Hospital Workflow Simulation Unity Client Project

by

Alexander Crichton

N6878296

15/06/2014

Unit Code: INN693

Unit Coordinator: Dr Ernest Foo

Project Supervisor: Dr Ross Brown

# Table of Contents

[Table of Contents i](#_Toc390478207)

[1 Introduction 1](#_Toc390478208)

[2 Project Background 2](#_Toc390478209)

[2.1 Technical Overview 2](#_Toc390478210)

[2.1.1 Client Simulation 2](#_Toc390478211)

[2.1.2 Database 3](#_Toc390478212)

[2.1.3 Socket Server 3](#_Toc390478213)

[2.1.4 Web Server 3](#_Toc390478214)

[2.1.5 Workflow Manager 3](#_Toc390478215)

[2.2 Reimplementation Motivations 3](#_Toc390478216)

[2.3 Chosen Development Solution 4](#_Toc390478217)

[3 Implementation Design 5](#_Toc390478218)

[3.1 Scope 5](#_Toc390478219)

[3.2 Graphic Client 5](#_Toc390478220)

[3.2.1 Environment 5](#_Toc390478221)

[3.2.2 Graphic User Interface 6](#_Toc390478222)

[3.3 Simulation Model 8](#_Toc390478223)

[4 Implementation Execution 9](#_Toc390478224)

[4.1 Relevant Interface Overview 9](#_Toc390478225)

[4.1.1 Asset Interaction Web Interface 9](#_Toc390478226)

[4.1.2 Veis Java Socket Server Console Interface 9](#_Toc390478227)

[4.1.3 YAWL Web Interface 9](#_Toc390478228)

[4.1.4 World State Table View 11](#_Toc390478229)

[4.1.5 Asset Service Routine Table View 12](#_Toc390478230)

[4.2 Unity Simulation Client Demonstration 12](#_Toc390478231)

[4.2.1 Client Start 12](#_Toc390478232)

[4.2.2 Connecting to the Veis Java Socket Server 13](#_Toc390478233)

[4.2.3 Registering as a Participant 13](#_Toc390478234)

[4.2.4 Launching the Case 13](#_Toc390478235)

[4.2.5 Completing the First Work Item 15](#_Toc390478236)

[4.2.6 Completing the Remaining Work Items 21](#_Toc390478237)

[4.2.7 Completing the Case 25](#_Toc390478238)

[4.2.8 Launching a Subsequent Case 26](#_Toc390478239)

[4.2.9 Cancelling a Case 27](#_Toc390478240)

[5 KNOWN ISSUES 29](#_Toc390478241)

[5.1 Issues Specific to this Implementation 29](#_Toc390478242)

[5.1.1 Unity DLL Conflicts 29](#_Toc390478243)

[5.1.2 Unity Type Threading Issues 30](#_Toc390478244)

[5.2 Issues Not Specific to this Implementation 31](#_Toc390478245)

[5.2.1 World State Reset Not Automated 31](#_Toc390478246)

[5.2.2 Veis Database Missing Some Asset Methods 31](#_Toc390478247)

[5.2.3 Simulation Client Connection with Veis Java Socket Server Unreliable 33](#_Toc390478248)

[5.2.4 Veis Java Socket Server Unnecessarily Computationally Intensive 34](#_Toc390478249)

[6 Possible Future Development 37](#_Toc390478250)

[6.1 Web Interface Integration 37](#_Toc390478251)

[6.2 Multiple Simultaneous Users 37](#_Toc390478252)

[6.3 Multiple Simultaneous Cases 37](#_Toc390478253)

[References 38](#_Toc390478254)

[Appendix A Project setup instructions 39](#_Toc390478255)

[Appendix B Simulation-YAWL Communication Process 40](#_Toc390478256)

[Appendix C Comparison of Development Solutions 41](#_Toc390478257)

# Introduction

The project referred to in this report is a continuation of previous efforts. The long-term goal of the project is to produce an interactive and collaborative virtual environment capable of simulating hospital scenarios for training and education purposes. The simulation utilises a Business Process Management Workflow tool to handle scenarios and to distribute tasks between participants. The client for user interaction was originally implemented in OpenSim.

This report details the re-implementation of the client in Unity. Section 2 outlines the project’s existing architecture, motivations for the client migration, and justification for continuing development in Unity. Section 3 explains the new Unity client implementation. Section 4 demonstrates the completed functionality of the new Unity client. Section 5 mentions any significant known issues. Section 6 suggests possible features that are viable in Unity.

# Project Background

Instructions for setting up the project can be found in

Appendix A.

## Technical Overview

Figure 1 illustrates the general project architecture, though it is logically and conceptually simpler to imagine a small hierarchy which is controlled by the Client Simulation (see Figure 2).

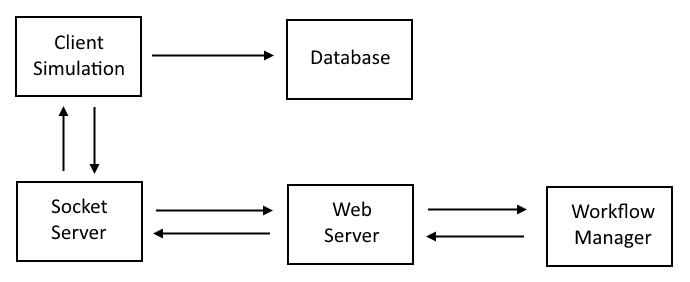


Figure . Top level project architecture.

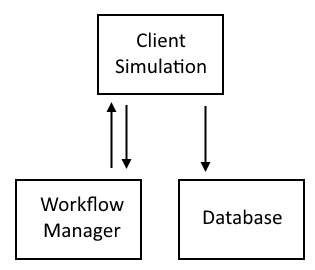


Figure . Project logical hierarchy controlled by the Client Simulation.

### Client Simulation

The Client Simulation (previously implemented in OpenSim) is technically one application, but should be logically regarded as having three parts: a Graphic Client, a Web Interface, and a Simulation Model (see Figure 3). The Graphic Client is the virtual environment that the user perceives. The Web Interface is how the user affects the state of assets in the environment; it is what updates the database. The Simulation Model handles all of the workflow tracking on the client-side, as well as all of the communication with the Workflow Manager; it reads the database for state changes.

Note that the separation of the Graphic Client and the Simulation Model is only logical; the Graphic Client is a standalone application that contains and references the Simulation Model via DLLs.

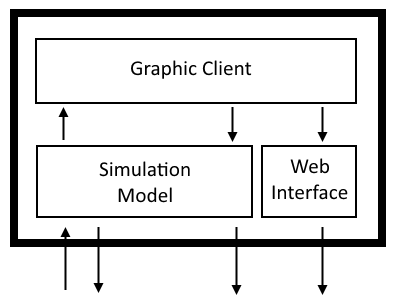


Figure . Logical architecture for the Client Simulation.

### Database

The database consists of three MySQL databases: veis\_knowledge\_base, veis\_logging, and veis\_world\_states. The Knowledge Base holds all of the constant preset values for the Simulation assets. The Logging database just records events that occur. The World State database holds the dynamic content such as asset states and asset actions to be performed. The Web Interface updates these values and the Simulation Model reads the changes.

### Socket Server

The Veis Java Socket Server is a small application that handles communication between the Simulation, which is a .NET application, and the Workflow Manager, which is a Java application.

### Web Server

This is a simple Windows, Apache, MySQL, and PHP (WAMP) server. The server is required for the Workflow Manager, but not for the database which is directly accessed by the Simulation.

### Workflow Manager

The Workflow Manager is an application called YAWL. It is a tool for defining and executing complex workflows. Its role in the project is to delegate tasks to participants for them to complete. It posts workflow data which is handled by the Veis Java Socket Server and passed to the Simulation. YAWL does not interact with the database; it is not aware of the state of workflow items and must be notified by the Simulation when they have been completed. A more sophisticated depiction of the communication process between the Simulation and YAWL can be found in Appendix B Simulation-YAWL Communication Process.

## Reimplementation Motivations

The previous client implemented in OpenSim was functional but littered with problems. It was comprised of an OpenSim server and a graphic client for user interaction. The OpenSim server is a large, pre-release, open source application with many features, a large portion of which is not relevant to this project. The graphic client was one of a number of third-party open source applications for connecting to virtual worlds.

The previous client was unreliable. In the long-term interest of the project a decision was made to investigate an alternative platform with stability, support, and efficient, dedicated tools.

## Chosen Development Solution

Many possible solutions for developing graphic clients exist, but they are not all equally suited to this project. The optimal development solution needs to suit both the technical needs and the relatively small scale of the project. The previous implementation in OpenSim suggests a solution with a toolset more closely aligned with the needs of the project.

