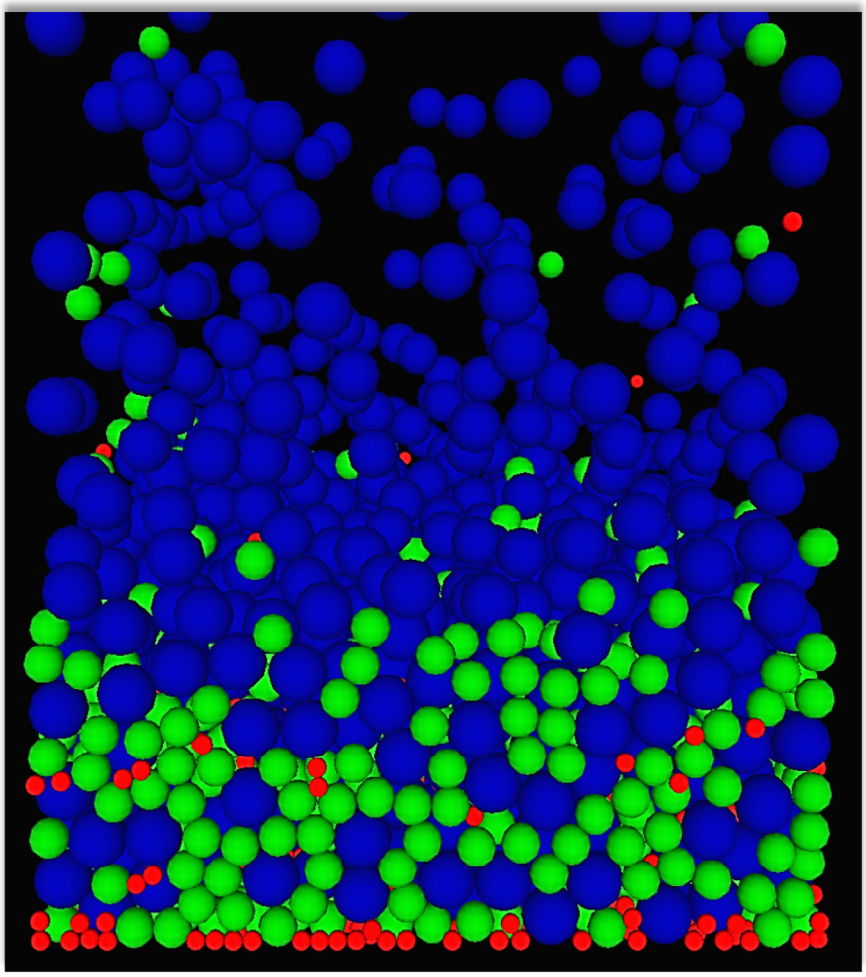
GPP Assignment 1

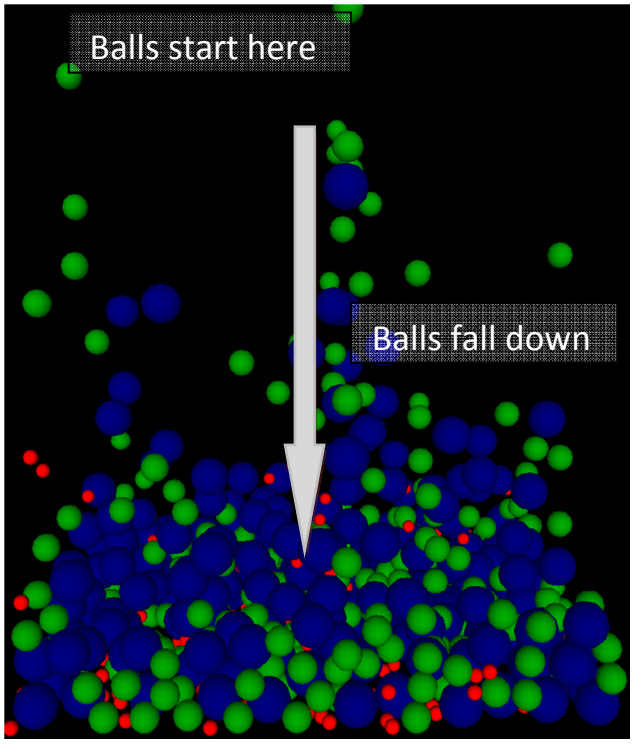
Architecture Specific Optimization

Objective: The objective of this assignment is to build a small interactive application that takes advantage of architecture specific coding optimizations for modern x86 based CPUs. These include (but are not limited to) Cache Coherency, Multi-threading and SIMD instructions. The application should perform some operation that would under normal circumstances not be able to run at suitable speeds without taking advantage of these optimizations. In this case the program will accelerate physics collisions for large object sets in real-time frame rates.

