Section 0
  Table of Contents 0-1

Section 1
  Introduction 1-1

Section 2
  Functional Requirements 2-1
    Level 1 - Required 2-1
    Level 2 - Primary Goals 2-1
    Level 3 - Secondary Goals 2-1
  Non-Functional Requirements 2-1
    Level 1 - Required 2-1
    Level 2 - Primary Goals 2-2
    Level 3 - Secondary Goals 2-2

Section 3
  Use Case Diagram 3-1
  Installer Use Cases 3-1
  User Use Cases 3-2
  Detailed Use Cases
    UCI1 - Install Qt 3-2
    UCI3 - Build Source 3-3
    UCU6 - Create Forces 3-3
  Scenarios
    UCI4 - Build 3-4
    UCI4b - Build insmodErr 3-4
    UCI4c - Build qmakeErr 3-5

Section 4
  Requirement Traceability Matrix 4-1

Section 5
  Screen Shots - Linux 5-1

Section 6
  Glossary 6-1

Section 7
  References 7-1

Section 8
  Team Member Contributions 8-1

Special thanks to our advisors, including, but not limited to:
  Dr. Fred Harris
  Dr. Sergiu Dascalu
  Brian Westphal
  Jeff Stuart
  Joe Jaquish

Without you, none of this would be possible.

This report was written using Open Office 1.1.2 on a Revision D iMac running Mac OS 10.3.8, thus proving that open source software can keep a computer running far longer than its sensible life span.
INTRODUCTION

Thraxion - What Is It?

Thraxion is a software project designed to allow the user to easily design, through the use of a computer mouse, a three dimensional scene that can be performed in real time. A rudimentary physics engine, already built by Jon Studebaker and Justin Gerthoffer, will be the basis for the physics controlling the graphics on the screen. Ease of use and making the existing engine more accurate are two of the major goals of this project.

The software will allow the building of scenes to display what would happen under a set of initial conditions set by the user. The environment will allow for spheres colliding with solid objects, distance constraints between objects and, hopefully, solid object collision. It will allow for the saving and loading of three dimensional scenes for running at later times. The users will be able to choose pre-built objects and create almost any environment they wish with them.

Almost anyone should be able to benefit from the program since ease of use is a major goal. Target audiences include younger students who might wish to see what they have learned about basic Newtonian physics in action, as well as programmers looking for an easy-to-use three dimensional collision simulator for a graphics engine.

The program will be novel in that it is basic, should be moderately fast, and will be easy to use but still create and manage complex three dimensional simulations. It will be perfect for quick demonstrations in 3D involving basic physics.

The potential for further development is unlimited. Some of the loftier goals we could achieve if the basic goals prove easy to obtain include:

- Use of binary-space partitioning trees to speed up simulation.
- User created objects, or user hand designed polygon surfaces.
- Accurate rigid body dynamics.
- An extremely realistic physics model.
- User interaction with live, moving scenes.
- Multi-platform support.

The project's major difficulty will be interacting with the windowing system. Accurately using the two dimensional nature of the computer mouse in a three dimensional world in an intuitive manner will prove difficult. Learning more about how to use Qt will be required as well. Bringing more speed to the project's physics engine will also be challenging, as the first version was built with speed of implementation as the primary consideration. Learning how to implement a more realistic and efficient physics engine will be necessary.

The major components of the technology are OpenGL and the Qt toolkit. The software will theoretically be cross-platform, as both OpenGL and Qt libraries exist for almost every operating system in common use. However, the software will primarily be built upon Linux. Mac OS X and Windows XP support will be explored, but primary development will not be on those systems.

