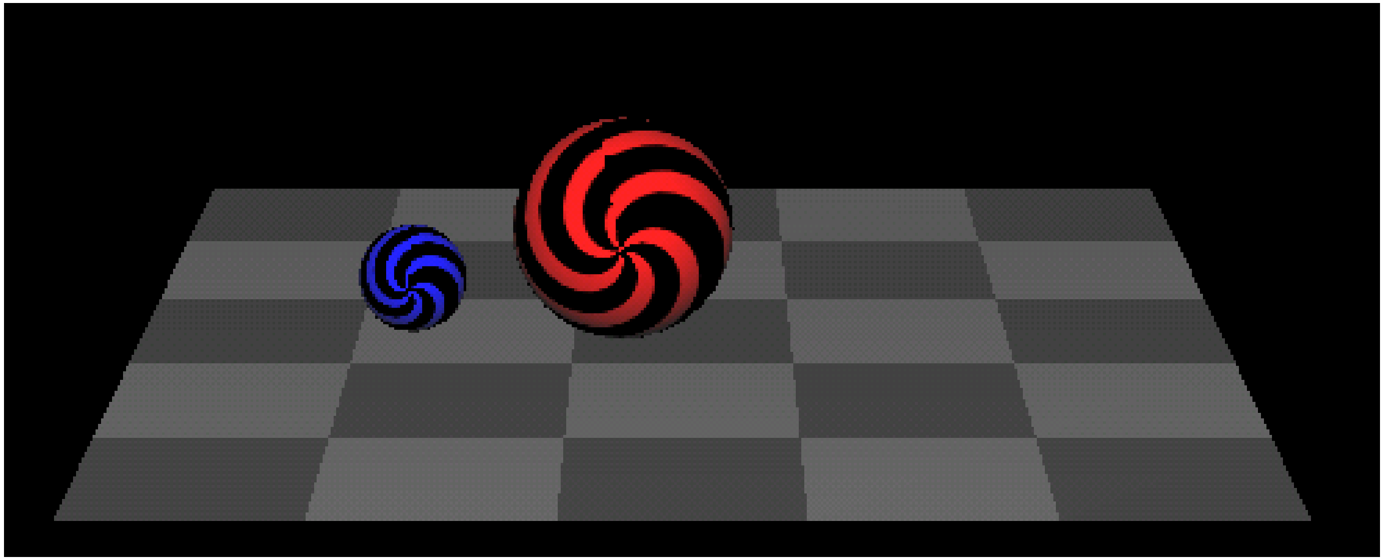
A Rigid-Body Simulator

Chris Van Horn

What’s a Rigid-Body Simulator?

- A rigid-body simulator calculates the positions and orientations of a group of rigid bodies that may collide with each other, over a given time span [4].
- A rigid body is an object, in this case a sphere, that experiences no deformations and no changes in size.



Demonstrations

- Bouncing Sphere
- Rolling Sphere/Sliding Sphere
- Collisions

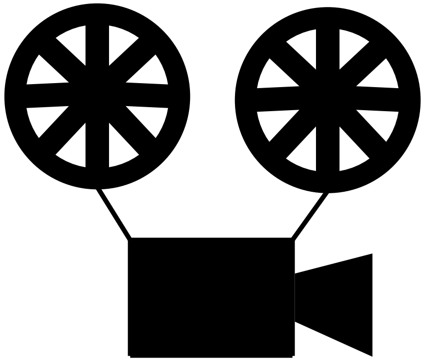
My Simulator Design

- Based on the concept of a film projector
- Each thread represents a major component of a film projector.

