Object Oriented Programming with Data Structures and Algorithms CS4D768

Dr Janusz Kulon

PhD, MSc, BSc

Lectures Plan

Introduction

OO & OpenGL fundamentals

Modelling simple physics laws of motion

Forces Accumulator D'Alembert principle

Collision Detection algorithm

Contact resolver algorithm

Contact generation algorithm, penetration resolution

Separating axes theorem, contact types

Bounding volume hierarchy, hierarchical grids, binary space partitioning

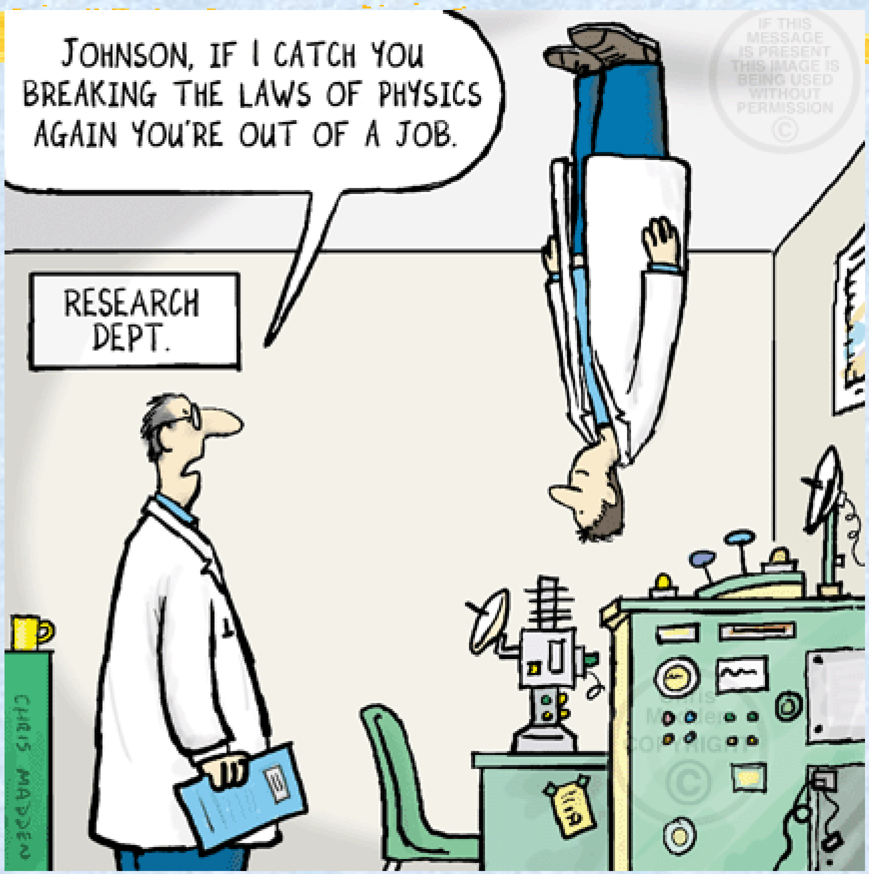
Recursive Dimensional Clustering

Minkowski space, Gilbert–Johnson–Keerthi algorithm

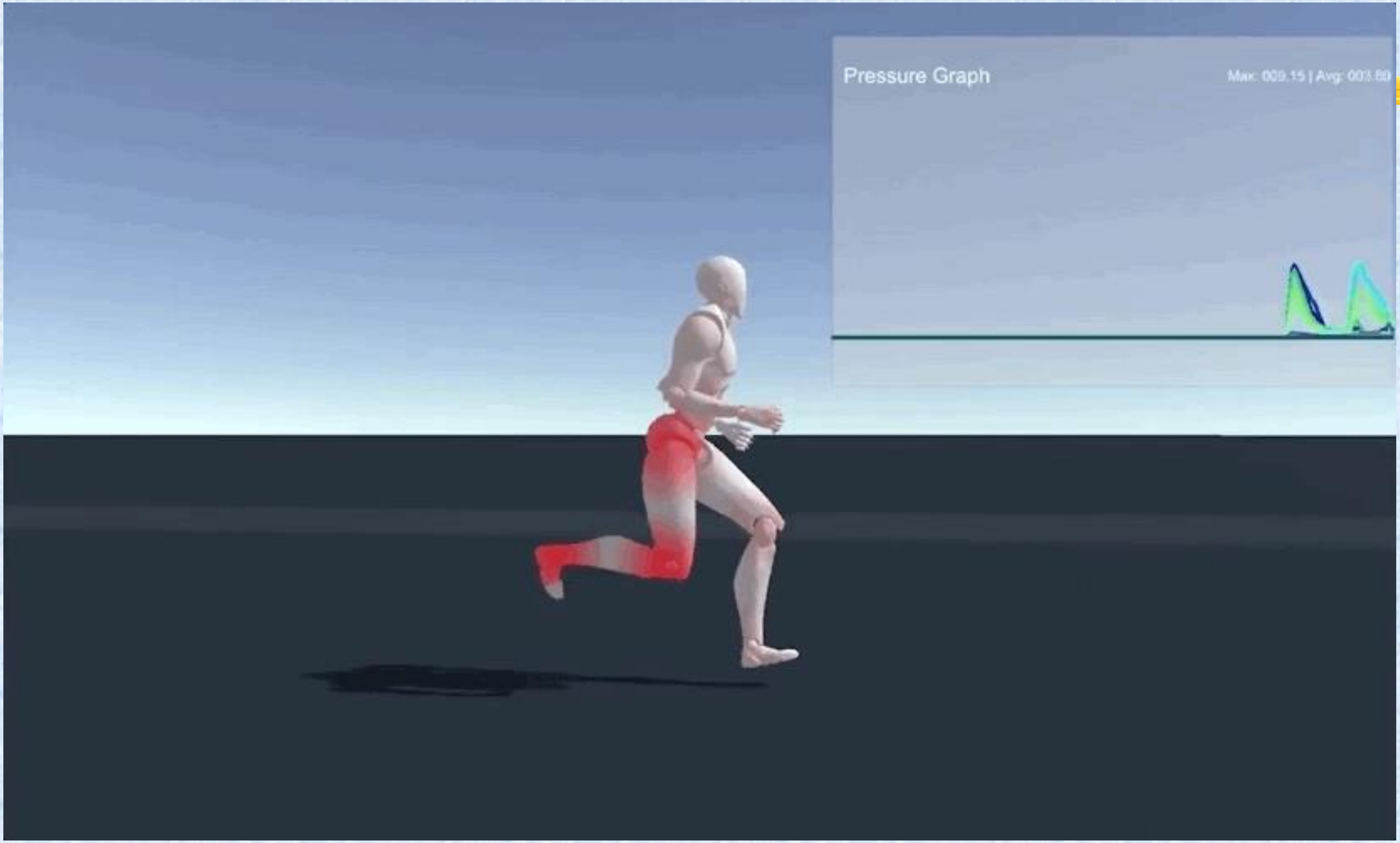
Project demo

Why Physics Simulation?

