Download the file NewtonTutorial.zip from the Newton website and unpack it anywhere in your windows machine, if you have not done so.

These tutorials are made for Win32, compiled with visual Studio 2003 and using Simple DirectMedia Layer (SDL) for Graphics, Sound and Input. All of the elements in the tutorials are cross platform, therefore it should be simple to build then on other platform like win64, Linux, and Mac OS. However this is a basic tutorial and we will not experiment with that right now.

After unpacking, go to the folder .../NewtonToturials and open the visual studio solution NewtonTutorials.sln

In the solution explorer right click on project Tutorial\_101\_GettingStarted and select Set as Startup Project, then click Rebuild Solution. All the dependencies are included in the download; therefore all projects should build with zero warning and zero errors.

Open up file Main.cpp

The first lines of code

#include <stdafx.h>

#include <dVector.h>

#include "OpenGlUtil.h"

#include "CreateScene.h"

#define USE\_VISUAL\_DEBUGGER

#define DEMO\_FPS\_IN\_MICROSECUNDS (int (1000000.0f/100.0f))

static int g\_currentTime;

static int g\_physicTime;

static int g\_timeAccumulator = DEMO\_FPS\_IN\_MICROSECUNDS;

static void\* g\_newtonDebugger;

static NewtonWorld\* g\_world;

static SceneManager\* g\_sceneManager;

static void ShutDown ();

static void\* AllocMemory (int sizeInBytes);

static void FreeMemory (void \*ptr, int sizeInBytes);

static void AdvanceSimulation (int timeInMilisecunds);

#pragma warning (disable: 4100) //unreferenced formal parameter

#pragma warning (disable: 4702) //unreachable code

Are just the necessary included files, constant and variables necessary for the project to work.

The first Line in function main

\_CrtSetDbgFlag(\_CRTDBG\_LEAK\_CHECK\_DF|\_CrtSetDbgFlag(\_CRTDBG\_LEAK\_CHECK\_DF));

Is a operating system memory leak check handler that indicates if some memory allocation was not freed on exit.

The following lines:

// initialize graphics system

if (!InitGraphicsSystem (800, 600)) {

exit (0);

}

Initializes a proper SDL Graphic System that will be use to visualize the physics scene. You are welcome to step into the function in debug mode to see how it is implemented, I will not spend time explaining the details, I will only say the code was taken almost verbatim from the SDL examples.

Next

// set a callback for destroying the world at termination

atexit (ShutDown);

A termination function is set, which will be called when the user decides to terminate the program. For most Graphics engines with a gracious ways to shutting down this is not necessary, but graphics libraries like SDL and GLUT do not terminate with grace and all termination code has to be added during forced exit termination. The termination function looks like this:

// On termination, the application must destroy the Newton world

void ShutDown ()

{

#ifdef USE\_VISUAL\_DEBUGGER

// destroy the debugger server

NewtonDebuggerDestroyServer (g\_newtonDebugger);

#endif

// destroy all rigid bodies, this is not necessary because Newton Destroy world will also destroy all bodies

// but if you want to change level and restart you can call this function.

NewtonDestroyAllBodies (g\_world);

// finally destroy the newton world

NewtonDestroy (g\_world);

// now we need to destroy the Graphics entities

Delete g\_sceneManager;

}

Essentially this destroys all objects created by the application in reverse order.

Next

</prev>

// now populate the world with Graphic and physical entities

CreateScene(g\_world, g\_sceneManager);

<prev>

Every Graphics engine has a way to encapsulate the various graphics components of a visual object into some kind of structure, in this tutorials will call those object Entities.

In turns these Entities are encapsulated into an even higher level object called Scene Manager. A Scene Manager is an object that encapsulates most of the functionality of graphics engine.

It manages the creation, destruction, user interface and motion of any graphic object. For the purposes of these tutorials the scene manager is a class that looks like this:

<prev>

class SceneManager

{

public:

SceneManager(void);

virtual ~SceneManager(void);

void CreateScene ();

Entity\* CreateEntity();

void SetIntepalationParam(dFloat param);

void Render ();

protected:

int m\_entCount;

dFloat m\_intepolationParam;

Entity\* m\_entityArray[MAX\_ENTITY\_COUNT];

};

</prev>

Basically is a very simple flat array of Entities.

