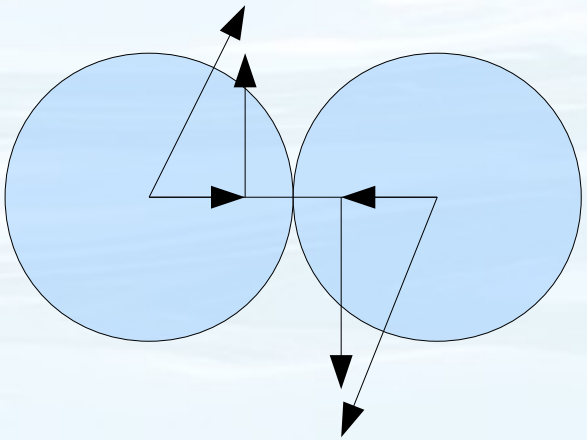
COMP1110 Assignment Crash Test Simulation

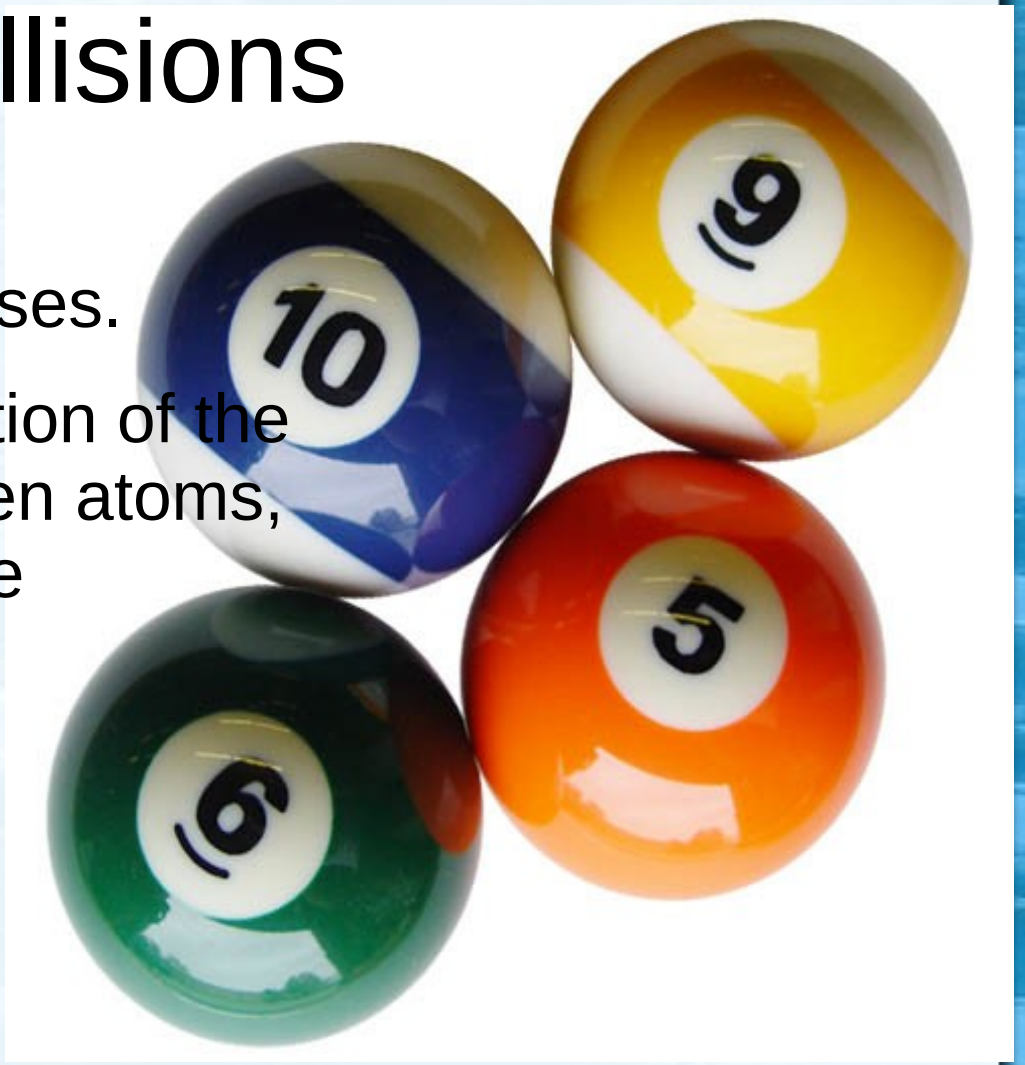
The Matter class and collision handling

- Matter is general definition for “atom” types
- Defined as collections of properties
- Mass, velocity, acceleration, position.
- Physics of collisions defined in this class in form of a method that updates the velocity of the particle.

Collisions

- All elastic
- Handles different masses.
- Algorithm finds projection of the velocity on line between atoms, then treats as one-Dee collision.





Collisions

- The solution to final velocities in one-Dee using conservation of momentum and kinetic energy (image courtesy of wikipedia).