- Realistic models
- More interactive human experience
- Emergent Behavior

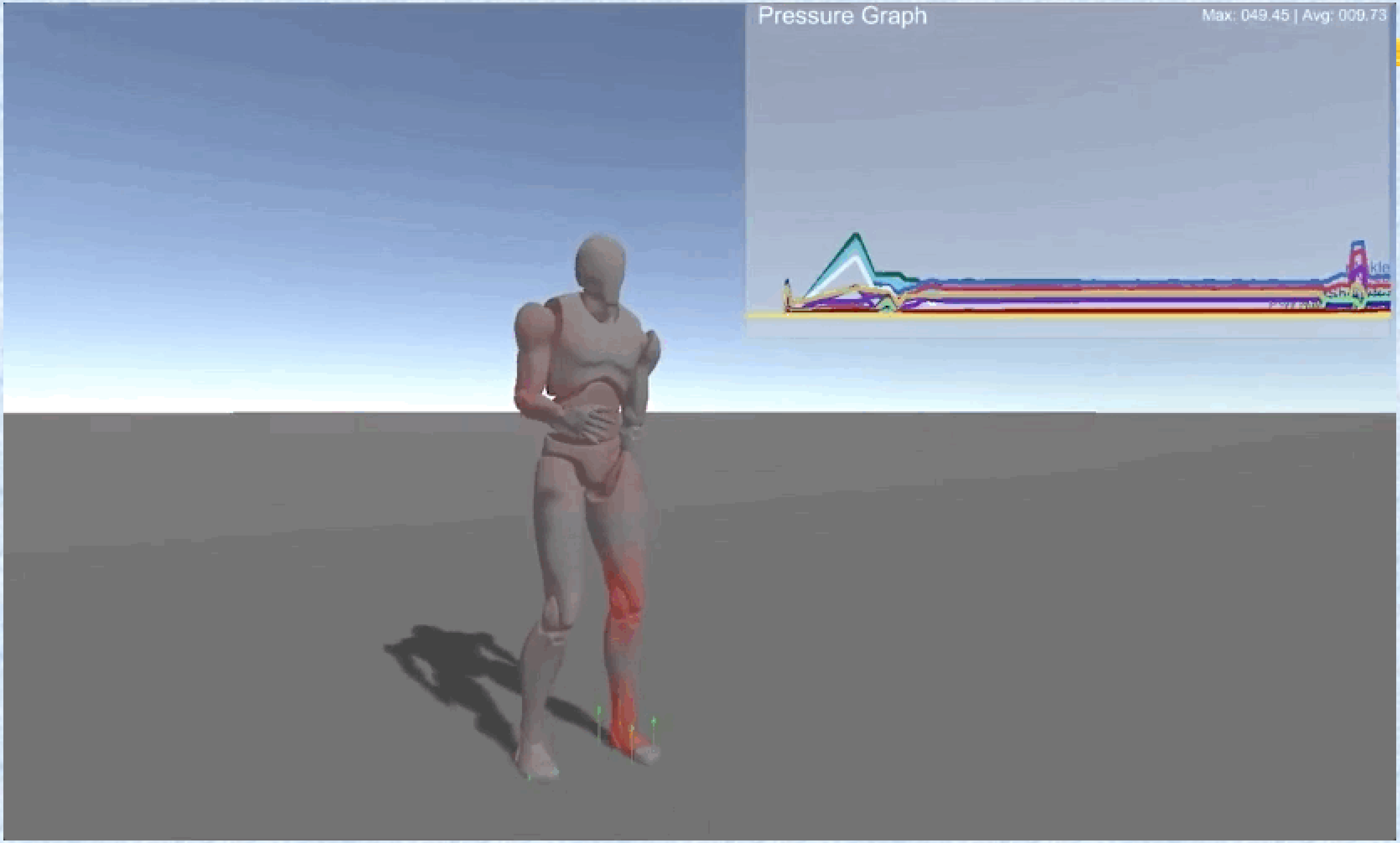


Health



https://blog.deepmotion.com/2018/08/27/how-can-physical-simulation-and-virtual-reality-improve-health/

Health



https://blog.deepmotion.com/2018/08/27/how-can-physical-simulation-and-virtual-reality-improve-health/

Game Physics

Coaster Rider; In most levels the amount of potential energy you start with is all you have to work with

Newtonian Mechanics

Pool 3D Training Edition is a unique 3D billiard game simulator that lets you improve your billiard gaming skills effectively by learning game physics on a PC

Armadillo Run is a build-and-simulate puzzle game in the.. The goal of the game is to guide the armadillo—it's basically a basketball—to the target area.

Fluid Dynamics

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} + \nu \nabla^2 \mathbf{u} + \mathbf{f}$$

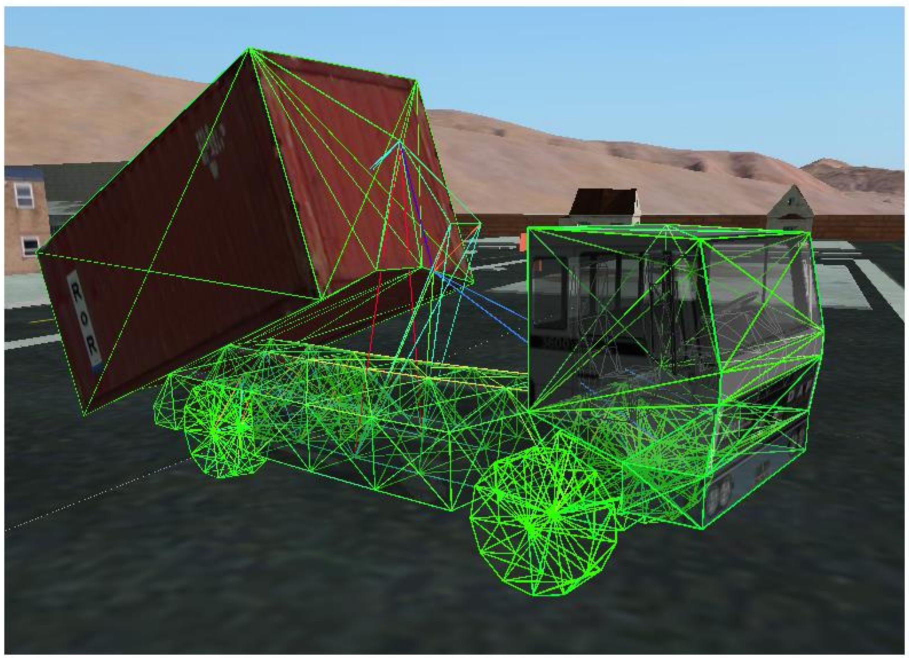
$$\frac{\partial \rho}{\partial t} = -(\mathbf{u} \cdot \nabla)\rho + \kappa \nabla^2 \rho + S$$

In **Ichor**, you fight without bullets or guns. You are just there, in the fluid. Everything is subject to change at a moments notice.

