Physics Lab - Teaching Package

Final Report

Adam Beecham
BSc. Computer Science

School of Computer Science and Informatics,
Cardiff University

Supervisor: Helen R. Phillips

Moderator: Alun D. Preece

Abstract

The physics lab project involves the design and implementation of a visual simulation of a number of scenarios that would typically be encountered in an A-level physics course. The emphasis of this project is on providing a way of visualising the actions of forces on objects and how these actions are reflected by the underlying equations.

The package would be used as a teaching aid by an instructor to demonstrate the application of these equations, which would otherwise be difficult or dangerous to perform. This package should also provide a more engaging method of teaching students the mathematical concepts behind various applications of force and help students to better understand these concepts. The aim of this project is to address the issues that students encounter when solving physics problems that require understanding of fundamental physics concepts and mathematical equations.

Acknowledgements

- Dr Helen R. Phillips, who originally proposed the idea of implementing a physics simulation system, and has supervised the development of the project.

- Mr John Ivins of croesyceiliog School, for providing feedback on the selection of appropriate physics concepts to use in simulations and evaluating the completed system.

Table Of Contents

Menu - Implementation

Menu - Testing

Momentum Simulation - Design

Momentum Simulation - Implementation

Momentum Simulation - Testing

Evaluation & Conclusion

Appendix A - Requirements Specification

Appendix B - Prototype Class Diagram

Appendix C - Feedback From Physics Teacher

Introduction

Requirements Specification Amendments

Following the development of the prototype system documented in the interim report, it was found that a number of changes to the original requirements specification were necessary as implementation of all the mentioned features would not be possible in the given timescale. The main alterations made are summarised below:

- The original requirements specification states that the system will have simulations covering concepts at GCSE and A-Level. To improve the feasibility of implementing a complete system in the time scale, the system will instead focus on A-level concepts, specifically projectile motion, conservation of momentum demonstrated via perfectly elastic collisions and pulley systems.
- The original requirements specification states the system will have a question mode and a demo mode. To save time in implementation, the functionality in these modes will be merged, so that all functionality will be available to the user without swapping between modes.
- Simulation setups will no longer be saveable as previously stated, but the user will still be able to write and save their own questions.

The full amended requirements specification is available in the appendices.

The Prototype System

The final system will build on much of the technology developed for the prototype outlined in the interim report. To summarise, the prototype system demonstrated a very basic version of the projectile motion simulation and demonstrated how the simulation could be integrated with a swing interface using a workaround that involved running the 3D application in a canvas. The prototype was primarily developed as a proof of concept, showing that it would be possible to meet the requirements using the JMonkeyEngine and java swing interfaces.

Introduction of New Technologies

Following development of the prototype system, a number of required technologies that were not previously mentioned in the interim report were identified. The following technologies are necessary for implementing all of the desired features specified by the requirements.

Nifty GUI

Nifty GUI is a GUI system especially developed for use in java openGL or LWJGL applications. Nifty provides a range of common GUI components such as sliders, text fields and panels as well as screen manager capabilities, allowing transitions between different screens. Typically, the layout of a nifty interface is done in XML, while transitions between screens and user interaction with Nifty interfaces are managed by java classes called controllers, which contain methods that are called when an event of some kind is detected. As JMonkey is based on LWJGL, it already has support for

nifty interfaces. JMonkey also provides a number of libraries to allow integration between nifty and its own SimpleApplication class, which would allow quick integration of nifty into Physics lab.

There are a number of reasons why Nifty GUI is a necessary component for physics lab. Using nifty will mean that development of a screen manager system will not be necessary and will save significant time when implementing the system interface. This screen management capability is necessary as the system will need many screens, one for each simulation and the menu, and will need to transition between these screens when the user selects a simulation or returns to the menu from a simulation.

The interim report detailed that in order to use a swing based interface, a workaround was necessary. This involved running the JMonkey application itself in a canvas, while adding a panel to the application frame with the swing interface. This was not an ideal solution as JMonkey applications are intended to run as self-contained programs and many desirable features were lost as a result. Using this method, each simulation would require its own JMonkey application, and the code structure would quickly become very untidy having to switch between several applications by starting and quitting them.

Using Nifty for the interface can prevent such problems as it runs inside the JMonkey application, rather than alongside it as swing does. Although Nifty offers fewer GUI components than swing, using Nifty enables desirable features such as running physics lab in full screen or windowed mode at many resolutions and anti-aliasing of 3D graphics. This also allows physics lab to run as a single JMonkey application and simulations can be developed as components of the application, rather than as individual applications.

Blender

Blender is an open source 3D modelling program that will be used to develop 3D models for physics lab. Blender provides a wide range of tools for 3D model development, including basic shapes that can be used as starting points for quick development of 3D models, texture mapping and illumination (Wikipedia). As the OgreXML model format is the preferred format for models used with JMonkey, models will be exported to JMonkey using blenders OgreXML exporter.

Development of custom 3D models will be necessary as free 3D models are scarcely available and are not suitable for the needs of this project. In addition, few models are available in the OgreXML format required by JMonkey. Models will be textured and their illumination models set with blender so that it will not be necessary to hard code these properties into Physics Lab and save development time. Developing custom models will ensure that they meet the requirements of the project but will require additional time to develop, meaning that less time will be available for coding.

JavaDB

JavaDB is a lightweight database solution that uses the open source Apache Derby database. JavaDB is included in the java SDK and will be used to develop an embedded database solution for the physics lab system. This database will be used for storing questions saved by users, as well as information on simulations in the system that will be retrieved from the database for display when the menu is opened.

JavaDB is an ideal solution for storing data as it is small in size and provides room for a large amount of data to be stored, and allows for easy expansion of the system in the future if required. In addition a database ensures data integrity that cannot be provided by a file based system that would require data integrity to be manually enforced if, for example, new simulations were added or outdated simulations were removed.

Projectile Motion Simulation – Design

Overview

As a basic version of the projectile motion simulation was developed as part of the prototype system, a number of existing classes used in the prototype will be expanded and integrated into the final system. These classes are:

Simulation – The abstract base class for all simulations. The simulation class contains methods and variables that provide the basic functionality required by all simulations, such as the `init()` and `update()` methods. Each simulation will extend on the Simulation class and overrides some of its classes to provide more specific functionality as needed.

Particle – The abstract base class for all particles. The particle class contains methods and variables that provide the basic functionality required by all particles. As each simulation will require its particles to behave differently, classes will be constructed for each type of particle. These classes will extend the Particle class and override its `update()` method, specifying how each particle should move and interact with its environment.

ProjectileMotionSim – The class that manages the projectile motion simulation. Currently, it is responsible for instantiating and updating a Projectile, and responding to user input that is received from the swing interface panel, such as playing and pausing the simulation.

Projectile – The class responsible for creating and updating a projectile particle. Currently, this class sets up a projectile and its geometry by calling the constructor of the particle class and updates its x and y position according to the projectile motion equations:

- $Y = U\sin(\theta) + at$
- $X = U\cos(\theta)$

Where:

- U = initial (launch) velocity
- a = acceleration (gravity, constantly pulling down at -9.81 m/s)
- t = time elapsed.
- θ = launch angle.

In order to incorporate a menu into the system so that the user may switch between simulations, the existing swing interface will not be included. Instead, the ***InterfacePanel*** and ***ProjectileMotionInterface*** classes will be dropped and replaced by nifty GUI controller classes to handle input and a nifty GUI XML document will be developed to layout the interface for the simulation.

Interface Design

As interface layout in nifty is different from swing and many features left out of the prototype, it was necessary to redesign the original projectile motion interface. Nifty interface layout is typically done using XML and consists of:

- **Screens** – Only one screen is visible at a time. The first screen to appear must be named 'start'. Screens hold all the elements of the interface that is currently active and interactions with the interface are handled by java controller classes.
- **Layers** – Layers hold panels and can be used for alignment and other effects such as overlapping.
- **Panels** – Panels hold interactive elements such as text fields and buttons. They are used for laying out an interface and can be nested but cannot overlap (Hohmuth 2011).

The diagram below shows the proposed layout for the interface of the complete projectile motion interface.

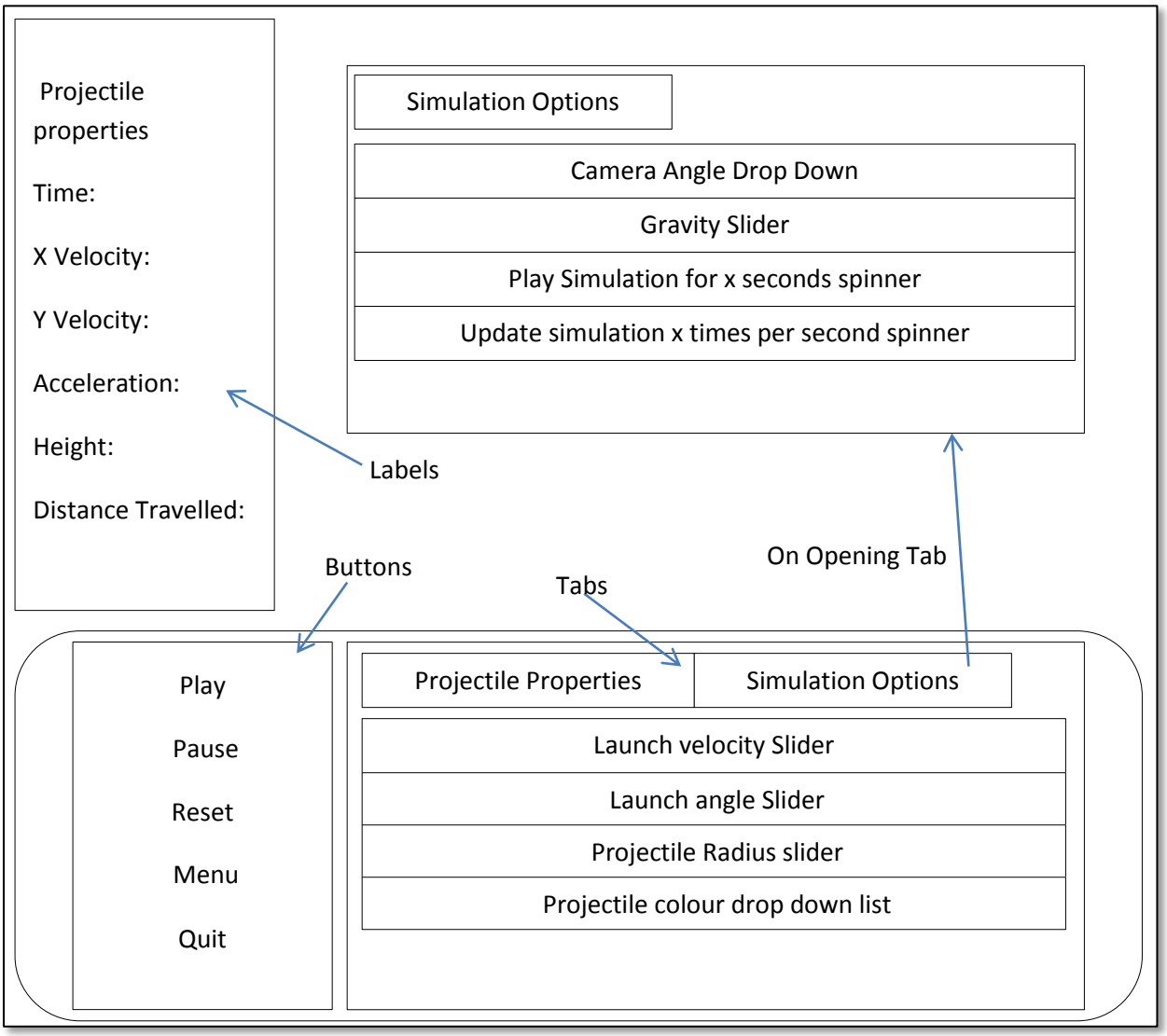


Figure 1 - Projectile Motion Simulation Interface

Due to the number of additional components not present in the prototype interface, components that allow interaction with the simulation have been grouped into two separate tabs. The projectile properties tab contains sliders for adjusting the projectiles launch angle, velocity and radius, as well as a list of colours that can be applied to the projectile. The simulation options tab allows the user to select from a range of camera angles that can be used to view the scene, a slider to change the strength of the gravity in the simulation and two spinners, one specifying how long the simulation should run for and another specifying how many times the projectile should update per second. A number of labels will display the projectiles properties at the given time and will be refreshed each time the particle is updated.

