

// this is the main iteration loop where calculations occur
void CSedSim::mainIteration(void)
{
      //increment iteration counter
      iterationNumber++;

      //determine the mass of the particle
      double pi=3.1415926535;
      double
particleVolume=(pi/3)*((particleDiameter/2)*(particleDiameter/2)*(particleDia
meter/2));
      double particleMass=particleVolume*particleDensity;
      particleMass/=1000000; /* convert to kilograms */

      //to determine sheer heights in the case of turbulence,
      //work left to right
      for(int i=0; i<bedWidth; i++)
      {
            if(!sheerTurbulence)
            {
                  //Turbulence is off, set sheer height above each column
```

```
                  sandbed[i].sheerHeight=sandbed[i].columnHeight+1;
            }
            else
            {
                  sandbed[i].sheerHeight=sandbed[i].columnHeight+1;
                  //sheer height is previous height * the downdraft
coefficient
                  if(i>0)
                  {
                        if(sandbed[i-1].sheerHeight-sandbed[i].sheerHeight>0)
                    sandbed[i].sheerHeight=sandbed[i-1].sheerHeight-
((sandbed[i-1].sheerHeight-
sandbed[i].sheerHeight)*(downDraftCoefficient/100));
                        else

      sandbed[i].sheerHeight=sandbed[i].columnHeight+1;
                  }
                  else
                  {
                        if(Wrapping)
                        {
                              if(sandbed[bedWidth-1].sheerHeight-
sandbed[0].sheerHeight>0)
                        sandbed[0].sheerHeight=sandbed[bedWidth-
1].sheerHeight-((sandbed[bedWidth-1].sheerHeight-
sandbed[0].sheerHeight)*(downDraftCoefficient/100));
                              else

      sandbed[0].sheerHeight=sandbed[0].columnHeight+1;
                        }
                        else

      sandbed[0].sheerHeight=sandbed[0].columnHeight+1;
                  }
            }
      }

      //now it is time to move the sand
      //we work backwards from the right of the bed
      for(int i=bedWidth-1; i>=0; i--)
      {
            //determine which index flying sand will potentially land
            int nextIndex;
            if(i==bedWidth-1 && Wrapping) nextIndex=0;
            else nextIndex=i+1;
            int prevIndex;
            if(i==0 && Wrapping) prevIndex=bedWidth-1;
            else prevIndex=i-1;

            //is there an open occupancy immediately to the right?
            if(sandbed[i].columnHeight > sandbed[nextIndex].columnHeight)
            {
                  //check to see if there is a sheer force
                  if(sandbed[i].sheerHeight-sandbed[i].columnHeight <=1)
                  {
                        /*There is a sheer force, impart momentum*/
```

```
                        //Determine the acceleration particle undergoes
                        double forceFriction =
((forceGravity/(iterationLength*1000))
*particleMass)/(interParticleViscosity/100);
                        double acceleration = ((appliedViscousForce *
sheerSpeed)+forceFriction) / particleMass;

      sandbed[i].particleVelocity+=(acceleration/(float(iterationLength)/1000
));

                        /*Check to see if the particle velocity is sufficient
to move to next cell*/
                        if(sandbed[i].particleVelocity >=
particleDiameter/1000)
                        {
                              sandbed[i].particleVelocity=0; /* possible
upgrade: instead of reseting velocity to 0, consider reducing it by a
collision force */
                              sandbed[i].columnHeight--;
                              sandbed[nextIndex].columnHeight++;
                        }
                  }
            }
            else /* a particle is occupying the immediate right, consider
rolling */
            {
                  if(rolling)
                  {
                        //check to see if there is a sheer force
                        if(sandbed[i].sheerHeight-sandbed[i].columnHeight
<=1)
                        {
                              /*There is a sheer force, impart momentum*/

                              //Determine the acceleration particle undergoes
                              double forceNormalFriction =
((forceGravity/(iterationLength*1000))
*particleMass)/(interParticleViscosity/100);
                              double forceRollingFriction = -
((appliedViscousForce*sheerSpeed)*particleMass)/(interParticleViscosity/100);
                              double
forceGravityPull=((forceGravity/(iterationLength*1000))*particleMass);