$$v_2 = \frac{u_2(m_2 - m_1) + 2m_1u_1}{m_1 + m_2}$$

$$v_1 = \frac{u_1(m_1 - m_2) + 2m_2u_2}{m_1 + m_2}$$

The Bonded Atom

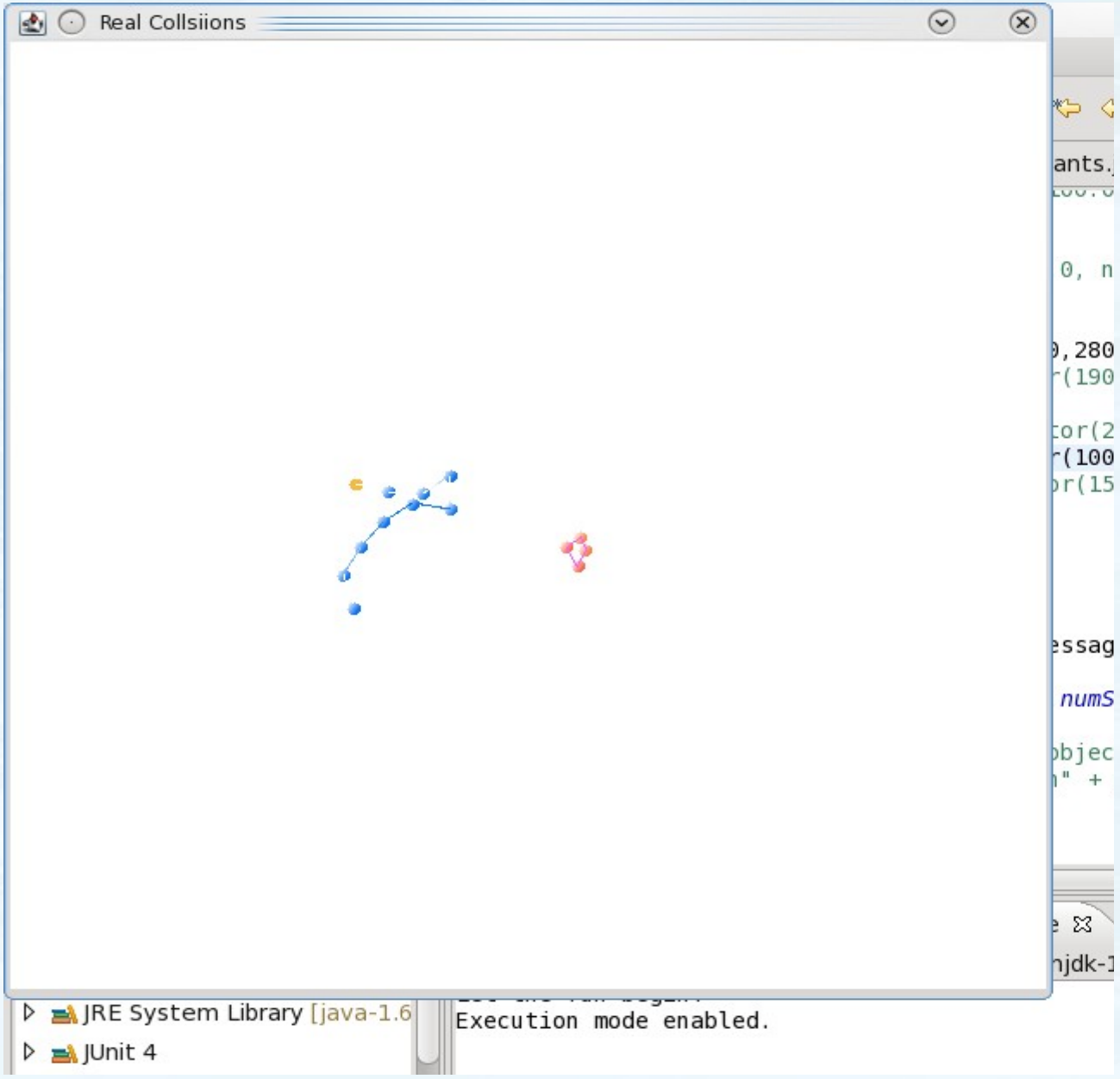
- Has a list of bonds
- Roughly approximates kinetic theory of matter.
- Bonds act like *Damped* springs, and break when they exceed certain length

The Bonded Atom: problems

- Initial bonds had no damping, caused *explosions* – a phenomena we were to become all too familiar with over the course of the term.
- Fixed (approximately) by bond damping.
- Bond accelerations altered centre of mass.
- Fixed by timeResolution and order of the execution of main.