A list of buttons is also provided for basic operations such as playing, pausing and resetting the simulation, returning to the menu and quitting the application. It should be noted that these buttons, as well as the simulation options and the particle properties on display, will be the same across all simulations as this will allow an abstract controller class to be defined and speed up implementation.

System Design Modifications

A number of alterations will be made to the structure of the existing application classes. More functionality will be given to the base ***Simulation*** and ***Particle*** classes so that less time is required to implement sub classes. As the ***Projectile*** class update algorithm is already correct, no changes will be made to it. Minor changes will be made to the ***ProjectileMotionSim*** algorithm in order to integrate the additional interface options.

In the prototype, the ***Simulation*** class was responsible for displaying the projectile properties to the user. To achieve better componentization, this responsibility will instead be handled by the new interface controller classes so that all interface components are handled together. In addition, the ***Simulation*** class will also include additional methods for handling the new interface options and for feeding particle data back to the new interface.

The ***Simulation*** class will use a generic List for storing particles. Rather than have sub classes manage and update their specific particles as previously done, they will instead instantiate the required type of particle and attach each particle to the list. This will allow much of the interaction between the projectile motion simulation and its particles (i.e. updating particles) to be abstracted to the ***Simulation*** class by exploiting inheritance and polymorphism.

The ***Particle*** class will no longer be responsible for keeping track of elapsed time. Though this was adequate in the prototype, it may cause discrepancies in simulations with multiple particles. The responsibility will now be given to the ***Simulation*** class and fed to each particle when its `update()` method is called by the simulation. The ***Particle*** class will also have additional methods to allow it to be targeted by the camera, have its radius adjusted and its colour changed.

As the new interface no longer requires the application to be run in a canvas, the ***PhysicsLab*** class will extend JMonkeys ***SimpleApplication*** class and the swing workaround will be removed. It will instead initialize the nifty interface by reading the XML layout document. Nifty will then start the appropriate controller class for the projectile simulation screen which will initialise the simulation and the interface and display the screen to the user.

The following class diagram shows the structure of the system thus far. For simplicity, get and set methods have been omitted. The original class diagram can be seen in the appendices.

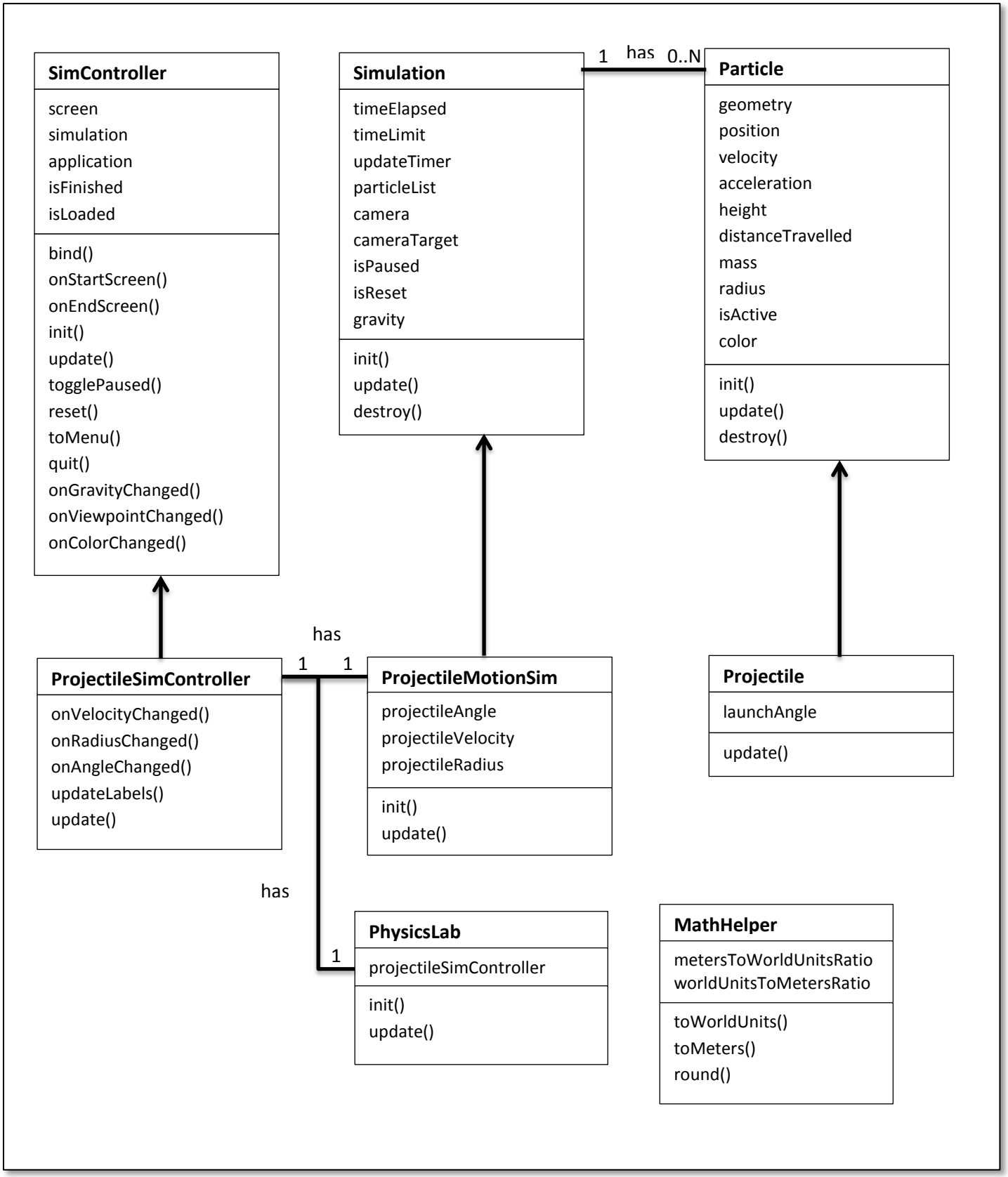


Figure 2 - First Iteration Class Diagram

As can be seen, the abstract base classes now include much more functionality than they previously did. Methods in the new controller classes prefixed with 'on' represent methods that perform an action in response to an event. an example of this is the `onStartScreen()` method, which can be used to load everything required by the screen prior to it being displayed. the controller classes also contain button methods such as `reset()` . As each screen has an interface and a simulation, each controller requires a reference to the active simulation and the application itself, as the application contains necessary components such as the scene graph and camera. The remaining classes are expanded versions of those used in the prototype. They specify additional methods and variables to include the previously discussed features.

Graphical Updates

Previously, the system exploited primitive 3D shapes included with the JMonkeyEngine framework. The final system will instead use models designed using Blender. The terrain will consist of two components; the runway, where the simulation will actually occur, will be a rectangular mesh textured with a tiled concrete image and the landscape. This will be used as a backdrop for the scene to make it more visually stimulating, but will not play any part in the simulation itself. Particles will remain unshaded sphere objects, in order to ensure they are unaffected by the lighting in the scene and can be clearly viewed from any angle.

Projectile Motion Simulation - Implementation

Overview

The updated implementation of the projectile motion simulation addresses many of the issues that were present in the prototype system, however some alterations were made from the design due to some unforeseen problems. One particular problem arose during implementation of the particle updates per second feature that allowed the users to speed up or slow down the particle. This caused unexpected behaviour, such as the particle not colliding with the runway and stopping out of the users view.

In order to resolve this issue, a new feature was implemented, whereby 'tracers' are dropped along the projectiles trajectory a specified number of times per second. Tracers inherit from the ***Particle*** class and hold data on the projectiles properties at the time the tracer was created. Tracers can be selected by clicking on them, which causes the interface to display the data held by the selected tracer for analysis by the user.

A previous issue, where the prototype displayed particle properties to approximately seven decimal places had also been resolved. Before values are sent to the interface, they are first rounded to two decimal places by the new `round()` method in the ***MathHelper*** class, giving a more reasonable level of accuracy. The projectile and simulation update algorithms remain essentially unchanged since the prototype system.

Interface Layout

As specified in the design, the interface layout was done using an XML document. Elements specify interface components themselves, while attributes set the properties of the given components. A sample of the interface layout below shows how the basic button controls are created.

```
<screen id="projectile_sim" controller="mygame.ProjectileSimController">
  <layer height="100%" width="100%" childLayout="vertical">
    <panel childLayout="horizontal" width="100%" height="30%">
      <panel style="nifty-panel" align="left" childLayout="center"
      width="20%" height="100%">
        <panel childLayout="vertical" valign="center" width="100%"
        height="90%">
          <control name="label" text="Basic Controls" align="center"/>
          <control name="button" label="Play/Pause" id="play"
          align="center" visibleToMouse="true" >
            <interact onClick="togglePaused()"/>
          </control>
          ...
```

```
          <control name="button" label="Reset" id="reset" align="center"
          valign="center" visibleToMouse="true" >
            <interact onClick="reset()"/>
          </control>
          ...
        </panel>
      </panel>
    </layer>
</screen>
```

Figure 3 - XML nifty interface layout

The opening element of the sample is the screen tag that holds all the other elements of the screen. This element has an 'id' attribute which allows it to be referred to by the application. It also specifies the controller class for the screen that will be responsible for loading the screen, handling input and closing the screen. The next element is a layer that contains the interface panels. Its attributes specify the dimensions as a percentage of the screen size and the layout of its children, specified as 'vertical'. This means that panels in the layer will be positioned in a top down manner, with each new panel placed lower on the screen than the previous one. The next component is a panel containing a nested panel that hold the buttons. As this panel should be visible, it is given a style attribute that is set to the default 'nifty-panel' (a grey panel with rounded edges) and specifies its alignment within the layer and the layout of its children.

Interactive elements are defined using control tags. these have a name that specifies the type of component, alignment attributes and further optional attributes such as text labels and whether or not the component should be clickable using the visibleToMouse attribute. Additionally, control elements can contain an interact element that specifies the method in the screens controller class to be called on a specific event. In the sample above, a button click triggers a call to the corresponding method, resulting in the appropriate action being performed. In order for control elements to be referenced and used by the corresponding controller class, they must have a defined 'id' attribute.

The layout of the remaining elements is done in a similar fashion. The complete interface is seen below.

Figure 4 - The Implemented Interface

Screen Controllers

The ***ProjectileSimController*** class is responsible for handling loading and interactions between the user and the simulation. Eventually, it will also be responsible for unloading the screen and returning to the menu when the user clicks the menu button, to allow switching between simulations. In order to enable the controller, it is first instantiated, along with a nifty interface object in the ***PhysicsLab*** class. The nifty interface then invokes a method specifying the XML layout document, the screen name and the specified controller that binds them together. The nifty interface is then added to the JMonkey GUI viewport so that it overlays the 3D canvas and is ready to use. The code for this is seen below.

```
@Override
public void simpleInitApp()     {
      //instantiate a controller and a nifty display
      projectileSimController = new
      ProjectileSimController((PhysicsLab)this);
      NiftyJmeDisplay niftyDisplay = new NiftyJmeDisplay(
      assetManager, inputManager, audioRenderer, guiViewPort);
      // Create a new NiftyGUI object
      Nifty nifty = niftyDisplay.getNifty();
      // Read XML and initialize custom ScreenController
      nifty.fromXml("Interface/PhysicsLabGUI.xml", "projectile_sim",
      projectileSimController);
      // attach the Nifty display to the gui view port as a processor
      guiViewPort.addProcessor(niftyDisplay);
    }
```

Figure 5 - Controller Setup

As no menu is yet present, the projectile motion simulation is run as soon as the program starts. The `onStartScreen()` method sets up some of the interface components and creates a new projectile simulation then calls the same parent ***SimController*** class method to setup the default components. The controllers update method is then called by the application update loop, which starts the simulation.