https://www.fun-motion.com/physics-games/ichor/

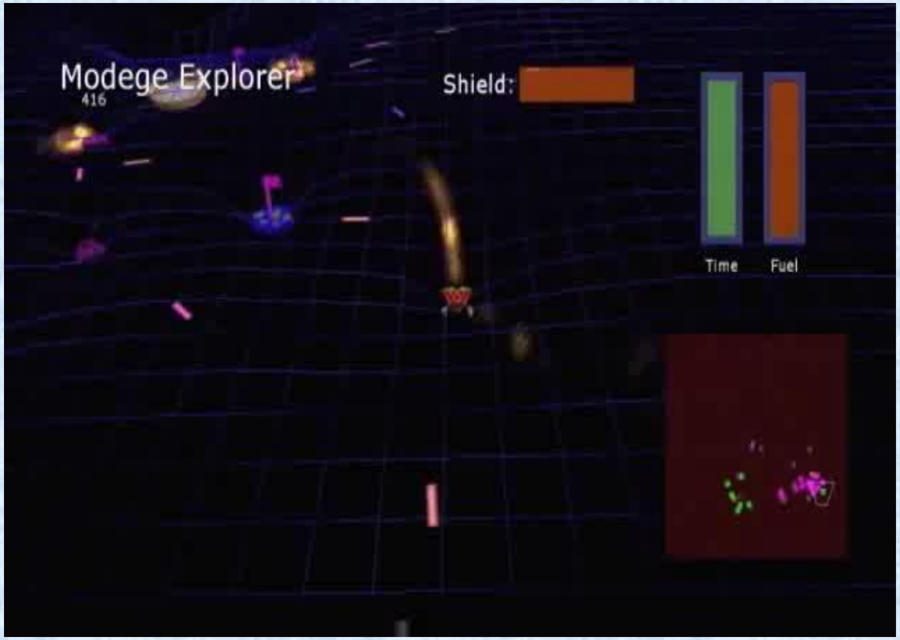
Rigs of Rods

is a free and open source vehicle-simulation which **uses soft-body physics** to simulate the motion destruction and deformation of vehicles



http://www.rigsofrods.org/

Relativistic Computer Games



The **Special Theory of Relativity** describes what happens to objects travelling at **speeds close to the speed of light**, and includes many fantastic and counter-intuitive effects such as **time dilation** and **length contraction**.

Particle systems

- A particle system is a graphics subsystem used to simulate certain natural phenomena such as **fire, smoke, sparks, explosions, dust, magic spells, trail effects**, etc.

Quantum Physics - qCraft

qCraft brings the principles of quantum physics to the world of Minecraft.

qCraft is not a simulation of quantum physics, but it does provide ‘analogies’ that attempt to show how quantum behaviors are different from everyday experience.

Newton’s laws of motion

1 **The first law: LAW OF INERTIA:**

- A body in motion will remain in motion unless a net force is exerted upon it.

2 **The second law: LAW OF ACCELERATION:**

- The net force of a particle is the rate of change of its linear momentum.
- Momentum is the mass of the body multiplied by its velocity.
- The force on a body is thus its mass multiplied by its acceleration (F = ma).

3 **The third law: LAW OF RECIPROCAL ACTIONS:**

- To every action there is an equal and opposite reaction.

Performance Tips and Tricks

Ski Stunt Simulator

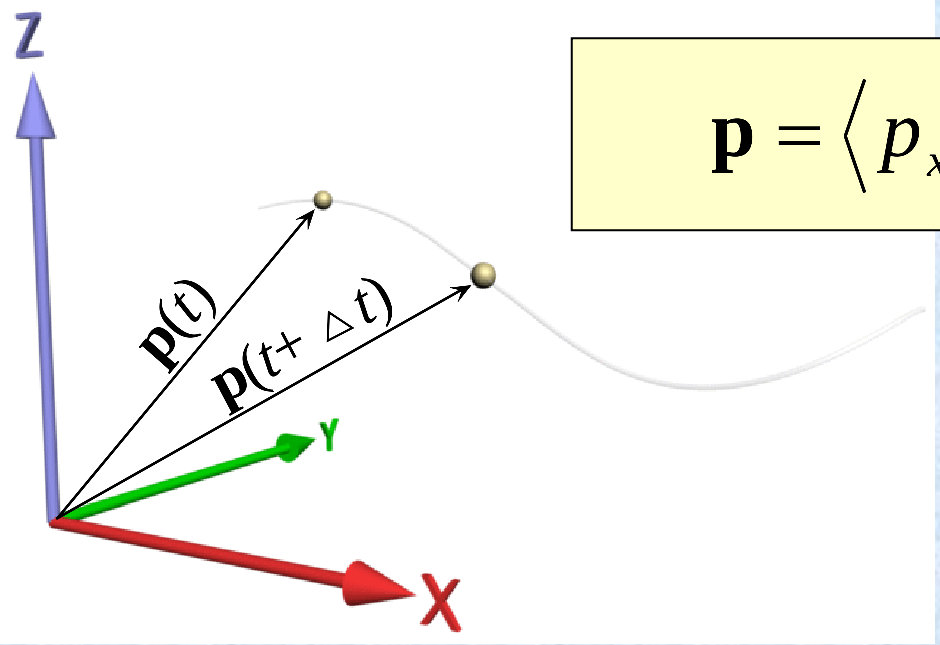
- Keep calculations simple
- Approximate complex volumes
- Do not simulate static objects
- rigid body kinematics

Time

- When designing an algorithm for use in a simple game it is more common to define time around the game's **frame rate** than in terms of seconds, minutes, hours, and so forth.
- More advanced games, such as 3D first-person shooters, require a **real time system operating** independently from the game's frame rate.
- Using **real time (seconds) as opposed to virtual time (frames)** is required when modeling movement and forces without the end result **being unrealistically influenced by changes in the game's frame rate.**

Position

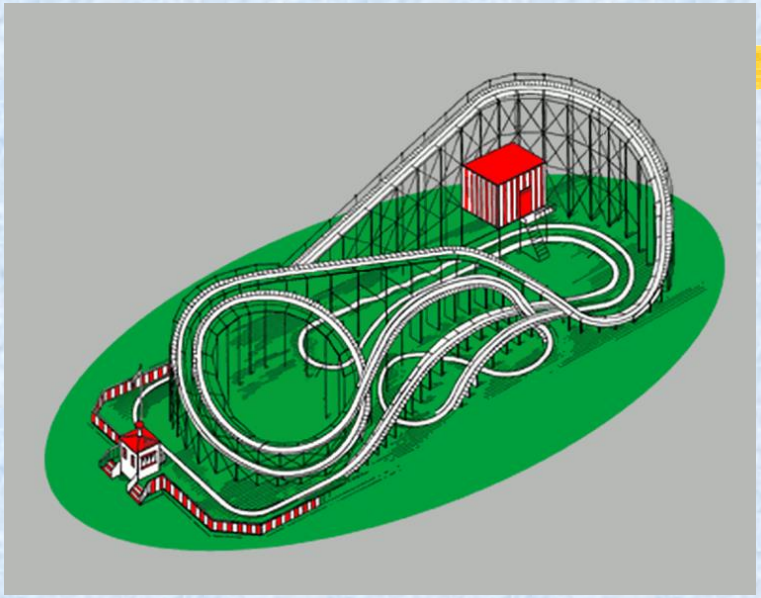
- Location of Particle in World Space
  - SI Units: meters (m)