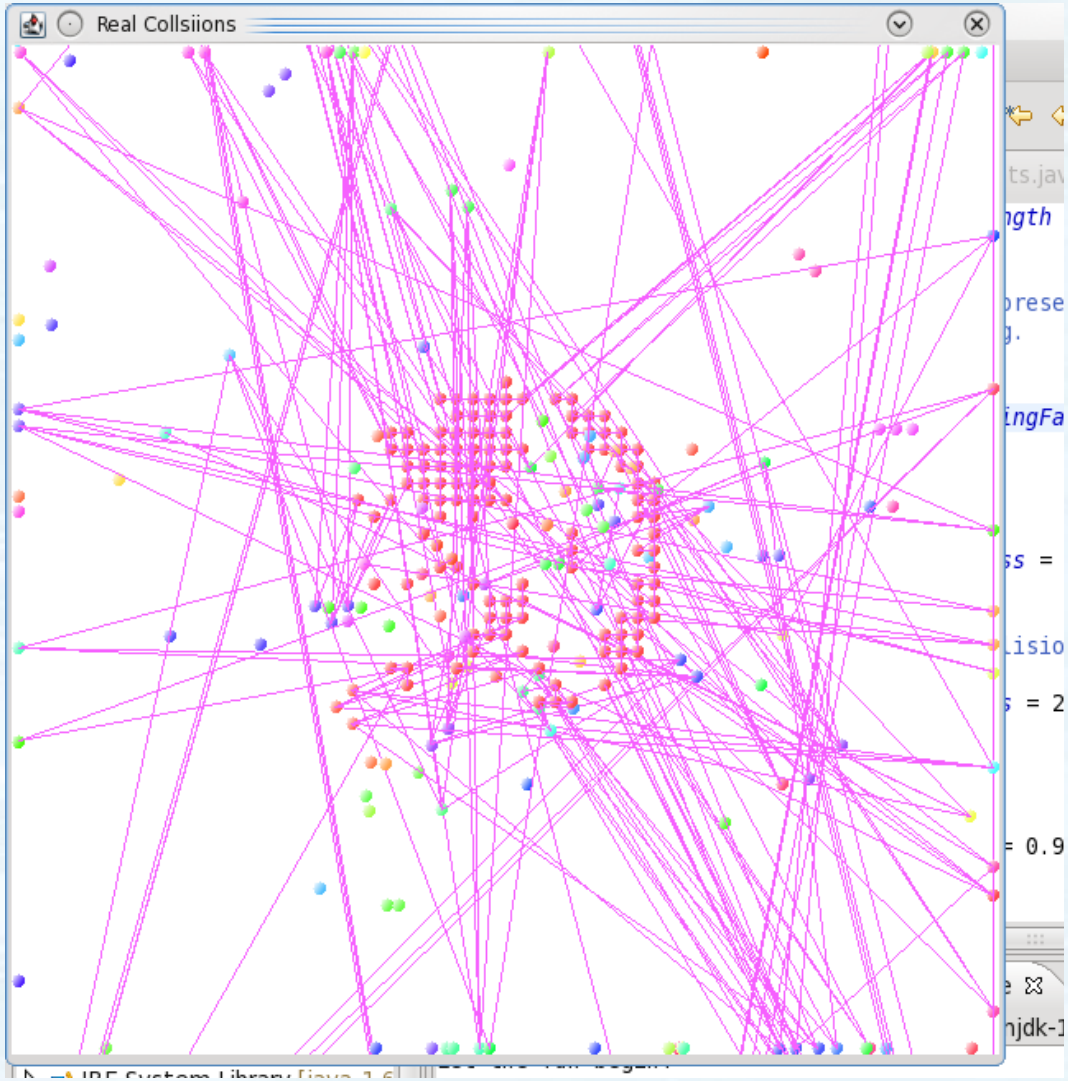
The Chain

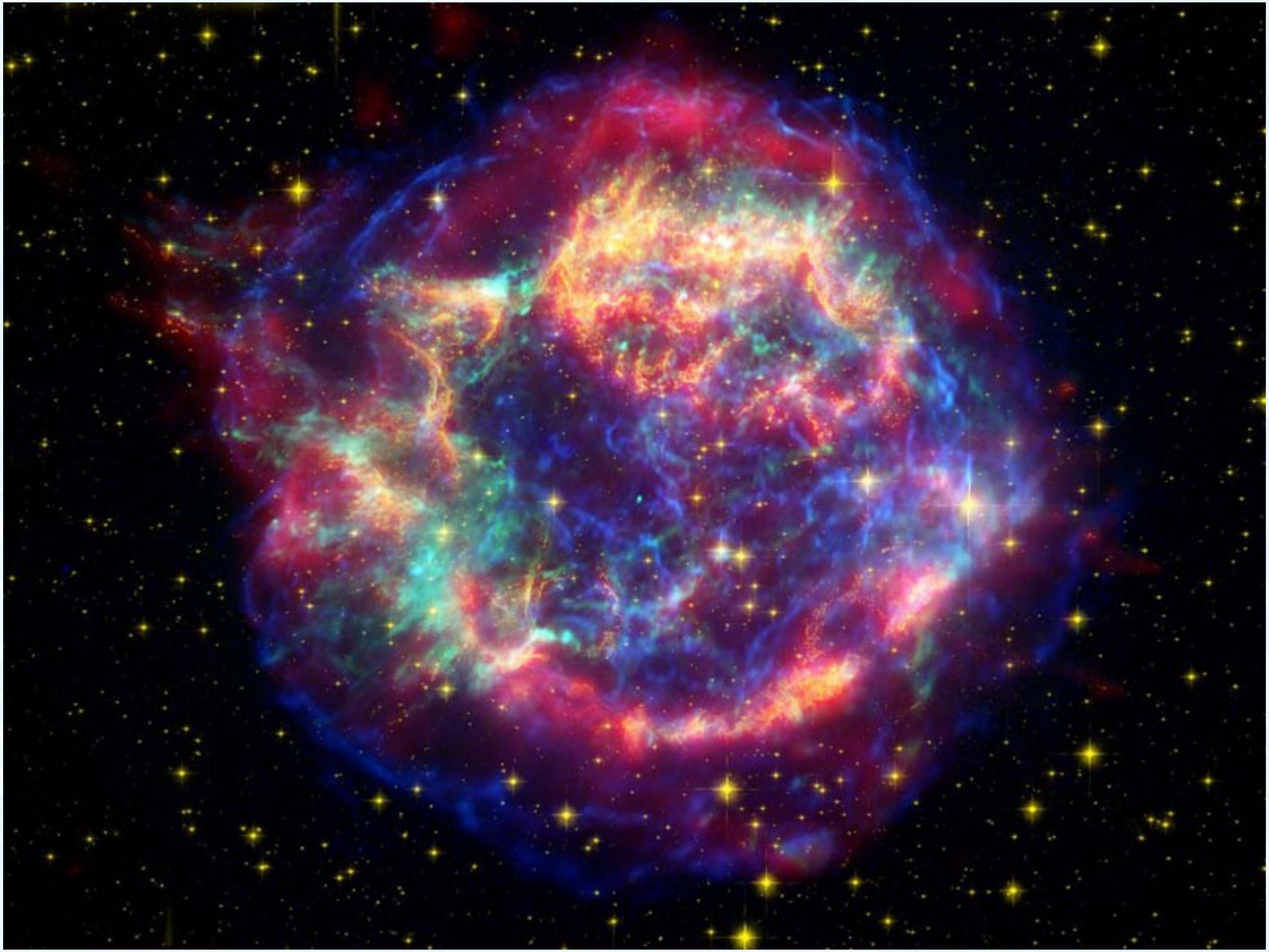
- A chain is a “Doubly linked list” of atoms.
- Starting with a Pivot, Another type of matter.
- The bonds differ to bonded Atom
- -never break, don't stretch (much)
- Particles very heavy


Real Collsiions
JRE System Library [java-1.6
JUnit 4
Execution mode enabled.

Construction: the world class

- Combinations of the atoms had to be made.
- Stored in an arrayList
- Circles
- Polygons
- Difficulty: each atom had to know what index of ArrayList to bond to.