The range of video game development tools offers more relevant and effective solutions. There are many to consider: Unreal Development Kit, CryENGINE, Game Maker Studio, and Source Engine are some of the more notable ones. The most appropriate has been deemed to be Unity. For a relevant feature comparison of various video game development solutions see Appendix C Comparison of Development Solutions.

# Implementation Design

As briefly mentioned in section 2.1.1 there are three important parts to the client: the Graphic Client, the Web Interface, and the Simulation Model. This section will describe what was implemented in the Graphic Client, and the relevant parts of the Simulation Model that were extended for this implementation. This section does not describe the entire Simulation Model that is found in the Veis libraries.

## Scope

The feature scope for this implementation was kept small as it was hard to predict what difficulties would arise during the migration from OpenSim to Unity. The goal was to recreate the basic functionality that was present in the previous implementation.

It was to be hosted locally; it was not necessary to be able to operate over a network. This meant it was not necessary to allow multiple simultaneous users. There was no need to cater for more than one case or more than one player avatar. The user needed to be able to complete the case by completing its work items.

All assets pertaining to the existing case were to be implemented. The method of interacting with assets was via the existing web interface. Assets that could move needed to retain that functionality.

The virtual environment needed to update with the Simulation. The environment could be rudimentary shapes, however it would ideally resemble a hospital and be navigable by the avatar. Unity Implementation

## Graphic Client

The Graphic Client is a Unity application which provides the virtual environment for users to interact with.

### Environment

The environment resembles a hospital with the rooms required by the asset specifications: the Emergency Room, Examination Room, Intensive Care, Radiology Room, and a final unnamed room (see Figure 4). The avatar starts in the bottom-left corner in the Emergency Room; this is where the PatientBill asset for the first work item is located.

Assets are distinguishable by their bright red, green, or blue colour. These are environment objects that the user can interact with. Place Markers are the less obvious transparent green objects. These are possible locations for certain assets, not for user interaction.

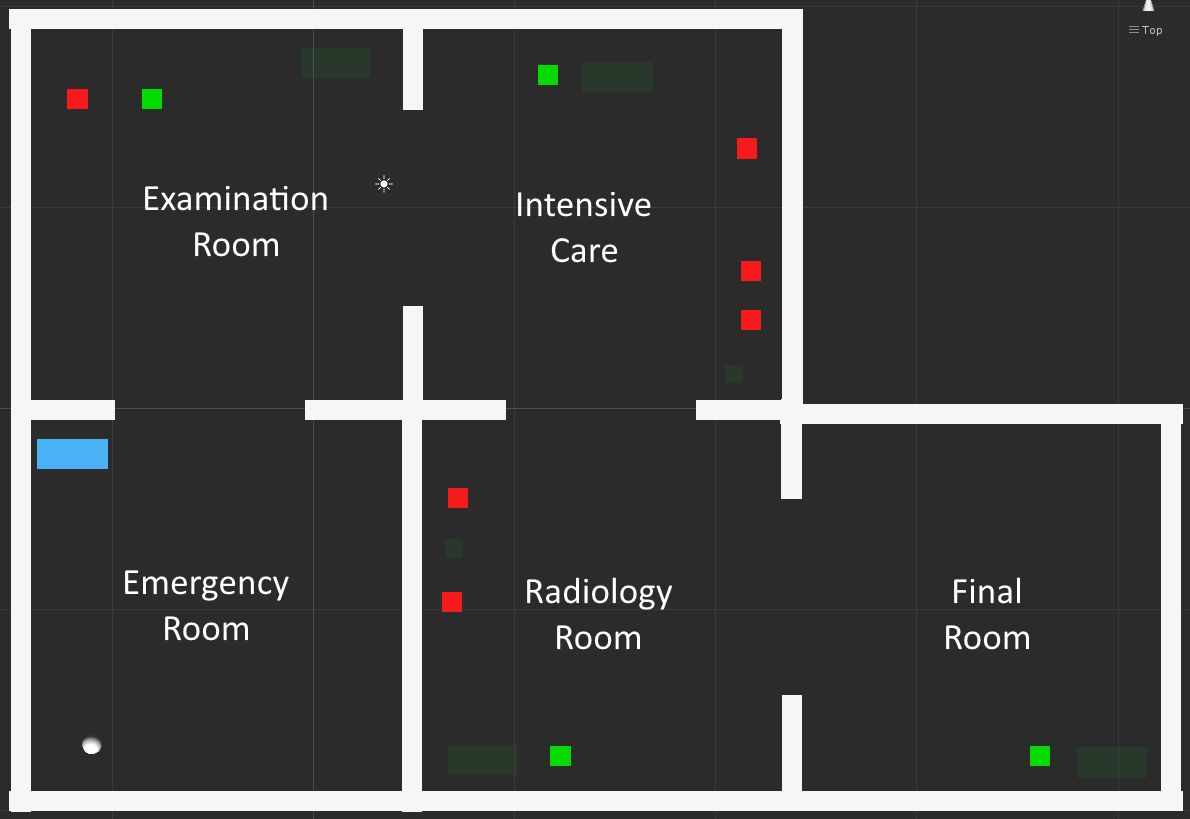


Figure . The Unity application environment Layout.

### Graphic User Interface

Figure 5 shows the Graphic User Interface (GUI) as seen by the user. The user sees the hospital environment and their avatar in the third person. The available simulation and case controls are located in the bottom-left corner of the interface (see Figure 6). Information about any current case and work items is located in the top-left corner of the interface. Particular assets can be easily identified by their asset name which is rendered as floating text layered on top of each visible asset (see Figure 8).

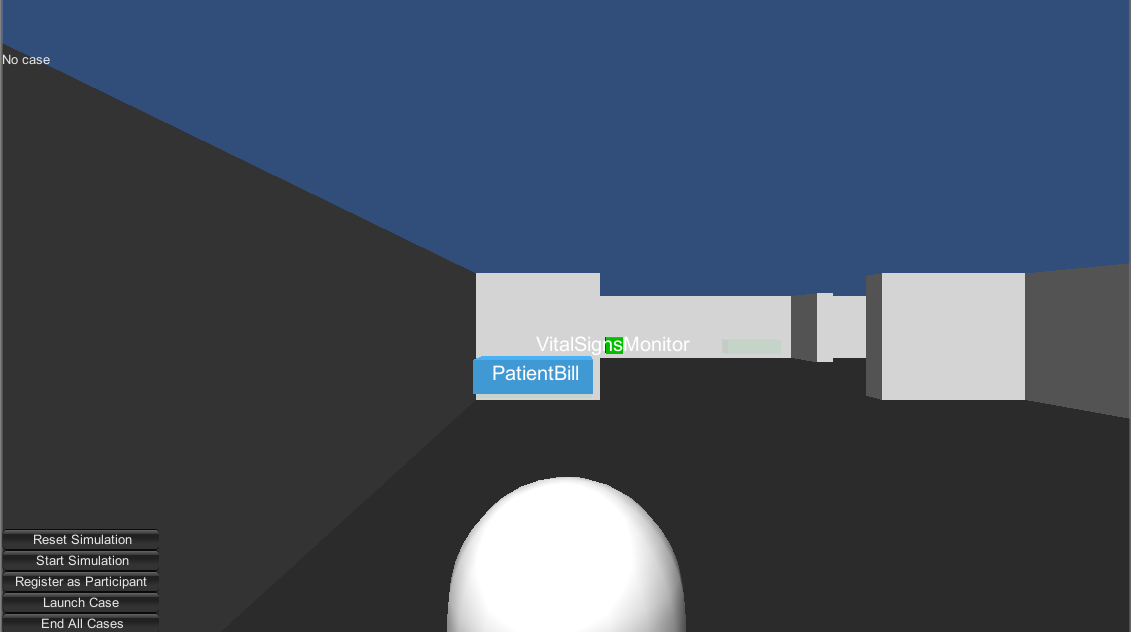


Figure . The Unity application Graphic User Interface.

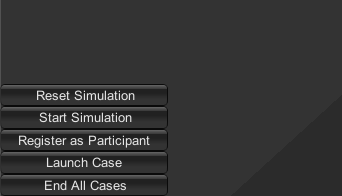


Figure . The Simulation controls available to the user.

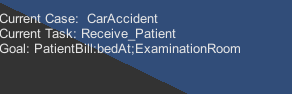


Figure . Case and work item information.

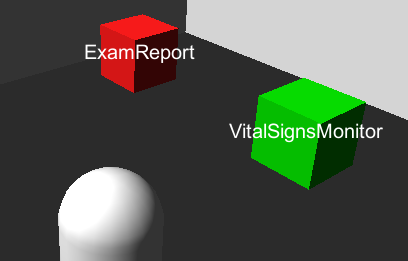


Figure . Floating labels for asset identification.

