Temperature Control Manufacturing Game

Software Requirements Document

Steve Swab, Matt Chell, Ralph Plett, Ian Trick

CMPT 385

Professor Rick Sutcliffe

Tuesday, November 23, 2010

Contents

1 Introduction

1.1 Purpose

This document contains software requirements and high-level specifications for the Temperature Control Manufacturing Game, the second mini-game in the Dye Merchant Game. The mini-game is a component internal to the entire game and is started from the main component, the shell. The contents of this document describe the nature of the mini-game and its properties as a subsystem in the rest of the shell.

1.2 Target Audience

This nature of this document involves the functional requirements and specifications of the components of the mini-game and a general overview of conventions required in the design. The sections vary in their level of technical detail; consequently, the document's relevance to any particular class of readers is variable from section to section. The chapters in the document are each described with their target audience based on their level of abstraction.

Chapter 2 gives a general description of the application and its place in the environment of the system. It describes the general operation of the mini-game and the kinds of users that the game will be designed to accommodate. This section is recommended for anyone, including users to developers, as it provides information on the overall objective from the development process.

Chapter 3 provides a technical illustration of the mini-game's internal component architecture. It contains object data and feature specifications. It is recommended as a reference for developers, testers, and those involved in producing documentation given this section's reference to overall program functionality requirements.

Chapter 4 offers detail on the testing processes that will be implemented to maintain precise and accurate program operation. It is recommended for developers, as they will be responsible for automated low-level regression testing, and testers who need to know how to direct the testing audience.

Chapter 5 contains more general requirements on the production of the mini-game. The information varies in technical abstraction and detail and is recommended for project developers and coordinators and managers.

Chapter 6 deals with development time and cost estimations and is recommended primarily for

project coordinators and managers to facilitate the regulation of developers and the development process.

1.3 Project Scope

The Temperature Control Manufacturing Game is a mini-game which the user plays to gain bonuses for the main game. In the main game, the player manufactures and sells a product. Playing the mini-game decreases the cost of the user manufacturing his/her product. The mini-game itself is responsible to challenge the user by giving him/her four pots, each on their own heating plate. The pots heat up, and the user must cool them down individually to prevent each pot from overheating an exploding. The user loses if all the pots are destroyed.

The main end goal of this particular mini-game is to, after the user has played, return a score to the shell derived from the user's performance. To successfully accomplish this, it must properly handle every stimulus from the user during its operation.

1.4 Definitions

Shell: The shell is the part of the game that encompasses each mini-game. It is not the entire application and it is not a mini-game itself. Although it has a UI with game-like features, it does not have the same mechanics that mini-games are intended to provide. It is the place in the entire game that the user is at when he is not in a mini-game. The user can launch mini-games from the shell. Mini-games interface with the shell. Their complete specification is in a different Software Requirement Document.

Blit: In Pygame, the term blit refers to the action of rendering an object onto a surface.

Pygame surface object: In Pygame, all rendering is done by blitting images or other drawings as pixels into a buffer on a surface object. From there, they can be sent to the screen where they become visible to the user.

Primary Pygame surface object: This is the Pygame surface belonging to the main window.

2 General Description

2.1 Product Perspective

The Temperature Control Manufacturing Game is one of six mini-games in the entire Dye Merchant Game. The user activates the mini-game from the shell and the program control is passed to the mini-game. Apart from a reference to the primary Pygame surface object that the mini-game will draw on, the mini-game shouldn't need any external interaction with other components after it is initialized because it doesn't need to change non-local data states and non-local states are not modified and don't require to be re-read while the mini-game is running. Because communication is limited, implementing a modular design is simplified. After the mini-game runs, it returns a value indicating the score that the user obtained for that game.

2.2 Basic Overview of Mini-game

This section briefly describes important routines during the operation of the mini-game for the purpose of generating user documentation and as an outline of the features that will be mentioned as the document progresses.

2.2.1 General mini-game procedure

The goal for the user is to prevent pots from overheating and exploding. There are four pots that rest on heating plates. The pots heat up over time and the player must lift individual pots up off the heating plates to cool them to prevent them from overheating. After 90 seconds, the mini-game ends and the user's score is sent back with the program control to the shell. At that point, the shell will probably display information about high scores and continue on its merry way.

