Multi-threaded Rendering and Physics Simulation

By Rajshree Chabukswar, Adam T. Lake,
and Mary R. Lee,
Intel® Software Solutions Group (SSG)

February 2005

Learn how to decouple rendering and physical simulation in a multi-threaded environment with a simple physical simulation demonstration. The sample code provided with this paper can either be used as an example or adapted directly into your game engine.

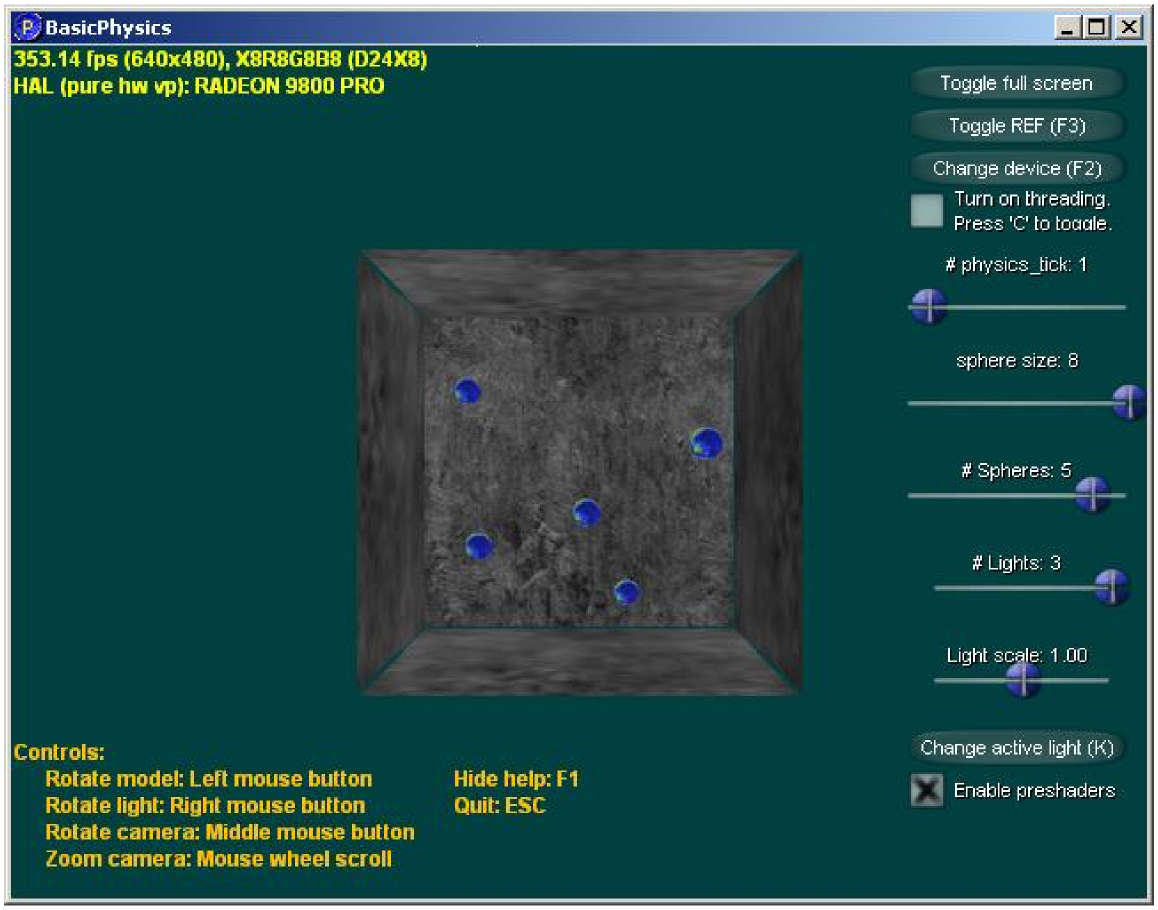
Introduction

This paper demonstrates bouncing spheres within a cube with sphere-sphere and sphere-plane collision detection mechanisms implemented using Newtonian physics. Initially, spheres collide with each other and the walls of the enclosing cube. After collision is detected, collision response code is executed to calculate the new velocity. The user has an option to run either a multi-threaded or single threaded version. Threading is implemented to compute physics equations in a separate thread while rendering takes place in the main thread.

The goal is to showcase how to use separate threads to perform CPU intensive physics computations independent of the actual rendering process. It is beneficial to use multiple threads to perform different tasks since future processor architectures are moving towards multiple cores. Hence, the performance of an application can be increased by having multiple threads work individually to take advantage of available CPU cycles. The paper first discusses the basics of this implementation followed by the physics concepts used in our demo. Future work and reference section is also included.

A number of papers and journals talk about collision detection algorithms. The following articles were used for the initial research and guidelines for this implementation: for sphere-sphere collision detection and response, the algorithms and techniques described in *Simple Bounding-Sphere Collision Detection* by Oleg Doperchouk   and *Pool of Hall Lesson* by Joe Den Heuvel and Miles Jackson provided insight. This paper talks about how to detect collision in 2D and 3D space. Another code-sample on povray.org*, *Rotation around an arbitrary axis* by Tor Olav Kristensen describes how to rotate a vector about an arbitrary axis in space. This algorithm was used while computing the final velocity vector in the sphere-plane collision response code. For this threading implementation, examples from MSDN* were used.