### Simulation Control

The Simulation controls previously shown in Figure 6 are used to get the Simulation going.

The Reset Simulation button just sets Simulation variables to their initial values. This affects local data tracking like registered participants and current work items; it does not affect externally stored data like asset states, which are in the database, and case and work item states, which are stored in YAWL. The Start Simulation button is currently redundant because when the Simulation resets it performs that function as well.

The Register as Participant button assigns a case participant to the user’s avatar. In this implementation there is only one case participant so the use is not able to make a selection.

The Launch Case button notifies YAWL to start the case.

The End All Cases button notifies YAWL to end any running case. This only ends locally tracked cases; it does not end cases that might have been started on YAWL outside of the Simulation.

### Asset Interaction

Assets in the environment can be interacted with by clicking on them. This will launch the Asset Interaction Web Interface (see Figure 9) which uses the asset name to load its state and available methods. The Asset Interaction Web Interface is the only way the user can affect assets. It directly updates the World State database which is then read by the Simulation Model. It updates asset values and inserts asset service routines if an action needs to be performed on an asset, such as moving it.

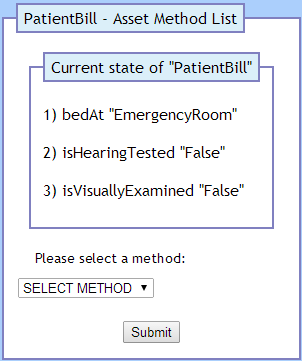


Figure . The first page of the Asset Interaction Web Interface.

## Simulation Model

The Simulation Model tracks running cases and participant work items. It queries the database and communicates with YAWL.

Register Participant - This process is handled internally within the Simulation. YAWL is not notified of any participants. The Simulation tracks the participants and passes work items to them. If the user does not register as a participant they will not be sent or be able to complete any case or work item.

# Implementation Execution

This section demonstrates the completeness of the Unity client implementation. Screenshots will be used extensively to convey the state of the client and other relevant interfaces.

## Relevant Interface Overview

This section explains any other relevant interfaces that will be shown.

### Veis Java Socket Server Console Interface

Figure 10 shows the console interface that is part of the Veis Java Socket Server. As this application passes messages between the Simulation and YAWL it also relays some information through this console interface for human viewing. This interface will be used to confirm some state changes as they are communicated between the Simulation and YAWL.

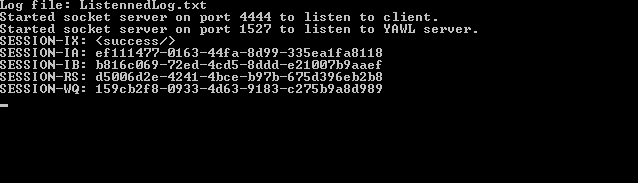


Figure . Veis Java Socket Server console window immediately after opening

### YAWL Web Interface

Figure 11 shows the Cases tab in the YAWL Web Interface. Here case specifications can be loaded, cases can be launched, and running cases can be viewed or cancelled. This interface will be used to confirm that cases are running or not.

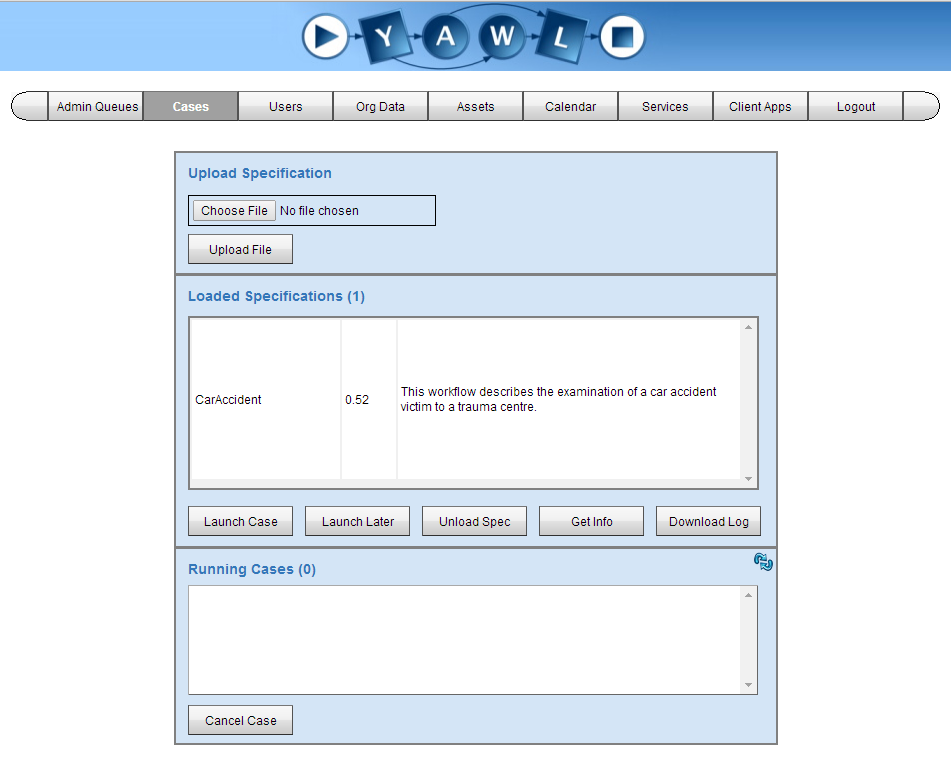


Figure . YAWL web interface view of case management section with no case running

Figure 12 shows the Admin Queues tab, and the Worklisted sub-tab, in the YAWL Web Interface. Here work items that are being handled by YAWL can be monitored. This interface will be used to confirm the state of various work items.

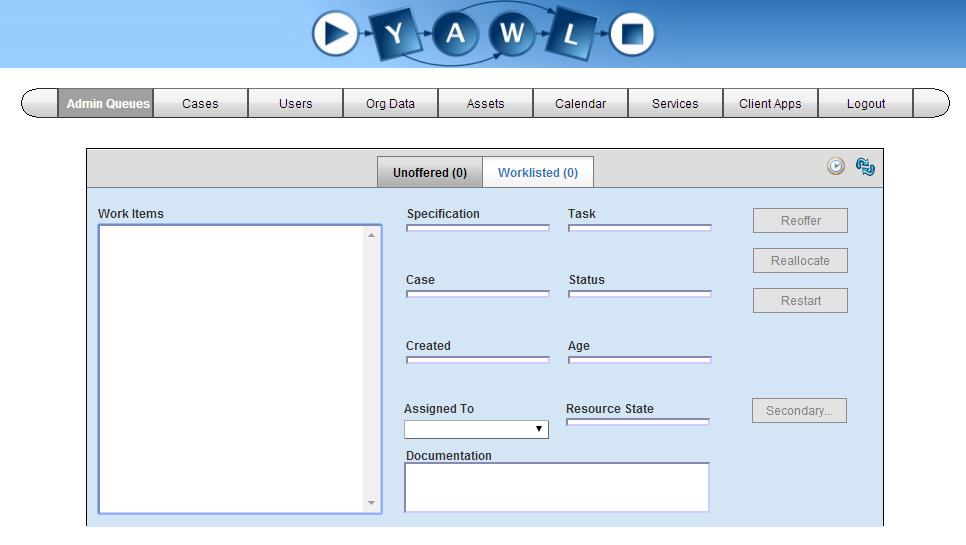


Figure . YAWL web interface view of current work items with no case running

### World State Table View

Figure 13 Shows a view of the ‘world\_states’ table, part of the Veis MySQL database, accessed through the PHPMyAdmin Web Interface. Simulation assets are defined in the Veis MySQL database. The ‘world\_states’ table stores various current states for those assets. This view will be shown to confirm the state of assets. Note that some of the assets listed in this table are not relevant to this simulation.

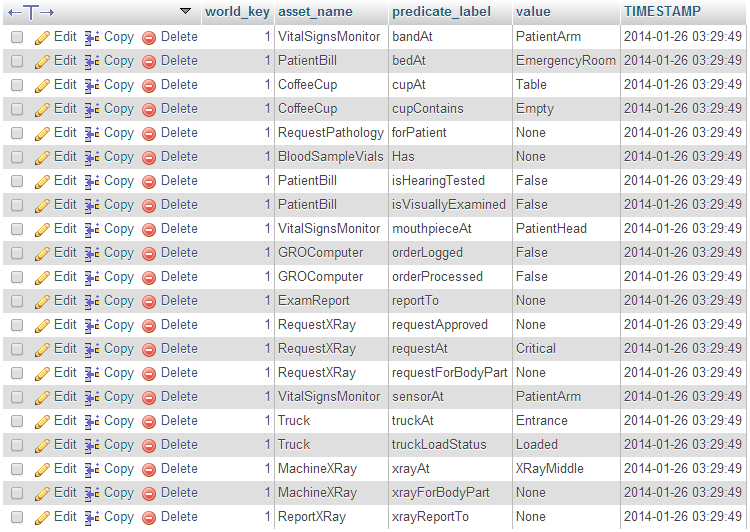


Figure . MySQL database table view of asset world states with its initial values

### Asset Service Routine Table View

Figure 14 shows a view of the *asset\_service\_routines* table. This table holds actions to be performed on assets such as moving to a new position. The actions are handled directly by the Simulation and then removed; as such the table is usually empty.