                              double
counterForcePerDiscreteParticle=forceNormalFriction+forceRollingFriction+forc
eGravityPull;
                              double totalCounterForce =
((sandbed[nextIndex].columnHeight-
sandbed[i].columnHeight)+1)*counterForcePerDiscreteParticle;

                              //the *2 is a tweak
                              double acceleration = ((appliedViscousForce *
sheerSpeed)+totalCounterForce) / (particleMass*2);


      sandbed[i].particleVelocity+=(acceleration/(float(iterationLength)/1000
));
```

```
                              /*Check to see if the particle velocity is
sufficient to move to next cell*/
                              if(sandbed[i].particleVelocity >=
particleDiameter/1000)
                              {
                                    sandbed[i].particleVelocity=0; /*
possible upgrade: instead of reseting velocity to 0, consider reducing it by
a collision force */
                                    sandbed[i].columnHeight--;
                                    sandbed[nextIndex].columnHeight++;
                              }
                        }
                  }
            }

            //Avalanching
            if(avalanching)
            {
                  //calculate the height of the next dropoff, if any
                  double thetar, thetal;
                  do
                  {
                        thetar=bedHeight;
                        thetal=bedHeight;
                        int dropoffr=sandbed[i].columnHeight-
sandbed[nextIndex].columnHeight;
                        if(dropoffr>0)
                        {
                              //There is a dropoff, calculate theta
                              thetar = atan( 1./double(dropoffr) );
                              if(thetar < avalancheTheta)
                              {
                                    sandbed[i].columnHeight--;
                                    sandbed[nextIndex].columnHeight++;
                              }
                        }
                        int dropoffl=sandbed[i].columnHeight-
sandbed[prevIndex].columnHeight;
                        if(dropoffl>0)
                        {
                              thetal = atan(1./double(dropoffl) );
                              if(thetal < avalancheTheta)
                              {
                                    sandbed[i].columnHeight--;
                                    sandbed[prevIndex].columnHeight++;
                              }
                        }
                  } while(thetar<avalancheTheta || thetal<avalancheTheta);
            }
      }

    //Notify the GDI that the simulation window needs repainting
      InvalidateRect(NULL, true);
}

void CSedSim::OnLButtonDown(UINT nFlags, CPoint point)
```

```
{
	//User has clicked the simulation window, indicating
	//it is time to reiterate
	mainIteration();
	CWnd::OnLButtonDown(nFlags, point);
}

void CSedSim::OnTimer(UINT nIDEvent)
{
	if(autoIterate)
		if(nIDEvent==100)
			mainIteration();
	CWnd::OnTimer(nIDEvent);
}

#pragma once
#include "Stdafx.h"
#include "vector"
#include "afxwin.h"
using namespace std ;

// CSedSim

class CSedSim : public CWnd
{
	DECLARE_DYNAMIC(CSedSim)

public:
	CSedSim();
	virtual ~CSedSim();

	struct Particle
	{
		double particleVelocity;
		double sheerHeight;
		int columnHeight;
	};
	typedef vector<Particle> CHARVECTOR;
	CHARVECTOR sandbed;

protected:
	DECLARE_MESSAGE_MAP()
public:

	// Simulation window registration and creation
	void LaunchWindow(void)
	{
		InitSimulation();
		LPCTSTR wndClass=
			AfxRegisterWndClass(CS_HREDRAW | CS_OWNDC | CS_VREDRAW,
	0, 0, AfxGetApp()->LoadIcon(IDR_MAINFRAME));
		this->CreateEx(0, wndClass, "Simulation Window", WS_BORDER |
WS_CAPTION | WS_MAXIMIZE | WS_POPUP | WS_SYSMENU | WS_VISIBLE, 100, 100, 300,
300, AfxGetMainWnd()->m_hWnd, NULL, NULL);
		//start timer if autoiterate function is selected
		if(autoIterate)
```