Interactions between the user and buttons are trivial, the user clicks a button and the button then calls the method defined in the layout document. However interactions with components that have changeable values, such as sliders, are more complex. The code sample below shows how interaction with the velocity slider is managed.

```
@NiftyEventSubscriber(id="p_vel")
public void onVelocityValueChange(final String id,
final SliderChangedEvent event)
{
   //get the value the user selected
   float val = event.getValue();
   //find the label displaying the velocity
   Element label = nifty.getCurrentScreen().findElementByName("ms_label");
   //update the label with the selected velocity rounded to 2 dp
   label.getRenderer(TextRenderer.class).setText(MathHelper.round(val, 2));
   //set the projectile velocity
   ((ProjectileMotionSim)sim).setInitialProjectileVelocity(val);
}
```

Figure 6 - Interacting with the Velocity Slider

First, `@NiftyEventSubscriber()` is used to subscribe to events triggered by the velocity slider using its 'id' attribute. The method below is then called every time the slider is moved by the user. This method gets the selected value and updates the label adjacent to it to show the user the value currently selected. The value is then sent to the ***ProjectileMotionSim*** class which uses the value to set the projectiles launch velocity.

The above procedure is used for all interface components excluding buttons. the element is subscribed to, and responds to selections by the user by retrieving the selected value and performing some action with the value.

Graphics

As specified in the design, 3D models were developed for the games terrain. In addition, a skybox was used to provide a backdrop for the simulation in order to make it a more immersive experience. Importing these assets was straightforward as JMonkey provides folders for the projects assets, they were saved to the appropriate folders and imported into the game by calling the asset managers `loadModel()` method, specifying the models directory. The skybox was created by using a .dds file and calling Jmonkeys built in `createSky()` method from the ***SkyFactory*** class.

To add further realism, a directional light and shadow renderer were also added to the simulation. As these features were included with the JMonkeyEngine, implementation of these required an

object of each type to be instantiated. The light source was then attached to the scene and the shadow renderer added to the viewport as a processor. As shadow generation can be computationally expensive, the terrain was set to receive shadows only, the particle to cast shadows but not receive them and the runway to both cast and receive shadows. The scene is shown below.

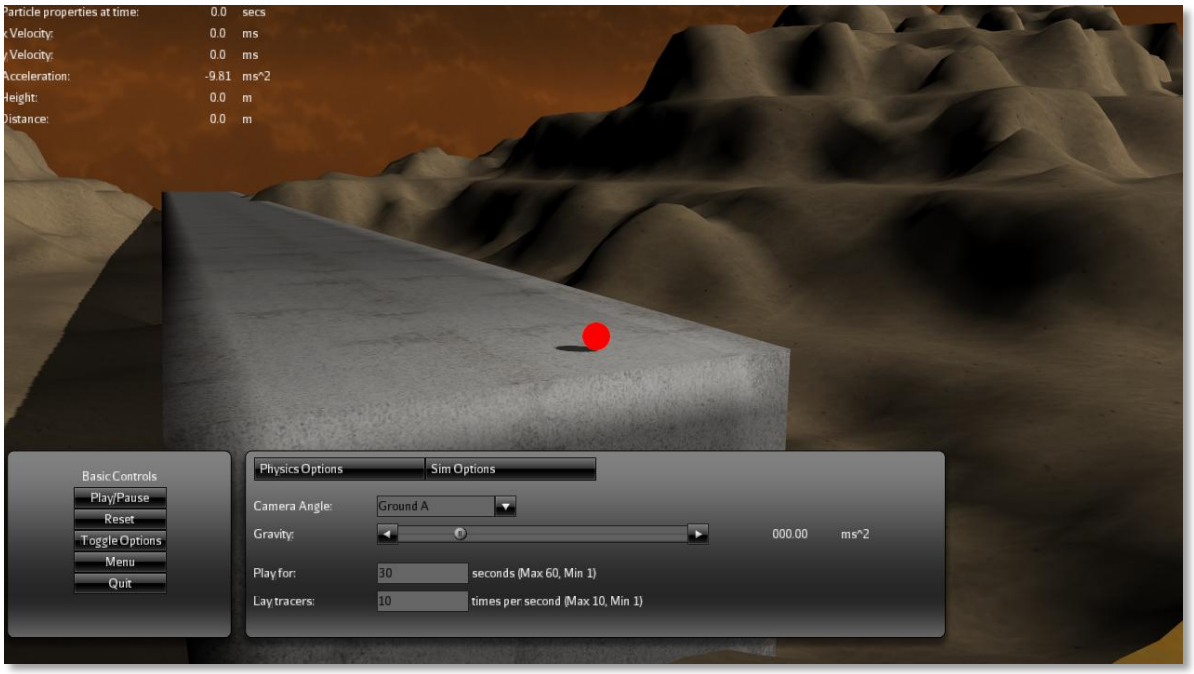


Figure 7 - Simulation with Terrain, Lighting and Shadows

Camera Angles

A number of changes have been made to the camera since the prototype. In the final implementation of the system, the user can choose to view the simulation from various camera angles by selecting them from a drop down list in the simulation options tab. The system still uses the built in camera, so once the user has chosen a view, they are able to move freely from the position if desired. When the user selects a viewpoint, the controller class determines the option selected using a switch statement and calls the ***Simulation*** class `setCamViewMethod()` which snaps the camera to the specified position. The implementation of the method is as follows.

```
public void setCamView(int view)
{
    //view from the runway
    if(view == GROUND_VIEW)
      app.getCamera().setLocation(new Vector3f(-60, 45, 600));
    //view from the side at a slightly elevated position
    if(view == SIDE_VIEW)
      app.getCamera().setLocation(new Vector3f(-200, 200, 500));
    //chase the particle
    if(view == CHASE_VIEW)
    {
```

```
//find the particle being chased then follow it
for(Particle p : particles)
       if(p.isCamChasing == true)
       {
         app.getCamera().setLocation(new Vector3f(p.position.x,
         p.position.y,  p.position.z - 15));
         break;
       }
  }
}
```

Figure 8 - setCamView method

When the user selects the chase view, the camera will not be controllable until a different view is selected. The camera will always target a particle, unless the user selects a tracer. The camera then locks on to the selected tracer until the user deselects it, at which point it will retarget the projectile.

Tracers

The most significant change since the prototype system is the introduction of tracers. Tracers hold data on particle properties at the time they were created. They can be selected by the user which causes data held by the tracer to be displayed in the interface instead of the particles data. Tracer functionality is implemented in the abstract simulation class so they can be used for all types of particles. They were developed to replace the ability to slow the number of updates to a particle each second, which caused problems during implementation. A significant advantage of this feature is that the tracers remain even after the simulation has finished, so the user may go back and analyse the particles trajectory and properties if desired.

A timer is used to keep track of when a tracer should be created in the update loop. When the timer reaches an interval (calculated as 1/times to spawn per second), a tracer is created and added to a list of tracers currently on the screen. The tracer is then attached to the scene graph for display via a tracer node, which is attached to the root node of the scene graph. This is done to simplify selection of tracers, as only objects attached to the tracer node will be selectable and the user cannot accidentally select the terrain. The code below shows how tracers are created.

```
//create a tracer if the timer has finished
if(tracerTimer >= tracerInterval)
{
   tracerTimer = 0;
   //create a tracer for each particle and add it to the list of active
   //tracers
   for(Particle p : particles)
   {
      tracers.add(new Tracer(p, new Material(app.getAssetManager(),
      "Common/MatDefs/Misc/Unshaded.j3md")));
      tracerNode.attachChild(tracers.getLast().getGeometry());
   }
}
```

Figure 9 - Tracer Creation

As tracers do not update in any way, they are easily selected by the user. An ***ActionListener*** was added to the ***Simulation*** class in order to respond to mouse clicks by the user. The listeners `onAction()` method gets the 2D coordinates of the click and converts it into 3D coordinates using the cameras position. The 3D coordinates are then used to calculate the normalised direction of the click which is then used to cast a ray into the scene. If the ray intersects a tracer, it is selected, its data is displayed and it becomes the camera target. If no intersections are made then the camera remains targeted at the projectile. The selection process code is a modified version of the mouse selection tutorial available on the JMonkey community website and is shown below.

```
//cast a ray and check for tracer collisions
Ray ray = new Ray(click3d, dir);
tracerNode.collideWith(ray, results);
// select a tracer if one was hit
if (results.size() > 0) {
   //deselect previous tracer
   if(selectedTracer != null)
       selectedTracer.setSelected(false);
       //if multiple tracers were hit, pick the closest one
       CollisionResult closest = results.getClosestCollision();
       Geometry selected = closest.getGeometry();
       //compare the geometry of the closest tracer with each active tracer
       for(Tracer t : tracers)
       {
          //if the geometry position matches that of the current tracer, we
          //know what tracer we have selected
          if(selected.getLocalTranslation().equals(t.position))
          {
              //found the tracer we need so exit the loop
              selectedTracer = t;
              break;
          }
       }
       //target the tracer
       camTarget = selectedTracer.getPosition();
       selectedTracer.setSelected(true);
} else {
       // No hits so deselect if a tracer was selected
    if(selectedTracer != null)
    selectedTracer.setSelected(false);
    selectedTracer = null;
}
```

Figure 10 - Tracer Selection

Projectile Motion Simulation – Testing

Projectile Testing

An important part of each simulation is ensuring that a particles movement or update algorithm is correctly calculating particle data being displayed to the user. As Physics lab is intended for use in an educational environment, data that is out of acceptable range of the correct answer could cause confusion among students and would not be useful.

In order to ensure the calculations within the projectiles update loop are correct, the simulation was run twice and the xVelocity, yVelocity , distance and height values generated by each run sampled at different times. These values were then compared against manually calculated values using the same times that the values from the simulation were sampled at. The results for each of these runs are shown below. Acceleration, launch velocity and angle data is omitted as it is specified by the user and not computed by the simulation.

First Run

The simulation was setup with the following properties:

- Gravity: -9.81 m/s^2
- Launch Velocity: 50m/s
- Launch Angle: 30 degrees

The table below compares the results computed by the simulation against those calculated manually.

Table 1 - Analysis of First Run

Time of Sample	Expected xVelocity	Simulation xVelocity	Expected yVelocity	Simulation yVelocity	Expected Distance	Simulation Distance	Expected Height	Simulation Height
0.72s	43.3m/s	43.3m/s	17.94m/s	17.95m/s	31.17m	31.13m	15.46m	15.44m
2.07s	43.3m/s	43.3m/s	4.7m/s	4.71m/s	89.63m	89.55m	30.73m	30.72m
3.6s	43.3m/s	43.3m/s	-10.32m/s	-10.33m/s	155.88m	155.93m	26.43m	26.42m
5s	43.3m/s	43.3m/s	-24.05m/s	-65.31m/s	216.51m	216.59m	2.38m	2.38m

Second Run

The simulation was setup with the following properties:

- Gravity: -13.72 m/s2
- Launch Velocity: 75.35 m/s
- Launch Angle: 66.25 degrees

The table below compares the results computed by the simulation against those calculated manually.