Figure . MySQL database table view of asset service routines.

## Unity Simulation Client Demonstration

This section demonstrates the functional aspects of the Unity Simulation Client. It makes extensive use of figures to confirm the validity of said functionalities. This demonstration assumes the following preconditions:

* An appropriate webserver is being hosted locally with MySQL and PHP
* The necessary Veis MySQL database has been set up
* The YAWL Web Service is running
* The necessary hospital case specification has been loaded
* The Veis Java Socket Server is running

### Client Start

As shown in Figure 5 when the Unity Simulation Client starts the user is positioned in the hospital environment. The section they start in is the Emergency Room. The only Simulation asset in this room is PatientBill. Initially the user is not registered as a participant, there is no case running, and there are no work items.

### Connecting to the Veis Java Socket Server

Figure 15 confirms that the Unity Simulation Client successfully establishes communications with the Veis Java Socket Server. The client is now able to send and receive messages to and from YAWL.

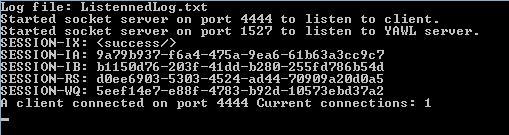


Figure . A Simulation client has successfully connected to the Veis Java Socket Server.

### Registering as a Participant

Figure 16 shows the user pressing the Register as Participant button in the Unity Simulation Client interface

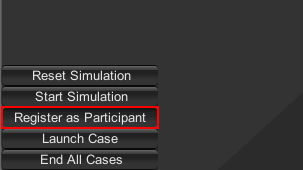


Figure . The user assigns a registered case participant to their avatar.

### Launching the Case

Figure 17 shows the user pressing the Launch Case button. Figure 18 shows the client interface updates to show the case, the first work item, and the work item’s condition for completion. The work item will not be completed until the World State table has been updated so that it matches those values.

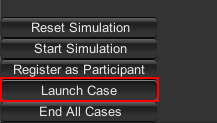


Figure . The user launches the case.

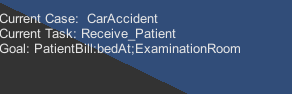


Figure . The interface updates to show the newly launched case, the user's first work item, and the work item’s condition for completion.

Figure 19 confirms the corresponding case has been launched and is running in the YAWL Web Interface. Figure 20 shows the corresponding work item has been delegated. Figure 21 shows the messages received from YAWL and sent to the Simulation by the Veis Java Socket Server. The first section is the case being sent from YAWL; the second section is the first work item being sent from YAWL; the third section is the Socket Server sending the work item to the Simulation.

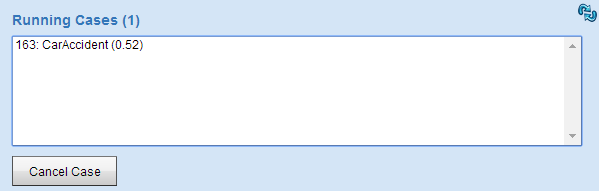


Figure . The YAWL Web Interface shows the case running.

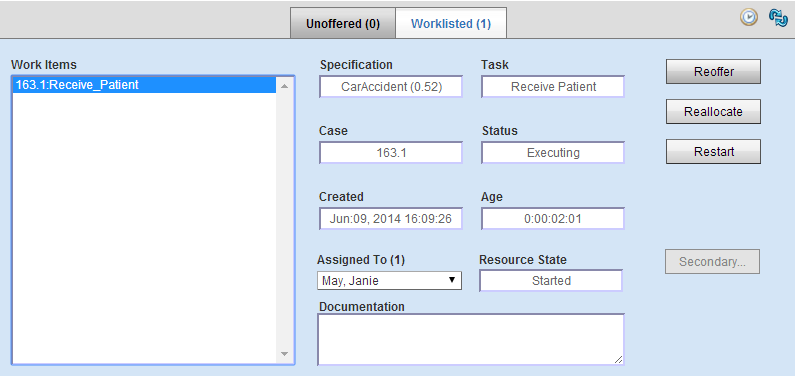


Figure . The YAWL Web Interface shows the work item delegated to the user.

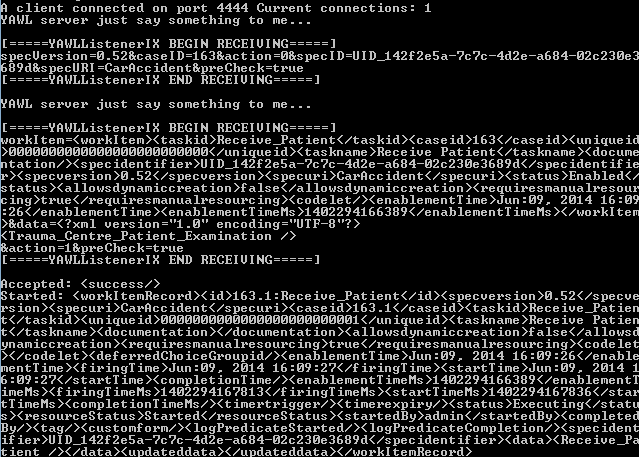


Figure . The Veis Java Socket Server shows the data received from YAWL and sent to the Simulation.

### Completing the First Work Item

Figure 22 outlines what is required to complete the first work item. It reads: for the asset *PatientBill*, its predicate label *bedAt* must have the value *ExaminationRoom*. Figure 23 outlines the Unity GameObject that represents the asset PatientBill in the client. The user can press on the GameObject to launch the asset interaction.

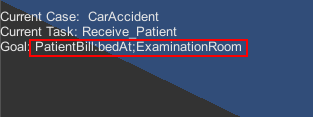


Figure . The client interface shows the asset state required to complete the work item



Figure . Outlining the asset PatientBill that is part of the first work item.

When the user presses on the PatientBill GameObject the Asset Interaction Web Interface is launched (see Figure 24). It selects all the predicate labels associated with the asset PatientBill and their current values. The user selects the asset method that corresponds with the predicate label for which they intend to change the value (see Figure 25). The user presses the Submit button and the next page loads.

The asset method is selected and the corresponding predicate label’s current value is shown (see Figure 26). The user selects from the available values the one they intend to assign to the predicate label (see Figure 27). The user presses the Submit button and the next page loads.

The selected value is assigned to the selected predicate label of the selected asset. The user is notified that the database has been updated successfully (see Figure 28). The user is finally prompted to close the Asset Interaction Web Interface (see Figure 29).

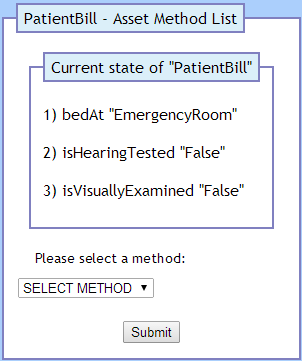


Figure . The Asset Interaction Web Interface when launched by pressing on the asset PatientBill.

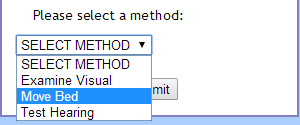


Figure . The available asset methods for the asset PatientBill.

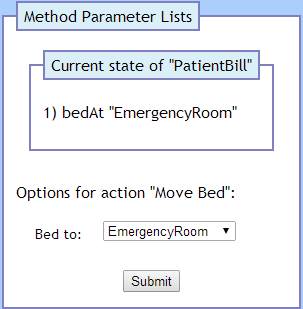


Figure . The current value for the selected predicate label bedAt.

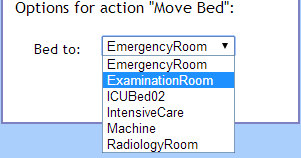


Figure . The available values for the predicate label bedAt.

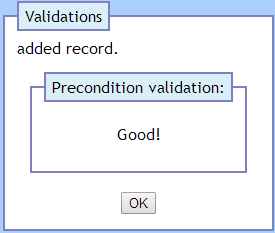


Figure . The Asset Interaction Web Interface has updated the corresponding asset value.