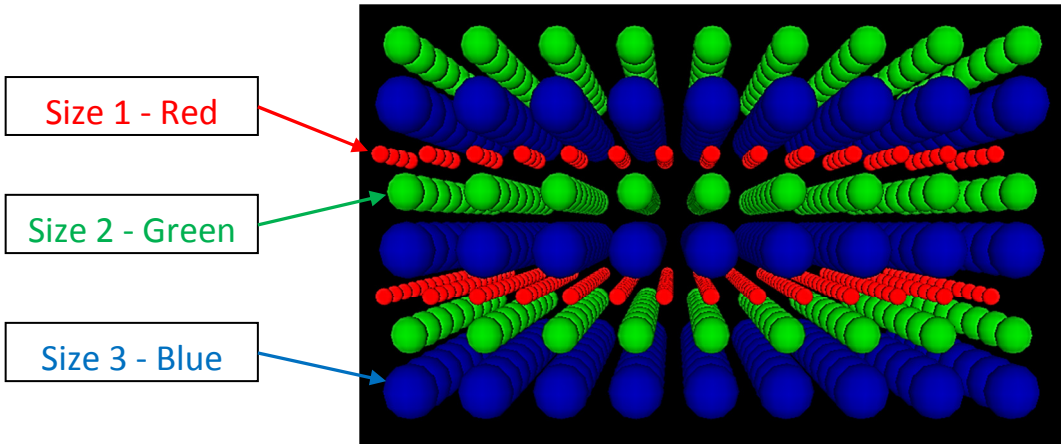


Programming assignment:

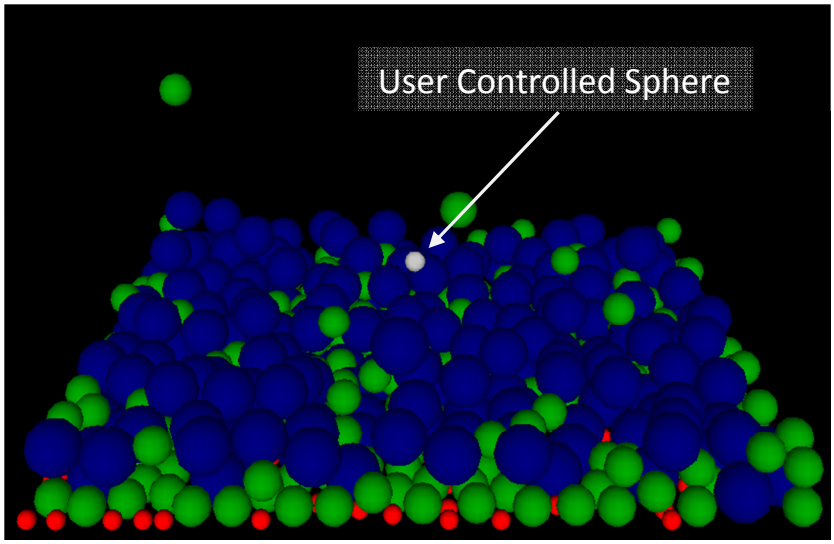
The program should simulate basic collision physics by simulating balls in a fish tank. The program will have a fixed sized invisible cube that functions as the fish tank (40 units wide). At program start up ~1000 balls will start pouring into the fish tank from above (based on acceleration of gravity – 9.81m/s^2). These balls should then accurately collide (inelastic collision with coefficient of restitution~= 0.6) and bounce around (but not out of) the fish tank in a physically correct manor.



There should be 3 different ball sizes (1 unit, 2 unit and 3 unit diameters with 1,2 and 3 unit masses respectively) evenly distributed between each and each ball size should be rendered in a different colour (red, green and blue respectively).



In the centre of the fish tank there should be a small user controllable sphere (1 unit wide, white). This sphere should accept user input and move around (6 degrees of movement) the fish tank accordingly. User input can be from the keyboard or mouse or any combination of those. As the sphere moves around it should accurately collide with any balls it comes in contact with causing them to move around.