$$\mathbf{p} = \left\langle p_x, p_y, p_z \right\rangle$$

  - Changes over time when object moves

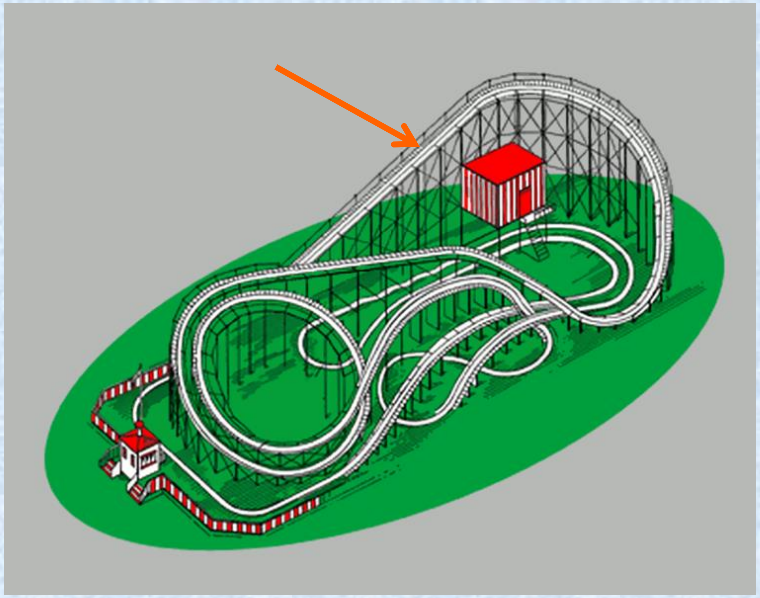
Scalar Positions



A position may be scalar if there's only one degree of freedom

We have to convert to 3D to display our object

Position translated to a point in a track system. That point is in 3D.



Scalar positions are relatively rare in 3D graphics systems

Velocity and Acceleration

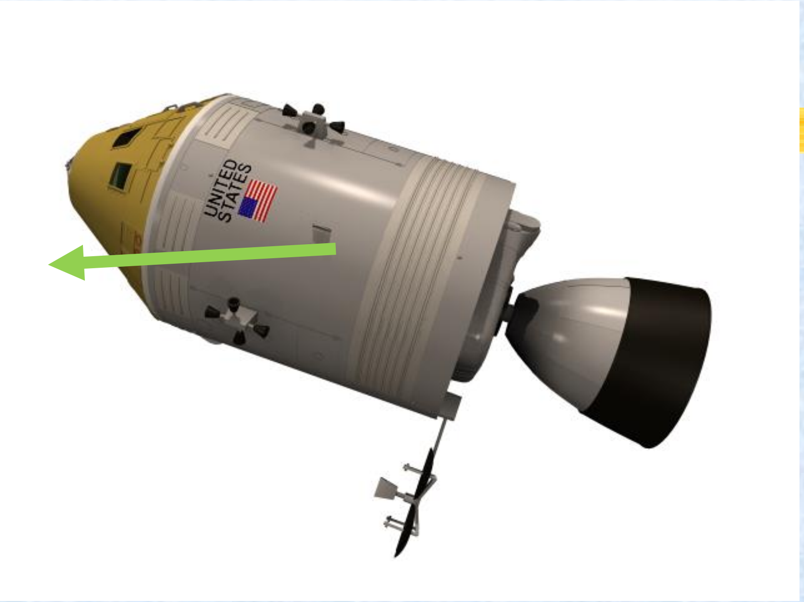
- Velocity (SI units: m/s)
  - First time derivative of position:

$$\mathbf{V}(t) = \lim_{\Delta t \to 0} \frac{\mathbf{p}(t + \Delta t) - \mathbf{p}(t)}{\Delta t} = \frac{d}{dt}\mathbf{p}(t)$$

- Acceleration (SI units: m/s^2)
  - First time derivative of velocity
  - Second time derivative of position

$$\mathbf{a}(t) = \frac{d}{dt}\mathbf{V}(t) = \frac{d^2}{dt^2}\mathbf{p}(t)$$

Velocity Scalar or Vector?



<u>Velocity Vector</u>

Velocity is described as a vector

Vector3 velocity;

<u>Velocity Scalar</u>

Velocity is a single number

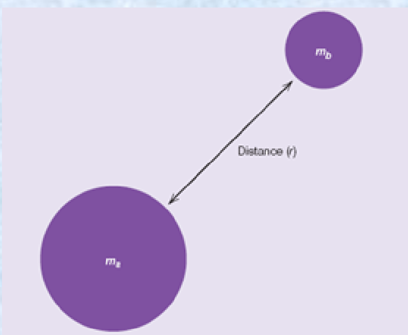
float velocity;

Usually means velocity is dependent on orientation

Velocity vectors are a more general solution in many applications

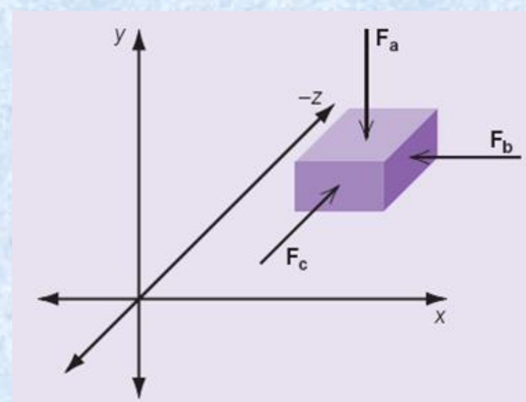
Force

- Force is the physical action exerted upon an object to accelerate it.

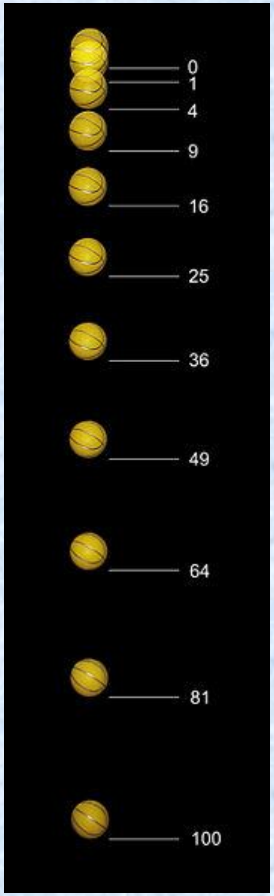


$$F = \frac{G \times m_a \times m_b}{r^2}$$

- Thrust
- Gravity
- Drag/Wind resistance
- Friction



Gravity



Force of gravity is an acceleration vector

(0, -980, 0) 980cm/sec^2

Implementation

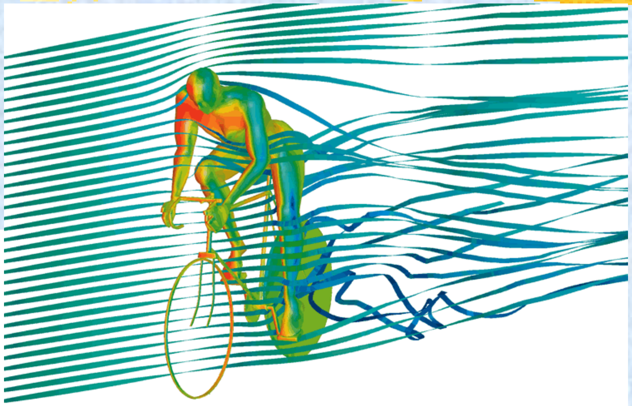
Just add to any other acceleration

acceleration += new Vector3(0, -980, 0);