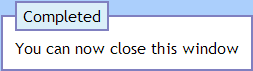


Figure . The Asset Interaction Web Interface has been completed for the asset.

The World State table is updated (see Figure 30). The asset PatientBill has its predicate label bedAt changed to ExaminationRoom. An action is inserted into the Asset Service Routine table (see Figure 31). In this instance the action is to move the asset PatientBill to the location ExaminationRoom.

The YAWL Web Interface shows the previous work item has been removed and new work items have been set (see Figure 32). The Veis Java Socket Server shows the messages from YAWL to the Simulation (see Figure 33). The first two sections indicate the new work items being sent from YAWL; the third section indicates the work item Visual\_Examination being sent to the Simulation; the fourth section indicates the previous work item being completed.

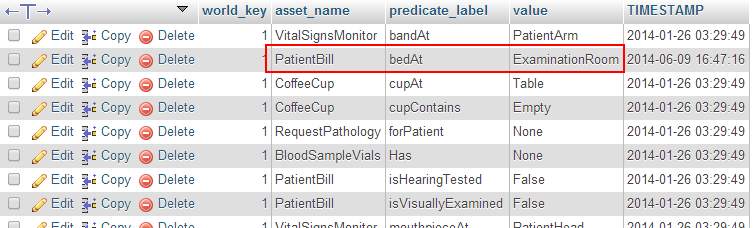


Figure . The World State table has been updated for the asset PatientBill.



Figure . The move action to be performed on the asset PatientBill.

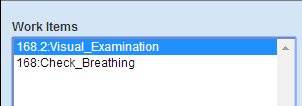


Figure . The work item Receive\_Patient has been completed and new work has been delegated.

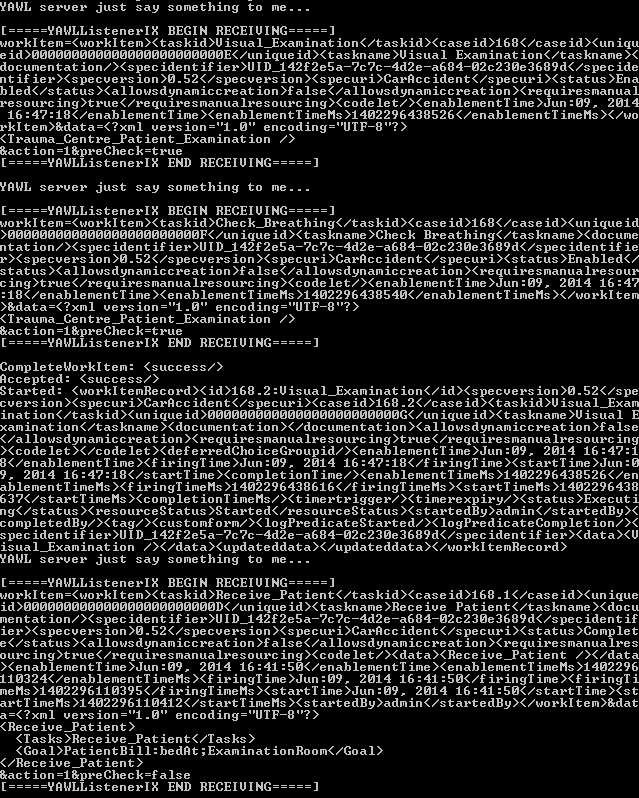


Figure . The Veis Java Socket Server passing new work items from YAWL to the Simulation.

The Simulation client updates to show the new work item and its predicate label value required to complete it (see Figure 34). The GameObject for the PatientBill asset moves in the Unity Scene (see Figure 35). Its new position is in the Examination Room as specified in the Asset Service Routine table. The service routine action is now removed from the table.

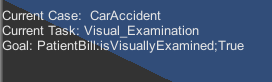


Figure . The Simulation client interface has updated to show the new work item Visual\_Examination and the action required to complete it.

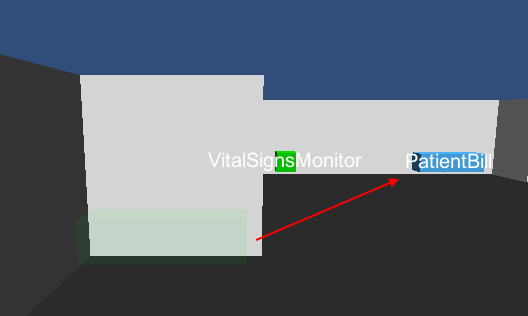


Figure . The GameObject for the asset PatientBill has moved to its new position.

### Completing the Remaining Work Items

To avoid unnecessary repetition of the processes involved in completing work items, work items will be summarised unless significant.

After completing the first work item the user moves into the Examination Room (see Figure 36). The assets in this room are:

* PatientBill
* VitalSignsMonitor
* ExamReport

The work items associated with the assets in this room are:

1. Check\_Breathing
2. Auditory\_Inspection
3. Check\_Pulse
4. Check\_Blood\_Pressure
5. Update\_Examination\_Report
6. Doctor\_to\_Nurse\_Hand\_Over

Completing the last work item moves the asset PatientBill to the Intensive Care room (see Figure 37).

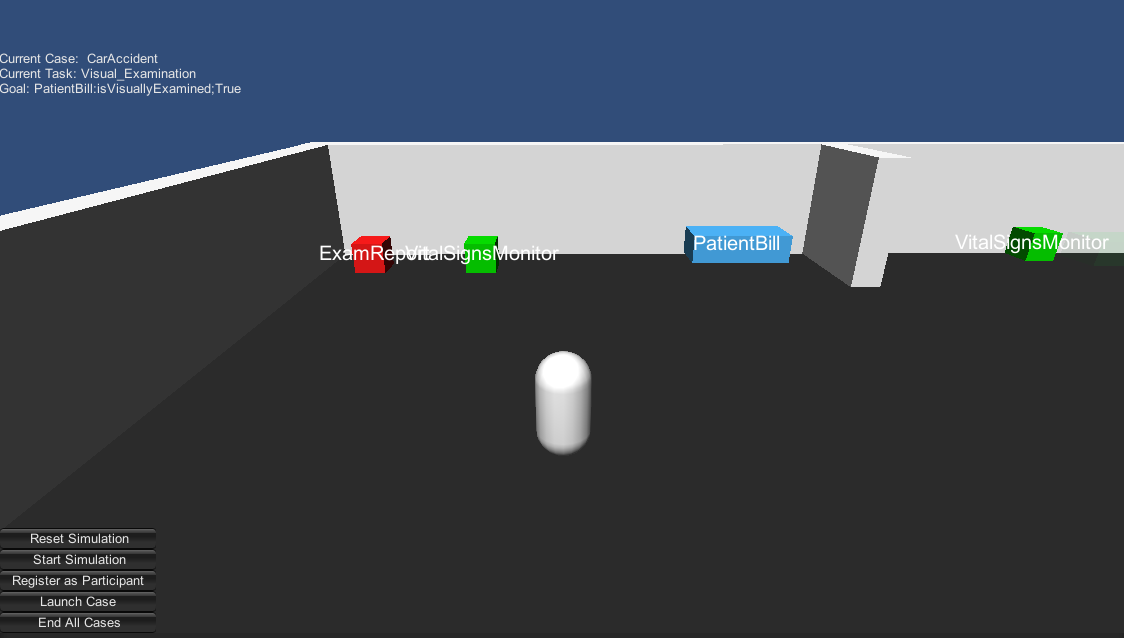


Figure . The user has moved their avatar to the Examination Room.



Figure . The asset PatientBill has moved from the Examination Room to the Intensive Care room.

The user moves into the Intensive Care room (see Figure 38). The assets in this room are:

* PatientBill
* VitalSignsMonitor
* BloodSampleVials
* RequestXRay
* RequestPathology

The work items associated with the assets in this room are:

1. Request\_X-Ray
2. Request\_Pathology
3. Take\_Blood\_Samples
4. Attach\_Vital\_Signs\_Monitor
5. Admit\_Patient\_to\_Intensive\_Care
6. Collect\_X-Ray\_Requests
7. Collect\_Patient\_from\_Intensive\_Care

Completing the last work item moves the asset PatientBill to the Radiology Room (see Figure 39).