Simulation Thread filmstrip generation

Rendering Thread light source and lens

Input Thread projector control

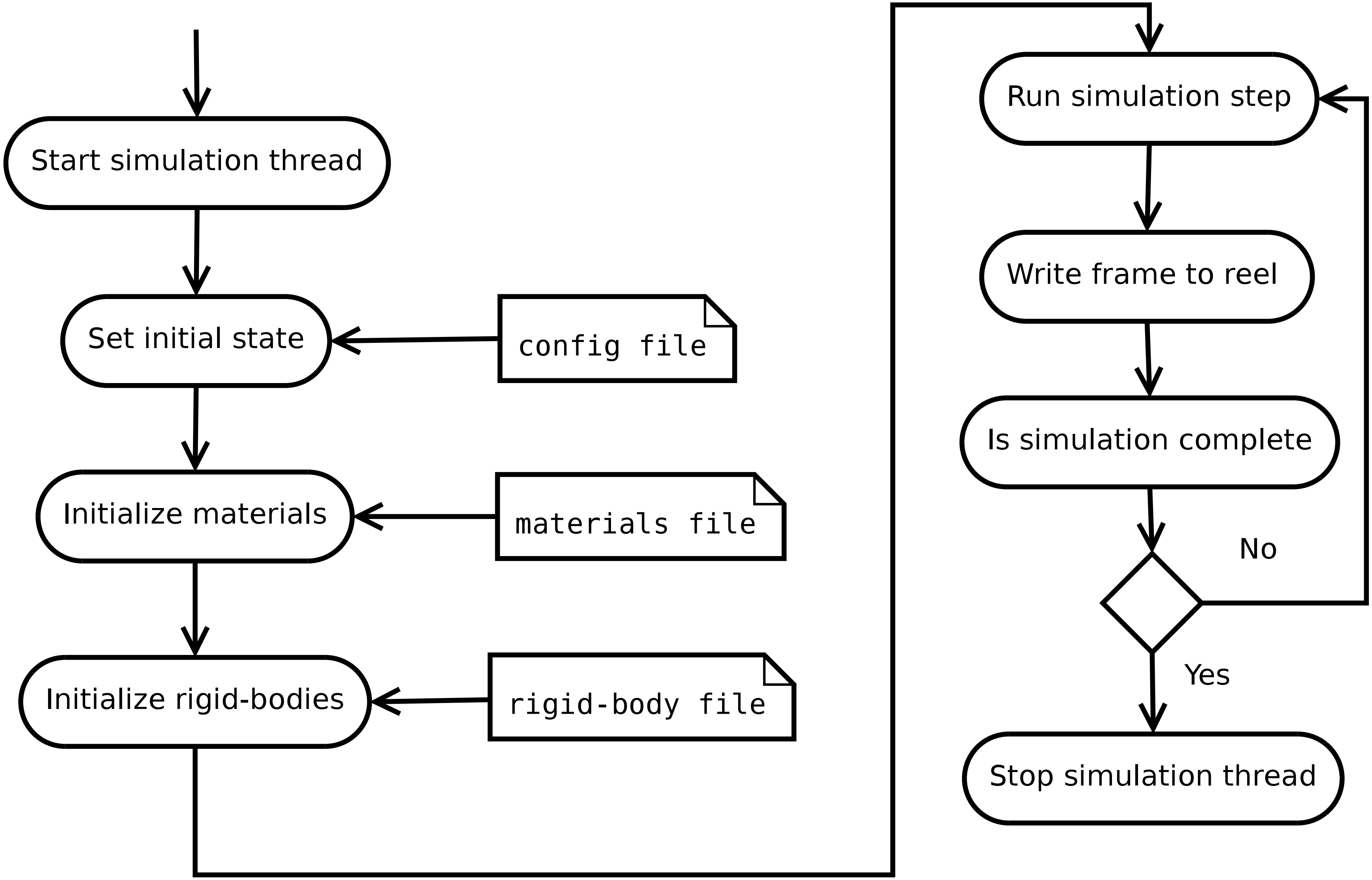


Simulation Thread

Steps taken by simulation thread:

1. Set initial state
2. Initialize materials
3. Initialize rigid bodies
4. Run simulation step
5. Write frame to reel
6. Check if simulation complete

Simulation Thread Flowchart



Start simulation thread

Set initial state

`config file`

Initialize materials

`materials file`

Initialize rigid-bodies

`rigid-body file`

Run simulation step

Write frame to reel

Is simulation complete

No

Yes

Stop simulation thread

Set Initial State

- Given its initial settings by the input thread
  - Number of time steps
  - Length of each time step
  - Gravity vector
  - Whether gravity is expressed as a force or an acceleration
  - If the simulation should pause on object collision
  - Location of the materials file and the rigid-body file
- The length of the simulation is determined by the number of time steps.

Initialize Materials

- Materials determine the friction of two objects when they collide.
- `materials.txt` contains the definitions of all materials.
- The materials are organized in a table.

Example Materials Table			
Material 1	Material 2	Static	Kinetic
Rubber	Rubber	1.5	1.0
Rubber	Concrete	1.0	0.5
Rubber	Steel	0.5	0.25