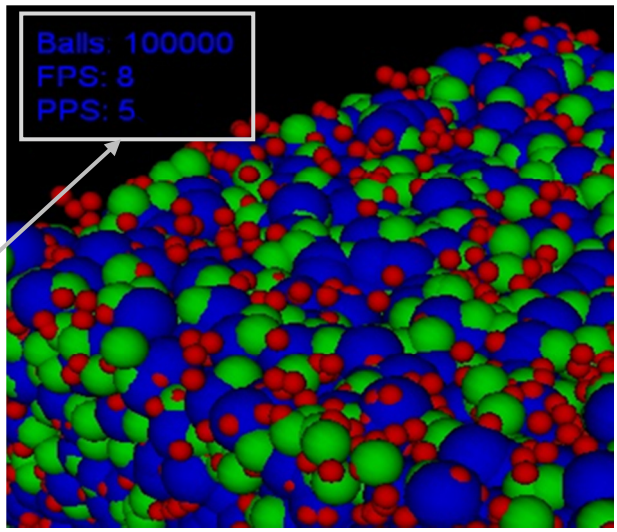


The user should also be able to add more balls to the fish tank through the press of a key (+~1000 balls at a time from the top).

For further testing there should be 2 program variants submitted. These include a basic un-optimized (standard C/C++) program as well as the optimized SSE and MT version. These versions should not be coded separately but instead code should be written so that each function has 2 variants which are selected from at compile time. Marks will be allocated based on the programs performance and demonstrated optimizing knowledge. It is required that you implement an accurate frame rate counter display as well as a counter that shows the number of complete physics runs (i.e. all collisions) that have been processed per second.

Statistics Display

FPS(frames per second)
PPS (Physics passes per second)



In order to visualise the physics collisions in real-time the output of the program needs to be displayed to the screen. This can be performed using any graphics rendering API as the methods used to display the real-time output will not be marked as it is simply used as an analysis tool. However the rendered output must use a window with dimensions 1280x1024. Complex lighting equations are not required as there will be no need for point lighting or shadowing. Only basic Normal shading is necessary (i.e. no flat shading is allowed). Each ball should be rendered to the screen using a polygon mesh with a tessellation factor of 12 in both the *u* and *v* directions (i.e. each sphere must be comprised by 12x12 surface patches).

In addition to the program it is also required to submit appropriate documentation. This includes a user manual and a Technical Report. The user manual should describe the program and most importantly detail the installation process and the specific user input controls. The Technical Report should detail how your program was written; this includes APIs used and the physics algorithm that was implemented. More importantly it should also describe all the optimisation techniques that were implemented in the program and why they were chosen.

Requirements:

- Runs on the lab PCs
- Must be done individually
- Use multi-threading
- Use SSE SIMD instructions
- Must dynamically change threads based on available CPU cores.
- Must provide an optimized and an un-optimized version for comparison.
- Real-time full screen rendered output (can use Direct3D, OpenGL or any other).

What to hand in:

- Zip folder containing 3 folders labelled Documentation, Executable and Source each containing their respective files.
- The Documentation folder should contain a Technical Report and a User Manual.
- The Source folder should contain the full source code for the program.
- The Executable folder should contain 2 executable (optimized and un-optimized version) that are capable of running on the lab machines without modification.

Due Date: Submission => 9:30am Friday 15th April 2010 (week 7)

Demonstration => During Lab Time (Week 11)

Marking Scheme:

Documentation:	10%
Code Quality:	40%
Requirements Coverage:	30%
Demonstration & Performance:	20%

Resources:

- Boost libraries download:
  http://www.boost.org/users/download/
  http://www.boostpro.com/download (precompiled MSVC libs installer)
- Boost Thread online reference:
  http://www.boost.org/doc/libs/1_38_0/doc/html/thread.html
- Boost Thread SDK example/tutorial:
  %BOOST_DIR%\libs\thread\example
  %BOOST_DIR%\libs\thread\tutorial
- MSDN Streaming SIMD Extensions (SSE) reference:
  http://msdn.microsoft.com/en-us/library/t467de55(vs.71).aspx
- Intel AVX Instruction Guide

Plagiarism is not tolerated by the Department – see the handout "Subject Guide Fourth Year Computer Science 2011" found in department office.

Policy on return of assignments: Submitted projects will not be returned to individuals but they are available in the lecturer's office for viewing the marking. Since the projects are very useful for display in job interviews, it is therefore necessary for students to make a facsimile of the entire project submission for their own records and use.

Real-Time Rigid Body Dynamics

Simulating many objects moving about in real-time requires deriving some algorithm that can be used to suitably approximate physically correct interactions (represented as an Ordinary Differential Equation). In this instance we are not concerned with relativistic effects and so we can use standard classical Newtonian physics. The objects involved in the physics system are all spherical and so for the purposes of this program it is irrelevant to take into account Angular Dynamics (rotational effects etc.). Thus the system only needs to take into account linear physics. Lastly the program does not need to support object deformations and so all entities within the program can be represented using Rigid Bodies. Putting this all together results in the program only needing to use a Classical Linear Rigid Body Dynamics system.