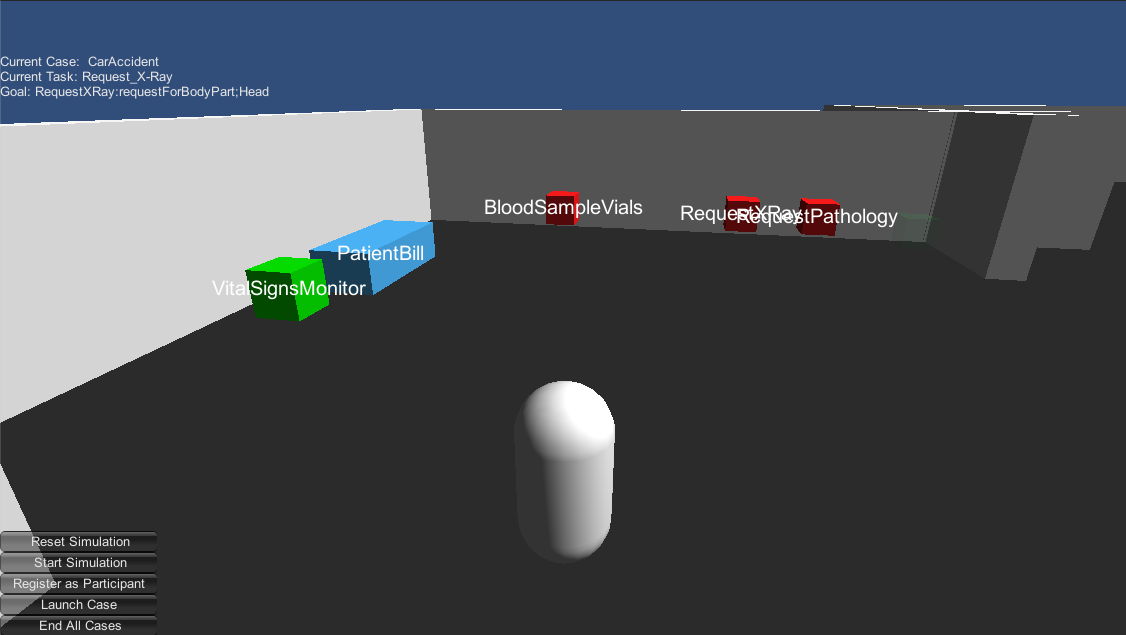


Figure . The user has moved their avatar to the Intensive Care room.

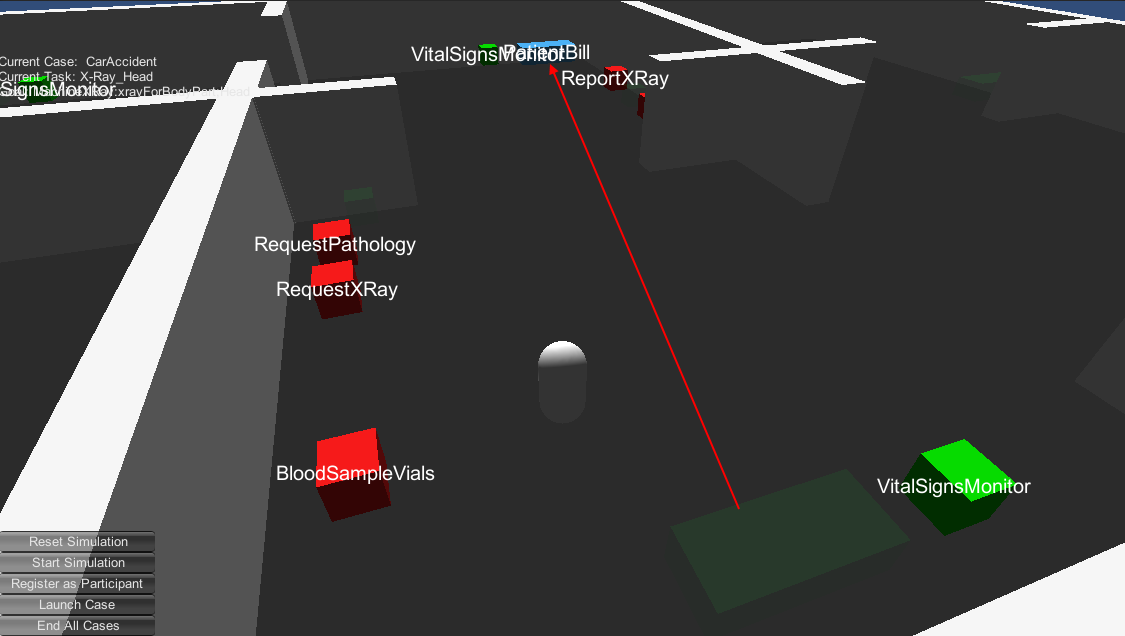


Figure . The asset PatientBill has moved from the Intensive Care room to the Radiology Room

The user moves into the Radiology Room (see Figure 40). The assets in this room are:

* PatientBill
* VitalSignsMonitor
* ReportXRay
* MachineXRay

The work items associated with the assets in this room are:

1. X-Ray\_Head
2. X-Ray\_Torso
3. Report\_X-Ray\_Results\_to\_Doctor
4. Complete\_XRay\_Duties

Completing the last work item moves the asset PatientBill to ICUBed02 (see Figure 41).



Figure . The user has moved their avatar to the Radiology Room.

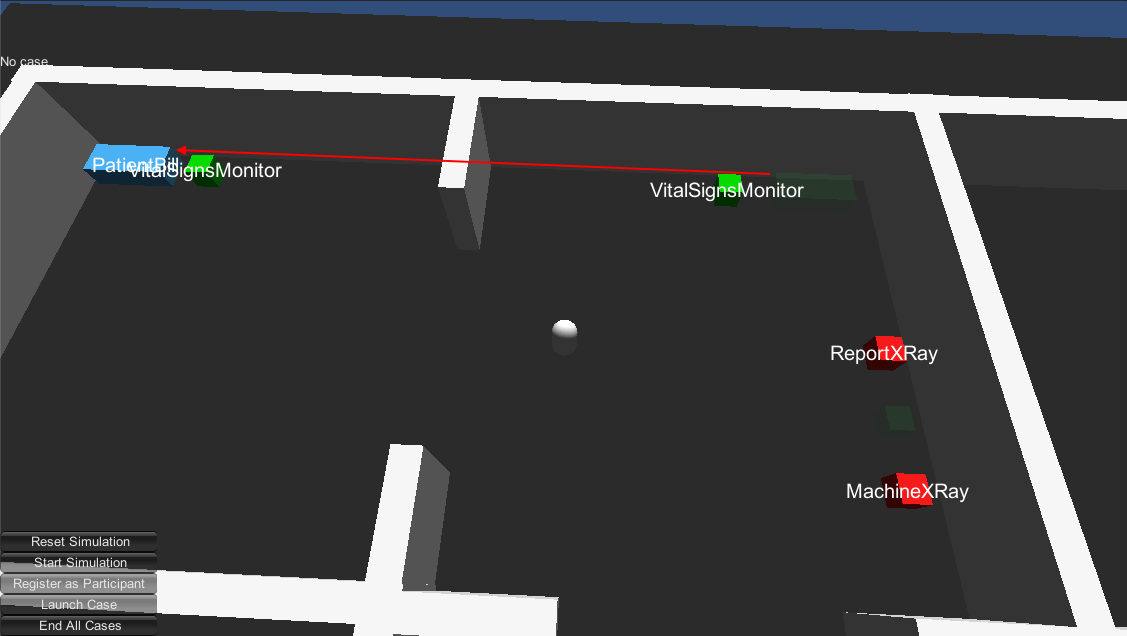


Figure . The asset PatientBill has moved from the Radiology Room to ICUBed02.

### Completing the Case

Moving the asset PatientBill to ICUBed02 is the final work item for the case. The case is now completed. The Simulation client interface updates and no longer shows a case (see Figure 42). The YAWL Web Service confirms that there are no longer any work items (see Figure 43) and there is no case running (see Figure 44). The Veis Java Socket Server indicates that the case has been completed (see Figure 45).



Figure . The Simulation client interface updates to show that there is no longer a case running.



Figure . The YAWL Web Interface shows that there are no remaining work items.

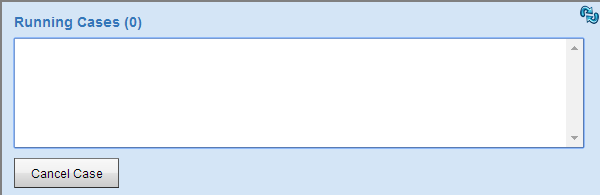


Figure . The YAWL Web Interface shows that there is no case running.

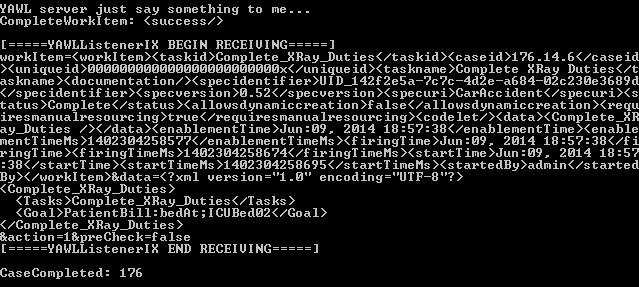


Figure . The Veis Java Socket Server shows the message being sent that the case has been completed.