```
                SetTimer(100, autoIterateLength, NULL);
        }
        // the maximum height of the sand bed in discrete units (grains)
        int bedHeight;
        // the width of the bed in discrete units (grains)
        int bedWidth;
        // initializes the simulation, and allocates memory required for
current parameters
        void InitSimulation(void);
        // Initial depth of the sandbed
        int bedDepth;
        afx_msg void OnPaint();
        afx_msg BOOL OnEraseBkgnd(CDC* pDC);
        // Designates whether or not sand forced off the right of the bed
returns leftmost
        bool Wrapping;
        // indicates whether r not to generate waveform during InitSimulation
        bool generateSinewave;
        // Magnitude of the sinewave to generate, if selected
        int SinewaveMagnitude;
        // The number of sinewave peaks, if selected
        int SinewavePeaks;
        // defines the length of time that each iteration is split into
        int iterationLength;
        bool autoIterate;
        // Speed (meters per second) of sheering medium
        double sheerSpeed;
        // newtons per meter per second per particle
        double appliedViscousForce;
        // indicates whether or not to incorporate simple turbulence into the
calculations
        bool sheerTurbulence;
        // percent of particle height fluid flows down bed slopes
        double downDraftCoefficient;
        // how often to automatically iterate the simulation in real time
        int autoIterateLength;
protected:
        // this is the main iteration loop where calculations occur
        void mainIteration(void);
public:
        afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
        // the particle diameter in millimeters
        double particleDiameter;
        // density of the particle material in milligrams per cubic millimeter
        double particleDensity;
        // Inter-Particle Viscosity % (0...100)
        double interParticleViscosity;

        // converts a double precision floating point number to a CString
        static CString doubleToString(double number)
        {
                int decimal, sign;
                CString ret=_fcvt(number, 10, &decimal, &sign);
                if(decimal>=0) ret.Insert(decimal, '.');
                else
                {
                        for(int i=0; i<abs(decimal); i++)
```

```
                        ret.Insert(0, '0');
                  ret.Insert(0, '.');
            }
            if(sign!=0) ret.Insert(0, '-');
            return ret;
      }
protected:
      // used by the GDI for outputting simluation settings
      CFont infoFont;
      // tracks how many iterations have elapsed
      int iterationNumber;
public:
      afx_msg void OnTimer(UINT nIDEvent);
      // indicates whether or not to incorporate avalanching
      bool avalanching;
      // the angle at which avalanching occurs
      double avalancheTheta;
      // indicates whether or not grains can role
      bool rolling;
      // force of gravity in m/s/s
      double forceGravity;
};


// sedimentDlg.cpp : implementation file

//updated May 02 - Matt Jallo
//updated April 25 - Matt Jallo

#include "stdafx.h"
#include "sediment.h"
#include "sedimentDlg.h"
#include "sedsim.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif


// CAboutDlg dialog used for App About

class CAboutDlg : public CDialog
{
public:
      CAboutDlg();

// Dialog Data
      enum { IDD = IDD_ABOUTBOX };

      protected:
      virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support

// Implementation
protected:
      DECLARE_MESSAGE_MAP()
};
```

```
CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
	CDialog::DoDataExchange(pDX);
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
END_MESSAGE_MAP()


// CsedimentDlg dialog



CsedimentDlg::CsedimentDlg(CWnd* pParent /*=NULL*/)
	: CDialog(CsedimentDlg::IDD, pParent)
{
	m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}

void CsedimentDlg::DoDataExchange(CDataExchange* pDX)
{
	CDialog::DoDataExchange(pDX);

	SetDlgItemInt(IDC_BEDWIDTH, SedSim.bedWidth, true);
	SetDlgItemInt(IDC_BASEHEIGHT, SedSim.bedDepth, true);

	CheckDlgButton(IDC_WRAPPING, SedSim.Wrapping);
	CheckDlgButton(IDC_SINEWAVE, SedSim.generateSinewave);

	SetDlgItemInt(IDC_MAGNITUDE, SedSim.SinewaveMagnitude, true);
	SetDlgItemInt(IDC_PEAKS, SedSim.SinewavePeaks, true);