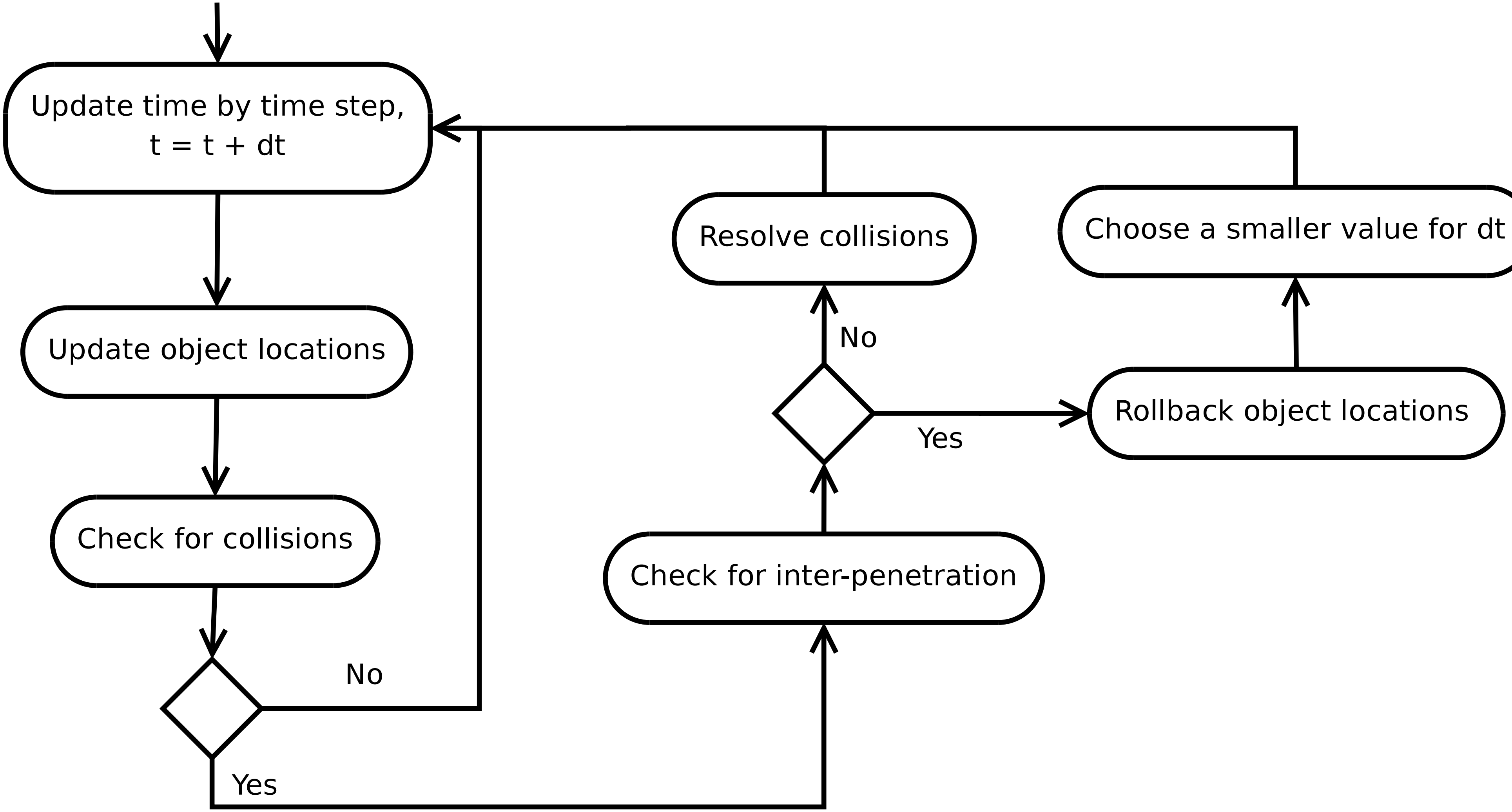
Initialize Rigid Bodies

- `objects.txt` defines the rigid bodies in the simulation.
- The following elements define a rigid body:
    - Type (sphere or plane)
    - Physical appearance (color and texture)
    - Initial position
    - Initial orientation
    - Type dependent attributes
    - Material composition
    - Center of mass
    - Initial force on the object

Run Simulation Loop

- Heart of the simulator
- Performs the following tasks for each iteration of the loop:
  1. Update simulation time
  2. Update each rigid body's location and orientation
  3. Check for and classify collisions
  4. Resolve collisions

Simulation Loop Flowchart



Update time by time step,
t = t + dt

Update object locations

Check for collisions

No

Yes

Check for inter-penetration

No

Yes

Resolve collisions

Rollback object locations

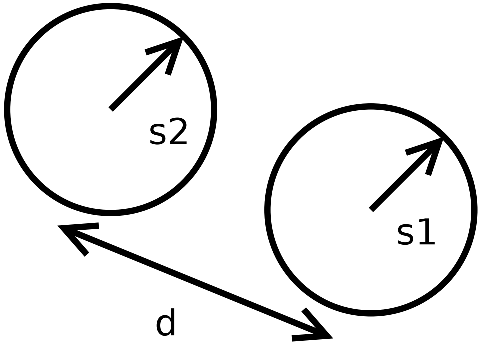
Choose a smaller value for dt

Update Locations and Orientations

- Derive initial-value problems using Newton's second law [3]
- Use $F = ma$ to derive an initial-value problem for updated velocity (v) and position (s)
  - $v_{t+\Delta t} = v_t + \frac{F}{m}\Delta t$, $v_{t=0} = v_0$
  - $s_{t+\Delta t} = s_t + v_{t+\Delta t}\Delta t$, $s_{t=0} = s_0$
- Use $M = I\omega$ to derive an initial-value problem for updated angular velocity (ω) and orientation (Ω)
  - $\omega_{t+\Delta t} = \omega_t + \frac{M}{I}\Delta t$, $\omega_{t=0} = \omega_0$
  - $\Omega_{t+\Delta t} = \Omega_t + \omega_{t+\Delta t}\Delta t$, $\Omega_{t=0} = \Omega_0$

Check for Collisions

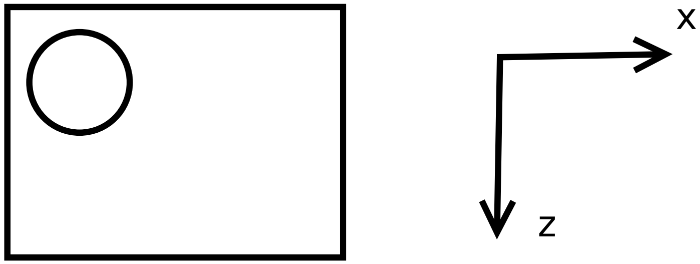
- Use the geometric properties of the objects to test for collisions
- Sphere - Sphere collision
  1. Compute distance (d) between the spheres
  2. Compare d to the sum of the spheres' radii (sr)
     - if $sr > d$, the spheres are overlapping
     - if $sr = d$, the spheres are touching
     - if $sr < d$, the spheres are not touching