### Launching a Subsequent Case

A case may be started as long as there is no current case running. After finishing a case the user may choose to launch another one (see Figure 46). The user does not have to register as a participant again because their current registration isn’t affected by case completion. The Simulation will handle the subsequent case without error, but the asset World State values are not reset, and assets that can move do not revert to their initial position (see Figure 47).

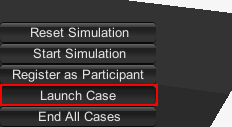


Figure . The user can press the Launch Case button to launch another case.

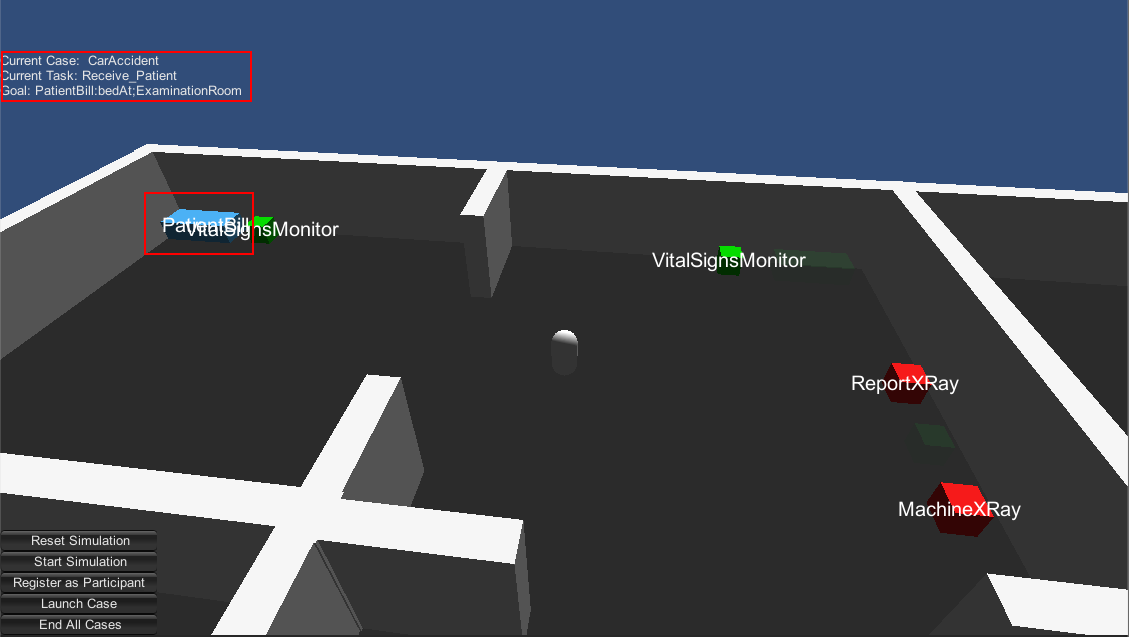


Figure . Launching a subsequent case works but dynamic assets will not move to their start position.

### Cancelling a Case

A running case may be cancelled regardless of what work items have been completed (see Figure 48). The Simulation client will show that there is no case running (see Figure 49). The YAWL Web Interface confirms that the case has been cancelled (see Figure 50).

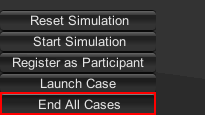


Figure . The user can press the End All Cases button to cancel a running case.



Figure . The Simulation client showing that no case is running after being cancelled.

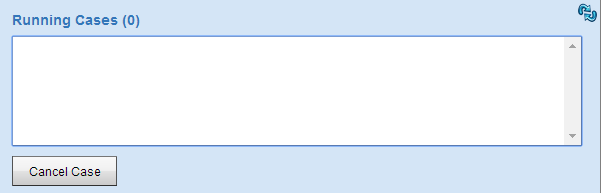


Figure . The YAWL Web Interface shows that there is no case running after being cancelled.

# KNOWN ISSUES

This section lists notable technical issues and – where possible – potential solutions to those issues.

## Issues Specific to this Implementation

These are issues that are present due to the project being implemented in Unity.

### Unity DLL Conflicts

The libraries referenced by the Veis, Veis.Data, and Spark DLLs cause errors with the Unity compiler (see Figure 51 and Figure 52). The Unity compiler output file indicates that the problem arises because these DLLs reference system libraries that are not supported by Unity (see Figure 52). The errors can be subdued by manually including the specific system libraries (see Figure 54). However, the manual inclusion of system libraries can cause conflicts with their Unity version counterparts when trying to use some Types (see Figure 55). The project will compile and run if no conflicting Types are used, but that removes access to some useful Types, for example System.Timers.Timer. One possible solution to these problems which stem from mismatched system references might be to recompile the Veis DLLs using the Unity versions of the necessary system libraries.

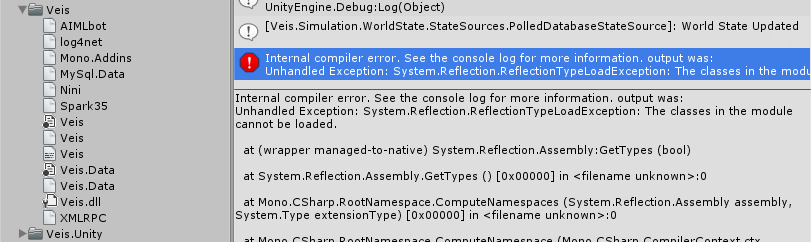


Figure . The Veis DLLs and their dependencies, and the error shown when Unity tries to compile them.

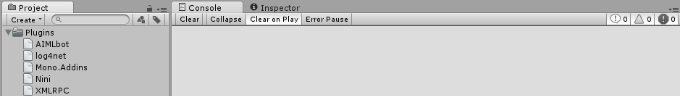


Figure . The Unity project will compile when the Veis, Veis.Data, and Spark DLLs have been removed.

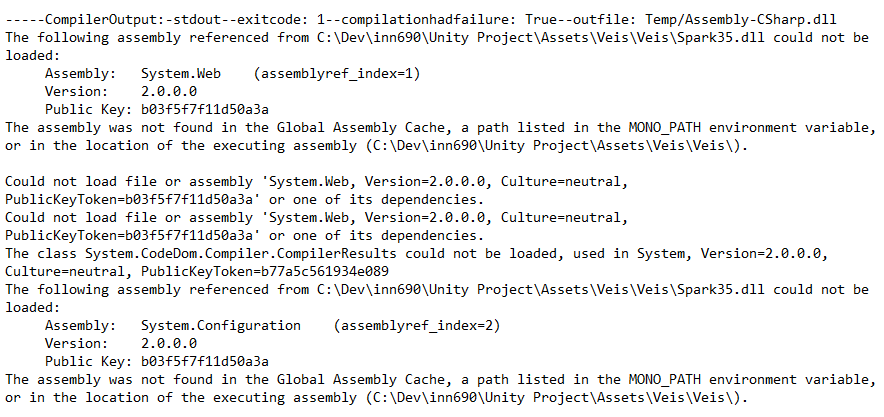


Figure . The Unity Editor log indicates that some of the Veis DLLs reference system libraries not supported by Unity.



Figure . The Unity project will compile without error if the error-inducing system references are manually included.

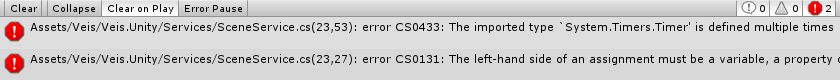


Figure . The Unity Editor will not compile when a Type is used that is contained in a system library that has both a Unity and non-Unity version present.

### Unity Type Threading Issues

Unity reference Types such as Transform are not thread-safe and can only be accessed from the main application thread. This has already caused issues with some of Veis’ architecture due to its use of threads (see Figure 56. Unity Types and methods cannot be accessed from outside of the main thread.); specifically, the Simulation updates the game world via the SceneService on a System.Timers.Timer, which raises events and is handled on a Thread Pool thread (not the main thread) [Microsoft 2014]. When the timer thread attempts to interact with Unity objects it breaks.

The problem has been overcome by using a very simple Producer-Consumer pattern. The timer thread queues instructions in the SceneService, and the main thread periodically checks and executes those instructions. Unfortunately as timers cannot be used due to their threaded nature the only way to get the main thread specifically to perform the actions is to have the call come from an Update method in the Unity scene. While the application does execute properly the tight coupling with the Unity scene is undesirable, and it is unknown how effective the Producer-Consumer pattern for handling Unity Type interaction will be as the project continues to grow.