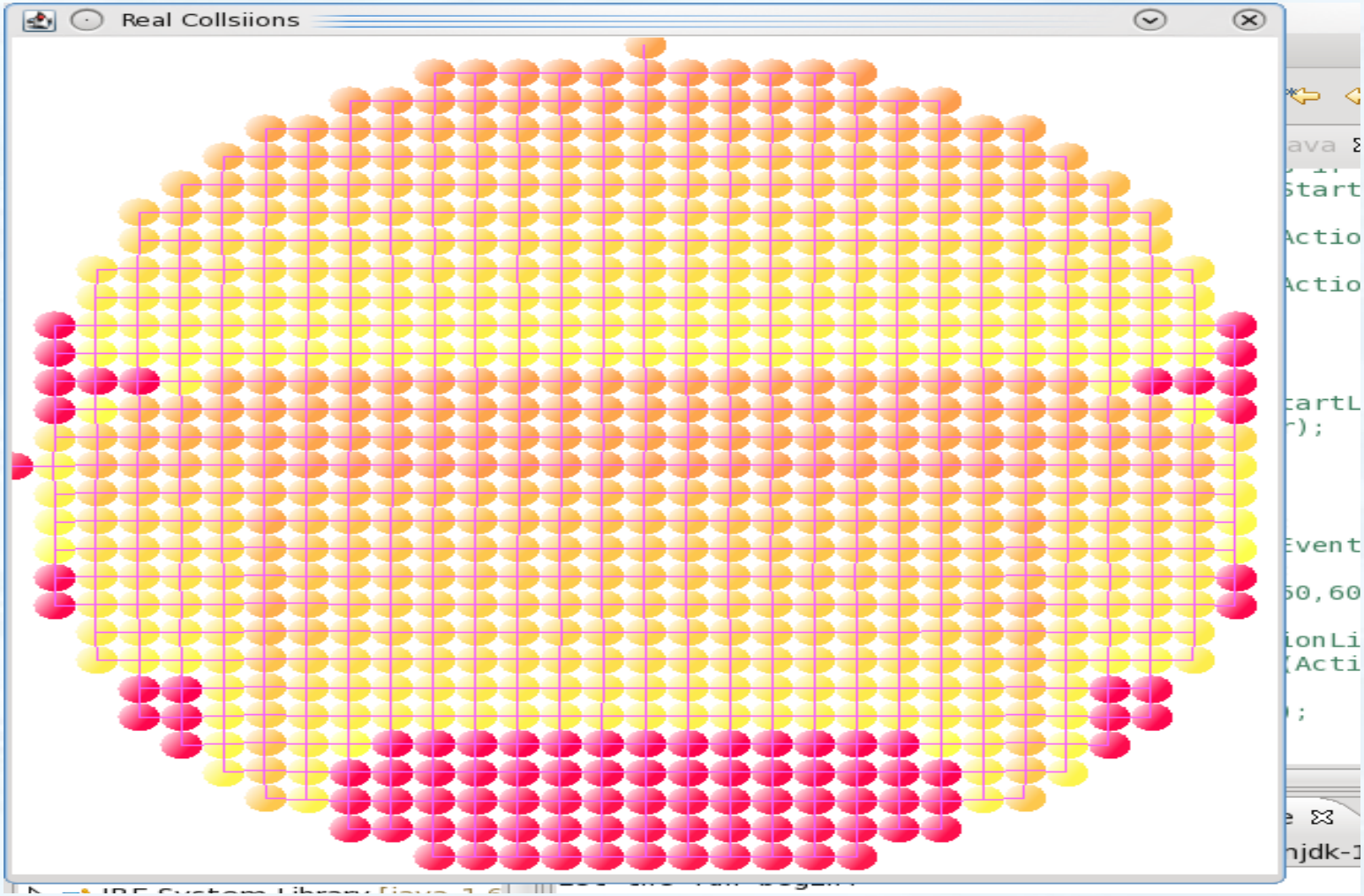

Real Collsiions

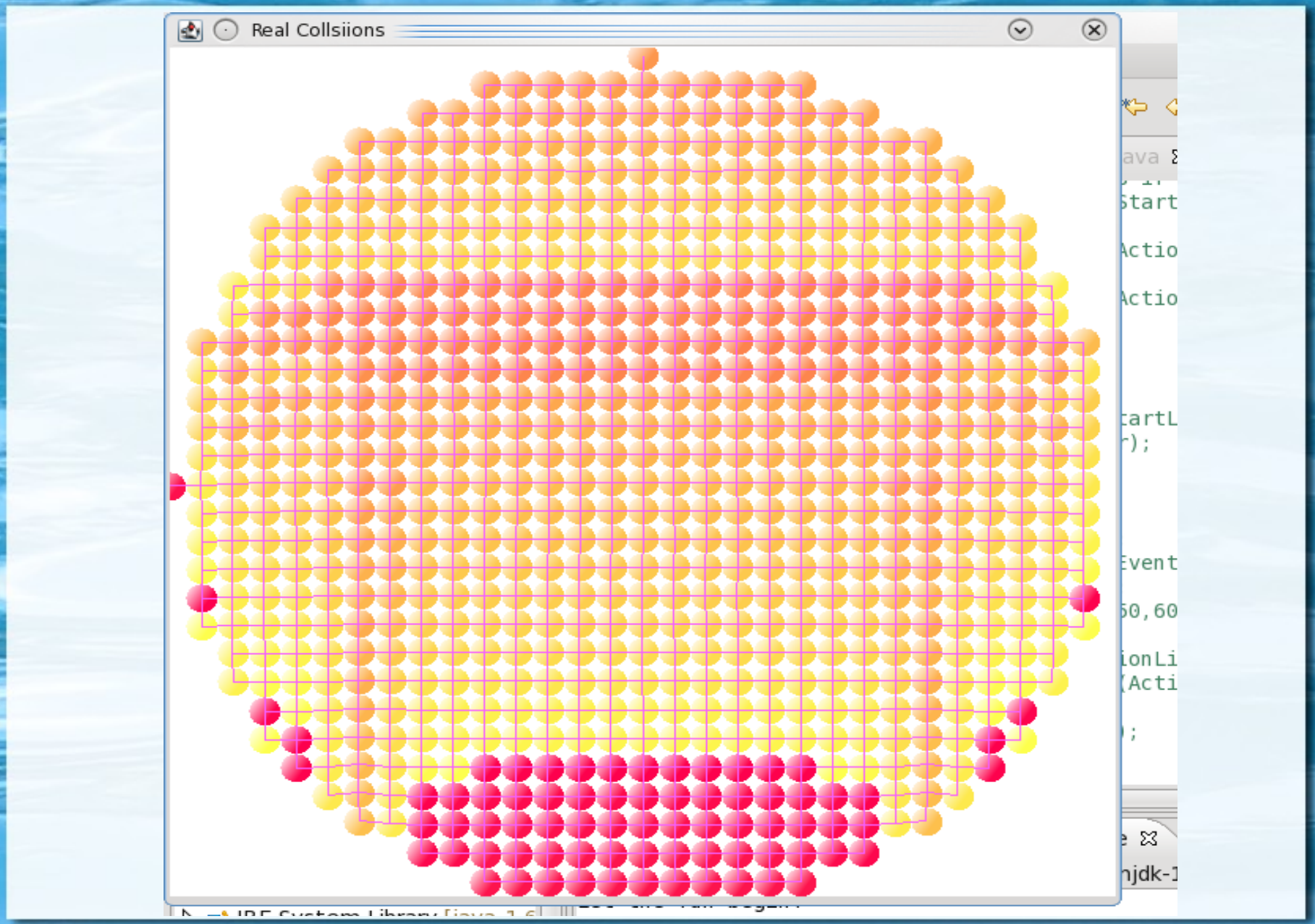


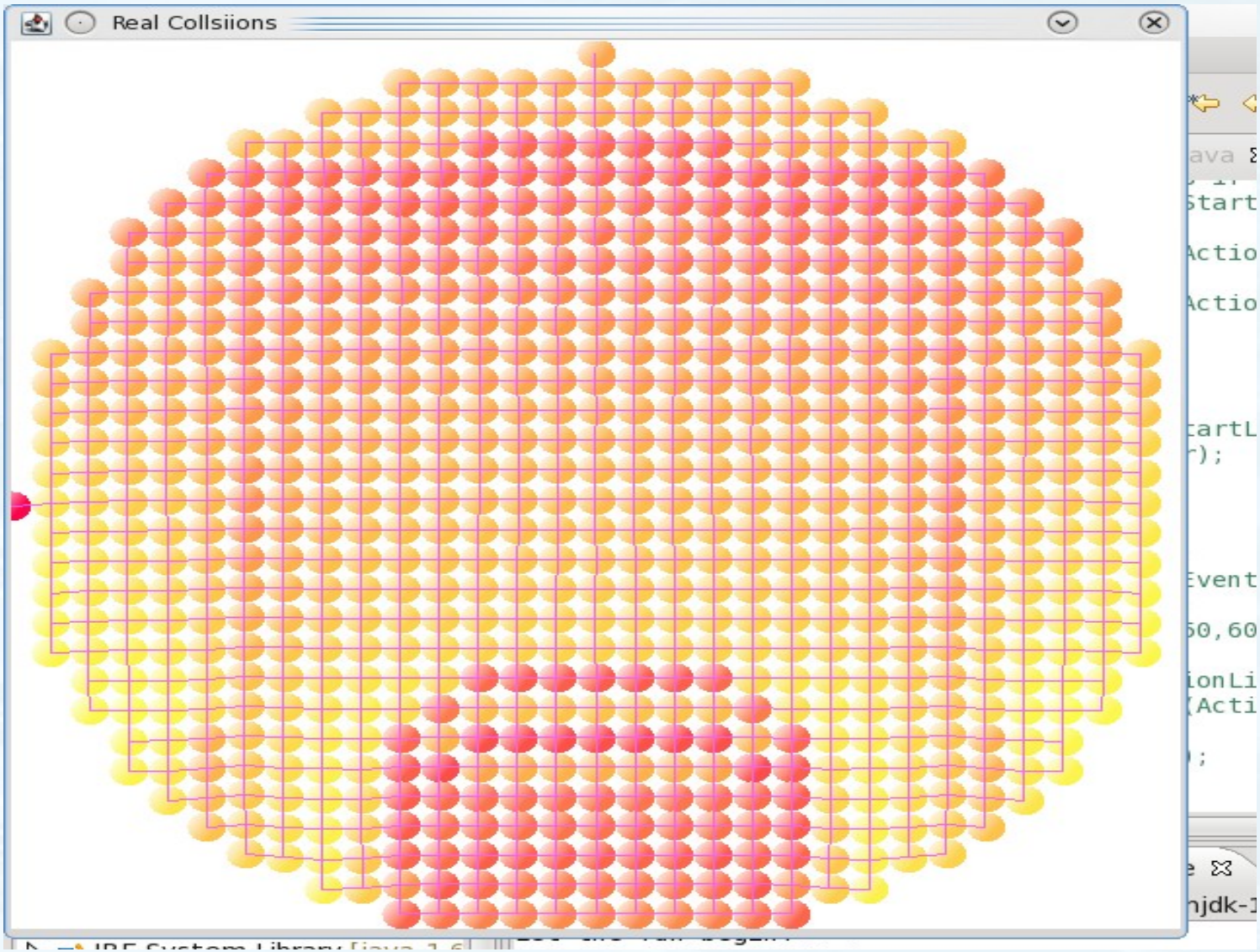
The awesome Graphics and their physical significance

- Each atom was coloured proportional to its velocity.
- Red tended to be slower.
- Blue faster.
- Additional lighting added to simulate light source in upper left corner.
- Allows us to see stress patterns in objects.