Table 2 - Analysis of Second Run

Time of Sample	Expected xVelocity	Simulation xVelocity	Expected yVelocity	Simulation yVelocity	Expected Distance	Simulation Distance	Expected Height	Simulation Height
0.42s	30.35m/s	30.35m/s	63.21m/s	63.25m/s	12.75m	12.65m	27.76m	27.57m
4.33s	30.35m/s	30.35m/s	9.56m/s	9.53m/s	131.4m	131.46m	170.02m	170.04m
6.23s	30.35m/s	30.35m/s	-16.51m/s	-16.45m/s	189.06m	188.94m	163.42m	163.48m
9.81s	30.35m/s	30.35m/s	-65.62m/s	-65.69m/s	297.7m	297.86m	16.4m	16.08m

Interpretation of Results

It can be seen from the data that there are small discrepancies between the expected values and simulation values for the yVelocity, distance and height data. As the difference between these values is so small, it is likely that the difference is due to rounding errors. Manual calculations were done using the gravity, launch velocity and launch angle data displayed by the interface. This data is first rounded to two decimal places before being displayed, however it is not rounded before being used by the simulation to calculate new values for the projectile, meaning that the values used by the simulation and those used for manual calculations are slightly different.

In order to resolve this problem, each of these values should be rounded before being used by the simulation, as well as prior to being displayed by the interface, as this would make calculations by the simulation correspond with calculations done manually, avoiding any future confusion by users.

Interface Testing

To ensure the projectile motion simulation interface works as intended, each of the components were tested and the observed outcome documented and compared with the expected outcome of using the component. The table below summarises the tests performed and their results.

Table 3 - Interface Testing Results

Tested Component	Expected Outcome	Observed Outcome
Play/Pause Button	Simulation should start with defined settings on first click, successive clicks should toggle pausing of the simulation.	As expected.
Reset Button	If the simulation has finished or is currently playing, the projectile should return to the start position, tracers should be cleared and the interface reset.	As expected.
Toggle Options Button	Should hide/ show the option panel on each click depending on whether it is already visible.	As expected.
Quit Button	Clicking should close the application.	As expected.
Launch Velocity Slider	Altering the slider position should change the launch velocity, causing the adjacent label to be updated with the selected value and cause the projectile to be launched at the specified velocity.	As expected, Though wrong value is displayed on application start until the slider is moved, at which point value displayed is correct.

Launch Angle Slider	Altering the slider position should change the launch angle, causing the adjacent label to be updated with the selected value and cause the projectile to be launched at the specified angle.	As expected, Though wrong value is displayed on application start until the slider is moved, at which point value displayed is correct.
Radius Slider	Altering the slider position should change the projectile radius, causing the adjacent label to be updated with the selected value and cause the projectile to be resized to the specified radius.	As expected, Though wrong value is displayed on application start until the slider is moved, at which point value displayed is correct.
Projectile Colour Drop Down	Selecting a value from the drop down list should cause the particle to immediately change to the specified colour.	Causes transparent layer overlaying the simulation to turn black and hide the simulation from view. With this layer removed, behaviour is as expected, but fade-in effect at the simulation start up is lost.
Camera Angle Drop Down	Selecting a value from the drop down list should cause the camera to snap to the selected camera angle.	As expected.
Gravity Slider	Altering the slider position should change the gravity, causing the adjacent label to be updated with the selected value and cause the projectiles yAcceleration to be affected by the selected value.	As expected.
Run Time Text Field	Entering a value into the field should cause the simulation to run either until the value (in seconds) is reached (minimum of 1, maximum of 60), or the projectile hits the floor. Values outside the accepted range will be converted to 1 is less than 1 or 60 if more than 60.	As expected, though if the user enters a value outside the accepted range the value is not corrected in the text field for the user to see.
Tracer Text Field	Entering a value into the field should cause tracers to be created the specified number of times per second behind the particle(minimum of 1, maximum of 10 per second). Values outside the accepted range will be converted to 1 is less than 1 or 10 if more than 10.	As expected, though if the user enters a value outside the accepted range the value is not corrected in the text field for the user to see.
Tracers	Selecting a tracer from the simulation should cause it to turn white in colour and replace the currently displayed data with data on the projectile at the time the tracer was created.	As expected, though tracers that appear further away do not always select when clicked.

In general, the interface behaves as expected. There are however some minor problems where details such as ensuring labels match initial slider values have been overlooked but are easily corrected.

The most significant problem occurred when selecting a colour from the drop down list, unexpectedly causing the screen to turn black and hide the simulation from view. This was due to a layer specified in the nifty XML document, set to appear initially as black and fade out to reveal the simulation on start up. Selecting a value from the drop down list appears to interrupt the rendering of the layer and cause it to become black again, hiding the simulation.

Unfortunately, the problem seems to be contained within nifty itself and as a result the only way to prevent it from occurring is to remove the layer and transition effect entirely.

Menu - Design

Overview

The menu screen will provide the user with the ability to select from a list of available simulations that they wish to use. Selecting a simulation from the menu will cause a brief description of the chosen simulation to be retrieved from an embedded database and displayed, providing further details about the simulation to the user. If the user then decides that they wish to use the selected simulation, it will be started upon clicking the start button. The user will also be able to quit the application from the menu by clicking a quit button.

If the user is currently viewing a simulation and wishes to switch to another simulation, each simulation screen will have a menu button in its interface that will unload the current simulation (to prevent the program eventually crashing), and return the user the menu when clicked.

When the application is first started, it will no longer immediately load the projectile motion simulation. A simple splash screen will be used as the start screen to introduce the user to the application. From this screen, the user can press start to then proceed to the menu, or immediately exit the application by pressing the quit button

Interface Design

The interface of the menu will be minimalistic and not require many components. The only functionality required for the menu is the ability to select a simulation, display simulation data (i.e. its name, description and image) and options for starting the selected simulation or quitting the application.

The list of simulations will be held by a nifty list box component, as this can display string values and can be configured for selection of single values from the list. Label and image elements will be specified in the menus XML layout for displaying simulation data. These will have ‘id’ attributes so that they can be referred to by the menu’s controller class and updated when the selected value of the list box is changed. The design for the menu’s interface can be seen below.

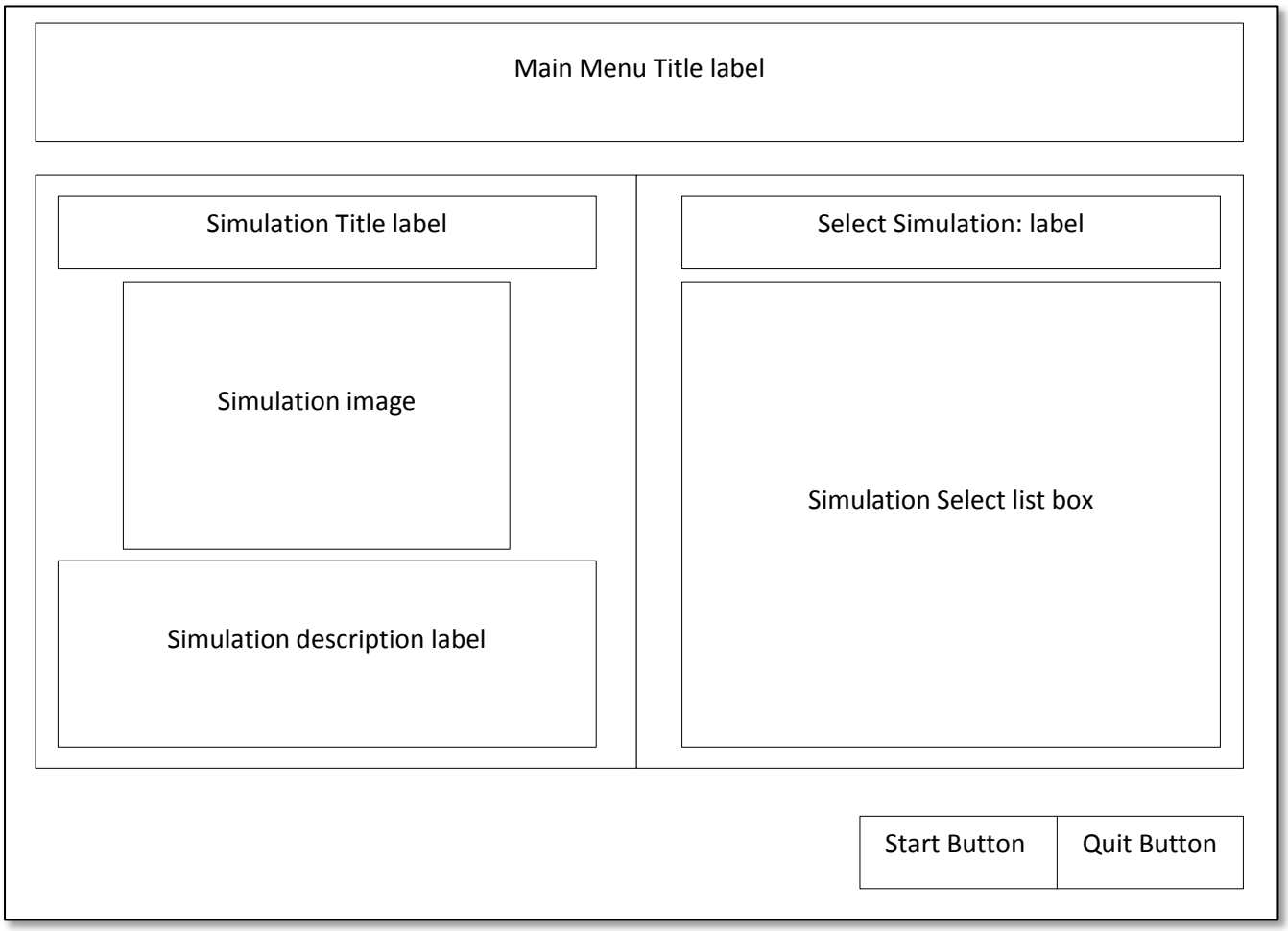


Figure 11 - Menu Layout

The middle section of the menu will have two separate panels that hold nested panels containing the interfaces components, such as the list box and the simulation data labels. Unlike the Projectile Motion Interface panels, each of These panels will be transparent and will be used strictly for layout purposes and no style or colour attributes will therefore be defined.

Database Design

The system database will be created using the tools provided with the standard java SDK. This provides a java derby database and libraries for making SQL queries in java. The database will hold data on simulations for the menu such as its title and description, question data such as the questions answer and the question description, as well as customised simulation setups. This will allow the system to be expandable at a later date, even if this functionality is not fully implemented within the timescale for completion of the project.

Simulation data for the menu will be retrieved from the embedded database on the first load of the menu using the java SQL library and stored in an array. This will prevent expensive connections to the database and improve the systems performance if the user makes multiple switches between simulations. In order to ensure the records in each table are unique, each record will have an ID number that will act as a primary key field. The table below summarises the information needed by the database.

Table 4 - Database Attributes

Attribute	Description
Simulation ID	Unique identifier for a simulation.
Simulation Title	The name of a simulation.
Simulation Description	A brief description of the contents of a simulation.
Simulation Image	The file path of the image to be displayed in the menu for a given simulation.
Question ID	Unique identifier for a user defined question.
Question Description	A description of the problem that will be displayed to the user
Question Answer	The answer to the question.
Sim Setup ID	Unique identifier for a user defined simulation setup.
Gravity setting	User defined gravity value.
Run time setting	How long the simulation should run for before stopping.
Tracer setting	How many tracers should be created per second.
Camera setting	The camera viewing angle.
Particle Setup ID	Unique identifier for a user defined particle setup.
Particle Velocity	The particles initial velocity.
Particle Angle	The particles initial angle.
Particle Mass	The particles weight in kilos.