As this program is focussed on maximum achievable performance certain approximations are considered acceptable providing they don't cause excessively noticeable errors. As such it is possible to use a 'snap shot' style physics system. In this type of system the full path of an object will not be infinitely traced and instead collision detection checks can be performed only at specific intervals along that path. This can cause errors in a system where the velocities of an object can greatly exceed their size. In this case it would be possible for an object to pass completely through another object between time steps without a collision being detected. In this program however all objects are of a similar size and it is assumed that there velocities will be bound within acceptable limits for the snap shot based approach to work.

Using the above assumptions it is possible to describe a physics object using nothing more than its position and its velocity. Using the snap shot approach requires that each of these attributes be updated at specific points in time. In a real-time system this is represented by the frame rate in which the time difference between rendering frames is used as the snap shot interval. However determining the new velocity and position for each snap shot requires solving an ODE. A basic Euler approach is the simplest method but it adds considerable error to the system. For this project it is instead recommended to use an O(Δt^4) approach such as the "Velocity Verlet" algorithm.

Assuming each objects attributes at a previous time (t) is represented by a position ($p(t)$), velocity ($v(t)$) and an acceleration ($a(t)$). Then using the Velocity Verlet algorithm the new position ($p(t+\Delta t)$) and velocity ($v(t+\Delta t)$) can be determined based on the amount of time that has elapsed (Δt) since t.

$$v\left(t+\frac{1}{2}\Delta t\right) = v(t) + \frac{1}{2}a(t)\Delta t$$

$$p(t+\Delta t) = p(t) + [v\left(t+\frac{1}{2}\Delta t\right)]\Delta t$$

$$a(t+\Delta t) = \frac{F(t+\Delta t, p(t+\Delta t), v(t+\Delta t))}{m}$$

$$v(t+\Delta t) = v\left(t+\frac{1}{2}\Delta t\right) + \frac{1}{2}a(t+\Delta t)\Delta t$$

where m is the mass of the object and $F(t+\Delta t, p(t+\Delta t), v(t+\Delta t))$ is the total force being applied to the object at time $t+\Delta t$.

This algorithm provides a simple yet reasonably accurate method to trace an objects path through a linear dynamics system. However it requires determining the value of $F(t + \Delta t, p(t + \Delta t), v(t + \Delta t))$ or more specifically the net force being applied to an object at a specific point in time. This force is the result of gravity and any collisions that may have occurred during the elapsed time interval Δt. To calculate this constant requires determining what collisions may have occurred and with which object they occurred with. This can be a complex task that is luckily greatly simplified when all objects in the system are spherical. A collision between 2 spheres can be detected by simply determining the distance between their 2 centres and comparing this with their combined radiuses. Finding any collisions that may have occurred for an object simply requires checking that object against all others in the system by using the above collision detection algorithm.

Once a collision (or collisions) has been detected the system now needs a way to determine the force being applied to the involved objects as a result of this collision. This can be solved using standard Kinematics equations; however they are only useful when very few collisions occur involving the same object. In a system when many objects will come to rest on top of others (i.e. such as this one) each object will actually be involved in multiple simultaneous collisions. In order to cater for this situation a good approach is to use a 'mass-spring' system. This approach treats the surface of each object as a collection of outward facing springs. Thus any collision with the surface of that object will apply a force directly along the axis of that spring. That force can be determined using Hooke's Law.

$$F\big(t + \Delta t, p(t + \Delta t)\big) = -kx$$

$$F\big(t + \Delta t, v(t + \Delta t)\big) = -bv_s(t + \Delta t)$$

where k controls the duration of the collision and b acts as the coefficient of restitution for the collision. Also x represents the distance the colliding objects are currently overlapping by.

It is important to note that the velocity $v_s(t + \Delta t)$ is the velocity associated with the collision in the direction of the spring. This means the velocity of the colliding object can't be used directly as it is most likely acting in a direction that is not parallel to the axis of the virtual spring. The axis aligned velocity v_s must be calculated by determining the combined velocity of the colliding objects in the direction of the collision axis.

For this program we want inelastic collisions (i.e. collisions that loose energy over time) so the value of b should be around 0.6. Note that the value of v_s is a directional vector so the resulting force will also be directional. To convert the positional force to a directional force the distance x must be multiplied by a unit vector d that points along the virtual spring's axis. Putting this all together we get an equation for the total force being applied to an object at a specific time interval.

$$F\big(t + \Delta t, p(t + \Delta t), v(t + \Delta t)\big) = \sum -kdx - bv_s(t + \Delta t)$$

This can be combined with gravity to get a total force that can be inserted into the Verlet equation for each object to update its position and velocity values each frame.