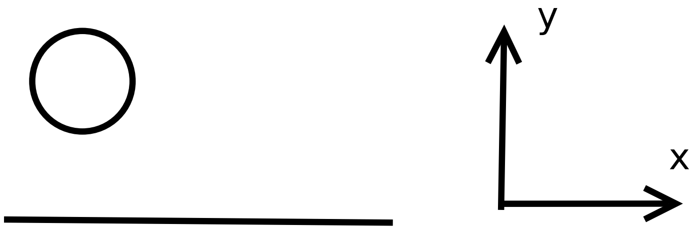


Check for Collisions, Continued

Sphere - Plane collision

1. Verify that the sphere is above the plane, using the plane normal
2. Compute the height of the sphere above the plane
3. Determine if the sphere and plane are
   - touching
   - not touching
   - overlapping





Check for Collisions, Continued

- Assume rigid-body a and rigid-body b are touching
- The relative velocity between rigid-body a and rigid-body b is v_r.
- The normal from rigid-body a to rigid-body b is N.
- We can classify the collision as follows [4]:
  - $v_r \cdot N < 0$, colliding contact
  - $v_r \cdot N = 0$, resting contact
  - $v_r \cdot N > 0$, separating contact

Resolve Collisions

- Use the definition of an impulse force (J) and the coefficient of restitution (R) to resolve the collisions [3]
  - $J = m(v_+ - v_-)$
  - $R = \frac{-(v_{a+} - v_{b+})}{v_{a-} - v_{b-}}$
- Rigid bodies involved experience an impulse force
- Rigid-body a receives a positive impulse, $J = m_a(v_{a+} - v_{a-})$.
- Rigid-body b receives a negative impulse, $-J = m_b(v_{b+} - v_{b-})$.

Resolve Collisions, Continued

- Use the impulse equations and the coefficient of restitution equation to calculate the results of the collision
- Solving for our three unknowns we have,
  - $J = \frac{-v_r(R+1)}{1/m_a+1/m_b}$
  - $v_{a+} = v_{a-} + \frac{Jn}{m_a}$
  - $v_{b+} = v_{b-} + \frac{-Jn}{m_b}$

Resolve Collisions, Continued

- Taking angular velocity and friction into account we have,
  - $J = \frac{-v_r(R+1)}{1/m_a + 1/m_b + (n \cdot (\frac{r_a \times n}{I_a} \times r_a)) + (n \cdot (\frac{r_b \times n}{I_b} \times r_b))}$
  - $v_{a+} = v_{a-} + \frac{Jn + \mu Jt}{m_1}$
  - $v_{b+} = v_{b-} + \frac{-Jn + \mu Jt}{m_2}$
  - $\omega_{a+} = \omega_{a-} + (r_a \times (Jn + \mu Jt))/I_a$
  - $\omega_{b+} = \omega_{b-} + (r_b \times (-Jn + \mu Jt))/I_b$
  - $t = (n \times v_r) \times n$
- Where n is the normal between the rigid bodies, I is the moment of inertia, μ is the coefficient of friction, and t is the unit vector tangent to the collision.

Resolve Collisions, Continued

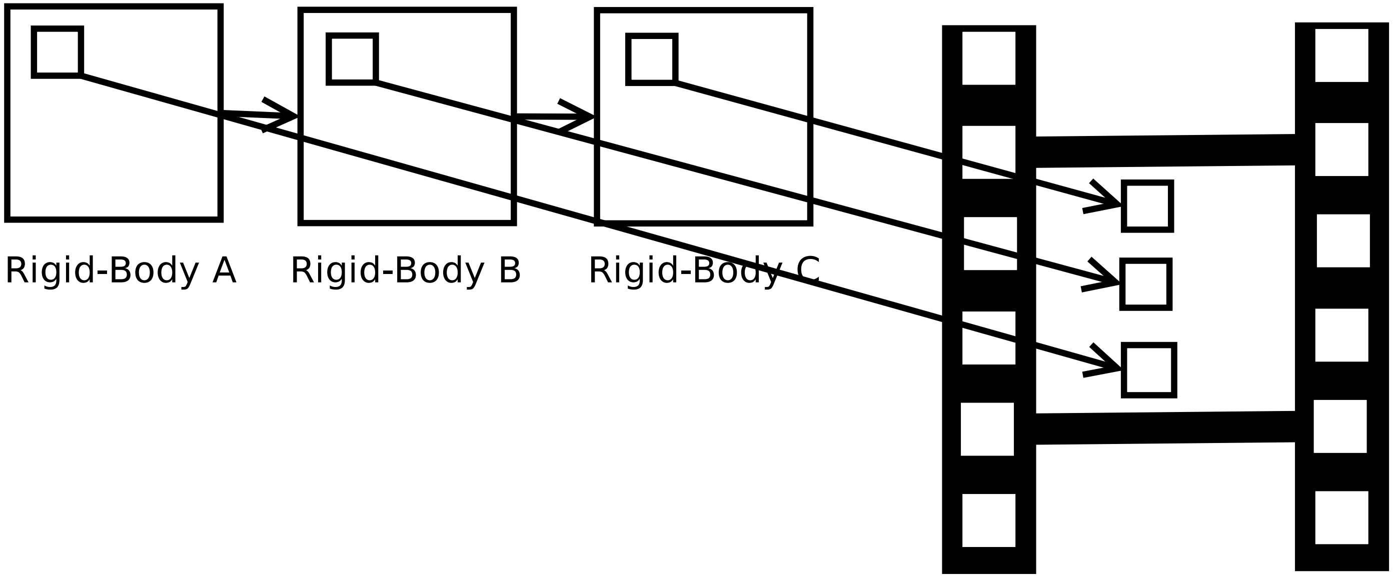
- Under resting contact we need to cancel out any forces along the normal between the rigid bodies
- This is to prevent rigid bodies from overlapping

Rollback

- If we find two rigid bodies overlapping we have to rollback to the beginning of the time step.
- Compute a smaller time step value
- $\Delta t_{new} = \Delta t/2$
- Restart the iteration of the simulation loop

Write Frame

- After each iteration of the simulation loop, we write a frame to the reel.
- Only the physical representation of each object is written to the frame.