Note that only simulation properties that appear across all simulations are saveable. This allows them to be used as 'preferences' that could be quickly loaded to speed up initial simulation setups. This also allows them to be used on any simulation rather than being limited to a specific type of simulation. This also applies to particle setups, however there is a risk that simulations added in future may require additional properties to be saveable, though these could be added at a later date if necessary. Any property attributes held by the database that are not required by a specific type of particle could be left blank.

Normalization

As the number of attributes in the database is quite small, normalization is quite simple. In order to put the database in first normal form, it must not have any repeating elements or groups of elements. This is achieved by separating the data into the three tables shown below. Notice that they do not have relationships with eachother as both setup tables are independent of one another and can be applied to any simulation.

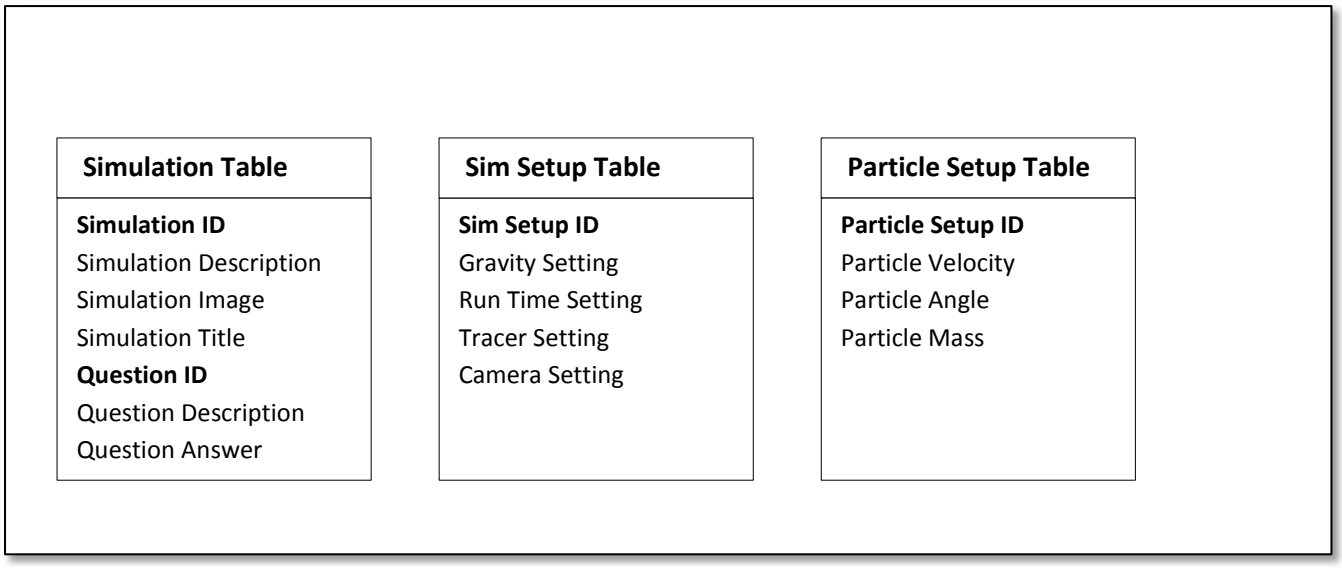


Figure 12 - First Normal Form

Attributes that form a primary key are highlighted in bold font. As can be seen, the simulation table has two attributes forming the primary key. This is not necessary, as each of the attributes in the table depend on only one of the primary key attributes and can therefore be normalized into second normal form. To do this, all attributes dependent on the Question ID attribute are placed into another table and the Simulation ID is included in the new table as a foreign key. This is required because a simulation can have many questions, and as such a relationship between the new Question table and the Simulation table is necessary. Second normal form is adequate for the needs of the system and there is no need for further normalisation. The final database structure is shown below.

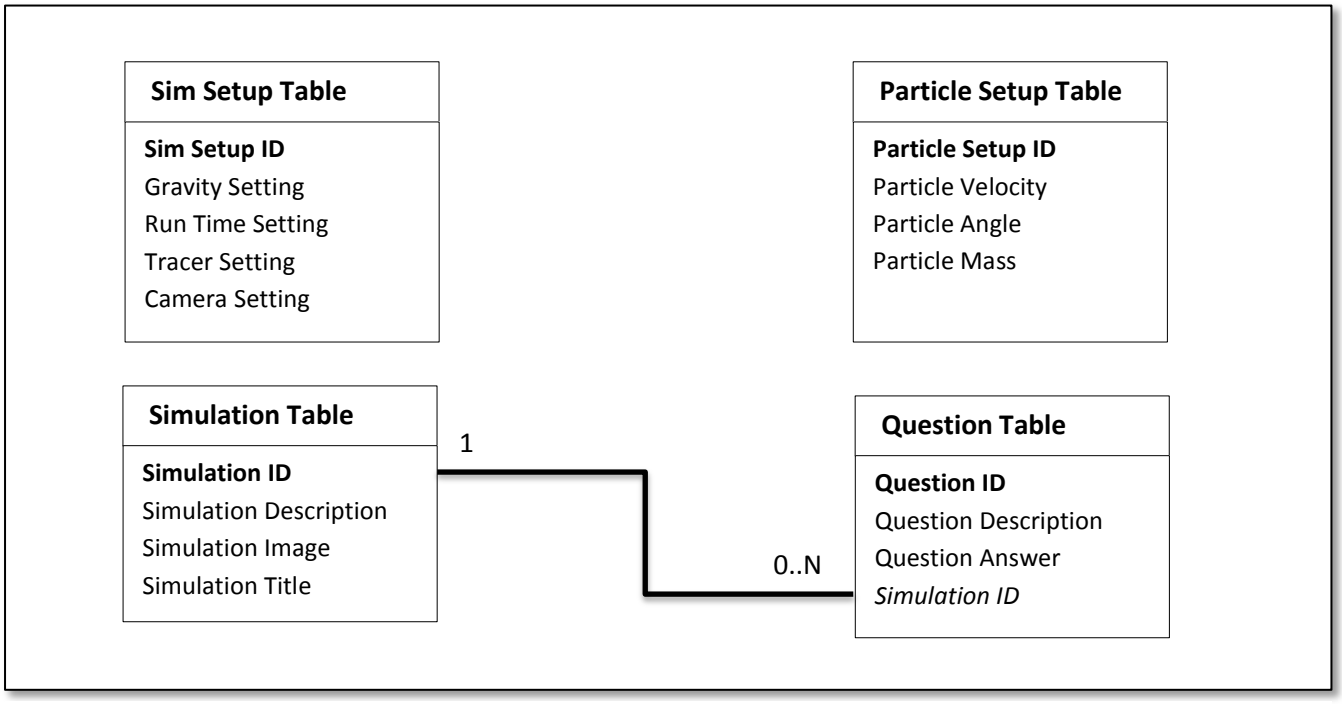


Figure 13 - Second Normal Form

Menu Structure

As with the Projectile Motion Simulation, the interface layout will be done using XML and a corresponding controller class will handle user interactions with the menu. As the ***PhysicsLab*** class holds the nifty interface, it will need to know which screen is active so that it can call the update method of that screens controller class in order to run the chosen simulation. This will be achieved by using java enumeration types, which will hold a number of values that will correspond to one of the screens in the interface. When a controller then becomes active, it sets the enumeration types value to the value corresponding to itself in its `onStartScreen()` method causing the ***PhysicsLab*** class to call its update method.

Menu - Implementation

Menu Layout

The layout of the menu was done in a similar manner to that of the projectile motion simulation. The main component of the menu is the List box holding the simulations that can be selected by the user. When the user selects a simulation from the list, its data is displayed in a panel next to the list box. The code for the list box is as follows.

```
<panel height="100%" width="30%" align="center" childLayout="vertical">
   ...
   <panel height="50%" width="100%" align="center" childLayout="center">
      <control name="listBox" id="sim_list" align="center" valign="center"
      horizontal="optional" displayItems="10" selection="single"
      forceSelection="true" visibleToMouse="true" />
   </panel>
</panel>
```

Figure 14 - List box layout

The List box is positioned by nesting two panels. The main panel is positioned horizontally adjacent to the panel holding the simulation data but is sized slightly smaller than half the screen width, as the simulation data panel holds much more data and needs more screen space as a result. The List box is added to the screen by adding a control element with the name ‘listbox’.

As it is necessary to refer to the List box in the ***MenuController*** class for handling user selections, it’s ‘id’ attribute is also specified as ‘sim_list’. The horizontal scroll bar of the interface is set to ‘optional’, meaning that it is only displayed when required, whereas the vertical scrollbar attribute is not specified and is always displayed by default. Finally, the list box is given enough space to display 10 simulations at one time, and is given ‘single’ selection mode, meaning only one item can be selected at a time. The ‘forceSelection’ attribute determines whether a mouse click on a list item immediately selects it and the ‘visibleToMouse’ attribute specifies whether or not the user can interact with the list box.

The rest of the menu is layout is done in a similar fashion. Each of the control elements are contained in a panel to give greater control over the positions of items on the screen. The menus title and background are provided by using XML image elements that specify the filename and dimensions for the image. Any necessary resizing is done automatically by nifty. The finished menu is shown below.

Figure 15 - Main Menu

Database Implementation

The database was constructed using a database plugin tool for the JMonkeyEngine, allowing tables to be contructed and relationships created using automatically generated SQL code. In order to allow interaction between the program and the database, a simple helper class ***DataSource*** was created by modifying a code sample from the book 'big Java' (Horstmann). It is used to read in database properties from a properties file, such as the databases url, and establish a connection to the database using the ***DriverManager*** class in the java sql library. The ***DataSource*** static `getConnection()` method provides a connection using the following code.

```
public static Connection getConnection() throws SQLException

{
   return DriverManager.getConnection(url, username, password);
}
```

Figure 16 - The DataSource.getConnection() Method

The method passes the data held by the ***DataSource*** class to the ***DriverManager*** class and creates a connection to the database, providing the data passed to the `getConnection()` method is correct.

As specified in the design, the menu's controller class makes a connection to the database in its `onStartScreen()` method on its first load in order to retrieve data for the menu and place it in an array for further use. This is achieved by exectuting a 'SELECT * FROM simulations' SQL query using functionality from the ***Statement*** class in the java SQL library. The code for this is detailed below.

```
if(isFirstLoad)
{
   //get the menu list box
   listBox = screen.findNiftyControl("sim_list", ListBox.class);
   Connection connection;
   try{
       //initialize the data source and create a database connection
       DataSource.init("src/mygame/database.properties");
       connection = DataSource.getConnection();
       try{
           //keep track of the number of results
           int count = 0;
           //prepare an SQL statement
           Statement statement = connection.createStatement();
            //put the results of the query in a result set
            ResultSet result = statement.executeQuery("SELECT * FROM
            simulations");
            //loop through the results and store them in the array
           while(result.next())
           {
               simData[count][0] = result.getString("name");
               simData[count][1] = result.getString("description");
               simData[count][2] = result.getString("image");
               count++;
           }
        }
       //always close the database connection
        finally{
           connection.close();
        }
   }
   catch(Exception e)
   {
       e.printStackTrace();
   }
   //add all simulations to the list box
   populateMenu();
}
```

Figure 17 - Connecting to the database

The code above is only executed once during each run of the application as the ***PhysicsLab*** class only needs to instantiate screen controllers when the program starts. This can be exploited to prevent having to connect to the database each time the menu is opened. The list box is also setup in the above code, as once all the items have been added to the list, they are not cleared when screens are switched unless done manually by the user.

As the program uses a derby database, it needs to know the location of the appropriate driver for the database to allow interaction with java. This was done by copying the derby driver .jar file from the java SDK and placing it in the simulations 'mygame' folder. The file path for the drivers was then added to the programs classpath using the following command line argument: ' -classpath src/mygame/derby.jar'.

