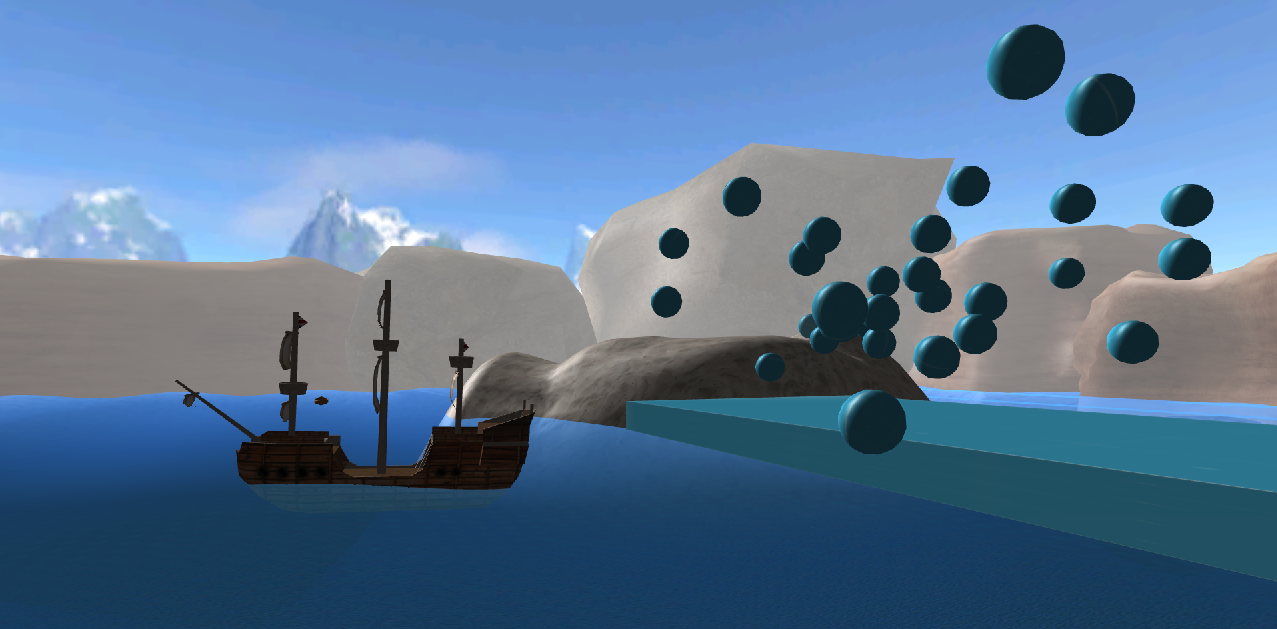
Further Games and Graphics Concepts Semester 2 Report

Anthony Sturdy

Staffordshire University



# Particle Model

The particle model in my application uses a net force, acceleration and velocity vector for motion. Each frame, the net force is calculated using multiple forces stored in a struct (Thrust, Gravity and Friction).

*void* ParticleModel::UpdateNetForce() {

    //Add thrust

    netForce.x += forces.thrust.x;

    netForce.y += forces.thrust.y;

    netForce.z += forces.thrust.z;

    //Add gravity

    netForce.x += forces.gravity.x \* (isColliding ? 0 : 1);

    netForce.y += forces.gravity.y \* (isColliding ? 0 : 1);

    netForce.z += forces.gravity.z \* (isColliding ? 0 : 1);

    //Multiply by friction (1 = no friction, 0 = infinity friction)

    netForce.x \*= forces.friction.x;

    netForce.y \*= forces.friction.y;

    netForce.z \*= forces.friction.z;

}

Next, the acceleration is calculated using the net force and dividing it by the mass. ***a*** is acceleration, ***f*** is the net force, *ti* is an instant of time and *m* is mass.