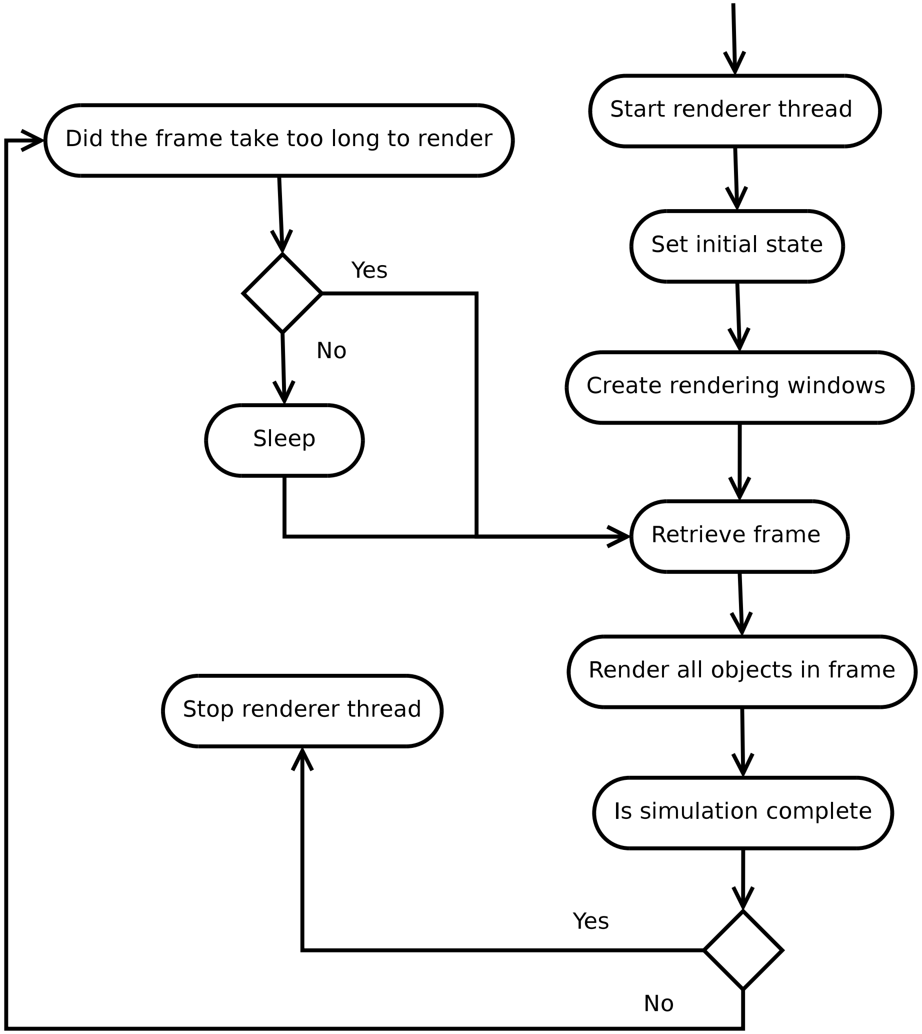
Check for Completion

- Performed after each iteration of the simulation loop
- The simulation thread stops when:
  1. It has completed the specified number of time steps
  2. The user has requested that the program stop

Renderer Thread

- Displays frames generated by the simulation thread
- It performs the following steps:
  1. Set initial state
  2. Create rendering windows
  3. Retrieve frame
  4. Render all objects in the frame
  5. Maintain timing
  6. Check for completion

Renderer Thread Flowchart

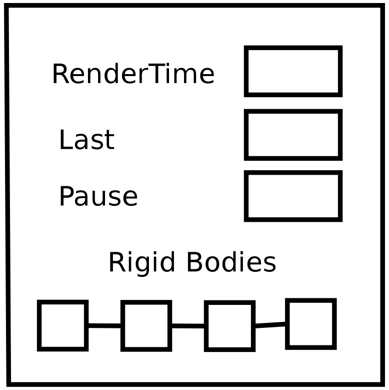


Set Initial State and Create Windows

- Initial settings given by the input thread
  - Number of renderers
  - Viewing frustum of each renderer
  - Camera position of each renderer
  - Type of each renderer
- There can be a maximum of four renderers
- Create a new window for each renderer

Retrieve a Frame

- Retrieve the next frame from the reel
- Information contained in the frame:
  - Length of time the frame should be rendered for, $targetTime$
  - Last frame flag
  - Pause flag
  - List of objects to render