Menu Controller

The ***MenuController*** class is structured in a way similar to the controller class for the projectile motion simulation. The most significant differences between these controllers is in the way user interactions are handled due to the different components. It has already been seen that part of the ***MenuControllers*** responsibility involves interacting with the embedded database. Its other main responsibility involves handling selections from the simulation list box and using the selection to display the appropriate information back to the user. This functionality is handled in a similar fashion to handling input from sliders. First, events from the list box are subscribed to, then the users selection is retrieved by the method below and used to update the simulation information displayed to the user. The code for this is shown below.

```
@NiftyEventSubscriber(id="sim_list")
public void onSimListBoxSelectionChanged(final String id, final
ListBoxSelectionChangedEvent<Integer> event)
{
      //retrieve the users selection from the event
      List<Integer> selection = event.getSelectionIndices();
      //only one selection at a time, so set the selected sim to the
      //first value in the list of results
      currentSim = selection.get(0);
      //display data on the selected sim
      Element title =
      nifty.getCurrentScreen().findElementByName("sim_title");
      Element description =
      nifty.getCurrentScreen().findElementByName("sim_desc");
      title.getRenderer(TextRenderer.class).
      setText(simData[currentSim][0]);
      description.getRenderer(TextRenderer.class).
      setText(simData[currentSim][1]);
      //show the image of the selected sim
      Element img = nifty.getCurrentScreen().findElementByName("sim_img");
      NiftyImage newImage =
      nifty.getRenderEngine().createImage(simData[currentSim][2], false);
      img.getRenderer(ImageRenderer.class).setImage(newImage);
  }
```

Figure 18 - Handling list box selections

The code above retrieves the index of the selected item in the list and passes the value to the 'currentSim' variable. This variable is then used to find the appropriate simulation data that was previously stored in an array and passes these values to each of the labels and image tags that display the data to the user. The 'currentSim' variable is then used in the start buttons `startSim()` method, to determine which of the simulations should be started via a switch statement when the user clicks it.

Menu - Testing

Interface Testing

As the menu is composed of only interface components, testing the menu consisted of checking each of the interactive components on the interface to ensure they worked correctly. The observed outcomes were documented and compared with the expected outcome of using each component. The table below summarises the tests performed and their results.

Table 5 - Menu Test Results

Tested Component	Expected Outcome	Observed Outcome
Simulation List Box	Clicking an item in the list should cause it to become highlighted, indicating a selection has been made. The simulation information displayed to the user should be updated, displaying information and an image of the selected simulation.	As expected. A problem occured when selecting simulations with longer descriptions. The description text overlapped other elements on the screen and did not stay within the dimensions specified in the XML layout.
Start Button	If a selection has been made from the list box, clicking the start button should cause the corresponding simulation to start and close the menu screen. Otherwise it should do nothing.	Simulations start correctly on the first click of the start button. However returning to the menu and pressing the button again causes the simulation to load with some of its interface components unresponsive.
Quit Button	Clicking the quit button should cause the application to close.	As expected.

A major problem was encountered when testing the menu. When starting the projectile motion simulation, returning to the menu and starting the same simulation again, the particle settings section of the interface failed to load correctly. Attempting to use any of the components in the section with the mouse did not work, yet they were still selectable by using the tab key and could be altered using the arrow keys. A possible way to resolve this issue would be to explicitly set each components 'visibleToMouse' attribute in the corresponding simulation controller class, as it appears to become disabled when restarting a simulation.

Momentum Simulation – Design

Overview

The momentum simulation is the second simulation that will be included in the system and will demonstrate the principle of conservation of momentum. This will be demonstrated by colliding two particles together in a perfectly elastic collision, whereby the kinetic energy of the two particles is preserved even after colliding.

The development of the momentum simulation will follow a process that is almost identical to the development of the projectile motion simulation. The only significant difference between the two simulations is that their particles behave differently and hence have differing update algorithms.

In order to implement the simulation, several new classes will be required. The ***MomentumSim*** class will be responsible for initializing and updating the simulation and will inherit its core functionality from the ***Simulation*** class in the same way as the ***ProjectileMotionSim*** class. The ***MomentumParticle*** class will be used to create the particles needed for the momentum simulation, containing the particles update code and checking for collisions between two momentum particles. It will inherit its core functionality from the ***Particle*** class, just as the ***Projectile*** class in the projectile motion simulation did. A ***MomentumController*** class will also be added to handle interactions with the simulations nifty interface.

Interface Design

The momentum simulations interface will be in the same format as the interface used in the projectile motion simulation. The settings available in the simulation options tab will be exactly as they were in the projectile motion simulation, providing changeable gravity, camera angles, run time and tracer settings. As the momentum simulation involves two particles, two tabs will be required providing sliders to change each particles mass and velocity, as these are the two properties that affect the outcome of the collision between the two particles. In addition, two sets of labels will be required, displaying each particles velocity throughout the simulation, so the user can see how this value changes after the two particles collide. The layout for each particles settings tab is shown below.

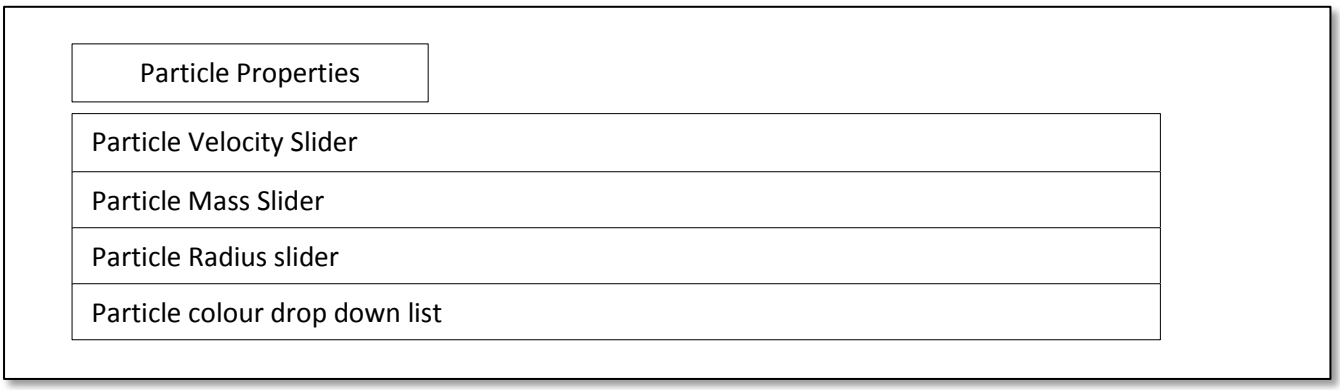


Figure 19 - Particle properties tab

The layout is similar to the projectile settings tab for the projectile motion interface. The difference in this tab is that the mass slider replaces the launch angle slider, as particles will collide in a straight line and therefore an adjustable launch angle is not required.

Momentum Update Algorithm

The momentum particle update algorithm will use the velocity and mass values set by the user from the interface in order to calculate the velocities of each ball after they have collided. Initially, each particle will move toward the other at the velocity specified by the user. when a collision between the two particles is detected, the mass and velocities of each particle will be used to calculate new velocities for each particle, causing them to move away from eachother as would be expected. Given that the first particle has an initial (before collision) velocity u_1 and mass m_1 and the second particle has initial velocity u_2 and mass m_2, then the final velocity v_1 of the first particle can be found using the equation:

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

The final velocity of the second particle v_2 can then be calculated by rearranging the equation:

$$v_1 - v_2 = u_1 - u_2$$

The momentum simulation update algorithm will update each of the momentum particles by calling the ***Simulation*** class update method and then check for a collision between the two particles. If a collision is detected, then the new velocities for each particle will be calculated by calling the ***MomentumParticle*** class `calcFinalVelocity()` method. The pseudo code for this is shown below.

```
Algorithim SimulationUpdate(timePerFrame)
Input: timePerFrame, the time taken to update from one frame
to the next.
Output: new position of particle, time elapsed and x velocity.
superclass.update(timePerFrame)
if playSimulation then
     if particle1 collidesWith particle2
     calcFinalVelocity(particle1, particle2)
if resetSimulation then
resetInterface()
resetParticles()
```

Figure 20 - Momentum Simulation Update Algorithm

The `calcFinalVelocity()` method in the update algorithm (above) will compute the new velocities for each particle according the equations explained above and will be part of the ***MomentumParticle*** class.

Momentum Controller

The momentum simulation controller class is very similar to the projectile motion simulation class previously implemented. It will inherit its core features from the abstract ***SimController*** class. As it has slightly differing options to the projectile motion simulations interface, the only difference between the two classes are the event handling methods for the additional components. Such similarities are beneficial as this has allowed most functionality to be abstracted to the ***SimController*** class and resulted in less development time for implementing controllers.

Momentum Simulation - Implementation

Interface Layout

The momentum simulation interface is composed of three tabs, two tabs holding the settings for each of the particles and one tab holding the same simulation settings present in the projectile motion simulation. The same button menu is also included and two separate sets of labels display data to the user regarding each of the simulations particles. Again, an XML document is used to specify the layout and types of components to be used in the interface. The XML sample below shows how sliders are added to the particle tabs of the interface.

```
<panel id="label_panel" childLayout="center" width="60%" height="100%">
      <control id="p1_vel" name="horizontalSlider" min="0" max="50"
      initial="10" stepSize="0.01" buttonStepSize="1" width="90%"
      align="left"/>
</panel>
```

Figure 21 - Slider controls

As with all interactive screen elements, the slider is defined using a control element. The 'name' attribute specifies that the control should be a horizontal slider and its 'id' attribute enables it to be referenced by the controller class for event handling. The 'min' and 'max' attributes set the minimum and maximum values that can be selected by the user. Finally, the 'stepSize' attribute sets the smallest possible increment of the slider to 0.01 and the 'buttonStepSize' attribute sets the amount to increment the slider value by if the user presses the buttons on either end of the slider, rather than dragging the slider with the mouse. The screen shot below shows the completed interface.

Figure 22 - Momentum Simulation Interface

Momentum Update Algorithm

With the exception of the differing update algorithms, the implementation of the momentum simulation was the same as that of the projectile motion simulation. The Momentum simulations update method updates each of the particles by making a call to the ***Simulation*** class update method, which loops through the list of particles attached to the scene and updates them. If the simulation is playing, the momentum simulation update method then checks if a collision between the two particles has occurred. If this is found to be the case, a call to the ***MomentumParticle*** class static `calcFinalVelocity()` method is made, which computes the velocity for each particle after they have collided. The code for this algorithm is below.

```
//update the particles through the simulation class
super.update(tpf);
//simulation is playing
if(!isPaused)
{
   //check for a collision if particles haven't yet collided
   if(particle1.collidesWith(particle2) && collided == false)
   {
       //calculate the velocity of the particles after the collision
       MomentumParticle.calcFinalVelocity(particle1, particle2);
      //only need to check for a collision once, else particles will get
      //stuck
       collided = true;
   }
}
```

Figure 23 - Momentum Simulation Update

As it is known that the two particles colliding in a straight line and both particles are spherical, collisions can be detected using a basic collision detection algorithm. the `collidesWith()` method shown above, checks if the distance of the center of each particle is less than the distance of the combined radii of the two particles. If this is the case, then the particles would appear to be touching and make the collision look more realistic. A boolean value of true is returned by the method if a collision is detected and false if otherwise. The method is shown below.

```
public boolean collidesWith(MomentumParticle other)
{
    //if the two particles are sufficiently close, a collision has occured
    if(position.distance(other.position) <= (radius + other.radius))
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Figure 24 - Collision Detection

Momentum Simulation Controller

Implementation of the momentum simulations controller class did not go as expected. As the simulation has two particles, each with its own set of tracers, selecting a tracer belonging to either of the particles caused the data labels of both particles to be overwritten with the tracers data. The desired outcome was that if a tracer was selected, the particle it belonged to would be identified and only the data for that particle would be overwritten. In order to correct this, alterations were made to the controllers `updateLabels()` method.

Originally, the method retrieved the both particles from the simulation, using its get methods to access the particles properties and pass them to the display labels. An if statement to check if a tracer was selected caused the method to instead set the labels to the tracers properties if it evaluated to true. This was found not to be sufficient when dealing with more than one particle, as this cause all labels to be overwritten with the data from the tracer. Instead, the method had to use two more complex if statements before updating each set of labels. These new if statements first check if a tracer is selected, then check that the tracer belongs to the particle whose labels are about to be overwritten. The overwrite is only allowed to proceed if both conditions are true. Part of the new method can be seen below .