The GUI of the BasicPhysics demo is shown below in Figure 1. Beyond the normal GUI, there is a checkbox to turn on threading. When checked, this separates the rendering and simulation into two threads. The physics tick slider allows the simulation to happen at a faster rate than the rendering. Finally, there are two sliders to adjust the size and number of spheres.


BasicPhysics

353.14 fps (640x480), X8R8G8B8 (D24X8)
HAL (pure hw vp): RADEON 9800 PRO

Controls:
Rotate model: Left mouse button
Rotate light: Right mouse button
Rotate camera: Middle mouse button
Zoom camera: Mouse wheel scroll
Hide help: F1
Quit: ESC

Turn on threading. Press 'C' to toggle.
# physics_tick: 1
sphere size: 8
Spheres: 5
Lights: 3
Light scale: 1.00
Enable preshaders

Figure 1: Physics demo with spheres bouncing inside a cube

A Look at the Equations

This section details the equations used for physics computation of sphere-sphere collision and sphere-plane collision, the display mechanism, and the multi-threading implementations.

Collisions occur when two moving objects or one moving object and one stationary object come in contact with each other. Then this collision affects the velocities of the objects. In this demo, a perfectly elastic collision is assumed where no energy is lost to factors like friction or deformation. Energy lost by one object will be gained by the other.

A chart of how collision detection works in this demo is shown below in Figure 2. First, the current position of the spheres is determined. Each sphere has its position compared against all the other spheres to check for possible collision. Second, each sphere has to check for collision against the six sides of the cube.

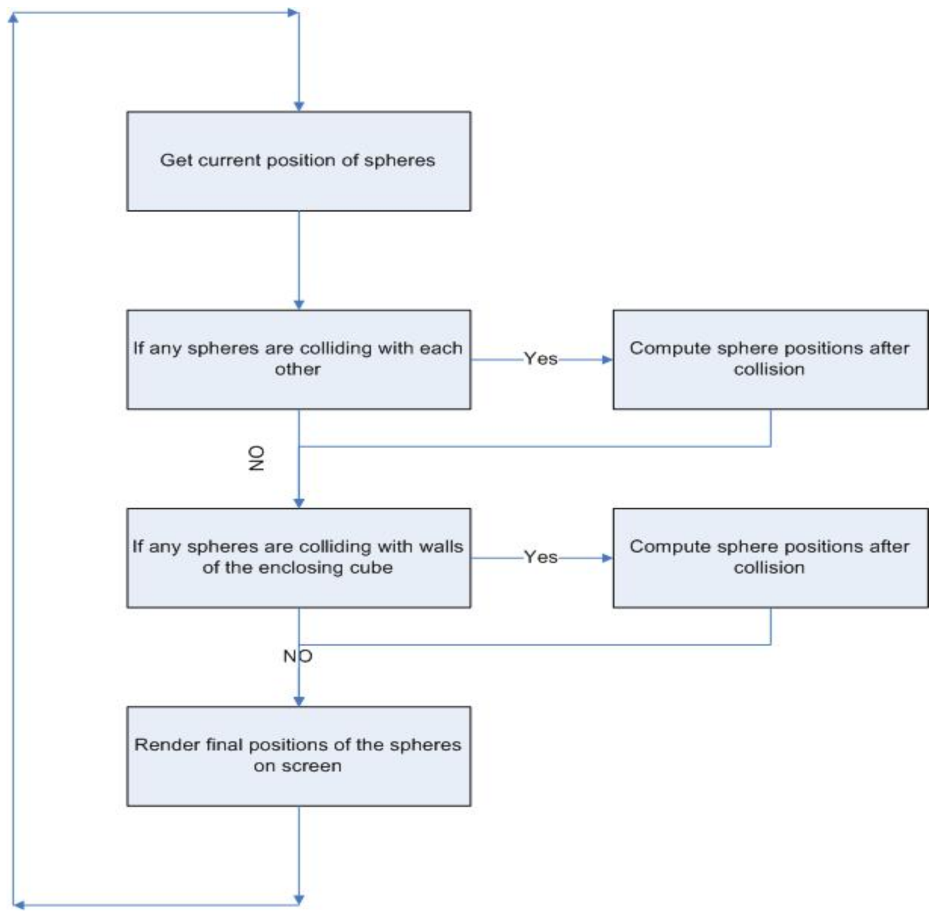

Get current position of spheres
If any spheres are colliding with each other
Yes
Compute sphere positions after collision
NO
If any spheres are colliding with walls of the enclosing cube
Yes
Compute sphere positions after collision
NO
Render final positions of the spheres on screen

Figure 2: Algorithm Overview

Sphere-Sphere collision detection and response

A simple scenario of sphere-sphere collision is demonstrated in Figure 3, which is based on Doperchouk's version. Sphere 1 is in the upper left and sphere 2 is in the lower right. Both have a sphere center and a velocity that is leading to imminent collision.