Computer graphics is a major field of research and practical use. Understanding it will help all of us to learn new programming techniques and give us experience in a cutting edge field.
REQUIREMENTS

FUNCTIONAL REQUIREMENTS
LEVEL 1 - REQUIRED

FR1.1: The user shall be able to open scenes for demonstration purposes.
FR1.2: The user shall be able to save scenes for demonstration purposes.
FR1.3: The user shall be able to add text comments to scenes.
FR1.4: The user shall be able to view a help file or page within the program.
FR1.5: The user shall be able to write scenes from within the program.
FR1.6: The user shall be able to switch between various graphical modes, including wireframe mode and solid mode.

FUNCTIONAL REQUIREMENTS
LEVEL 2 - PRIMARY GOALS

FR2.1: The user will have at least four views in the graphical user interface, including an isometric view and three plane-sliced views.
FR2.2: The user will be able to pause scenes.
FR2.3: The user will be able to manipulate scenes while paused.
FR2.4: The user will be able to create scenes using a graphical user interface for the designer.
FR2.5: The user will be able to save scenes using a graphical user interface for the designer.
FR2.6: The user will be able to open scenes using a graphical user interface for the designer.
FR2.7: The user will have a set of pre-defined objects to build scenes with.

FUNCTIONAL REQUIREMENTS
LEVEL 3 - SECONDARY GOALS

FR3.1: The user may be able to reverse the flow of time of scenarios.
FR3.2: The user may switch between a scripted scenario editing mode and a purely graphical scenario editing mode.
FR3.3: The user may modify environment variables using the graphical user interface.
FR3.4: The user may view properties (initial force, initial acceleration, etc.) through a properties window in the program.
FR3.5: The user may be able to modify the lighting of scenarios for aesthetic effect.
FR3.6: The user may be able to move items around interactively during scenes.

NON-FUNCTIONAL REQUIREMENTS
LEVEL 1 - REQUIRED

NFR1.1: The program shall run on Slackware 10.
NFR1.2: The program shall run on Gentoo.
NFR1.3: The program shall run on Debian Sarge.
NFR1.4: The program shall be built using gcc 3.3.
NFR1.5: The program shall be built using Qt 3.3.
NFR1.6: The program shall require less than 128 megabytes of random access memory for proper functioning.
NFR1.7: The program shall run all scenes with at least a 25% speed increase upon completion compared to the original speed of the program at the beginning of this project.
NFR1.8: The program shall implement mouse and keyboard user interaction.
REQUIREMENTS

NFR1.9: The program shall run 5 limbs with one solid at a rate equal to or greater than 30 frames per second using a Pentium 4 running at 2 GHz with 512 MB of RAM running Slackware 10.

NON-FUNCTIONAL REQUIREMENTS

LEVEL 2 - PRIMARY GOALS

NFR2.1: The program will run on Mac OS 10.3.8 without using X11.
NFR2.2: The program’s source will compile without editing the Makefile on Linux.
NFR2.3: The program will use a scripting language for scene editing instead of manual compilation of scene files.
NFR2.4: The program will run all scenes with at least a 50% speed increase upon completion compared to the original speed of the program at the beginning of this project.
NFR2.5: The program will, on Mac OS X machines, look like a native Mac OS X application, fully mimicking the Aqua look and feel common on most Mac OS X applications.

LEVEL 3 - SECONDARY GOALS

NFR3.1: The program may run on Windows XP.
NFR3.2: The program may use a graphical user interface for scene editing, instead of requiring user knowledge of a scripting language.
NFR3.3: The program may run any scene at a rate of 30 frames per second on the machine described in NFR1.9.
NFR3.4: The Mac OS X implementation of the program may use the standard Mac OS X installer for installation of the program, instead of a completed .app bundle or requiring compilation of source code.
NFR3.5: The program may use a tree-based collision detection system to optimize the code.
USE CASES

USE CASE MODEL

THRAXION

INSTALL QT

GRAB SOURCE

BUILD SOURCE

CREATE ROPE

CREATE SPHERE

CREATE SOLID

CREATE STATIONARY OBJECT

CREATE LIMBS

CREATE FORCES

SAVE SCENE

OPEN SCENE

RUN DEMO

START SCENE

STOP SCENE

EDIT FORCES

REWIND SCENE

START PROGRAM

STOP PROGRAM

USE CASES - INSTALLER

UCI1: Install Qt
Install Qt package on target system.