The pots themselves have indicators of their actual temperature and the temperature at which they become ruined. As the pots heat up, their appearance becomes less saturated. As they reach the point at which they blow up, they begin bubbling which produces a visual and auditory warning.

At any point during the game players can pause and refer to the instructions. They can also quit the game. How this affects their score at the level of the shell is currently undefined.

2.3 Target Users

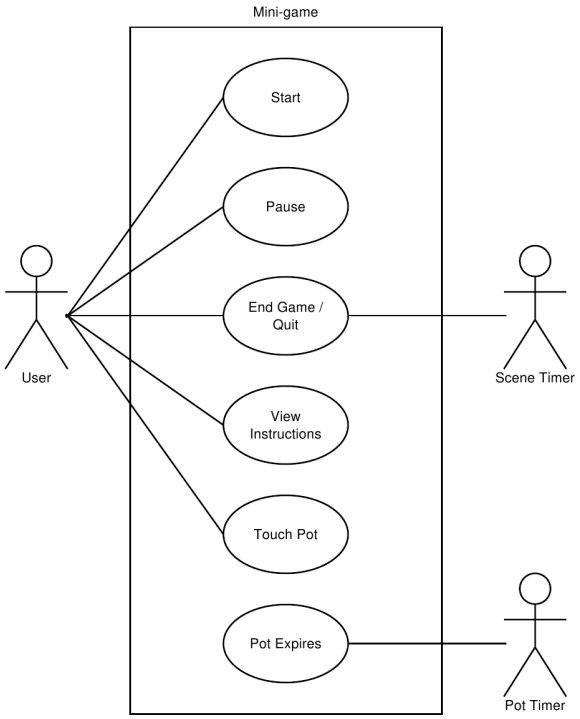
The mini-game is aimed at anybody with basic computer operating knowledge. Instructions will be provided so that people with the ability to operate a computer can learn how to play the mini-game. The nature of this project is not such that potential users are limited to a small class, such as doctors or system administrators. There are no strict rules on who the game is designed for. Only that it be accessible for whoever is playing.

3 Component Architecture

This section describes the component design and organization of critical objects in the mini-game. Each documented component includes a statement of functional requirements in the form of defining use cases, a preliminary data dictionary, and corresponding diagrams. The use cases can serve to define testing procedures for comparing the final product to the specifications of this document.

3.1 Use Case Diagram

This use case diagram is a summation of the use cases in the preceding sections.



3.2 Scene Object

The scene object is the primary component in the mini-game. It encompasses miscellaneous GUI elements and timers. The scene starts at the launch of the mini-game, and closes when the mini-game is over.

3.2.1 Functional Requirements

Some functional requirements generated from the need to make the game practical, rather than just from use cases, must be specified.

Firstly, the mini-game should have a minimal learning curve. That is, the user shall be able to learn how to play the game in less than four minutes from the mini-game's first run. This should be achieved by displaying instructions to the user the first time he launches the mini-game. Additionally, a practice game may be required.

Secondly, the component loading time must be under three seconds for systems above the minimum hardware requirements defined in the glossary.

Use Case: User Starts

Actors: User.

Description: The user starts the mini-game after it loads.

Data: The start button will control a boolean variable. When true, the mini-game will play. When false, the mini-game will be paused.

Stimulus: A command which propagates from the user, by a keyboard or mouse-click event.

Response: Scene prompts user for confirmation and quits if confirmed.

Upon loading, the mini-game will be in a paused state. The start button will be displayed. When it is activated, the mini-game will begin.

Functional Requirement Definition

The user shall be able to start the game only after the mini-game has finished loading.

Upon activating the start button, the game shall start.

Testing The Requirement

In order to test the requirements, we must load the mini-game and check if the mini-game is in a paused state. If the mini-game is in a paused state, we can test to see that the start button starts the mini-game.

Use Case: User Pauses

Actors: User.

Description: User pauses the mini-game.

Data: A flag for the running/paused state of the scene will be modified.