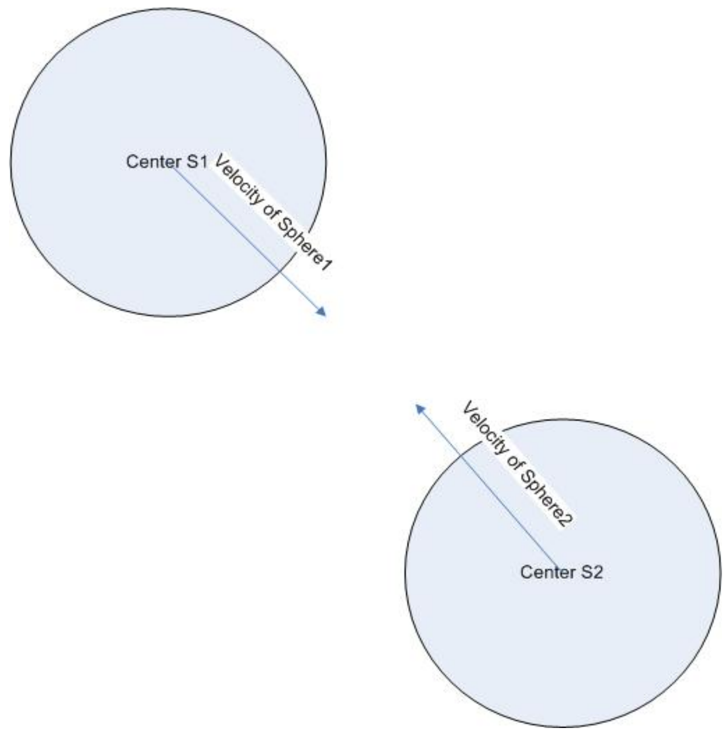


Figure 3: Sphere-Sphere collision detection

To find out if two moving spheres are colliding or not, first we need to compute relative distance between the two spheres. As seen in equation 1, the relative distance between the two spheres is the following:

$$relative\_distance = center\_s1 - center\_s2$$

Equation 1: Relative distance between two spheres

If the sum of the radii is greater than or equal to the relative distance, then the spheres are already colliding and hence we need to resolve the collision.

One of the early tests to see if the spheres are moving away from each other is to examine the relative movement of the spheres. If the relative movement shows that the spheres are going away from each other, the two spheres would not collide and the computation can be skipped.

If the spheres are moving towards each other and not already colliding, it is necessary to find out if the collision can occur before the end of the next frame. For this, relative velocity of the spheres can be determined by equation 2:

$$relative\_velocity = sphere1\_velocity - sphere2\_velocity$$

Equation 2: Relative velocity between two spheres

There is a possibility that a collision will occur before the end of the frame. To check this, compare the relative distance between the spheres with the relative velocity multiplied by the update interval. If the relative distance is smaller, then the fraction of time until collision must be computed with equation 3:

$$time\_fract = \frac{relative\_dis\tan ce}{relative\_velocity}$$

Equation 3: Time fraction at which collision occurs

Next, the remaining time needs to be calculated while resolving the collision. This is used to compute the next positions of the spheres after collision:

$$time\_remain = update\_time - time\_fract$$

Equation 4: Time remaining after collision occurs

Collision response is computed by taking into account the laws of physics, conservation of momentum, and conservation of energy[1]. From figure 3, each sphere has mass m_1 and m_2, while v'_1 and v'_2 are velocity vectors of the spheres after collision. The law of conservation of momentum is as follows:

$$m_1 * v_1 + m_2 * v_2 = m_1 * v'_1 + m_2 * v'_2$$

Equation 5: Conservation of momentum

Any momentum lost by either of the spheres will be gained by the other sphere. If sphere 1 loses momentum, it will be gained by sphere 2. To compute magnitude of R, we can use the law of conservation of energy:

$$\frac{m_1 * v_1^2}{2} + \frac{m_2 * v_2^2}{2} = \frac{m_1 * v_1'^2}{2} + \frac{m_2 * v_2'^2}{2}$$

Equation 6: Conservation of energy

The resulting motion of the two spheres after collision is shown in Figure 4. Each sphere will bounce back from each other with new velocities going away from each other after the collision.

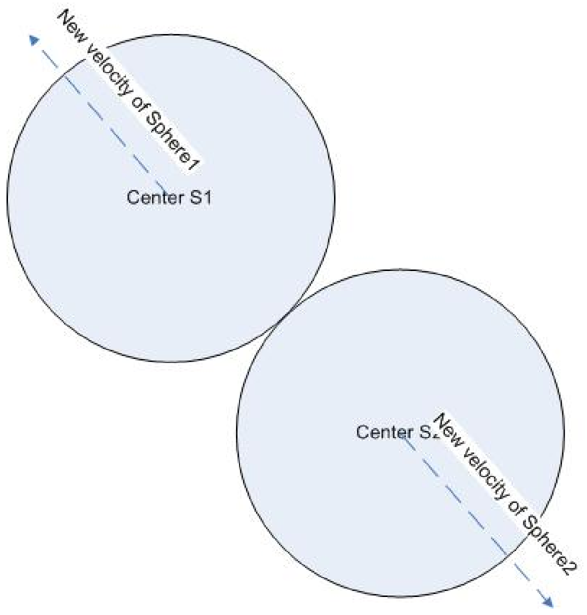


Figure 4: Sphere-Sphere collision response

Sphere-Plane collision detection and response

In checking for sphere-plane collision, each plane of the six cube sides must be checked to find if a sphere will be colliding with a face. The early exit test would be to find a position of the sphere for the next frame. If the position is within bounds of the cube, then the sphere can move without any collision. If this is not the case, then the plane of the cube with which collision will happen must be computed. The dot product of the face normal with the velocity vector must be computed. If the dot product is greater than zero, then the sphere is moving away from that particular face. If the dot product is zero, then the sphere is moving parallel to the face.