No matter what mass the object has, it will always accelerate at the same rate due to gravity

Drag

$$F_D = \frac{1}{2} \rho v^2 C_d A,$$



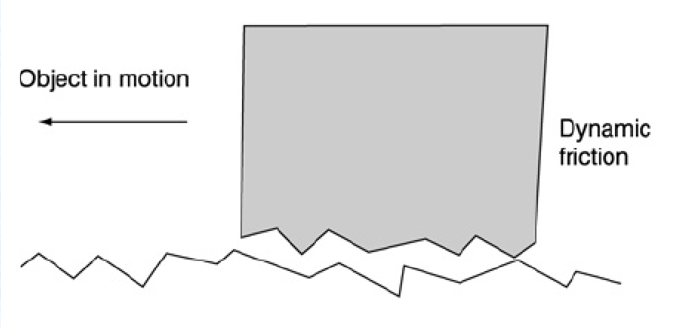
$\mathbf{F}_D$ is the force of drag,
ρ is the density of the fluid,[3]
v is the speed of the object relative to the fluid,
A is the reference area,
C_d is the drag coefficient (a dimensionless parameter, e.g. 0.25 to 0.45 for a car)

Implementation

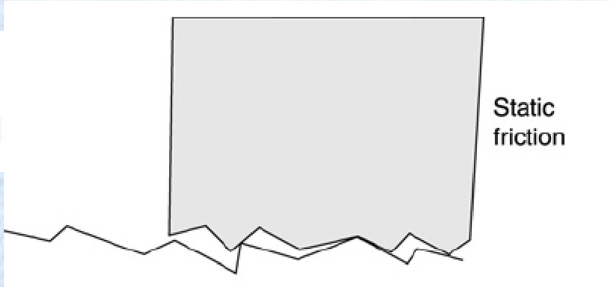
force = velocity * -drag
acceleration += force / mass

Friction

$$F_{frict} = \mu \bullet F_{norm}$$



$$F_{frict\text{-}static} \leq \mu_{frict\text{-}static} \bullet F_{norm}$$



Implementation

To handle friction in our simulation we must first understand what friction is doing in terms of velocity.

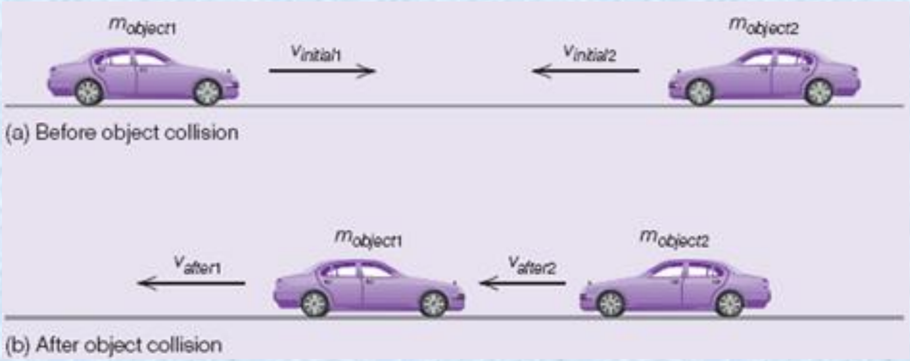
Momentum

- Momentum is the product of mass and velocity, i.e. a property inherent to objects in motion.

$$P = m \times v$$

- Conservation and transfer of momentum.

$$\Delta p_{object1} = -\Delta p_{object2},$$



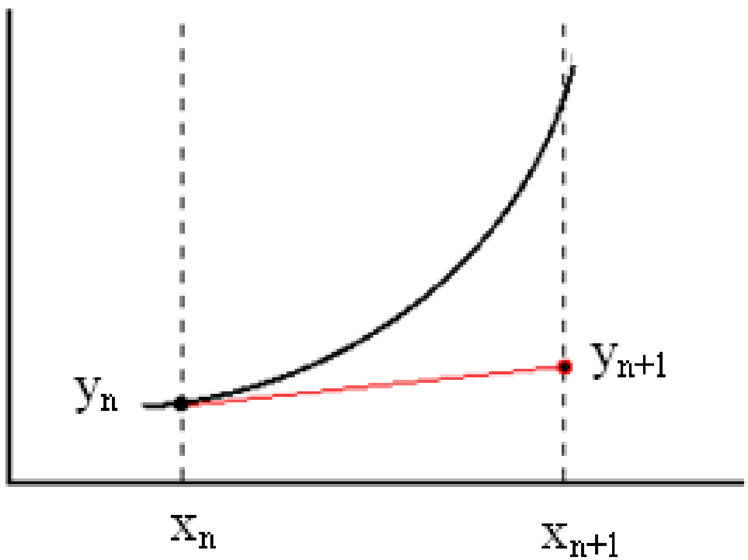
Physics Simulation

- **The Cycle of Motion**
  - Force, $\mathbf{F}(t)$, causes acceleration
  - Acceleration, $\mathbf{a}(t)$, causes a change in velocity
  - Velocity, $\mathbf{V}(t)$ causes a change in position
- **Physics Simulation**
  - Solving variations of the above equations over time to emulate the cycle of motion

Integration methods

- Euler’s Method
- Midpoint Method
- Runge-Kutta method

Euler’s Method



To solve a first order ODE $\dfrac{dy}{dx} = f(x,y)$

Given the initial condition $y(x_0) = y_0$

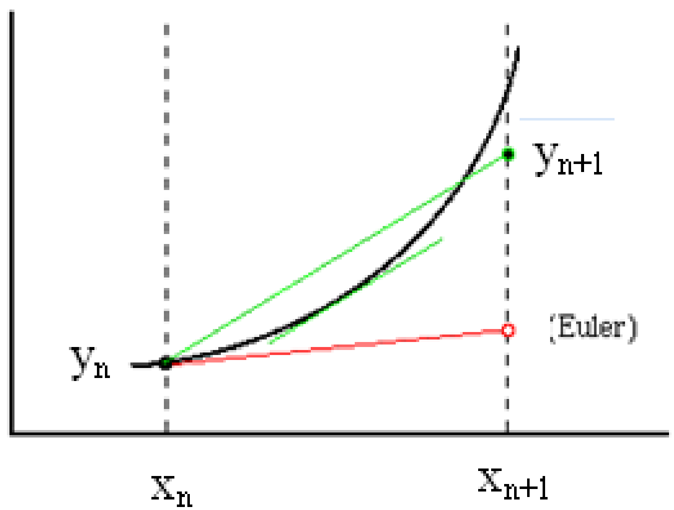
and pick the marching step h,

$$k_1 = hf(x_n, y_n)$$

$$\Rightarrow y_{n+1} = y_n + k_1$$

The Euler integration scheme is the simplest available in the simulator. It is computationally cheap, but because it is only first order its error term is second order in the timestep, making it a relatively inaccurate integration method.