Stress patterns




Real Collsiions

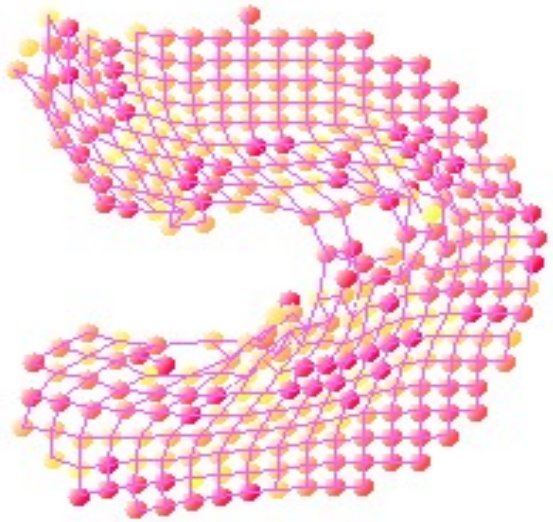

Real Collsiions

Consequences of the approximations

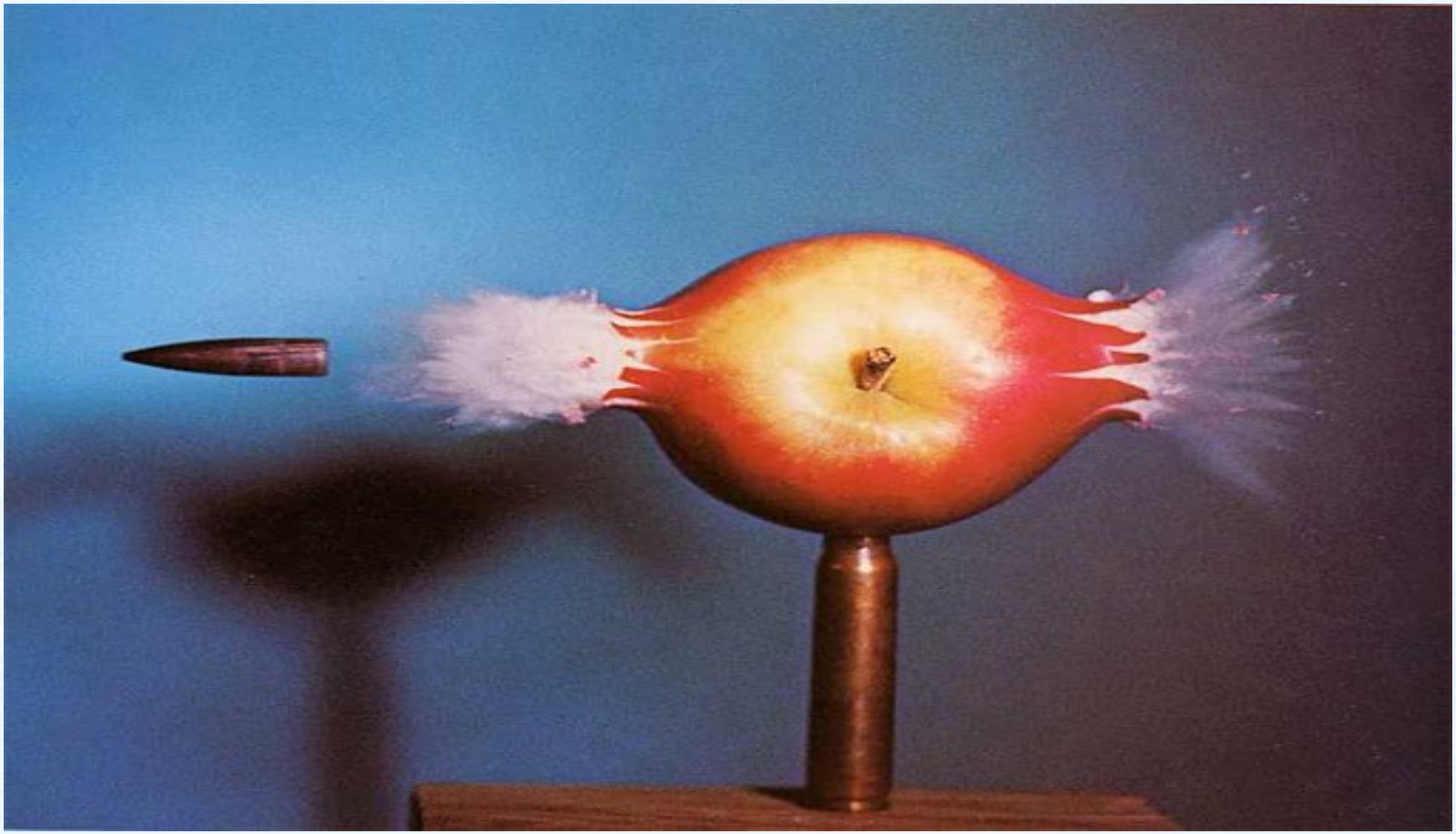
- *Jitter* -apparently random vibrations in Bonded Atoms.
- Damping fixed some.
- TimeResolution fixes.
- Need infinite Time Resolution.