UCI2: Grab Source
Acquire source code from web site or authors.

UCI3: Build Source
Using source code acquired in UCI2, build it, either using a Makefile or qmake.

UCI4: Build
Same as UCI3 only set up for different scenarios.

USE CASES - USER

UCU1: Create Ropes
Create ropes to bind two or more objects together in motion, like a necklace.

UCU2: Create Sphere
Create a uniformly round object with radius r.

UCU3: Create Solid
Create a polygonal solid of variable sides and dimensions (length, width, height).

UCU4: Create Stationary Object
Create stationary object of infinite mass, such that it is immovable.

UCU5: Create Limbs
Create a limb around which groups of objects may be attached like muscles. Essentially non-flexible ropes.

UCU6: Create Forces
Create initial force vectors to cause object, once scenario starts, to head in direction of vector with force.
USE CASES

UCU7: Save Scene
Save a user-created or user-edited scene.

UCU8: Open Scene
Open a user-created or user-edited scene.

UCU9: Run Demo
Run a demo scene provided in the program.

UCU10: Start Scene
Begin running a user-created or user-edited scene. Similar to UCU9.

UCU11: Stop Scene
Pause scene before completion, i.e. before all objects have reached a state of rest.

UCU12: Edit Forces
Change the various forces in effect in a scene, such as vector forces, gravity, friction, etc. May be possible after UCU11 is initiated by the user.

UCU13: Rewind Scene
Change the flow of time of a scene such that it reverses, like a movie.

UCU14: Zoom In
Zoom in on a given point in space using the scroll button on the mouse or keyboard commands.

UCU15: Zoom Out
Zoom out on a given point in space using the scroll button on the mouse or keyboard commands.

UCU16: Start Program
Begin running Thraxion executable - exact actions performed are platform-specific.

UCU17: Stop Program
End running of Thraxion, either through the use of provided "Quit" command or platform-dependent TERM or KILL command equivalent.

DETAILED USE CASES

UCI1 - INSTALL Qt

<table>
<thead>
<tr>
<th>Use case: Install Qt</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID: UCI1</td>
</tr>
</tbody>
</table>

Actors:
Installer

Preconditions:
1. Installer is installing Qt on a system with a windowing environment.
2. Installer is installing Qt on a system capable of supporting Qt.

Flow of events:
1. The use case starts when the target system for installation has started and is ready to accept input.
2. If the installer downloads the source for Qt to compile
   2.1 The installer selects a destination for the source file to download to.
   2.2 The installer runs the ./configure script.
   2.3 The installer runs the make script.
3. If the installer downloads a binary installation package of Qt
   3.1 The installer selects a destination for the installer to download to.
   3.2 The installer executes the binary installation package.

Postconditions:
1. Qt is installed.
**USE CASES**

### DETAILED USE CASES

#### UCI3 - BUILD SOURCE

**Use case: Build Source**

**ID:** UCI3

**Actors:**
- Installer

**Preconditions:**
1. gcc 3.3 is installed on target machine.
2. UCI1 is complete.
3. UCI2 is complete.

**Flow of events:**
1. The use case starts after the Installer completes all the listed Preconditions.
2. If qmake is properly functioning
   2.1 Installer uses ./build script which builds source.
3. If qmake is not properly functioning
   3.1 Installer edits Makefile to link to Qt directories.
   3.2 Installer runs make.