Midpoint Method



To solve a first order ODE $\dfrac{dy}{dx} = f(x, y)$

Given the initial condition $y(x_0) = y_0$

and pick the marching step h,

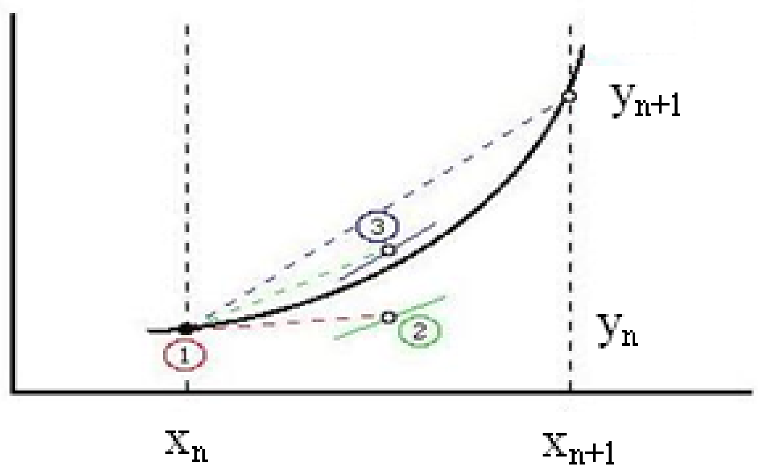
$$k_1 = hf(x_n, y_n)$$

$$k_2 = hf(x_n + \frac{h}{2}, y_n + \frac{k_1}{2})$$

$$\Rightarrow y_{n+1} = y_n + k_2 .$$

The Midpoint method is similar to the Euler method in that it starts by taking an Euler "trial step." It then uses the values obtained by the trial step to take real steps according to the formulae shown. This method also has an error term that is third order, but it is more computationally expensive than the Euler method.

Runge-Kutta method



To solve a first order ODE $\dfrac{dy}{dx} = f(x,y)$

Given the initial condition $y(x_0) = y_0$

and pick the marching step h,

$$k_1 = hf(x_n, y_n)$$

$$k_2 = hf(x_n + \frac{h}{2}, y_n + \frac{k_1}{2})$$

$$k_3 = hf(x_n + \frac{h}{2}, y_n + \frac{k_2}{2})$$

$$k_4 = hf(x_n + h, y_n + k_3)$$

$$\Rightarrow y_{n+1} = y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6}.$$

The Runge-Kutta method is a fourth order integration technique similar to an expanded Midpoint scheme. This method takes four trial steps and uses their weighted average to advance the particle according to the formulae shown above. It is highly accurate, having a fifth order error term, but it is computationally expensive

Order of Operations

Accumulate Forces

```
Vector3 force = engineDirection * engineThrust +
                velocity * -drag;
```

Accumulate Accelerations

```
Vector3 acceleration = force / mass +
                new Vector3(0, -980, 0);
```

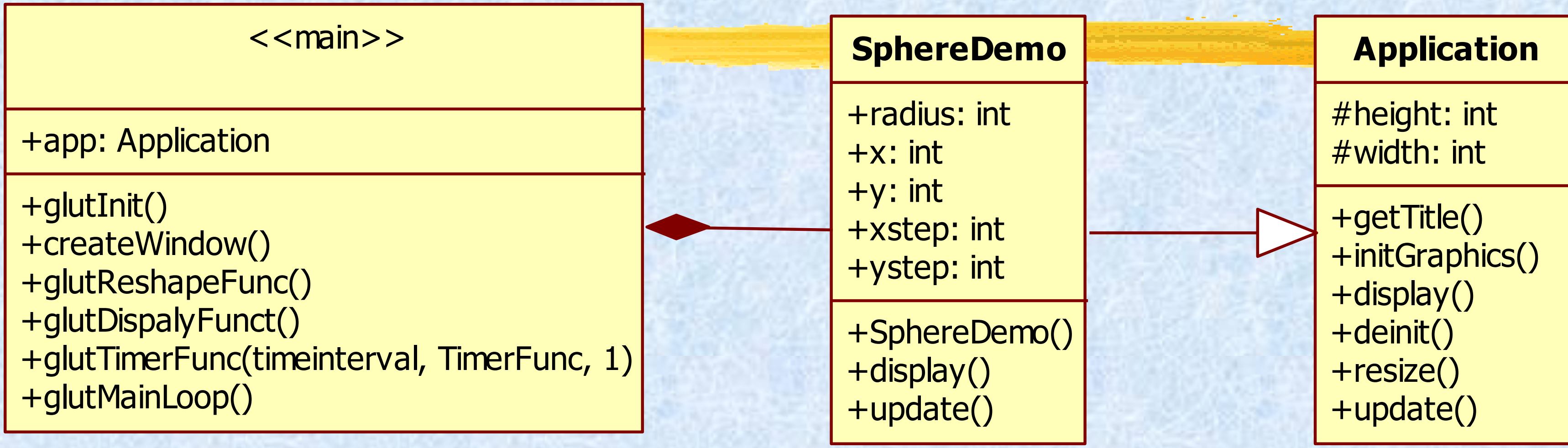
Velocity Euler Step

```
velocity += acceleration * deltaTime;
```