Success

- Despite issues approximately simulates some types of matter accurately
- Some examples



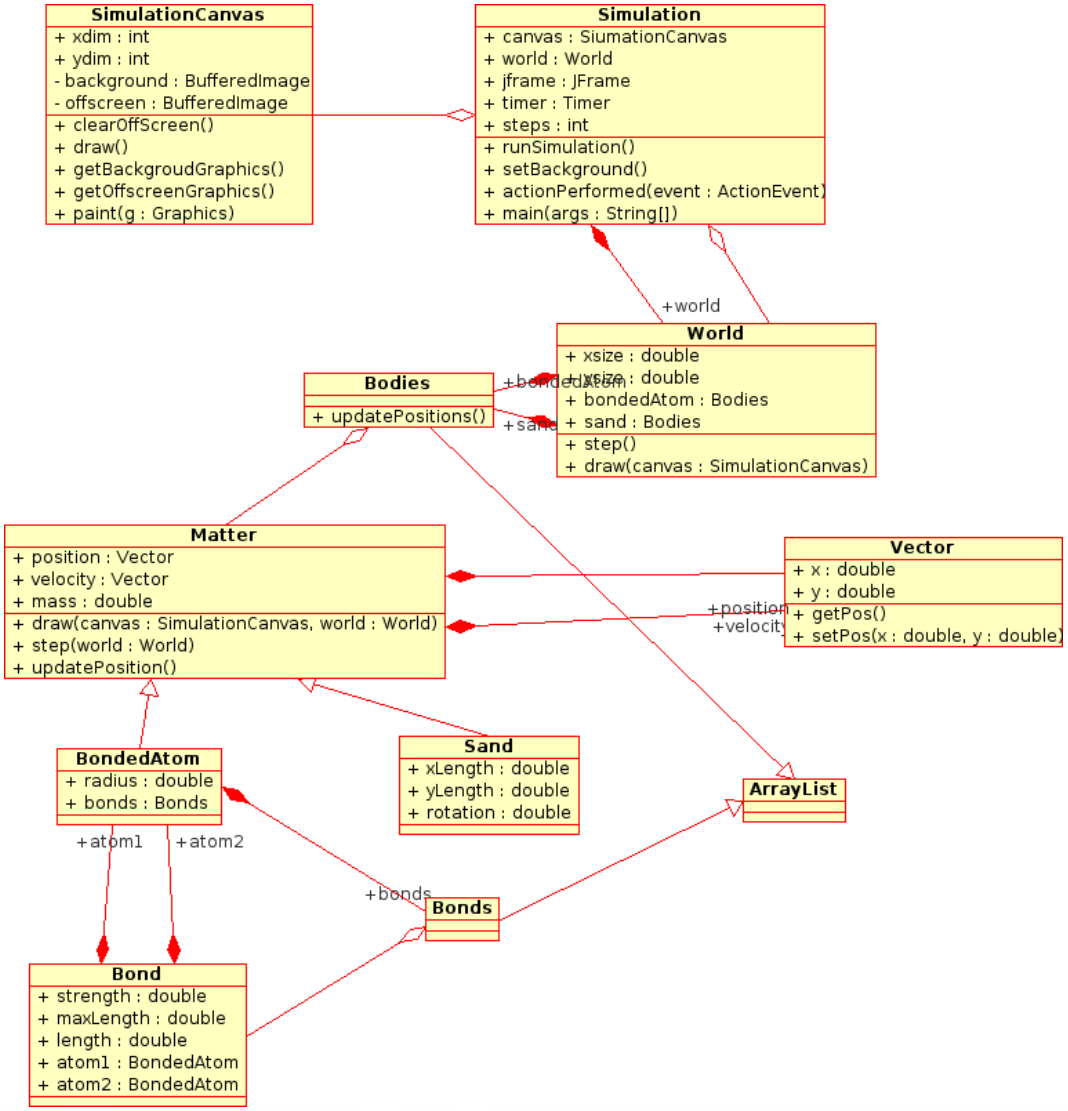
The Real thing



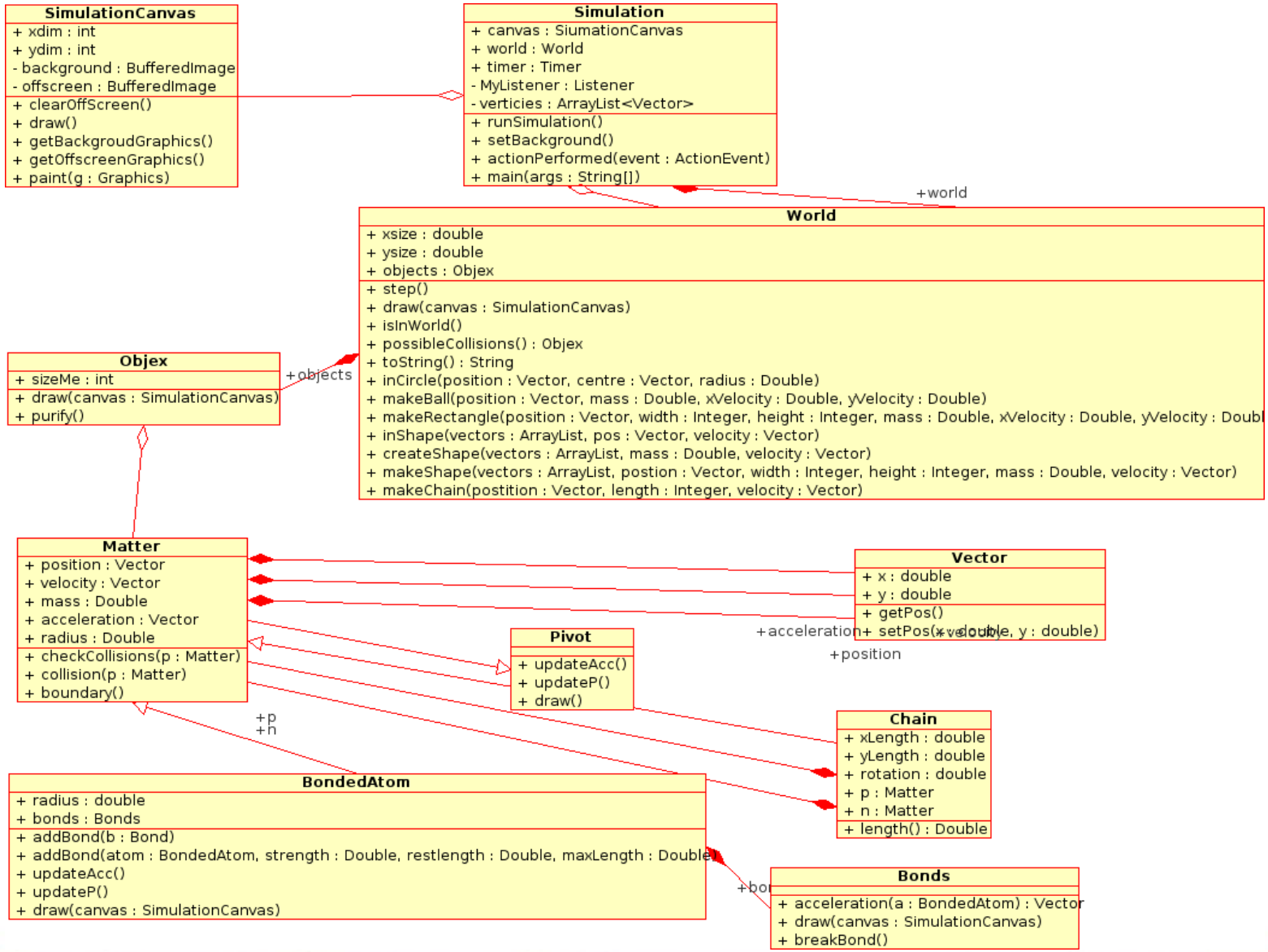
Design

- We ended up stick approximately to our original UML design, though it's definately safe to say we added a great many more methods to certain classes once it became apparent what was needed.
- We also added a few more classes – notably replacing 'Sand' with 'Pivot' and 'Chain'.

Before


SimulationCanvas
+ xdim : int
+ ydim : int
- background : BufferedImage
- offscreen : BufferedImage
+ clearOffScreen()
+ draw()
+ getBackgroudGraphics()
+ getOffscreenGraphics()
+ paint(g : Graphics)
Simulation
+ canvas : SiumationCanvas
+ world : World
+ jframe : JFrame
+ timer : Timer
+ steps : int
+ runSimulation()
+ setBackground()
+ actionPerformed(event : ActionEvent)
+ main(args : String[])
+world
World
+ xsize : double
+ ysize : double
+ bondedAtom : Bodies
+ sand : Bodies
+ step()
+ draw(canvas : SimulationCanvas)
Bodies
+ updatePositions()
Matter
+ position : Vector
+ velocity : Vector
+ mass : double
+ draw(canvas : SimulationCanvas, world : World)
+ step(world : World)
+ updatePosition()
Vector
+ x : double
+ y : double
+ getPos()
+ setPos(x : double, y : double)
+position
+velocity
BondedAtom
+ radius : double
+ bonds : Bonds
+atom1
+atom2
Sand
+ xLength : double
+ yLength : double
+ rotation : double
ArrayList
+bonds
Bonds
Bond
+ strength : double
+ maxLength : double
+ length : double
+ atom1 : BondedAtom
+ atom2 : BondedAtom

After


SimulationCanvas
+ xdim : int
+ ydim : int
- background : BufferedImage
- offscreen : BufferedImage
+ clearOffScreen()
+ draw()
+ getBackgroudGraphics()
+ getOffscreenGraphics()
+ paint(g : Graphics)
Simulation
+ canvas : SiumationCanvas
+ world : World
+ timer : Timer
- MyListener : Listener
- verticies : ArrayList<Vector>
+ runSimulation()
+ setBackground()
+ actionPerformed(event : ActionEvent)
+ main(args : String[])
+world
World
+ xsize : double
+ ysize : double
+ objects : Objex
+ step()
+ draw(canvas : SimulationCanvas)
+ isInWorld()
+ possibleCollisions() : Objex
+ toString() : String