RenderTime

Last

Pause

Rigid Bodies

Render All Rigid Bodies

- Render each rigid body in each renderer
- If the pause flag is set, the rendering thread pauses
- Time how long it takes to render each frame $renderTime$.
- If $renderTime >= targetTime$, start rendering the next frame
- If $renderTime < targetTime$, sleep for $(targetTime - renderTime) ms$

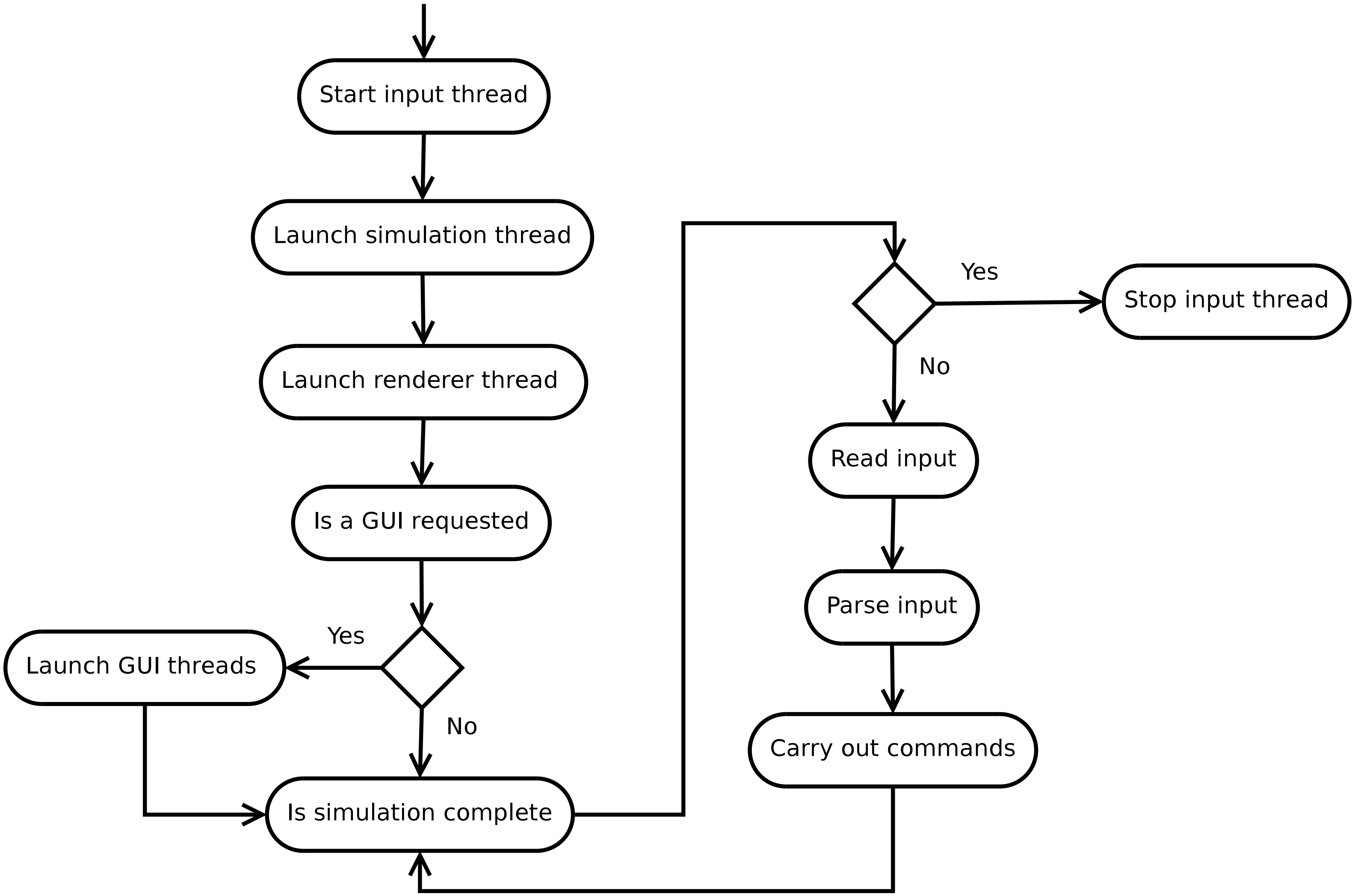
Check for Completion

- Closes if the user has requested that the simulator stop
- Pauses if the last frame flag has been set
- When in pause state the thread sleeps for each iteration of the loop

Input Thread

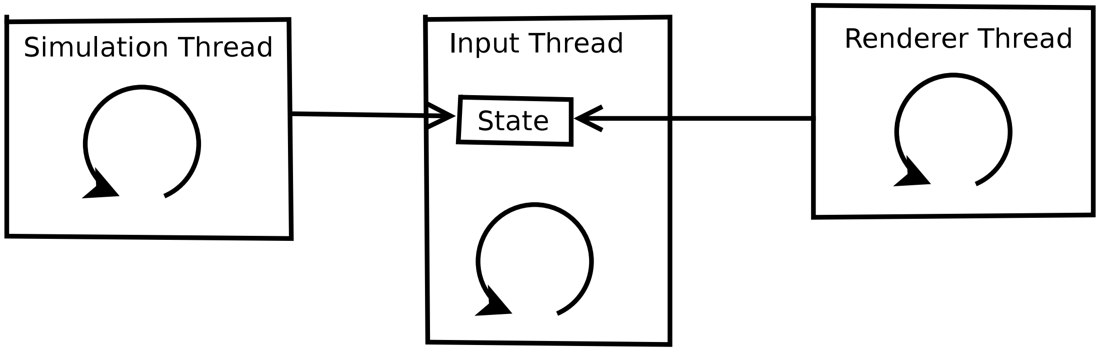
- Retrieves and parses user input
- It performs the following steps:
    - Launch subordinate threads
    - Launch GUI threads
    - Read and parse input
    - Carry out commands
    - Check for completion