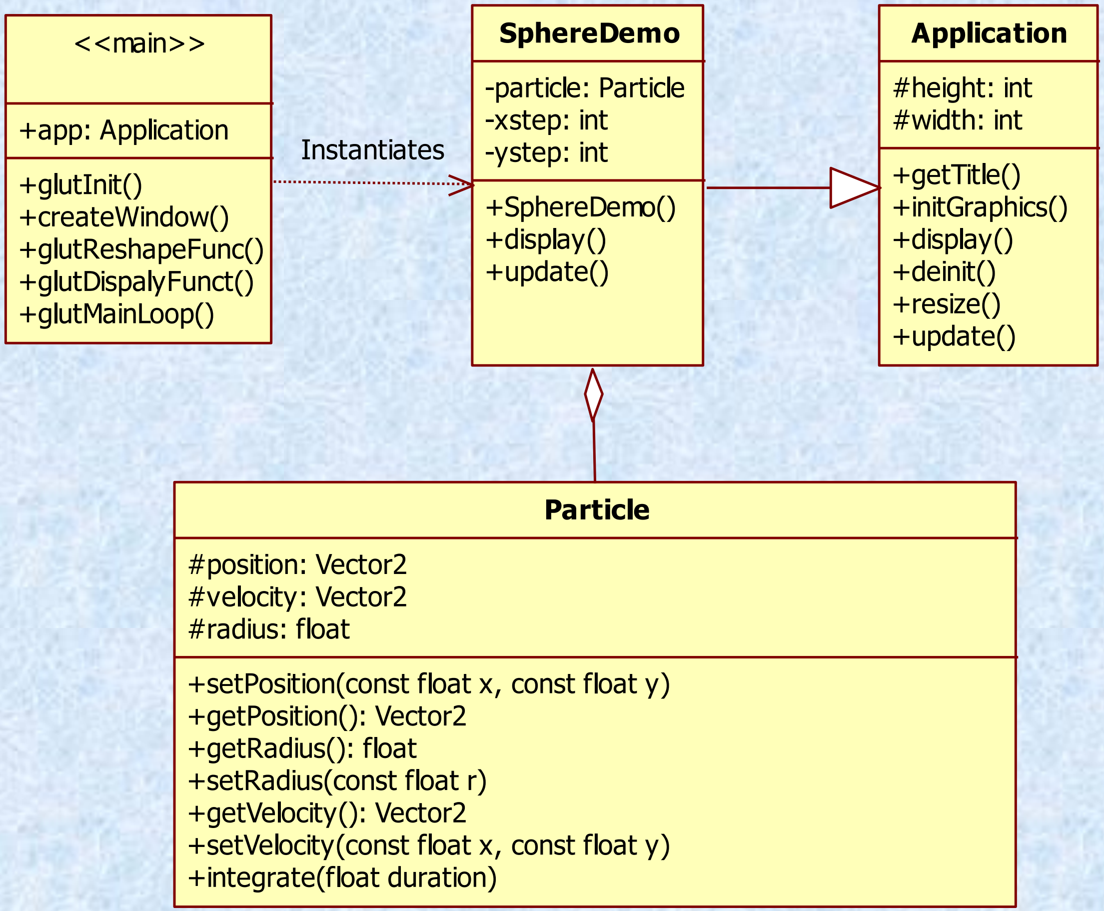
Position Euler Step

```
position += velocity * deltaTime;
```

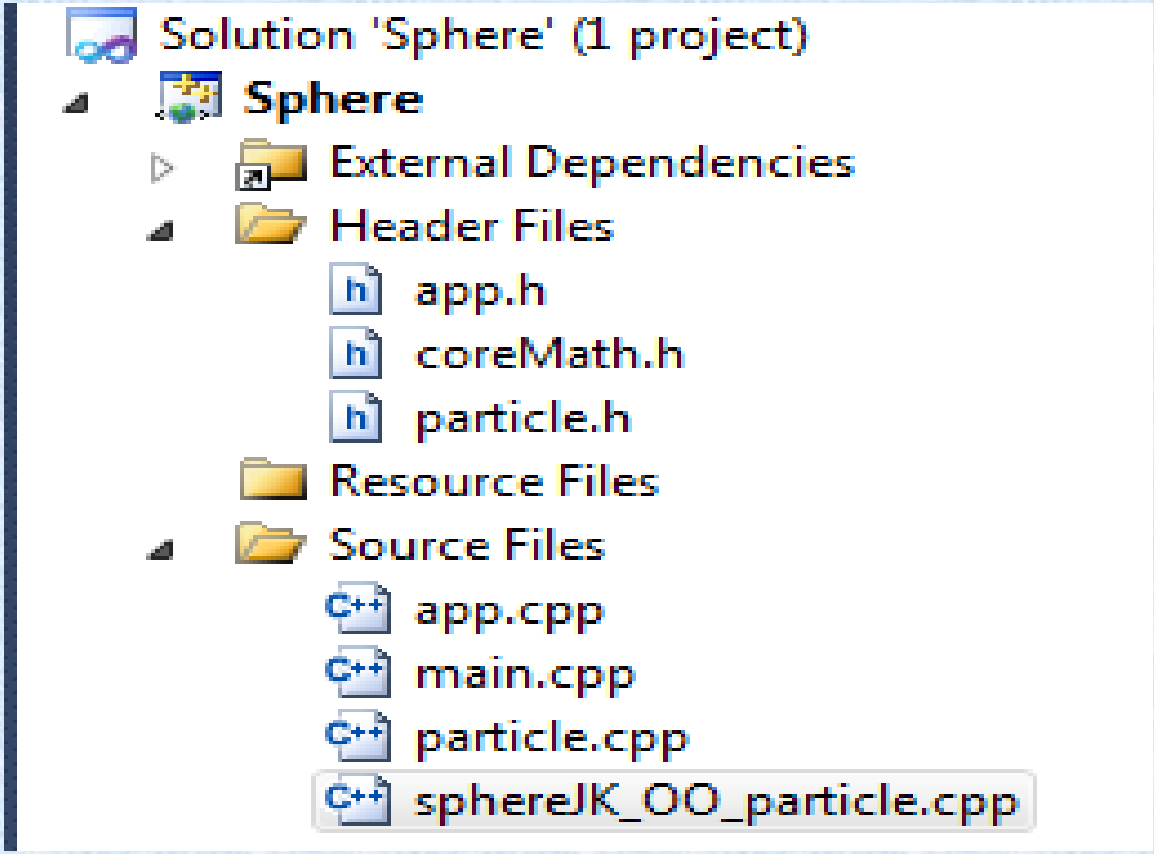
Tutorial – particle class



Tutorial – particle class



Tutorial – particle class



Tutorial – coreMath.h

```
class Vector2
    {
    public:
         /** Holds the value along the x axis. */
        float x;

        /** Holds the value along the y axis. */
        float y;


    public:
        /** The default constructor creates a zero vector
        Vector2() : x(0), y(0) {}

      /** Adds the given vector to this. */
      void operator+=(const Vector2& v)
      {
          x += v.x;
          y += v.y;
      }

     /** Subtracts the given vector from this. */
     void operator-=(const Vector2& v)
     {
         x -= v.x;
         y -= v.y;
     }
```

```
/** Multiplies this vector by the given scalar. */
void operator*=(const float value)
{
    x *= value;
    y *= value;
}
```

```
/** Gets the magnitude of this vector. */
float magnitude() const
{
    return sqrt(x*x+y*y);
}
```

```
/** Turns a non-zero vector into a vector of unit length. */
void normalise()
{
    float l = magnitude();
    if (l > 0)
    {
        (*this) *= ((float)1)/l;
    }
}
```

Tutorial – particle class

```
#ifndef PARTICLE_H
#define PARTICLE_H

#include "coreMath.h"

    class Particle
    {
    protected:
     Vector2 position;
     Vector2 velocity;
     float radius;

      public:
        void setPosition(const float x, const float y);
        void setRadius(const float r);
        Vector2 getPosition() const;
        float getRadius() const;
        void integrate(float duration);

        void setVelocity(const float x, const float y);
        Vector2 getVelocity() const;

       };
```

Tutorial – particle class

```cpp
#include "particle.h"
#include <math.h>

void Particle::setPosition(const float x, const float y)
{
    position.x = x;
    position.y = y;
}

void Particle::setRadius(const float r)
{
    radius = r;
}

Vector2 Particle::getPosition() const
{
    return position;
}

float Particle::getRadius() const
{
    return radius;
}
```