However, if the dot product is less than zero, then the distance between the center of sphere and plane is computed. As shown in figure 5, it is the distance between plane and center of sphere S1. If this distance is same as radius of the sphere, then the sphere is already colliding with that particular face and hence we return true.

If the distance between the center of sphere and plane is less than the distance the sphere can travel in one frame update depending on velocity, then the sphere can only move for a fraction of time. The exact time when collision happens in the given time frame is the following:

$$time\_fract = \frac{distance\_from\_center - radius}{velocity}$$

Equation 7: Time fraction at collision

Afterwards, the remaining time is calculated from the update interval, so that the distance traveled in the time remaining after collision can be found:

$$time\_remain = update\_time - time\_fract$$

Equation 8: Time remaining after collision

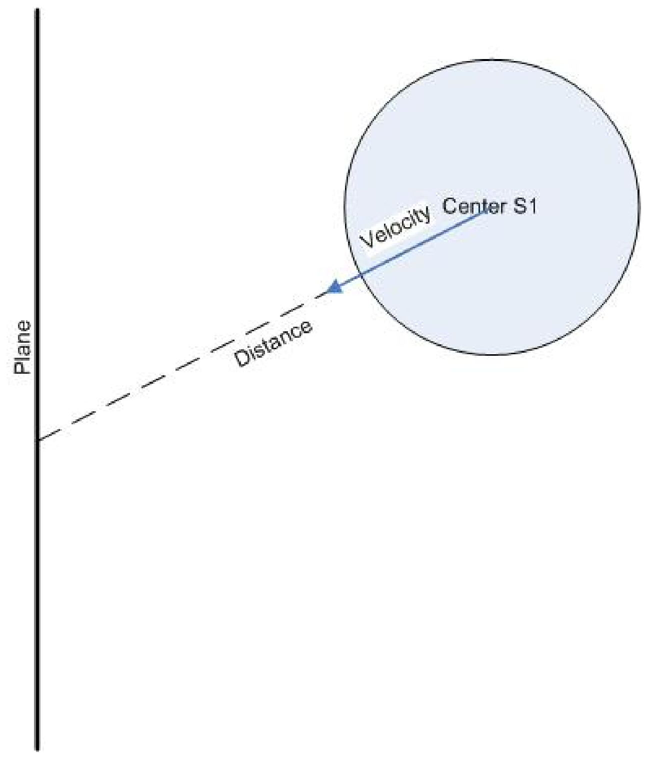


Figure 5: Sphere-Plane collision detection

Once the collision has been detected, it is resolved by finding the new velocity vector of the sphere. First, angle of incidence is computed between the velocity vector and normal of the plane. It is computed as follows:

$$\cos(\alpha) = \frac{v_1 \bullet Normal}{|v_1| * |Normal|}$$

Equation 9: Angle of incidence

To compute the final velocity, the initial velocity vector v_1 is rotated around the normal by 2*α. The resulting vector will be the final velocity vector after the collision.

From figure 6, α will be angle between v_1 and the normal of the plane at the point of contact. The normal of the plane needs to be computed separately.

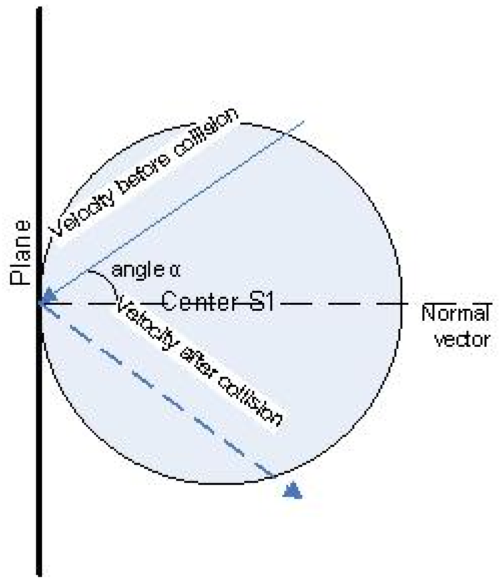


Figure 6: Sphere-Plane collision response

Physics timing and scheduling

As previously mentioned in the previous sections,there are several scenarios where collisions can occur before the end of a frame or a time interval. First, depending on how fast each sphere is moving, the collision might happen during frame update interval. Second, while detecting sphere-plane collision, the initial position of the sphere might be inside the cube. Later at the next frame update, the sphere may pass though the wall depending on the velocity at which it is moving.

This can be resolved by keeping track of distance between the spheres for sphere-sphere collision and distance between sphere and plane for sphere-plane collision. If this distance is less than how much a sphere can move in one frame update interval, the update interval is divided into two parts: the fraction of the time until the collision happens and the remaining time is then used to compute the new position of the sphere after collision happens.

For the physics scheduler, a variable called `physics_tick` is used. This is a simple implementation of the scheduler which keeps track of number of times physics need to be computed before drawing. By default, physics is computed for every frame update and new position is rendered every frame. This is shown in Figure 7, where N=1. The user can choose to render the spheres every N frames where N being the number of times physics is computed before actually rending to screen. The second part of Figure 7 shows where N=3; in this case, the physics simulation is computing three frames ahead and then submitting the information to be rendered.