**Postconditions:**
1. Thraxion is installed on target machine.

#### UCU6 - CREATE FORCES

**Use case: Create Forces**

**ID:** UCU6

**Actors:**
- User

**Preconditions:**
1. At least one sphere, solid object, or limb has been created.

**Flow of events:**
1. The use case starts when either a sphere, solid object, or limb has been created by the User.
2. The User selects "Edit Forces" on a new object that has no current forces.
3. While the User chooses to add forces to an object
   3.1 The user selects a direction for the new force to act on the object from.
   3.2 The user selects a magnitude for the new force.

**Postconditions:**
1. At least one new force is added to the object.

**Alternate flow 1:**
1. The user does not add a force.
**USE CASES**

**SCENARIOS**

**UCI4 - BUILD PRIMARY**

<table>
<thead>
<tr>
<th>Use case: Build</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ID:</strong> UCI4</td>
</tr>
<tr>
<td><strong>Actors:</strong> Installer</td>
</tr>
<tr>
<td><strong>Preconditions:</strong></td>
</tr>
<tr>
<td>1. gcc 3.3 is installed on target machine.</td>
</tr>
<tr>
<td>2. UCI1 is complete.</td>
</tr>
<tr>
<td>3. UCI2 is complete.</td>
</tr>
<tr>
<td><strong>Primary scenario:</strong></td>
</tr>
<tr>
<td>1. Use case begins after listed preconditions have been met.</td>
</tr>
<tr>
<td>2. Installer runs ./build script.</td>
</tr>
<tr>
<td><strong>Secondary scenarios:</strong></td>
</tr>
<tr>
<td>insmodErr</td>
</tr>
<tr>
<td>qmakeErr</td>
</tr>
<tr>
<td><strong>Postconditions:</strong></td>
</tr>
<tr>
<td>1. Thraxion is successfully installed on the system.</td>
</tr>
</tbody>
</table>

**SCENARIOS**

**UCI4b - BUILD insmodERR**

<table>
<thead>
<tr>
<th>Use case: Build</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ID:</strong> UCI4b</td>
</tr>
<tr>
<td><strong>Actors:</strong> Installer</td>
</tr>
<tr>
<td><strong>Preconditions:</strong></td>
</tr>
<tr>
<td>1. The use case begins in step 2 of the use case Build when the compiler does not recognize the C library function insmod(). This is common on Mac OS X versions of gcc 3.3.</td>
</tr>
<tr>
<td>2. The Installer finds the cmath.h header file on the target system (usually in or near /usr/include/gcc/darwin/3.3/c++/cmath).</td>
</tr>
<tr>
<td>3. Installer edits the cmath file, commenting out the line near the end that says &quot;#undef insmod&quot;.</td>
</tr>
<tr>
<td>4. Installer, from the directory in which the Thraxion source files resides, runs &quot;make clean&quot;.</td>
</tr>
<tr>
<td><strong>Postconditions:</strong></td>
</tr>
<tr>
<td>1. Use case returns to the beginning of step 2 of the use case Build.</td>
</tr>
</tbody>
</table>
### Use case: Build Primary

**Secondary scenario:** qmakeERR

**ID:** UCI4c

**Actors:**
- Installer

**Preconditions:**
1. The use case begins in step 2 of the use case Build when the build script is unable to find qmake.
2. Installer finds the location of the qmake application.
3. Installer creates a symbolic link in the /bin folder (or wherever the target machine’s PATH variable is pointing to) to the qmake application.
4. Installer runs "make clean" in the folder containing the Thraxion source files.

**Postconditions:**
1. Use case reverts to step 2 of use case Build.
Key:

Yellow box: To meet requirement, use case is or may be required.
Red box: For use case to exist, requirement must be met.
Above is a screen shot of the main Demo tab with an arbitrary scene loaded.

Above is a screen shot of the information tab.

More screen shots may be found at the project web site at:

http://www.cse.unr.edu/~colborne
Qt: A graphical user interface library that is largely platform independent. It allows programs to be built with its many widgets and then run on platforms such as OS X, various types of Linux, and Microsoft Windows.