Tutorial – particle class

```
class SphereDemo : public Application
{
int xstep;
int ystep;
Particle particle;

public:
    SphereDemo();
    virtual void display();
    virtual void update();
};

SphereDemo::SphereDemo()
{
particle.setPosition(0,0);

particle.setVelocity(100,101);


particle.setRadius(10);
xstep = 2;
ystep = 2;
width = 600;
height = 600;
}
```

Tutorial – particle class

Particle | setPosition(const float

```cpp
void Particle::integrate(float duration)
{

position += velocity*duration;

}

void Particle::setVelocity(const float x, const float y)
{
    velocity.x = x;
    velocity.y = y;
}

Vector2 Particle::getVelocity() const
{
    return velocity;
}
```

Tutorial – particle class

```
void SphereDemo::update(void)
{

    float radius = particle.getRadius();
    float duration = timeinterval/1000;

    particle.integrate(duration);

    Vector2 position = particle.getPosition();
    Vector2 velocity = particle.getVelocity();

    // Reverse direction when you reach left or right edge
    if(position.x> Application::width-radius || position.x < -Application::width + radius)
        particle.setVelocity(-velocity.x,velocity.y);

    // Reverse direction when you reach top or bottom edge
    if(position.y > Application::height -radius|| position.y < -Application::height + radius)
        particle.setVelocity(velocity.x,-velocity.y);
```

Further improvements?

```
void SphereDemo::update(void)
{

    float radius = particle.getRadius();
    float duration = timeinterval/1000;

    particle.integrate(duration);

    Vector2 position = particle.getPosition();
    Vector2 velocity = particle.getVelocity();

    // Reverse direction when you reach left or right edge
    if(position.x> Application::width-radius || position.x < -Application::width + radius)
        particle.setVelocity(-velocity.x,velocity.y);

    // Reverse direction when you reach top or bottom edge
    if(position.y > Application::height -radius|| position.y < -Application::height + radius)
        particle.setVelocity(velocity.x,-velocity.y);
```

Further improvement?

```
void SphereDemo::updateClean(void)
{
    float radius = particle.getRadius();
    float duration = timeinterval/1000;

    particle.integrate(duration);

    Vector2 position = particle.getPosition();
    Vector2 velocity = particle.getVelocity();

    // Reverse direction when you reach left or right edge
    if(position.x> Application::width-radius || position.x < -Application::width + radius)
        particle.setVelocity(-velocity.x,velocity.y);

    // Reverse direction when you reach top or bottom edge
    if(position.y > Application::height -radius|| position.y < -Application::height + radius)
        particle.setVelocity(velocity.x,-velocity.y);

    if(position.x > (Application::width - radius))
            position.x = Application::width - radius;
    else if(position.x < -Application::width + radius)
            position.x = -Application::width + radius ;

    if(position.y > (Application::height - radius))
           position.y = Application::height - radius;
    else if(position.y < -Application::height + radius)
           position.y = -Application::height + radius;
```

Further improvement

SphereDemo | SphereDemo()

```
// sphereJK_OO.cpp
// single particle + simple physics


#include <gl/glut.h>// OpenGL toolkit
#include <app.h>// OpenGL toolkit
#include "particle.h"

class SphereDemo : public Application
{
Particle particle;
float AccTime;
float FallTime;
float Max_velocity;

public:
    SphereDemo();
    virtual void display();
    virtual void update();
    void box_collision_resolve(Particle &particle);
    bool out_of_box_test(Particle particle);
    void out_of_box_resolve(Particle &particle);

};
```

Further improvement

```
void SphereDemo::update(void)
{

    float radius = particle.getRadius();
    float duration = timeinterval/1000;
    AccTime += duration;

    particle.integrate(duration);
    box_collision_resolve(particle);
    if(out_of_box_test(particle)) out_of_box_resolve(particle);


    Application::update();
}
```

Further improvement

```
void SphereDemo::update(void)
{

    float radius = particle.getRadius();
    float duration = timeinterval/1000;
    AccTime += duration;

    particle.integrate(duration);
    box_collision_resolve(particle);
    if(out_of_box_test(particle)) out_of_box_resolve(particle);


    Application::update();
}
```

Tutorial – particle class

Particle ▾ | setPosition(const float

```
void Particle::integrate(float duration)
{

position += velocity*duration;

}

void Particle::setVelocity(const float x, const float y)
{
    velocity.x = x;
    velocity.y = y;
}

Vector2 Particle::getVelocity() const
{
    return velocity;
}
```

Further improvement

```
void SphereDemo::update(void)
{

    float radius = particle.getRadius();
    float duration = timeinterval/1000;
    AccTime += duration;

    particle.integrate(duration);
    box_collision_resolve(particle);
    if(out_of_box_test(particle)) out_of_box_resolve(particle);


    Application::update();
}
```

Further improvement

```
// detect if the particle colided with the box and produce a response
void SphereDemo::box_collision_resolve(Particle &particle)
{
    Vector2 position = particle.getPosition();
    Vector2 velocity = particle.getVelocity();
    float radius = particle.getRadius();

    float w = Application::width;
    float h = Application::height;

    // Reverse direction when you reach left or right edge
    if(position.x> w-radius || position.x < -w + radius)
        particle.setVelocity(-velocity.x,velocity.y);

    if(position.y > h -radius|| position.y < -h + radius)
        particle.setVelocity(velocity.x,-velocity.y);
}
```

Further improvement

```
void SphereDemo::update(void)
{

    float radius = particle.getRadius();
    float duration = timeinterval/1000;
    AccTime += duration;

    particle.integrate(duration);
    box_collision_resolve(particle);
    if(out_of_box_test(particle)) out_of_box_resolve(particle);


    Application::update();
}
```

Further improvement

```
bool SphereDemo::out_of_box_test(Particle particle)
{
    Vector2 position = particle.getPosition();
    Vector2 velocity = particle.getVelocity();
    float radius = particle.getRadius();
    if ((position.x > Application::width-radius) ||  (position.x < -Application::width  + radius)) return true;
    if ((position.y > Application::height-radius) || (position.y < -Application::height + radius)) return true;

    return false;
}
```

Further improvement

```
void SphereDemo::update(void)
{

    float radius = particle.getRadius();
    float duration = timeinterval/1000;
    AccTime += duration;

    particle.integrate(duration);
    box_collision_resolve(particle);
    if(out_of_box_test(particle)) out_of_box_resolve(particle);


    Application::update();
}
```

Further improvement

```
void SphereDemo::out_of_box_resolve(Particle &particle)
{
    Vector2 position = particle.getPosition();
    Vector2 velocity = particle.getVelocity();
    float radius = particle.getRadius();


    if(position.x > Application::width - radius)        position.x =  Application::width-radius;
    else if(position.x < -Application::width + radius)  position.x = -Application::width + radius;

    if(position.y > Application::height - radius)        position.y = Application::height-radius;
    else if(position.y < -Application::height + radius)  position.y = -Application::height + radius;

    particle.setPosition(position.x,position.y);
}
```