*acceleration = netForce / mass*

acceleration = XMFLOAT3(netForce.x / mass,

                        netForce.y / mass,

                        netForce.z / mass);

The final step in the particle model’s update is to move the object using constant acceleration. The position is updated, acceleration is applied to velocity and friction is then also applied to the velocity.

The velocity is updated using the following equation with ***v*** being velocity, *dt* being deltaTime and *ti –* 1being a previous instant in time:

*newVelocity = prevVelocity + (acceleration \* deltaTime)*

velocity = XMFLOAT3(velocity.x + (acceleration.x \* deltaTime),

velocity.y + (acceleration.y \* deltaTime),

velocity.z + (acceleration.z \* deltaTime));

The position is updated using the following equation, with ***p*** being position:

*newPosition = prevPosition + prevVelocity \* deltaTime + 0.5 \* acceleration \* deltaTime \* deltaTime*

transform->SetPosition(XMFLOAT3(transform->GetPosition().x + (velocity.x \* deltaTime + 0.5f \* acceleration.x \* deltaTime \* deltaTime),

transform->GetPosition().y + (velocity.y \* deltaTime + 0.5f \* acceleration.y \* deltaTime \* deltaTime),

transform->GetPosition().z + (velocity.z \* deltaTime + 0.5f \* acceleration.z \* deltaTime \* deltaTime)));

# Rigidbody

The Rigidbody system in my application uses Inertia and Torque with Quaternion calculations to simulate physics rotations.

The first calculation performed by the class is to calculate the object’s Inertia Tensor. The Moment of Inertia is how difficult it is to change an object’s rotational speed. Different shaped object’s will have different Inertia Tensors, my application calculates the Inertia Tensor for a rectangular prism, but depending on the shape of the object determines which type of Inertia Tensor should be calculated:

XMMATRIX Rigidbody::CalculateInertiaTensor(*float* *w*, *float* *h*, *float* *d*) {

    XMFLOAT3X3 matrix = XMFLOAT3X3();

    matrix.\_11 = 0.0833f \* particleModel->GetMass() \* (h \* h + d \* d);

    matrix.\_22 = 0.0833f \* particleModel->GetMass() \* (w \* w + d \* d);

    matrix.\_33 = 0.0833f \* particleModel->GetMass() \* (w \* w + h \* h);

    return XMLoadFloat3x3(&matrix);

}

To rotate an object, a function called ApplyForce is called, which takes two parameters: Force direction, and the point of which force is applied relative to the object’s centre of mass.

*void* ApplyForce(XMFLOAT3 *forceDir*, XMFLOAT3 *pointForceApplied*);

These two variables are used to calculate Torque, with ***P****f* being the point of force, and *f* is the force being applied.

XMVECTOR Rigidbody::CalculateTorque(XMFLOAT3 *forceDirection*, XMFLOAT3 *pointForceApplied*) {

    return XMVector3Cross(XMLoadFloat3(&pointForceApplied), XMLoadFloat3(&forceDirection));

}

Angular Acceleration is then calculated:

XMVECTOR Rigidbody::CalculateAngularAcceleration() {

    XMMATRIX inverseMatrix = XMMatrixInverse(&XMMatrixDeterminant(inertiaTensor), inertiaTensor);

    return XMVector3Transform(torque, inverseMatrix);

}

After Angular Acceleration is calculated, Angular Velocity can be calculated and damping is applied (*d* being damping):

XMVECTOR Rigidbody::CalculateAngularVelocity(*float* *deltaTime*) {

    return angularVelocity + angularAcceleration \* deltaTime;

}

XMVECTOR Rigidbody::ApplyDamping(*float* *deltaTime*) {

    return angularVelocity \* angularDamping \* deltaTime;

}

Following the Angular Velocity calculations, the Angular Orientation can be calculated using a quaternion. The Angular Velocity vector is added to the quaternion, the quaternion is then normalised and CalculateTransformMatrixColumnMajor() is called, resulting in a final rotation matrix which is used when the world matrix is calculated for the object.