```
public void updateLabels()
{
    //retrieve the first particle
    Particle p = (MomentumParticle)sim.particles.get(0);
    //is a tracer selected and does it belong to this particle?
    if(sim.selectedTracer != null && sim.selectedTracer.parent == p)
       //if yes, overwrite the first particle with the selected tracer
       p = sim.selectedTracer;
    //update the display labels
    Element label =
    nifty.getCurrentScreen().findElementByName("p1_x_velocity");
    label.getRenderer(TextRenderer.class).setText("" + MathHelper.round(
    -p.getXVelocity(), 2));

    ...
    //retrieve the second particle
    p = (MomentumParticle)sim.particles.get(1);
    //overwrite the particle if a tracer is selected and belongs to this
    //particle
    if(sim.selectedTracer != null && sim.selectedTracer.parent == p)
       p = sim.selectedTracer;
    //update the display labels
    label = nifty.getCurrentScreen().findElementByName("p2_x_velocity");
    label.getRenderer(TextRenderer.class).setText("" + MathHelper.round(
    -p.getXVelocity(), 2));
    ...

}
```

Figure 25 - updateLabels method

For simplicity, the momentum simulation only allows one tracer to be selected at a time. Even though it would be desirable to be able to select a tracer from each particle and be able to compare them, this functionality would require a complete rewrite of the Tracer selection algorithm in the ***Simulation*** class (detailed in the projectile motion implementation section).

Momentum Simulation - Testing

Particle Testing

In order to ensure the calculations within the momentum particles update method are correct, the simulation was run five times and the xVelocity values generated for each particle by each run sampled after the two particles collided. These values were then compared against manually calculated values for each of the particles. The results for each of these runs and the setups for each are shown below. mass and initial velocity values are omitted as they are specified by the user and not computed by the simulation.

Run 1 Setup

- Particle A initial velocity: 7 m/s
- Particle A mass: 15kg
- Particle B initial velocity: -3 m/s
- Particle B mass: 10kg

Run 2 Setup

- Particle A initial velocity: 13.15 m/s
- Particle A mass: 21.1kg
- Particle B initial velocity: -15.1 m/s
- Particle B mass: 23.7kg

Run 3 Setup

- Particle A initial velocity: 40 m/s
- Particle A mass: 50 kg
- Particle B initial velocity: -50 m/s
- Particle B mass: 20.62kg

Run 4 Setup

- Particle A initial velocity: 0 m/s
- Particle A mass: 15.91 kg
- Particle B initial velocity: -30.19 m/s
- Particle B mass: 9.9kg

Run 5 Setup

- Particle A initial velocity: 25 m/s
- Particle A mass: 20 kg
- Particle B initial velocity: -25 m/s
- Particle B mass: 20kg

Table 6 - Particle Test Results

Run No.	Expected Particle A Final Velocity	Actual Particle A Final Velocity	Expected Particle B Final Velocity	Actual Particle B Final Velocity
1	-1m/s	-1m/s	9m/s	9m/s
2	-16.74m/s	-16.74m/s	11.51m/s	11.51m/s
3	-12.56m/s	-12.56m/s	77.44m/s	77.44m/s
4	-23.16m/s	-23.16m/s	7.03m/s	7.03m/s
5	-25m/s	-25m/s	25m/s	25m/s

As the table shows above, the momentum particle update algorithm correctly calculates final velocity values for each of the particles in every run to two decimal places. Note that negative values refer to the direction of the particles movement, where particle A is always assumed to be the forward moving particle.

Interface Testing

To ensure the momentum simulation interface works as intended, each of the components were tested and the observed outcome documented and compared with the expected outcome of using the component. The table below summarises the tests performed and their results.

Table 7 - Interface Testing Results

Tested Component	Expected Outcome	Observed Outcome
Play/Pause Button	Simulation should start with defined settings on first click, successive clicks should toggle pausing of the simulation.	As expected.
Reset Button	If the simulation has finished or is currently playing, the projectile should return to the start position, tracers should be cleared and the interface reset.	As expected.
Toggle Options Button	Should hide/ show the option panel on each click depending on whether it is already visible.	As expected.
Quit Button	Clicking should close the application.	As expected.
Menu Button	Clicking the button should return the user to the menu.	As expected but restarting the simulation causes unresponsiveness of the interface.
Particle A Velocity Slider	Altering the slider position should change the initial velocity of particle A, causing the adjacent label to be updated with the selected value.	As expected.

Particle A Mass Slider	Altering the slider position should change the mass of particle A, causing the adjacent label to be updated with the selected value.	As expected.
Particle A Radius Slider	Altering the slider position should change particle As radius, causing the adjacent label to be updated with the selected value and cause the particle to be resized to the specified radius.	As expected, Though wrong value is displayed on application start until the slider is moved, at which point value displayed is correct.
Particle A Colour Drop Down	Selecting a value from the drop down list should cause the particle to immediately change to the specified colour.	As expected.
Particle B Velocity Slider	Altering the slider position should change the initial velocity of particle B, causing the adjacent label to be updated with the selected value.	As expected.
Particle B Mass Slider	Altering the slider position should change the mass of particle B, causing the adjacent label to be updated with the selected value.	As expected.
Particle B Radius Slider	Altering the slider position should change particle Bs radius, causing the adjacent label to be updated with the selected value and cause the particle to be resized to the specified radius.	As expected, Though wrong value is displayed on application start until the slider is moved, at which point value displayed is correct.
Particle B Colour Drop Down	Selecting a value from the drop down list should cause the particle to immediately change to the specified colour.	As expected.
Camera Angle Drop Down	Selecting a value from the drop down list should cause the camera to snap to the selected camera angle.	As expected.
Gravity Slider	Altering the value of the slider should have no effect on this simulation As only movement along the x axis is considered.	As expected.
Run Time Text Field	Entering a value into the field should cause the simulation to run either until the value (in seconds) is reached (minimum of 1, maximum of 60), or the projectile hits the floor. Values outside the accepted range will be converted to 1 is less than 1 or 60 if more than 60.	As expected, though if the user enters a value outside the accepted range the value is not corrected in the text field for the user to see.
Tracer Text Field	Entering a value into the field should cause tracers to be created the specified number of times per second behind the particle(minimum of 1, maximum of 10 per second). Values outside the accepted range will be converted to 1 is less than 1 or 10 if	As expected, though if the user enters a value outside the accepted range the value is not corrected in the text field for the user to see.

	more than 10.	
Tracers	Selecting a tracer from the simulation should cause it to turn white in colour and replace the currently displayed data with data on the projectile at the time the tracer was created. Any currently selected tracers should be deselected if a new Tracer becomes selected.	As expected.

The results of interface testing show that only a few minor problems exist with labels not being correctly set at start up. The majority of significant issues that could have arisen were addressed when testing the projectile motion interface and have therefore been avoided for the momentum simulations interface.

One potentially significant problem that remains however is that when returning to the menu and opening the same simulation that was just closed, some of the settings tabs become unresponsive to mouse events. Attempts were made to resolve this when testing the interface but were not successful. The cause of the problem is unclear and is possibly a bug within nifty, as this is still a relatively new technology.

Evaluation & Conclusion

Results

Having implemented the majority of the functionality specified in the system requirements, it was decided that a number of features should be dropped from the system, in favour of ensuring existing features and simulations were complete and working to a good standard. The main features that were dropped from the system were the pulley systems simulation, which was only partially implemented and the question system allowing users to create and answer questions on each simulation.

I believe that the reason for having to drop these components is down to an over ambitious requirements specification for the given development timescale. During the development of the system there were also a number of unexpected changes to the development process that were unaccounted for by the time plan, including the redevelopment of the interface with nifty and the implementation of the tracer feature. These unplanned events resulted in significant delays in development and no alterations were made to the time plan to counteract the effects of the delays, which meant that other desirable features had to be excluded from the system.

The final system does have a number of strengths. Although not without problems, I believe the nifty interface to be a much more suitable choice for the interface for the system than a java swing one. A particular benefit is niftys built in screen management capabilities as this served the systems needs for switching between simulations very well. Use of nifty also saved considerable time in having to develop a custom screen manager system that would have been required for a swing interface.

Another positive aspect of the system is its 3D graphics. I feel that this was a good development choice as it provides a more immersive experience and would likely be more interesting to an A level student as a result. In addition, displaying particle properties and developing selectable tracers was also a strong point of the system, as this allows for a much more detailed analysis of a particles behaviour and would be beneficial in an educational environment.

A drawback of the system was the use of XML interface layouts. Although this method of development was relatively quick to learn, this benefit was quickly negated by the loss of the ability to exploit java inheritance with the original swing interface. As a result of this, the XML layout specifies identical components for each of the screens multiple times and could have been avoided if it was implemented in java.

I also feel that given many features specified by the requirements were dropped, the implementation of a database for storing the systems data was unnecessary and adds a significant loading time to the menu screen. As the system currently only requires menu data to be loaded from the database, I feel a simple system for reading in menu data from text files would have been a more suitable and faster option. Despite this, the use of a database does provide the possibility of future expansions such as new simulations or the ability to create and save custom questions without having to worry about designing a new system for storing data.

Feedback

In order to test the systems suitability for its intended audience, a meeting was arranged with an A-level physics teacher in order to demonstrate the systems capabilities. After viewing both the projectile motion and momentum simulations, he provided both positive and negative feedback on several aspects of the system.

A particular positive point noted was that he noted the simulations were highly relevant to the curriculum for students in his school and that the simulations would be useful for students to compare their own calculations with or for demonstrations using a projector or interactive whiteboard. These comments suggest that the system would indeed be appropriate as an aid to active learning, which was one of the key aims for developing the system.

Some negative points the teacher noted were that some aspects of the interface may be distracting for students and were not necessary such as the ability to change the colour of particles. His concern was that such features would take the focus from the mathematical concepts behind the simulations, and that the system would become more like a game than an educational program. A summary of the teachers evaluation can be found in the appendices.

Conclusion

Overall, I believe the system has satisfied its main aim of providing a range of simulations that could be used in a physics class to help students understand mathematical concepts in physics. This has been enforced by the feedback given by an A-level physics teacher who was generally positive about the system and felt that it would be beneficial for physics students.