Input Thread Flowchart



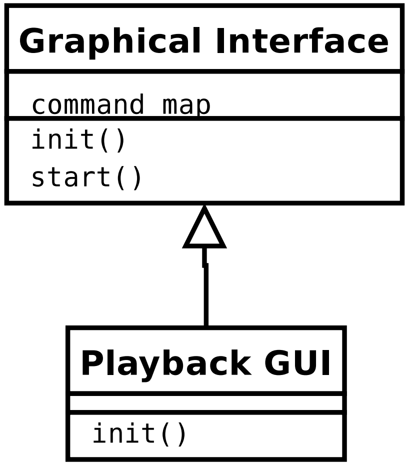
Launch Subordinate Threads

- Read `config.txt` to get the required settings for the other threads
- Launch the simulation thread
- Launch the renderer thread
- Set the initial simulation state



Launch GUI Threads

- Each GUI type inherits from `GraphicalInterface`
- `GraphicalInterface` is an empty window
- The subtypes are responsible for adding elements to the empty window
- Each subtypes operates in its own thread



Graphical Interface

`command_map`

`init()`

`start()`

Playback GUI

`init()`

Read, Parse, and Execute Input

- The command-line interface and each GUI interface listens input
- The input value is looked up in a command map

<table>
<tr><th colspan="2">Command Map</th></tr>
<tr><th>Input</th><th>Command</th></tr>
<tr><td>quit</td><td><code>SimulationState.setFinished()</code></td></tr>
<tr><td>help</td><td><code>CommandLineInterface.help()</code></td></tr>
</table>

- The matching command is called

Check for Completion

- The input thread stops when the user requests that it stop
- When the user requests the simulator shutdown, the simulation state is updated
- Before completely shutting down, the GUI windows are closed

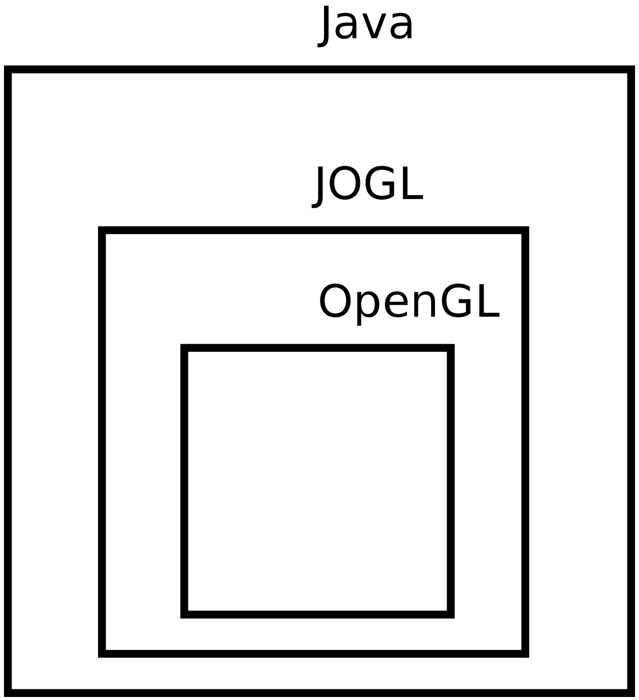
New Technologies Learned

I learned some new technologies during the development of the simulator.

- OpenGL
- Ant
- Design Patterns
- Java Generics

OpenGL

- Used to present output of the simulation
- The simulator uses JOGL [2].
- JOGL is a binding layer between Java and OpenGL



Java

JOGL

OpenGL

OpenGL - Rendering a Square

```
GL gl = drawable.getGL();
gl.glBegin(gl.GL_QUADS);
gl.glColor3f(1.0f, 0.0f, 0.0f);
gl.glVertex3f(-1.0f, 1.0f, 0.0f);
gl.glVertex3f(-1.0f, -1.0f, 0.0f);
gl.glVertex3f(1.0f, -1.0f, 0.0f);
gl.glVertex3f(1.0f, 1.0f, 0.0f);
gl.glEnd();
```

Ant

- Build tool written in Java [1]
- Uses XML for build files
- Originally developed for the Tomcat project