XMMATRIX Rigidbody::CalculateAngularOrientation(XMFLOAT3 *position*, *float* *deltaTime*) {

    XMFLOAT3 av; XMStoreFloat3(&av, angularVelocity);

    quaternion.addScaledVector(vector3(av.x, av.y, av.z), deltaTime);

    quaternion.normalise();

    XMMATRIX resultMatrix = XMMATRIX();

    CalculateTransformMatrixColumnMajor(resultMatrix, vector3(position.x, position.y, position.z), quaternion);

    return resultMatrix;

}

Finally, I also apply the damping to the current force being applied, which along with the damping applied previously, slows the object over time.

# Choices made to Simulated Effects and Behaviour

The physics in my application are used for the Particle System. Each particle system stores a pool of GameObjects, which can then be set to active and have physics applied once Emit() is called. I have a ParticleInfo struct which stores information about the current particle being emitted:

*struct* ParticleInfo {

    XMFLOAT3 position;

    XMFLOAT3 scale;

    XMFLOAT3 thrust;

    XMFLOAT3 friction;

    XMFLOAT3 gravity;

    XMFLOAT3 initVelocity;

    ID3D11ShaderResourceView\* texture;

*float* lifeTime = 10.0f;

*float* lifeTimeRemaining = 0.0f;

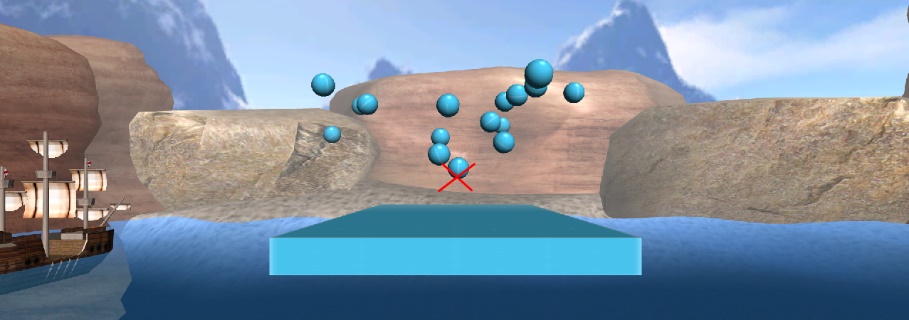
};

This Particle Information is applied across every particle emitted, but can also be specified for each individual Emit() call using another implementation of the function.

*void* Emit();

*void* Emit(ParticleInfo& *info*);

The Particle System currently in the scene set up to fire the particle in the air upon emission, this done by setting the initial velocity, which the Particle Model then uses to simulate the particle flying in the air until gravity causes the particle to begin falling. A random X and Z value is also added to the initial velocity to give the particle system some variation.

I have placed a block below the point of emission to demonstrate collision. In the picture to the left, you can see the particles being emitted (the red cross illustrates the point of emission) along with the block placed in the scene.

Upon collision, the particles bounce slightly before tapering off and stopping. Some bounce and continue off the edge of the block, which causes them to keep falling.

The collision in my application can handle Sphere vs Sphere, Sphere vs AABB and AABB vs AABB. I decided not to have Particles check for collisions with one another as they are all emitted from a single point, which would cause problems and appear ‘buggy’. Instead, Scene Objects are separated from Particle Objects for collision checks, so the collision checks being performed are Scene Objects vs Scene Objects, and Scene Objects vs Particle Objects. To optimise collision, I have multiple checks where collision calculations should not be check between two objects. The current checks in place to skip collision checks are:

* If either object A or B are *not* active
* If object A is the same as object B (this can happen due to each object being checked against each other object, this check does not need to be performed for Particles)
* If *both* object A and B are static (if neither object can move, there is no need to check for collision on each other)

Collision between two spheres requires the position and radius of both spheres. The distance is then calculated, and if the distance is smaller than the sum of both radii, the object’s are colliding.