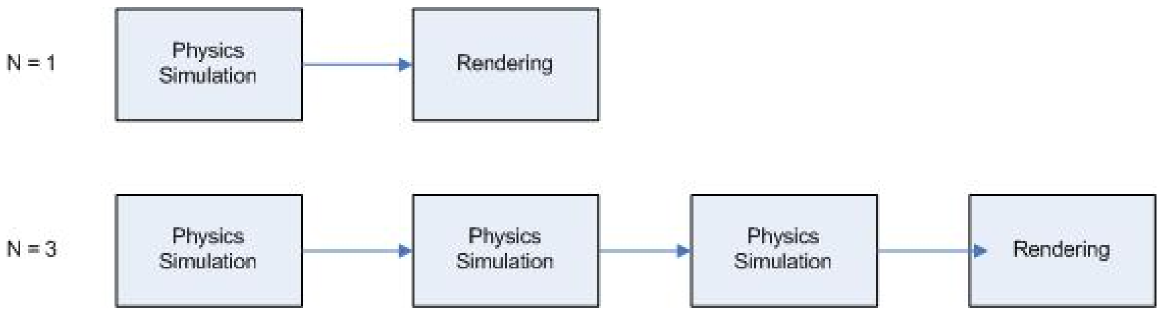


Figure 7: Physics tick diagram

Multi-threading

There are several stages to the graphics pipeline. With multi-core processors on the roadmap, it is possible to separate the physics simulation calculations from the rest of the game such as rendering, AI, logic, and keyboard/mouse input. Additionally, since there are more computing resources available on a single system, it would be advantageous to separate work to farm out to the different processor cores. In this scenario, there is a main thread to handle initialization, rendering, and other aspects while there are additional threads for the physics calculations.

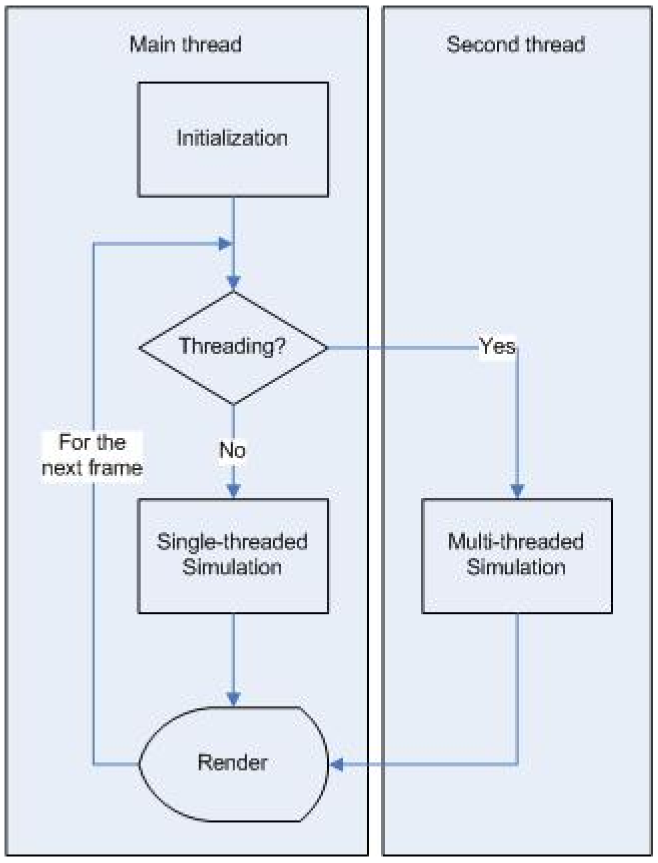


Figure 8: Threading model diagram

Separating the physics simulation work into a different thread than the main thread allows the calculations for sphere-sphere collision detection and sphere-plane collision detection to be done in sync with the frame rate or ahead of the frame rate. In Figure 8, the boxes marked "simulation" is where the collision detection takes place. If threading is not chosen, then all the work is done in a single thread. Otherwise, the physics work wakes up other threads which are waiting on a signal.

Managing multi-threading performance must account for thread overhead and workload limitations. Increasing the number of spheres in a world can consume a great deal of time in calculating each object's location as well as the likelihood of colliding with another sphere or a plane. The work can be partitioned by data separating the objects into different threads or by function splitting the sphere-sphere collision from the sphere-plane collision. Due to load imbalance, there is more work with sphere-sphere collision as the number of spheres increase whereas the sphere-plane collision is limited to checking the bounds of the six sides of a cube for each sphere.

Theory Implementation

We'll now take a closer look at the implementation of the theory discussed in the previous section. This includes how the equations are written, the GUI interaction, and the multi-threading of the physics simulation.

Sphere-Sphere Collision

The code sample below shows how the sphere-sphere collision detection mechanism is implemented.

```
bool IsCollidingSphereToSphere()
{
//relative velocity of two spheres
relative_vel = Sphere2_Vel - Sphere1_Vel;

r = Sphere1_Radius + Sphere2_Radius;

relative_position = Sphere2_Center – Sphere1_Center;

//checks if two spheres are already colliding
if ((Sqr_relative_position - (r*r)) <= 0)
                    return true;

//Few more early escape tests: check if the relative //movement of the two spheres show that they
are moving //away from each other. If yes, the exit returning false

float rel_distance = sqrt(Sqr_relative_pos) - (r);
//This is the case where we need to check if the collision //happens between two update intervals
(frames)
if (rel_distance < (sqrt(Sqr_relative_vel)*update_interval))
{
                    time_fract = rel_distance/sqrt(Sqr_relative_vel);
                    time_remain = update_interval – time_fract;

                    if ((Sqr_relative_pos - (r*r)) <= 0) {
                              return true;
                    }
}

return ((relative_movement * relative_movement - ((Sqr_relative_pos - r*r) - Sqr_relative_vel)) >
0);
}
```