	SetDlgItemText(IDC_ITERATIONLENGTH,
CSedSim::doubleToString(SedSim.iterationLength));
	SetDlgItemText(IDC_AUTOITERATELENGTH,
CSedSim::doubleToString(SedSim.autoIterateLength));
	CheckDlgButton(IDC_AUTOITERATE, SedSim.autoIterate);

	SetDlgItemText(IDC_SHEERSPEED,
CSedSim::doubleToString(SedSim.sheerSpeed));
	SetDlgItemText(IDC_VISCOUSFORCE,
CSedSim::doubleToString(SedSim.appliedViscousForce));
	CheckDlgButton(IDC_TURB, SedSim.sheerTurbulence);
	SetDlgItemText(IDC_TURBCOEFF,
CSedSim::doubleToString(SedSim.downDraftCoefficient));

	SetDlgItemText(IDC_PARTICLEDIAMETER,
CSedSim::doubleToString(SedSim.particleDiameter));
	SetDlgItemText(IDC_PARTICLEDENSITY,
CSedSim::doubleToString(SedSim.particleDensity));
	SetDlgItemText(IDC_PARTICLEVISCOSITY,
CSedSim::doubleToString(SedSim.interParticleViscosity));
```

```
	CheckDlgButton(IDC_AVALANCHE, SedSim.avalanching);
	SetDlgItemText(IDC_AVALANCHETHETA,
CSedSim::doubleToString(SedSim.avalancheTheta));
	CheckDlgButton(IDC_ROLLING, SedSim.rolling);

	SetDlgItemText(IDC_FG, CSedSim::doubleToString(SedSim.forceGravity));
}

BEGIN_MESSAGE_MAP(CsedimentDlg, CDialog)
	ON_WM_SYSCOMMAND()
	ON_WM_PAINT()
	ON_WM_QUERYDRAGICON()
	//}}AFX_MSG_MAP
	ON_BN_CLICKED(IDC_CMDBEGIN, OnBnClickedCmdbegin)
	ON_BN_CLICKED(IDC_EXIT, OnBnClickedExit)
END_MESSAGE_MAP()


// CsedimentDlg message handlers

BOOL CsedimentDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// Add "About..." menu item to system menu.

	// IDM_ABOUTBOX must be in the system command range.
	ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
	ASSERT(IDM_ABOUTBOX < 0xF000);

	CMenu* pSysMenu = GetSystemMenu(FALSE);
	if (pSysMenu != NULL)
	{
		CString strAboutMenu;
		strAboutMenu.LoadString(IDS_ABOUTBOX);
		if (!strAboutMenu.IsEmpty())
		{
			pSysMenu->AppendMenu(MF_SEPARATOR);
			pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX,
strAboutMenu);
		}
	}

	// Set the icon for this dialog.  The framework does this automatically
	//  when the application's main window is not a dialog
	SetIcon(m_hIcon, TRUE);			// Set big icon
	SetIcon(m_hIcon, FALSE);		// Set small icon

	// TODO: Add extra initialization here

	return TRUE;  // return TRUE  unless you set the focus to a control
}

void CsedimentDlg::OnSysCommand(UINT nID, LPARAM lParam)
{
	if ((nID & 0xFFF0) == IDM_ABOUTBOX)
	{
```

```
			CAboutDlg dlgAbout;
			dlgAbout.DoModal();
		}
		else
		{
			CDialog::OnSysCommand(nID, lParam);
		}
}

// If you add a minimize button to your dialog, you will need the code below
//  to draw the icon.  For MFC applications using the document/view model,
//  this is automatically done for you by the framework.

void CsedimentDlg::OnPaint()
{
		if (IsIconic())
		{
			CPaintDC dc(this); // device context for painting

			SendMessage(WM_ICONERASEBKGND,
reinterpret_cast<WPARAM>(dc.GetSafeHdc()), 0);