*distance = sqrt((x1-x2)\*(x1-x2) + (y1-y2)\* (y1-y2) + (z1-z2)\* (z1-z2))*

*float* dist = sqrt((pos1.x - pos2.x) \* (pos1.x - pos2.x) +

                    (pos1.y - pos2.y) \* (pos1.y - pos2.y) +

                    (pos1.z - pos2.z) \* (pos1.z - pos2.z));

Sphere vs AABB collision is more complex than Sphere vs Sphere but still uses the distance equation. Unlike the Sphere vs Sphere collision, the distance from one object to another can’t be used due to the corners of a cube being further away from the centre than the middle of the edges. Instead, the closest point of the Box to the Sphere needs to be found, then the distance calculation is performed from the Sphere to that point nearest to the Sphere. It also doesn’t check if the distance is less than the sum of two radii as the Box doesn’t have a radius, instead it just checks if the distance is less than the radius of the Sphere. The closest point is found by getting the minimum and maximum points of the Box, then for each X, Y and Z, it gets the higher value between the Box minimum point, and the lower value between the position of the sphere and Box’s maximum point. Explaining through text is confusing and difficult to translate from code though, so the following code snippet should be clearer:

//Min / Max values of box

XMFLOAT3 bMin = { pos2.x - (b.w / 2), pos2.y - (b.h / 2), pos2.z - (b.d / 2) };

XMFLOAT3 bMax = { pos2.x + (b.w / 2), pos2.y + (b.h / 2), pos2.z + (b.d / 2) };

//Get box's closest point to sphere centre

*float* x = max(bMin.x, min(pos1.x, bMax.x));

*float* y = max(bMin.y, min(pos1.y, bMax.y));

*float* z = max(bMin.z, min(pos1.z, bMax.z));

//Distance between box closest point and sphere centre

*float* dist = sqrt((x - pos1.x) \* (x - pos1.x) +

                    (y - pos1.y) \* (y - pos1.y) +

                    (z - pos1.z) \* (z - pos1.z));

Finally, AABB vs AABB simply checks if the minimum points and maximum points of either box are overlapping. It’s very similar to a 2D AABB collision check, only with a third axis added.

//Min / Max values

XMFLOAT3 b1Min = { pos1.x - (b1.w / 2), pos1.y - (b1.h / 2), pos1.z - (b1.d / 2) };

XMFLOAT3 b2Min = { pos2.x - (b2.w / 2), pos2.y - (b2.h / 2), pos2.z - (b2.d / 2) };

XMFLOAT3 b1Max = { pos1.x + (b1.w / 2), pos1.y + (b1.h / 2), pos1.z + (b1.d / 2) };

XMFLOAT3 b2Max = { pos2.x + (b2.w / 2), pos2.y + (b2.h / 2), pos2.z + (b2.d / 2) };

//3D overlap check

return ((b1Min.x <= b2Max.x && b1Max.x >= b2Min.x) &&

        (b1Min.y <= b2Max.y && b1Max.y >= b2Min.y) &&

        (b1Min.z <= b2Max.z && b1Max.z >= b2Min.z));

The only limitation to my collision system is that it cannot handle an Oriented Bounding Box (OBB), meaning if an object rotates, the collision area will not. This is usually checked using the Separated Axis Theorem (SAT), but unfortunately due to time constraints, I didn’t have time to implement this collision as it’s much more complex than Sphere and AABB collision. It is something I’d like to research in the future though. Another improved collision method could be Triangle vs Triangle collision, which checks if two Triangles are intercepting, this can be very slow though, as checking every triangle against every triangle can amount to a huge amount of calculations. If this collision method is used, it can be coupled with a simplified mesh used only for collision, as well as another Broad Phase of collision, such as using an Octree to detect when two objects (or in this case, triangles) could *possibly* be colliding before performing the actual collision checks.

# UML Class Diagram