Development of the prototype system was a highly beneficial component of the development process. The prototype enabled potential issues that could have been damaging to the quality of the final system to be identified and resolved before this could happen. I believe this greatly contributed to the success of the project as this allowed a much clearer visualisation of how the final system might look and whether it would be able to meet the specified requirements from an early stage.

Though it would have been desirable for the system to be useable as a self-contained tool for active learning in physics, I feel that the system satisfies its requirements to a good standard. Given a larger timescale to implement more of the desirable features identified in the requirements, I believe that the system could have become a much more useful tool for students and would have been suitable for students to use without supervision from an instructor.

As the system currently stands, I feel that it would be suitable for use in a classroom environment under supervision of an A-level physics instructor, as an aid to active learning rather than a self-contained solution. This is because the system does not currently provide any explanations of the mathematical formulae involved but provides a visual demonstration of how they work. It is assumed that an A-level instructor would be on hand to explain the simulations and the required mathematical concepts to students while using the system.

Future Work

As the system is not fully complete, there is a great deal of room for expansions to be made. The simulation only has two working simulations at present, covering a small portion of the A-level physics syllabus. Using the current systems framework, adding additional simulations could be done quickly as the major functionality for all simulations is contained in the abstract ***Simulation*** class. This would allow future developers to concentrate on developing update algorithms, particularly with regards to implementing the equations required for realistic particle movement. Additional simulations could cover further physics concepts at A-level such as restitution, friction or centrifugal forces. The system could also include simpler GCSE physics simulations, covering concepts such as introductory acceleration and velocity.

Further expansions to the system could also include adding extra functionality to the system. This might include a built in wiki, providing detailed explanations on the math behind each of the simulations which could be written using XML to maintain a consistent format across all articles and parsed using java XML api's.

The system could also benefit from the question system that was not implemented in this project. Adding such a system would greatly benefit the systems value if used for active learning, as this would enable students to engage with the system more effectively than is possible at present. As the database structure already exists for this system, development could focus on implementing the system immediately without having to design a storage solution.

A number of utilities could be added to the system to improve its ability to act as a self-contained active learning solution. An embedded calculator could be useful to the system if combined with a question system as previously mentioned, as this would enable users to calculate their answers to questions. The calculator could be accessible from a simulations button menu and be tailored for solving questions for each specific type of simulation and would be quicker than using a regular calculator.

One final expansion to the system could be to integrate a physics engine into the system. Adding a physics engine would provide a greater level of realism to how particles move and interact with the environment. A good option for this would be the java bullet physics engine as this is already integrated with the JMonkeyEngine framework used by the system. This was not included in the system in this project as it was deemed too realistic for simple A-level simulations and would have caused the system to become much more CPU intensive, making it unsuitable for older computers.

Reflection On Learning

During development of the physics lab project, I was required to become familiar with several technologies and concepts that I had previously never encountered and had no experience of using. These technologies include Nifty-GUI, in particular its XML layout system and menu controllers, as well as java database integration and the practice of using update loops in each of the simulations. Working with these new technologies has given me an appreciation of the importance of researching existing technologies that may be useful if included in the development of a system to meet its objectives. It has also given me confidence that I am able to learn to use technologies that are new to me, a skill that could be beneficial in a working environment where unforeseen problems may arise.

I have also gained an understanding that keeping a good time plan, which accounts for the fact that unforeseen problems will inevitably occur is an important factor in the success of a project. The time plan that I developed for my project was overambitious and as a result led to me falling short of my specified deadlines from an early stage in the project. In addition, I allowed little room for addressing unexpected problems and the combination of these factors had a damaging impact on the outcome of the project as several planned features had to be dropped. This problem could have been resolved by taking more care in developing a more realistically achievable set of requirements and allowing more time to deal with any problems that might occur during development.

A key component of the projects implementation was the use of inheritance and polymorphism. I feel that exploiting these traits of object oriented programming were crucial factors of the projects success, as development of particle and simulation classes was significantly shortened by abstracting many of the common functionalities of each to abstract base classes. By using such techniques, I have also learned the importance of good design as by using these techniques, careful consideration had to be given to the relationships between each of the classes in the system. This also led me to produce class diagrams to help visualise such relationships, which I feel were an important part of the systems design.

Overall, I feel that the development of the physics lab project has given me the confidence to cope with the complexity of developing large scale systems. It has also given me an understanding in the importance of good design and testing of the system, and that this can have a significant outcome on the success of a project. I have also gained an insight into development using 3D graphics technologies and the importance of an appropriate interface. I believe that the skills I have gained from this project will be of great benefit to me in any future career.

Appendix A – Requirements Specification

The full and amended requirements specification below will be used to evaluate the success of the project.

Functional Requirements

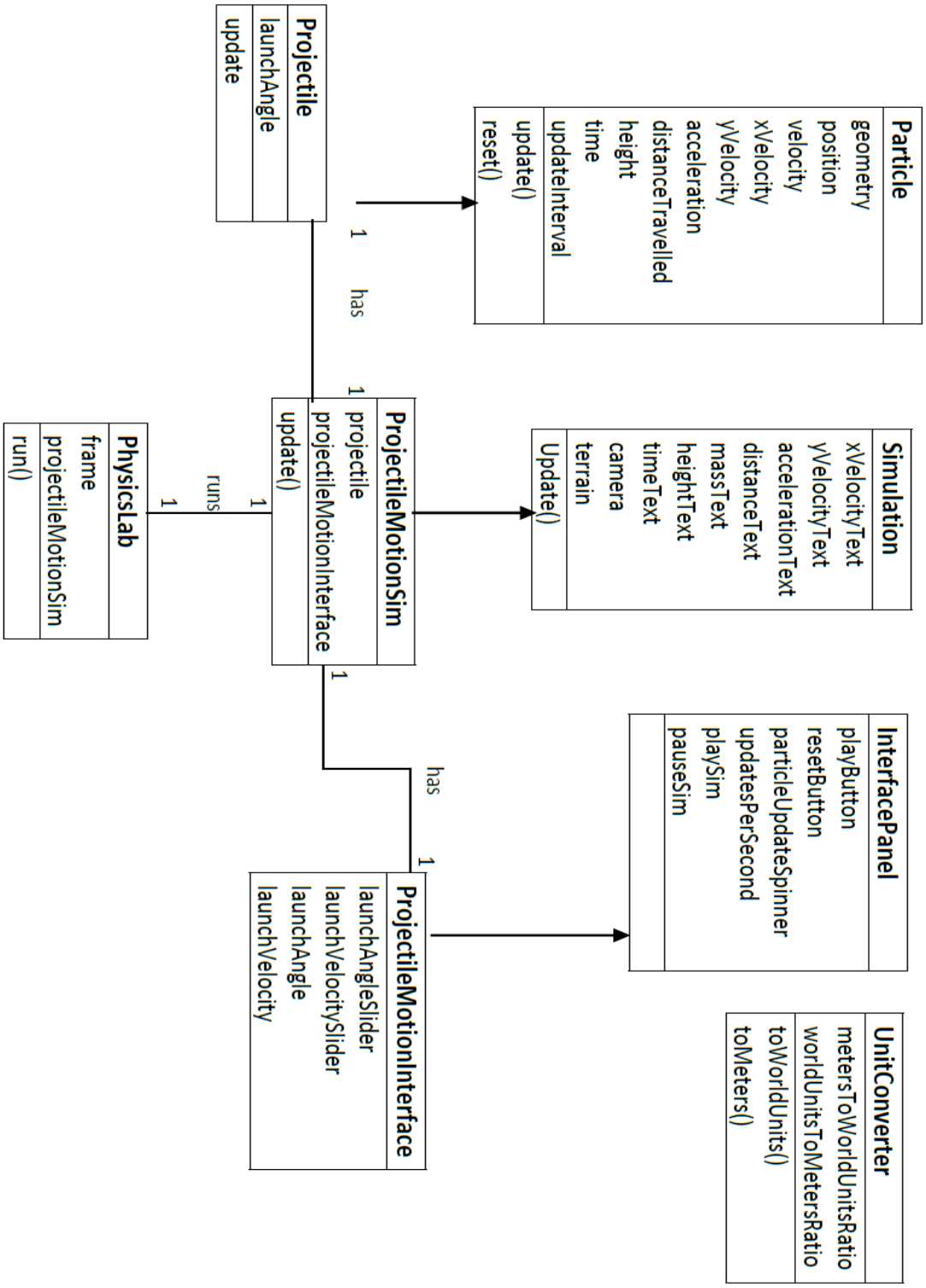
- The system shall have simulations that demonstrate concepts covered in A-level physics such as:
    - Equations of motion, including Projectile motion
    - Pulley systems
    - Collisions and momentum
- Simulations will have 3D graphics.
- Simulation particles will conform to the appropriate physics for the given simulation (e.g. projectiles will move as defined by the corresponding projectile motion equations).
- Simulations will allow interaction through the manipulation of an object's properties via the user interface. Properties that will be changeable will include initial velocity, mass and launch angles.
- Simulations will have a camera that can be moved around the scene by the user so that the scene can be viewed from various angles.
- Simulations will display data related to the objects mass, acceleration and velocity in a graphical manner so that it is easier for a student to understand the underlying mathematical concepts.
- Simulations should have functionality that will allow the user to create questions on the given simulation (automatically or manually) and provide a means to answer and check these questions.
    - The user should be able to save any questions produced by the user for that simulation.
- The system shall have a menu from which will display a list of available simulations, including an image to help identify the simulation.
- If a simulation is selected, the system shall display the simulation, as well as an interface providing options to adjust the simulation as required. These options will always include (but will not be limited to):
    - play, pause and reset options for the simulation
    - return to menu and exit application options
- Where appropriate the interface will allow the user to filter the information displayed on a given object, such as its velocity, so that a user can focus on any given area they choose.
- The system will provide help for how to navigate and use components appropriately. As the system is designed for use by physics instructors, it is assumed that no help needs to be provided on the underlying physics of each simulation.

Non-Functional Requirements

- The system will be suitable for use by novice users. This will be achieved with the following:
    - Risk of user error will be minimised by restricting user input using GUI components such as sliders, combo boxes and spinners. Only values within a defined range will be allowed.
    - The system will be developed with consideration to the Principles of usability[2] (Dix et al.):
        - Learnability - the interface will be simplistic, components logically grouped and behave in a predictable manner. If the user alters values, then the system will display these changes to the user immediately wherever possible. Common GUI components will be used so that the system feels familiar to the user.
        - Flexibility - The user will be able to customize the system by being able to alter values in the simulation and to show/hide non-essential components as needed.
        - Robustness - The system will provide feedback to the user through progress bars when busy processing and will warn the user when irreversible changes are about to be made. The user will be able to reset simulations restore default settings, and also save setups as desired.
- Simulations will be sorted in the menu by topic and level (GCSE or A-level).
- The system will ideally be usable on linux, windows and mac environments.
- Simulations will be graphically detailed, using models, textures and lighting techniques in order to be as appealing to students as possible.
- The user should be able to easily exit the program at any point.
- Simulations should have a frame rate of at least 30fps, so that they appear to animate smoothly.

Appendix B – Prototype Class Diagram

The following diagram shows the structure of the prototype system. This is different to the structure of the projectile motion simulation in the final system and the interface classes are no longer included in the system.



Appendix C – Feedback From Physics Teacher

The following document summarises the feedback given by Mr John Ivins on the final system. It outlines his thoughts on the system, as well as some suggestions he felt could benefit the system.

Observations on Mechanics project

- Very engaging user interface suitable for AS and A2 students.
- Projectiles and momentum applications address relevant parts of the AS and A2 curriculum (AQA)
- Very easy to see how the application could be used to teach projectiles and momentum, either:
    - As a class demonstration using a projector and white board.
    - As an engaging way for students to verify their calculations.
- Suggestions:
    - Would benefit from having the output information made more obvious on both applications.
    - A few of the features could be deleted or hidden in a setup menu as they would distract students (changing the colour and shape of the ball.

References

Hohmuth, J. 2009. *About Nifty GUI* [Online]. Available at: http://sourceforge.net/apps/mediawiki/nifty-gui/index.php?title=About_Nifty_GUI [Accessed: 09 April 2012]

Hohmuth ,J et al. 2011. *Nifty GUI 1.3.1 The Missing Manual* [Online]. Available at: http://nifty-gui.lessvoid.com/ [Accessed: 12 April 2012]

Horstmann, C. 2008. *Big Java, 3rd Ed.* Hoboken, NJ : Wiley

JMonkeyEngine. 2012. *Picking Tutorial* [Online]. Available at: http://jmonkeyengine.org/wiki/doku.php/jme3:beginner:hello_picking [Accessed: 16 April 2012]

Wikipedia. 2012. *Blender (Software)* [Online]. Available at: http://en.wikipedia.org/wiki/Blender_%28software%29 [Accessed: 10 April 2012]