			// Center icon in client rectangle
			int cxIcon = GetSystemMetrics(SM_CXICON);
			int cyIcon = GetSystemMetrics(SM_CYICON);
			CRect rect;
			GetClientRect(&rect);
			int x = (rect.Width() - cxIcon + 1) / 2;
			int y = (rect.Height() - cyIcon + 1) / 2;

			// Draw the icon
			dc.DrawIcon(x, y, m_hIcon);
		}
		else
		{
			CDialog::OnPaint();
		}
}

// The system calls this function to obtain the cursor to display while the
user drags
//  the minimized window.
HCURSOR CsedimentDlg::OnQueryDragIcon()
{
		return static_cast<HCURSOR>(m_hIcon);
}

void CsedimentDlg::OnBnClickedCmdbegin()
{
		SedSim.bedWidth=GetDlgItemInt(IDC_BEDWIDTH, NULL, FALSE);
		SedSim.bedHeight=SedSim.bedWidth * .7;
		SedSim.bedDepth=GetDlgItemInt(IDC_BASEHEIGHT, NULL, FALSE);

		SedSim.Wrapping=IsDlgButtonChecked(IDC_WRAPPING);
		SedSim.generateSinewave=IsDlgButtonChecked(IDC_SINEWAVE);

		SedSim.SinewaveMagnitude=GetDlgItemInt(IDC_MAGNITUDE, NULL, FALSE);
```

```
	SedSim.SinewavePeaks=GetDlgItemInt(IDC_PEAKS, NULL, FALSE);

	CString temp;
	GetDlgItemText(IDC_ITERATIONLENGTH, temp);
	SedSim.iterationLength=atof(temp);
	GetDlgItemText(IDC_AUTOITERATELENGTH, temp);
	SedSim.autoIterateLength=atof(temp);
	SedSim.autoIterate=IsDlgButtonChecked(IDC_AUTOITERATE);

	GetDlgItemText(IDC_SHEERSPEED, temp);
	SedSim.sheerSpeed=atof(temp);
	GetDlgItemText(IDC_VISCOUSFORCE, temp);
	SedSim.appliedViscousForce=atof(temp);
	SedSim.sheerTurbulence=IsDlgButtonChecked(IDC_TURB);
	GetDlgItemText(IDC_TURBCOEFF, temp);
	SedSim.downDraftCoefficient=atof(temp);

	GetDlgItemText(IDC_PARTICLEDIAMETER, temp);
	SedSim.particleDiameter=atof(temp);
	GetDlgItemText(IDC_PARTICLEDENSITY, temp);
	SedSim.particleDensity=atof(temp);
	GetDlgItemText(IDC_PARTICLEVISCOSITY, temp);
	SedSim.interParticleViscosity=atof(temp);

	SedSim.avalanching=IsDlgButtonChecked(IDC_AVALANCHE);
	GetDlgItemText(IDC_AVALANCHETHETA, temp);
	SedSim.avalancheTheta=atof(temp);
	SedSim.rolling=IsDlgButtonChecked(IDC_ROLLING);

	GetDlgItemText(IDC_FG, temp);
	SedSim.forceGravity=atof(temp);

	SedSim.LaunchWindow();
}

void CsedimentDlg::OnBnClickedExit()
{
	exit(0);
}

// sedimentDlg.h : header file
//

#pragma once
#include "sedsim.h"

// CsedimentDlg dialog
class CsedimentDlg : public CDialog
{
// Construction
public:
	CsedimentDlg(CWnd* pParent = NULL); // standard constructor

// Dialog Data
	enum { IDD = IDD_SEDIMENT_DIALOG };
```

```
	protected:
	virtual void DoDataExchange(CDataExchange* pDX);	// DDX/DDV
support


// Implementation
protected:
	HICON m_hIcon;
	CSedSim SedSim; //Simulator Class & Window

	// Generated message map functions
	virtual BOOL OnInitDialog();
	afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
	afx_msg void OnPaint();
	afx_msg HCURSOR OnQueryDragIcon();
	DECLARE_MESSAGE_MAP()
public:
	afx_msg void OnBnClickedCmdbegin();
	afx_msg void OnBnClickedExit();
};
```