Sphere-Plane Collision

Here the code sample demonstrates how sphere-plane is detected:

```
          bool IsCollidingSphereToPlane()
{
//Early escape test to see if sphere can move without //colliding. If they can, return false



//If not, check the distance between the sphere and the //plane to see if that is 0
                    for (int i=0; i<6; i++)
                    {
                    //dot product of velocity of sphere with normal
//if this is 0, then the velocity vector is parallel to the //plane, hence cannot collide

                              float unit_normal_vel_dot = normal . velocity;
                              if (unit_normal_vel_dot < 0.0f)
                              {
                                        //This computes equation of the plane
                                        float D1 = normal * point_on_plane;
                                        float normal_pos_dot = normal . center_S;

//Compute distance from center of sphere to //plane
float distance = (D1-normal_pos_dot)/unit_normal_vel_dot;

//This if handles cases if sphere is already //touching the box
if (distance <= radius1)
                                        {
```

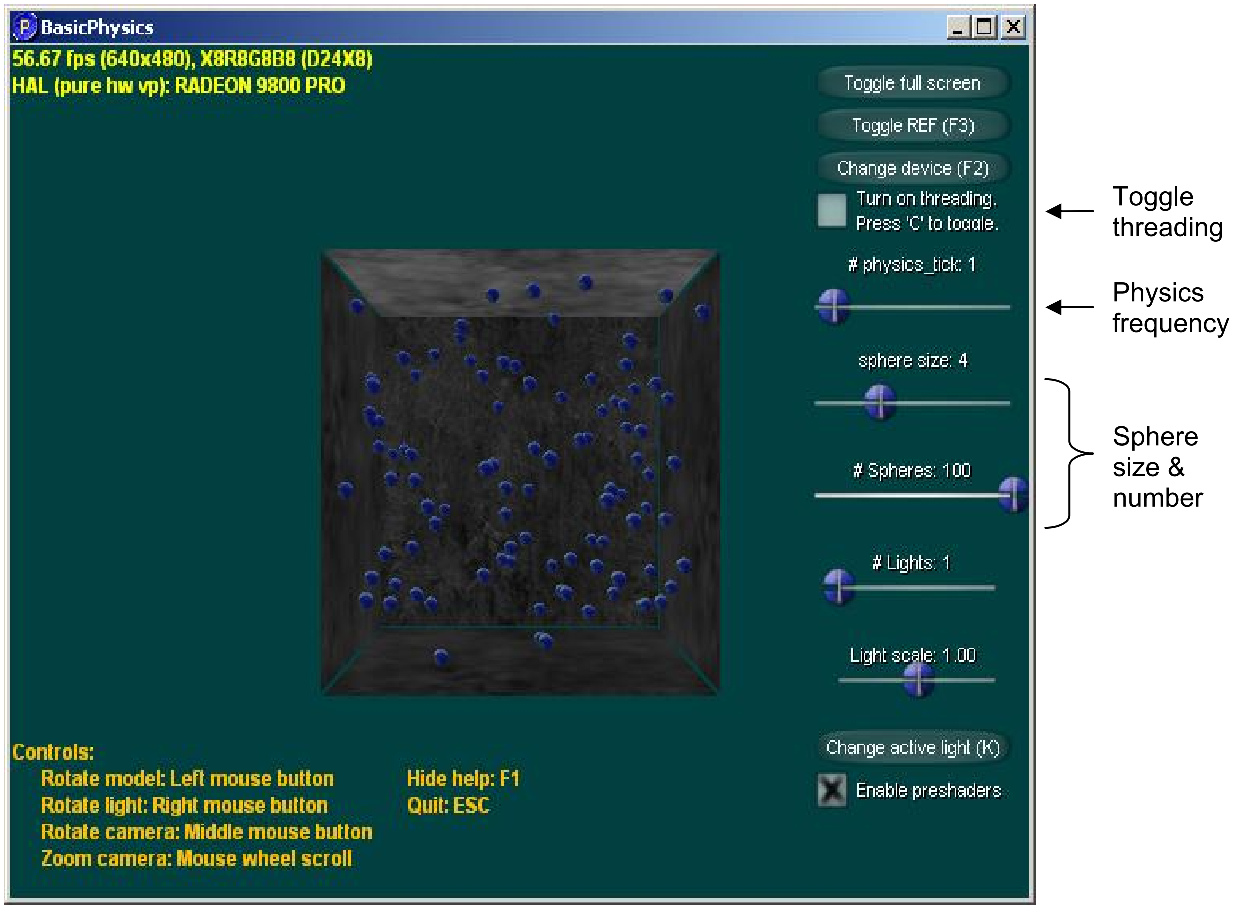
```
                                        collide = true;
                                        //Store the normal at this face
                                }
                                break;
                        }
                        else
                        {
                                float Projected = velocity * update_interval;

//This if handles the case where sphere moves //distnace x in given update_interval
                                //But x is less than distance from plane.

                                if (Projected > distance)
                                {
                                   collide = true;
                                   time_fract = (distance–radius)/velocity;
                                   time_remain = update_interval - time_fract;
                                }
                                break;
                        }
                }
                return collide;
}
```

User Interface

The GUI for this multi-threaded physics demo is adapted from the BasicHLSL demo from the Direct X 9 SDK*. In Figure 9, there is a checkbox to enable/disable threading for the physics simulation, a slider bar to adjust the frequency of physics calculations which also freezes movement when set to zero, and sliders to adjust the size and number of spheres present. The light sliders are a remnant of the original BasicHLSL demo to change the number and brightness of the light available. This allows all the sphere properties to be modified in real-time.