+ inCircle(position : Vector, centre : Vector, radius : Double)
+ makeBall(position : Vector, mass : Double, xVelocity : Double, yVelocity : Double)
+ makeRectangle(position : Vector, width : Integer, height : Integer, mass : Double, xVelocity : Double, yVelocity : Doubl
+ inShape(vectors : ArrayList, pos : Vector, velocity : Vector)
+ createShape(vectors : ArrayList, mass : Double, velocity : Vector)
+ makeShape(vectors : ArrayList, postion : Vector, width : Integer, height : Integer, mass : Double, velocity : Vector)
+ makeChain(postition : Vector, length : Integer, velocity : Vector)
Objex
+ sizeMe : int
+ draw(canvas : SimulationCanvas)
+ purify()
+objects
Matter
+ position : Vector
+ velocity : Vector
+ mass : Double
+ acceleration : Vector
+ radius : Double
+ checkCollisions(p : Matter)
+ collision(p : Matter)
+ boundary()
Vector
+ x : double
+ y : double
+ getPos()
+ setPos(x : double, y : double)
+acceleration
+position
Pivot
+ updateAcc()
+ updateP()
+ draw()
+p
+n
Chain
+ xLength : double
+ yLength : double
+ rotation : double
+ p : Matter
+ n : Matter
+ length() : Double
BondedAtom
+ radius : double
+ bonds : Bonds
+ addBond(b : Bond)
+ addBond(atom : BondedAtom, strength : Double, restlength : Double, maxLength : Double)
+ updateAcc()
+ updateP()
+ draw(canvas : SimulationCanvas)
Bonds
+ acceleration(a : BondedAtom) : Vector
+ draw(canvas : SimulationCanvas)
+ breakBond()

Scaling

We found that our program used an O(n^2) algorithm, due to the nesessity of comparing each element against every other element in order to detect collisions.

More Scaling

We tried to reduce the number of comparisons nessecary by implementing what is known as a QuadTree.

A QuadTree works by recursively partitioning a space into four equal quadrants, and placing elements in each, so that you only need to compare against elements in the same quadrant.

A Quadtree containing 1000 objects. (9 levels, leaf size 8×8px)

And Yet More Scaling

While this method is still O(n^2), since you could theoretically cram all your elements into one quadrant, in practice it tends to improve performance by a factor of about 50 to 100.

Unfortunately, we were unable to complete the implementation in time.