The following lines, initializes the default settings for a Newton world

// create the Newton World

g\_world = NewtonCreate (AllocMemory, FreeMemory);

// configure the Newton world to use iterative solve mode 0

// this is the most efficient but the less accurate mode

NewtonSetSolverModel (g\_world, 1);

// use the standard x87 floating point model

NewtonSetPlatformArchitecture (g\_world, 0);

// set a fix world size

dVector minSize (-500.0f, -500.0f, -500.0f);

dVector maxSize ( 500.0f, 500.0f, 500.0f);

NewtonSetWorldSize (g\_world, &minSize[0], &maxSize[0]);

The function NewtoCreate passes the pointer to memory allocation functions that will take care of all the engine memory allocation. These functions are optional, you can pass NULL but you cannot pass one or the other, if you decided to control memory allocation then you must write a memory allocation and a memory free function. At the simplest level these function should look like this:

// this is the callback for allocating Newton memory

void\* AllocMemory (int sizeInBytes)

{

return malloc (sizeInBytes);

}

// this is the callback for freeing Newton Memory

void FreeMemory (void \*ptr, int sizeInBytes)

{

free (ptr);

}

The following lines show a new feature in SDK 2.00

// initialize Newton Visual Debugger

#ifdef USE\_VISUAL\_DEBUGGER

g\_newtonDebugger = NewtonDebuggerCreateServer ();

#endif

It is the visual debugger, at the time of the writing these tutorial the Visual Debugger Listener is not yet fully implemented.

Next

// now populate the world with Graphic and physical entities

CreateScene (g\_world, g\_sceneManager);

For these tutorials the Entity class looks like this:

class Entity

{

public:

struct SubMesh

{

unsigned m\_textureHandle;

int m\_indexCount;

unsigned short\* m\_indexArray;

};

Entity(void);

virtual ~Entity(void);

void LoadMesh (const char\* name);

virtual void Render (dFloat interpolationParam);

// these are the element to represent the position and orientation state of a graphics object in the world

dMatrix m\_matrix; // current interpolated visual matrix

dVector m\_curPosition; // position one physics simulation step in the future

dVector m\_prevPosition; // position at the current physics simulation step

dQuaternion m\_curRotation; // rotation one physics simulation step in the future

dQuaternion m\_prevRotation; // rotation at the current physics simulation step

// These are the elements for a simple vertex list index list graphics object.

int m\_subMeshCount;

int m\_vertexCount;

dFloat \*m\_uv;

dFloat \*m\_vertex;

dFloat \*m\_normal;

SubMesh \*m\_subMeshes;

};

This class in not part of Newton and it is not part of any Graphics engine, It is just an easy way to group graphics functionality in this tutorial, please do not use this class as part of the physics engine.

This is where the application will create and save all graphics scene,

For this simple tutorial we create a simple scene with a big box as floor and two other small boxes floating on the air.

</prev>

void CreateScene (NewtonWorld\* world, SceneManager\* sceneManager)

{

Entity\* floor;

Entity\* smilly;

Entity\* frowny;

// Create a large body to be the floor

floor = sceneManager->CreateEntity();

floor->LoadMesh ("FloorBox.dat");

// add some visual entities.

smilly = sceneManager->CreateEntity();

smilly->LoadMesh ("Smilly.dat");

smilly->m\_curPosition.m\_y = 10.0f;

smilly->m\_prevPosition = smilly->m\_curPosition;

// add some visual entities.

frowny = sceneManager->CreateEntity();

frowny->LoadMesh ("Frowny.dat");

frowny->m\_curPosition.m\_x = 0.5f;

frowny->m\_curPosition.m\_z = 0.4f;

frowny->m\_curPosition.m\_y = 10.0f;

frowny->m\_prevPosition = frowny->m\_curPosition;

}

<prev>

Subsequences tutorials will go straight to this function to explain the process of adding physics properties to the graphics scene.

Next we add the application main loop.

// run the main application loop until the user terminates the demo

for (;;) {

// Draw the screen.