56.67 fps (640x480), X8R8G8B8 (D24X8)
HAL (pure hw vp): RADEON 9800 PRO

Turn on threading. Press 'C' to toggle.
# physics_tick: 1
sphere size: 4
Spheres: 100
Lights: 1
Light scale: 1.00
☒ Enable preshaders

Controls:
Rotate model: Left mouse button
Rotate light: Right mouse button
Rotate camera: Middle mouse button
Zoom camera: Mouse wheel scroll
Hide help: F1
Quit: ESC

Figure 9: Physics GUI

Multi-threading

This implementation has two threads. At start-up, the first thread did the rendering and simulation. Once a user selects the threading checkbox, then the second thread is signaled to begin computing the physics calculations while the first one takes care of the remainder of the demo: rendering, keyboard input, and so on. A function called `PhysicsSimulation()` does all the physics calculations. Depending on if threading has been enabled by the user, either the main thread calls `PhysicsSimulation()` or the second thread does the work in parallel. Once threading is disabled, the second thread is put to sleep and the `PhysicsSimulation()` call returns back to the main thread. This way, the overhead of thread creation is kept only to the application initialization.

Events are used to signal when a thread is sleeping or awakened to begin computing work. By default, the second thread is created at the start of run-time and put to sleep. If threading is enabled, then the `thStartThreading` event is signaled and the thread wakes up to begin the physics computation. When it is complete, the `thStopThreading` event is signaled and the thread is put back to sleep. By avoiding creating a thread for each frame, the thread creation overhead is avoided. In the main thread, synchronization occurs with `WaitForSingleObject` to sync the sphere position calculation and the rendering.

Thread creation

There is a high overhead cost associated with thread creation. In this implementation, it is reduced by creating a thread at application initialization and using this for the duration of the entire runtime. There are other implementations, such as thread pools, discussed in at the end of this paper in Future Work. The parameters of `CreateThread` are the following: the first parameter is a pointer to a security attributes structure. Setting this to NULL means that the handle cannot be inherited by child processes. The `dwStackSize` is listed second: if this is set to 0, then the new thread uses the default size for the executable. Third is a pointer to the function to be executed by the thread. The `lpParameter` is a pointer to the function arguments. The `dwCreationFlags` control how the thread is created; if it is set to 0, the thread runs immediately after creation. Finally, the `lpThreadId` is a pointer to a variable that receives the thread identifier.

```
HANDLE CreateThread(
                LPSECURITY_ATTRIBUTES lpThreadAttributes,
                SIZE_T dwStackSize,
                LPTHREAD_START_ROUTINE lpStartAddress,
                LPVOID lpParameter,
                DWORD dwCreationFlags,
LPDWORD lpThreadId
);
```

In this scenario, a single call to `CreateThread` is sufficient to create a worker thread. When the application begins, it immediately enters a sleep mode until it is called to complete some computation work. Sample code from the demo is included below. The thread is created to run with default properties and call the `thPhysicssimulation` function.

```
thThread = CreateThread(NULL,0,thPhysicsSimulation,0,0,&threadId);
```

In addition, there need to be two events to signal to the thread when to wake up to begin computation or when to go back to sleep. Also at application initialization, there are two events created: these automatically reset automatically when thread waiting on them has been released.

```
thStartThreading = CreateEvent(NULL, FALSE, FALSE, NULL);
thStopThreading = CreateEvent(NULL, FALSE, FALSE, NULL);
```

The `thPhysicsSimulation` function simply waits on the `thStartThreading` signal before it begins its computational work. It is an infinite loop that is then put to sleep after the computation is complete. This is part of the design so that the demo could switch from a non-threaded mode into a threaded mode. Without this necessity, it is probably unnecessary to signal the thread to go back to sleep once the computation is done.

```
DWORD WINAPI thPhysicsSimulation(LPVOID n)
{
                while (1) {
                        WaitForSingleObject(thStartThreading,INFINITE);
                        PhysicsSimulation();
                        SetEvent(thStopThreading);
                }
                return 0;
}
```

Thread execution

The thread created at application initialization remains at 0% CPU utilization until a user selects that threading be enabled in the physics GUI. This then signals an event that is designed to wake up the thread to start its work.

```
        if (gThreadingEnabled)
                SetEvent(thStartThreading);
else
        PhysicsSimulation();
```

The function `PhysicsSimulation()` calculates the new sphere positions and checks to see if it will collide with other spheres or planes. If collision will occur, then it also computes the new positions and velocity after impact. However, if threading has been enabled, then the `thStartThreading` event sends a signal to the `thPhysicsSimulation()` function to begin its computational work. Once complete, the `thStopThreading` event is triggered and the thread goes back to sleep. This also eliminates the need to delete threads.