Widget: A control object or display object that is standardized throughout a program. A scrollbar or drop down menu are examples of widgets. An OpenGL widget is what allows an OpenGL scene to be displayed in Thraxion.

OpenGL: A graphics standard that allows the user to use a series of calls to construct things like polygons on the screen. OpenGL is what is used to display the scenes through Thraxion.

Translation: The act of moving an object in OpenGL using predefined matrix manipulation functions.

Rotation: The act of rotating an object around the x, y, or z axis in OpenGL using predefined matrix manipulation functions.

Scaling: The act of changing an object's size in OpenGL using predefined matrix manipulation functions.

Transparency: Applying an alpha value to an object in OpenGL to produce a glass-like effect on objects.

Picking: OpenGL term for selecting an object that has been rendered to the screen with a mouse.

Texture: An image that is mapped over a polygon. For instance, a wood material like image could be mapped over a big flat polygon to make the polygon appear to look like a large piece of wood.

Material: Set of Red, Green, Blue, and Alpha values applied to lighting types such as ambient, specular, diffuse, shininess and glow to create color on 3D models.

Scene: A layout of objects created from particles and polygons with various vector properties that can be animated and manipulated by the user.

Isometric: Three dimensional perspective view used in both for playback and design of scenes.

Verlet Integration: Method for numerically integrating equations of motion common in game physics engines. In Thraxion, it is responsible for moving the particles and applying various forces.

Particle: The basic building block of the particle system. Has a radius, mass, and can be moved around in a scene in any way.

Constraint: Constraints keep particles together. A constraint is a hard set distance that two particles must stay from each other. Constraints let objects appear to be solid by holding multiple particles together and having polygons built amongst the various particles.

Polygon: The main object that makes up the graphical display. In Thraxion, a polygon is always a three-sided object, and the ends of the polygon are made up of up particles with no radius.
**Vector:** In Thraxion, a vector is simply a piece of data having an X, Y, and Z component in three dimensional space. They are the same thing as a point to the computer, but named differently so that they can be understood better.

**Collision Detection:** Term used to describe algorithms that decide how objects will react with each other after a collision has been detected.

**Bounding Sphere:** Term used to describe a large invisible sphere placed around an object used to make collision detection simpler and faster.

**Collision:** In Thraxion, it's when one Bounding Sphere travels into the space of another Bounding Sphere.
Our domain specific reference, it gives a quick overview of OpenGL, how it works, and how to do basic actions with it. This is highly useful for Thraxion, which makes heavy use of OpenGL technology to render the images in the scenes. OpenGL is also a cross platform technology, much like Qt.

Another domain specific book, *Programming with Qt* goes through the steps required to create a Qt program, including the introduction of concepts such as widgets, signals, slots, and more. Extremely useful.

One possible collision detection algorithm that may be attempted in Thraxion, it uses a bounded volume tree, instead of the current compare all to all technique employed in Thraxion, to determine whether objects have collided or not. This particular algorithm is described with a heavy emphasis on speed.

Another possible collision detection algorithm, this one also uses bounded volume trees.

However, the goal is to minimize the amount of memory required without affecting speed through the use of convex hulls and broader-grained collision detection mechanisms.

In order for Qt applications to work as native Aqua applications, it is not enough for a Mac OS X developer to simply compile and run the program as a UNIX program. In fact, doing this will disable the menu bar and prevent the program from operating effectively. Instead, it is required to save the executable and its various files in an application bundle. The above web site details how to do this.
Justin Gerthoffer: Primary focus is currently on optimizing the existing Thraxion code to make it run more efficiently and speedily. Partially responsible for creating the original Thraxion program.

Jon Studebaker: Primary focus is currently on creating the graphical user interface for Thraxion. Partially responsible for creating the original Thraxion program.

David Colborne: Primary focus is currently on learning about the Thraxion system, handling the paperwork for the project (including the layout of this report), and porting Thraxion to Mac OS X as a native application.