AdvanceSimulation (GetTimeInMicrosenconds ());

}

This is the simplex main simulation loop we can come up with, it advances the Physical and Graphical simulation by calling AdvanceSimulation with a time elapsed since the last loop in microseconds.

If you lock at function AdvanceSimulation, you will find that this function has two parts.

First parts updates physical simulation at a fix time step and collect the external inputs to the application. It also calls the Visual debugger to visualize the Physical Scene on the external Visual debugger.

int deltaTime;

dFloat fps;

dFloat physicTime;

dFloat interpolationParam;

// get the time step

deltaTime = timeInMilisecunds - g\_currentTime;

g\_currentTime = timeInMilisecunds;

g\_timeAccumulator += deltaTime;

physicTime = 0;

// advance the simulation at a fixed time-step

while (g\_timeAccumulator >= DEMO\_FPS\_IN\_MICROSECUNDS)

{

// sample time before the Update

g\_physicTime = GetTimeInMicrosenconds ();

// do a newton update

NewtonUpdate (g\_world, dFloat (DEMO\_FPS\_IN\_MICROSECUNDS) \* 1000000);

// calculate the time spent in the physical Simulation

g\_physicTime = GetTimeInMicrosenconds () - g\_physicTime;

// Process incoming events.

ProcessEvents ();

// call the visual debugger to show the physics scene

#ifdef USE\_VISUAL\_DEBUGGER

NewtonDebuggerServe (g\_newtonDebugger, g\_world);

#endif

// subtract time from time accumulator

g\_timeAccumulator -= DEMO\_FPS\_IN\_MICROSECUNDS;

physicTime ++;

}

Basically this is a loop designed to run the physics update at a fixed time-step. It keeps track of the delta time accumulated since the last call, and every time the accumulated time is equal or larger to one physical time step, represented by const DEMO\_FPS\_IN\_MICROSECUNDS, the loop issues a call to a ProcessEvent, NewtonUpdate, NewtonDebuggerServe and the time step is subtracted from the time accumulator.

ProcessEvent is the function in charge of collecting the user inputs for controlling the simulation. NewtonUpdate is the Newton Engine main update functions, and is does all of the physical work on all physics entities. NewtonDebuggerServe displays the physical state of the world to the external visual debugger.

The second part on this function updates the Camera, Render the Graphics Scene and display some useful information on the screen.

</prev>

// Clear the color and depth buffers.

glClear (GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT );

// calculate the interpolation parameter for smooth rendering

g\_sceneManager->SetIntepolationParam(dFloat (g\_timeAccumulator) / dFloat(DEMO\_FPS\_IN\_MICROSECUNDS));

// do the rendering here entire Scene

g\_sceneManager->Render ();

// display the frame rate

fps = 1000000.0f / dFloat (deltaTime);

physicTime = g\_physicTime \* dFloat(1.0f / 1000000.0f);

Print (dVector (1.0f, 1.0f, 0.0f, 0.0f), 10, 10, "fps: %6.2f physic time (milliseconds): %6.2f", fps, physicTime);

// show GL rendered Scene

glFlush();

SDL\_GL\_SwapBuffers( );

<prev>

Here the most important part is that the rendering does not happen at the same rate than the physics, because of this, the rendering system maintains two transformation states: the old state and the state one after DEMO\_FPS\_IN\_MICROSECUNDS has passed. The graphics system will calculate the correct transformation for visualization by interpolating the fraction of time elapsed between the two stages. The fraction of elapsed time is calculated in a variable interpolationParam

</prev>

// calculate the interpolation parameter for smooth rendering

g\_sceneManager->SetIntepolationParam(dFloat (g\_timeAccumulator) / dFloat(DEMO\_FPS\_IN\_MICROSECUNDS));

<prev>

This ends this tutorial. We have created a Graphics framework that will serve as our representation of the minimum functionality of must graphics engines.

We integrated the Newton Engine in this hypothetical graphics engine, and believe it or not it is this simple to integrate Newton in to a graphics engine.

Next tutorials will demonstrate how to add Physical properties to each of the entities in of the graphics world.