Multi-threading results

Perfmon* was used to verify that the physics calculations are indeed shifting from one thread to the other when the checkbox is toggled. The first version of the multi-threaded demo had two threads: one was the main thread for rendering keyboard input, and the other thread for physics simulation. Once the GUI checkbox was toggled, then the second thread that had been sleeping up to this point was signaled to complete the physics simulation instead of relying on the first thread to complete this task. In Figure 10 below, the single threaded version pegs the processor at 99-100%. In the multi-threaded version, the rendering thread uses about 15% of the processor while the physics simulation uses 85%.

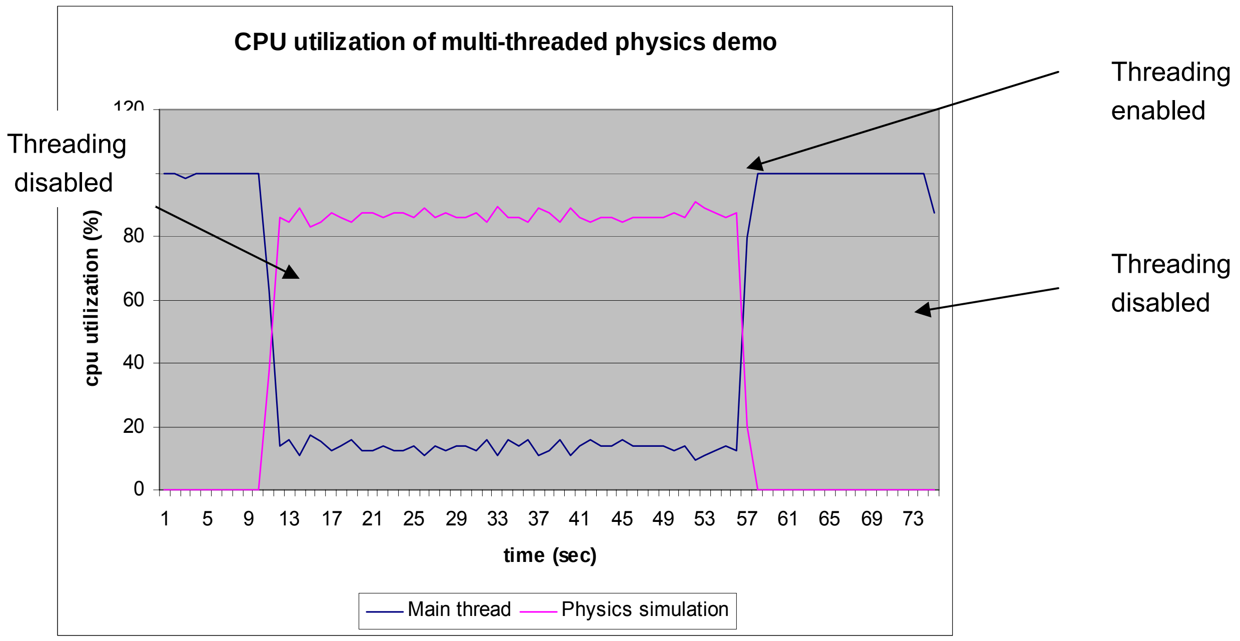


Figure 10: CPU utilization between threads

Future Work

In the future, a thread pool could be used to have an arbitrary number of threads set. This would aid in creating dynamic threading. For instance if the number of spheres were to increase from 100 to 200, then this work could be split into 4 threads if there are four logical processors present.

Additionally, comparison of data vs. functional decomposition: Currently, this demo could easily use data decomposition by separating the number of spheres in half. A functional decomposition model would be created to separate the sphere-sphere collision detection and the sphere-plane collision detection. This would help processor architects compare performance of these two different threading models.

References and Related Links

References

Jones, Wendy. 2004. *Beginning DirectX9.* Premier Press.

Watt, Alan. 2000. *3D Computer Graphics Addition*. Wesley.

Chris Hecker. 1996. *Physics Simulations*. Game Developer Forum.

Pool of Hall Lesson

http://www.gamasutra.com/features/20020118/vandenhuevel_03.htm

Rotation around an arbitrary axis

http://news.povray.org/povray.advanced-users/thread/%3C3C06D0B8.9DE747A0%40hotmail.com%3E/

Simple Bounding-Sphere Collision Detection

http://www.gamedev.net/reference/articles/article1234.asp

Related Links

APG Software Collision Detection

http://www.apgsoftware.co.uk/gl/collide.html

Microsoft DirectX 9.0 SDK and sample applications
http://msdn.microsoft.com/downloads/list/directx.asp

About the Authors

Rajshree Chabukswar is an Application Engineer working on client enabling. (Mobile Application Enabling group) Prior to working at Intel, she obtained an MS in Computer Engineering at Syracuse University, NY.

Adam Lake is a Sr. Software Engineer in the Software Solutions Group specializing in the next generation computer graphics algorithms and architectures. Prior to working at Intel he obtained a Masters in Computer Science at the University of North Carolina at Chapel Hill studying computer graphics and virtual reality. Previously he worked at Los Alamos National Laboratory in the Applied Theoretical Physics and Computational Science Methods group. He has several publications in computer graphics to his name. In his spare time he is a mountain biker, road cyclist, hiker, camper, avid reader, snowboarder, and a Sunday driver.

Mary Lee is an Application engineer in the Mobile Application Enabling team of Software & Solutions Group Goal that enables client platforms through software optimizations and architecture influence. Prior to working at intel she obtained a Bachelor degree in Computer Science from Purdue University.