Ant Example

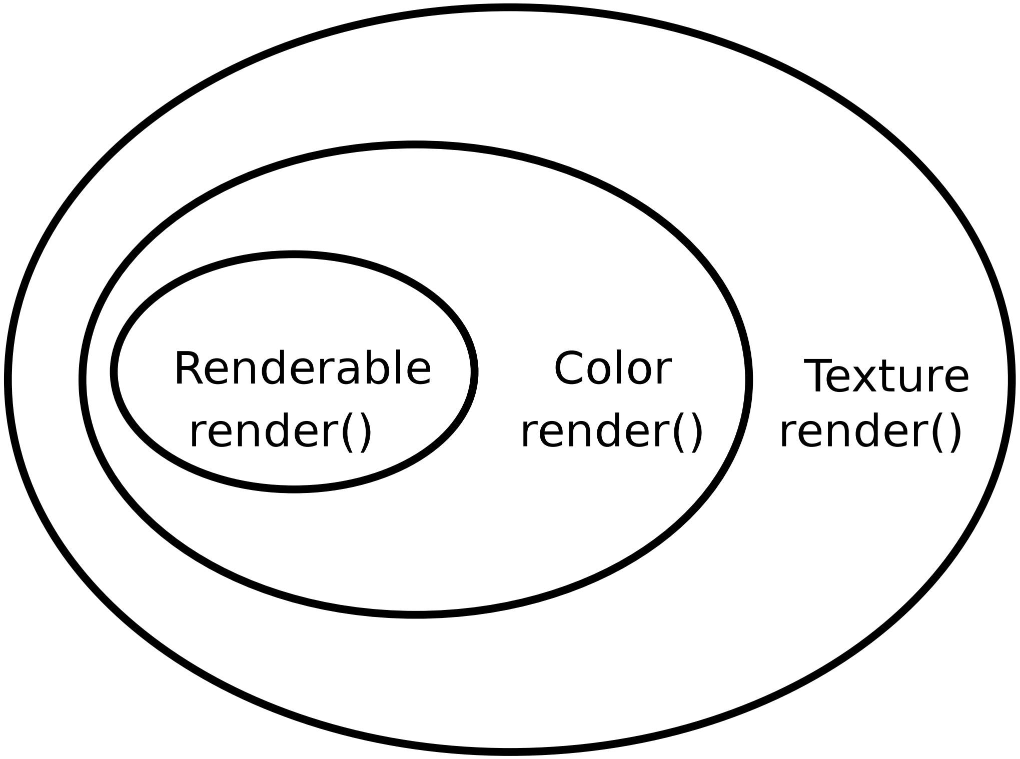
```
<?xml version="1.0"?>
<project default="default" name="Rigid Body Simulator">
    <description>
        Build file for my rigid body simulator.
    </description>
    <property name="srcDir" location="src"/>
    <property name="libDir" location="lib"/>
    <target name="default" depends="rbs"/>
    <target name="rbs_wall">
        <mkdir dir="${libDir}"/>
        <javac srcDir="${srcDir}" classpath="${libDir}" destDir="${libDir}
            <compilerarg value="-Xlint"/>
        </javac>
    </target>
    <target name="rbs">
        <mkdir dir="${libDir}"/>
        <javac srcDir="${srcDir}" classpath="${libDir}" destDir="${libDir}
    </target>
</project>
```

Design Patterns

- General solutions to common software problems [5]
- Following design patterns were used:
  - Decorator Pattern
  - Factory Pattern
  - Singleton Pattern

Decorator Pattern

Allows the program to dynamically modify the behavior of an object



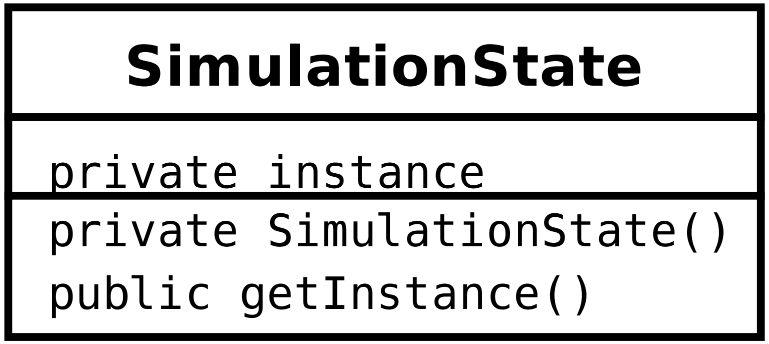
Renderable
render()

Color
render()

Texture
render()

Singleton Pattern

- Ensures that only one instance of a class can be created for the life of the program
- `SimulationState` is a singleton



SimulationState

```
private instance
private SimulationState()
public getInstance()
```

Factory Pattern

- Allows the program to create an object without knowing its exact class, until runtime
- Simulator uses a factory to create different renderer types
- The exact type is not known until the file `config.txt` has been read

Java Generics

- Constitute an extension to the Java programming language
- Introduced in JDK version 1.5
- Commonly used to specify the types of objects a container can contain
- Ensures compile-time type safety

Future Improvements

- Organize rigid-body file
- Selective rollback
- Add different object types
- Mount the camera to objects
- Add a free-roaming camera
- Fully simulate gravity

References

[1] Apache Ant Freqently Asked Questions, 2008. `http://ant.apache.org/faq.html`.

[2] The JOGL API Project, 2008. `https://jogl.dev.java.net`.

[3] David M. Bourg. *Physics for Game Developers*. O'Reilly Media Inc., 1005 Gravenstein Highway North, Sebasopol, CA 95472, 2002.

[4] David H. Eberly. *Game Physics*. Morgan Kaufmann Publishers, San Francisco, CA 94111, 2004.

[5] Eric Freeman and Elisabeth. *Head First Design Patterns*. O'Reilly Media Inc., 1005 Gravenstein Highway North, Sebasopol, CA 95472, 2004.