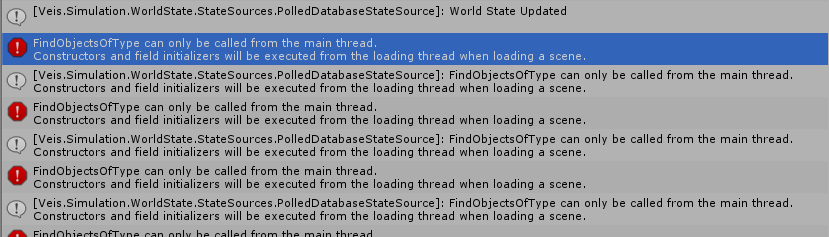


Figure . Unity Types and methods cannot be accessed from outside of the main thread.

## Issues Not Specific to this Implementation

These are issues that were either present prior to this implementation or have not presented themselves due to the project being implemented in Unity.

### World State Reset Not Automated

The methods for starting and resetting the Simulation only affect the Simulation properties such as the registered participant and tracked cases; these methods do not alter the World State database in any way. This means that there is no convenient way to set the World State database to its initial state; it must be done manually by reloading the World State database in PHPMyAdmin. The current situation is inefficient, and resetting the database automatically when the case begins or via the client interface should not be very difficult.

### Veis Database Missing Some Asset Methods

Many assets have methods that can be enacted upon them (see Figure 57). Some of these methods are intended to move assets between locations. When an asset is to be moved it is communicated to the Simulation via a service call (see Figure 58). The possible locations for moving assets are defined in another table (see Figure 59). However the only asset relevant to the simulation that has a service call is PatientBill. The assets MachineXRay and RequestXRay both have move methods but are lacking service calls and location values. It is unclear whether these assets have not had these values entered yet, or whether they were actually not intended to be moved. At the moment the solution has been to place these assets wherever they are needed.



Figure . Methods that can be enacted upon assets.

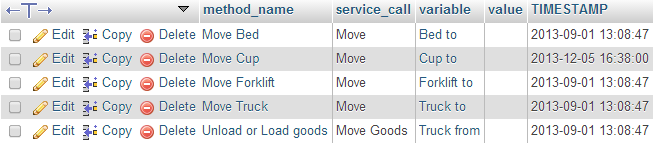


Figure . Services to be called when an asset method is to be enacted.

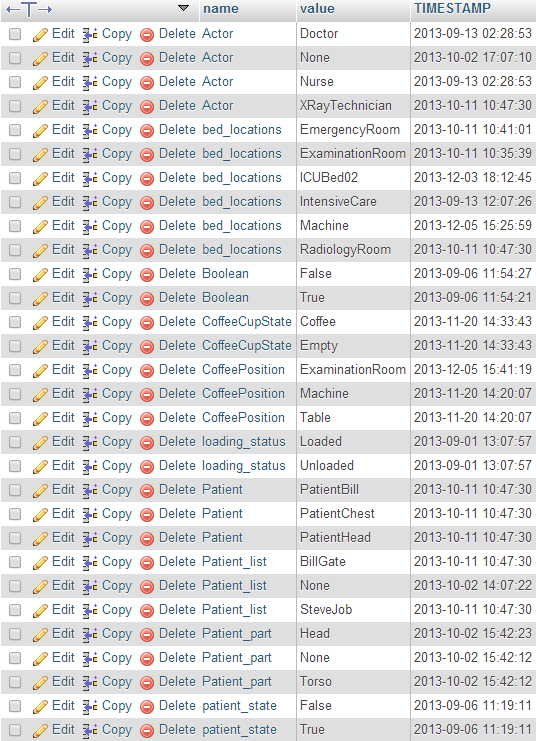


Figure . Possible values that asset predicate labels can be assigned.

### Simulation Client Connection with Veis Java Socket Server Unreliable

During case run-throughs the Veis Java Socket Server would occasionally lose the client connection (see Figure 60, Figure 61, and Figure 62). The problem appears to occur in the MessageHandler class which handles communications with the client; coincidentally, the disconnections seemed to occur when the Simulation would complete numerous goals in quick succession (which can happen when work item goal conditions are still satisfied from being completed in a previous case).

A potential diagnosis could be that there are synchronisation issues in the Socket Server when sending and receiving work items quickly. Perhaps the message handling process is lengthy and gets interrupted by messages from the client. The ConcurrentModificationException might suggest that some values are being accessed simultaneously – perhaps across threads – which also suggests that the Socket Server isn’t handling work item communication in a reliable manner.

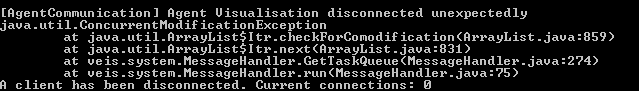


Figure . Veis Java Socket Server losing its client connection unexpectedly.



Figure . Veis Java Socket Server losing its client connection unexpectedly.

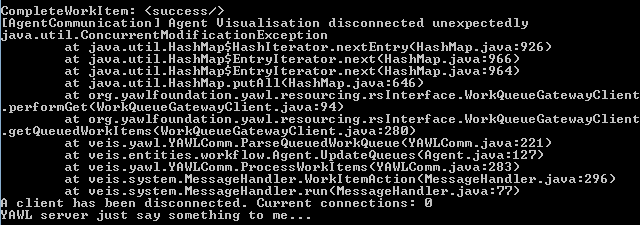


Figure . Veis Java Socket Server losing its client connection unexpectedly.

### Veis Java Socket Server Unnecessarily Computationally Intensive

The computer this implementation was developed on has an idle CPU core temperature of ~55ºC (see Figure 63). This temperature is maintained while running the necessary web server and YAWL application. However, after running the Veis Java Socket Server for a minute or two the core temperatures rise quite dramatically. When a client connects the core temperatures rise to dangerously high levels of ~100 ºC (see Figure 64).

The exact cause of the extreme core temperature increase is unknown. According to the source code the Socket Server application runs a thread for listening to YAWL and a thread for each client connection. There are some while loops that may be the cause (see Figure 65 and Figure 66). Some brief investigation suggests that these could be a form of “Busy Wait” which is a computationally intensive method of doing nothing while waiting for conditions. Although it doesn’t directly hinder development there is a clear unnecessary strain on host machines that might have a relatively simple solution such as a CyclicBarrier.

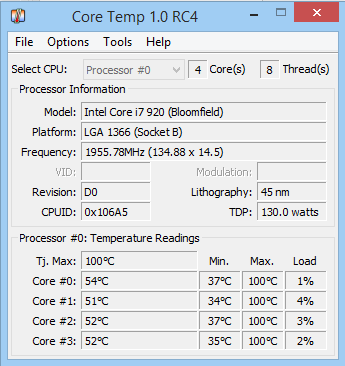


Figure . Host CPU core temperatures while not running Veis Java Socket Server.

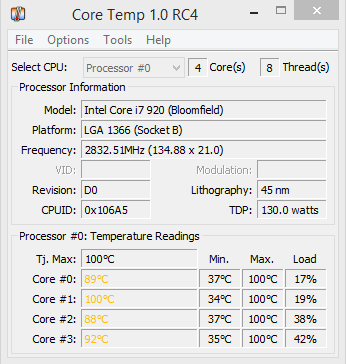


Figure . Host CPU core temperatures while running Veis Java Socket Server.

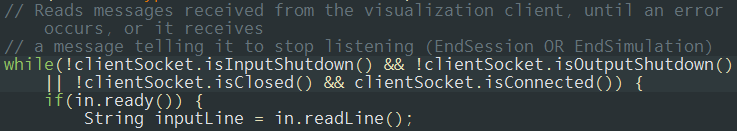


Figure . Potential "Busy Wait" while loop in Veis Java Socket Server class YAWLSocketServerIX.

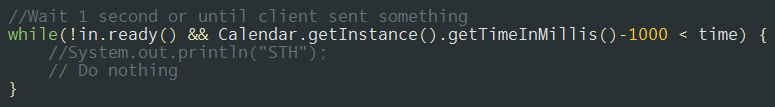


Figure . Potential "Busy Wait" while loop in Veis Java Socket Server class MessageHandlerOn4444.

# Possible Future Development

## Web Interface Integration

## Multiple Simultaneous Users

## Multiple Simultaneous Cases

# References

Microsoft. 2014. Timer Class. Retrieved June 11, 2014 from <http://msdn.microsoft.com/en-us/library/system.timers.timer.aspx>

# Appendix A Project setup instructions

# Appendix B Simulation-YAWL Communication Process

# Appendix C Comparison of Development Solutions