Stimulus: A command which propagates from the user, by a keyboard or mouse-click event.

Response: The mini-game pauses. This includes freezing the timers and blocking user events on pots. Some indication should be displayed to show the game is paused.

The mini-game component should allow the user to pause the mini game at any point during run-time. An indication of the paused state may be a visual overlay (possibly with text) and/or a menu with additional actions for the user. Possible actions for the menu may include; changing options, returning to the shell, resuming the game.

Functional Requirement Definition

The user shall be able to pause the game at any time.

Testing the Requirement

Testing the functionality of this function is very simple as it will affect the GUI by freezing the current state.

Use Case: User Exits

Actors: User.

Description: The user quits the mini-game at any point in its duration.

Stimulus: A command which propagates from the user.

Response: Scene prompts user for confirmation and quits if confirmed.

The mini-game component should allow the user to quit the mini-game at any point during runtime. A some event, such as a keyboard press or mouse click, will trigger an action which activates a quitting procedure in the scene. The scene may prompt the user for confirmation and, if the user accepts, end the mini-game and pass program control to a quitting procedure in the scene.

Functional Requirement Definition

The user shall be able to quit the mini-game at any time.

Testing The Requirement

Validating this specification is straightforward. The stimulus for the behaviour operates on the GUI level. Therefore, checking that the application conforms to the specifications can be

done through interaction with the GUI. We can check that particular events trigger the necessary actions and that the confirmation prompt quits or resumes the game as it is told.

Use Case: Show Instructions

Actors: User.

Description: The user can find help and instructions at any point during the mini-game's runtime. Additionally, the instructions will be provided automatically upon the mini-game's first run.

Stimulus: Displaying information either relevant to the mini game loading or requested.

Response: Displaying information relevant to the mini game loading or requested.

The game should have a list of questions very likely to be asked, or a FAQ with their response. Also, during loading time of the mini game instructions should be displayed with the objectives since the nature of each mini game will be a little different. User should be able to access this information through the main menu game or the menu within the game.

Functional Requirement Definition

The mini-game shall present instructions to the user on the first run. The user shall be able to view instructions upon demand.

Testing The Requirement

Testing would involve validating that the required information would be displayed at the appropriate triggers.

Use Case: Time Expires

Actors: Timer.

Description: The time runs out and the mini ends.

Data: Number of pots left not blown up. Amount of time left.

Stimulus: No time left in the game.

Response: Mini-game ends and application control returns to the shell.

If all of the time runs out, the mini-game has ended. The mini-game should let the user know their score and possible prompt them if they want to retry or quit. If the user chooses to retry, then the mini game should restart else go back to the shell game.

Functional Requirement Definition

If the time runs out, the mini game should end and the user be prompt if the user want to

retry.

Testing The Requirement

Testing should be straightforward since it is simple interaction with the mini game. To test we could put the timer at a low time and set some of the pots to be blown and see if the correct action are taken at the correct time.

3.2.2 Data Dictionary

Identifier	Description	Type
pots	An array of pot objects	list<Pot>
running	A flag used for pausing and resuming the game	boolean
game_over_timer	A timer that counts down from 90 seconds, the game is over when it reaches zero	Timer
surface	Pygame Surface object given by shell, used for drawing on	pygame.Surface
held_pot	A reference to the the pot currently being held. Is NULL when all pots on on heating plates.	Pot

3.3 Pot Object

The pot object is the class which contains relevant data and method for each pot. Since pots and heating plates have one-to-one relationships, heating plates can be included in the pot object. Such that the methods for drawing each pot will also include drawing of the heating pad.

3.3.1 Functional Requirements

Use Case: User Interacts With Pot

Actors: User.

Description: The user lifts up the pot or places one down.

Data: The pot is the user is interacting with. If the pot is blown up.

Stimulus: The users clicks with a pot.

Response: Conditions, such as the pot's blown up state, are validated. The pot animates. If the pot is lifted, it begins to cool. If it is lowered, it begins to heat.

Users can click and hold their cursor down on pots to lift them up. If a user tries to lift up a

pot that has blown up, their input is rejected. Once a pot is lifted, it starts to cool. When the user releases the mouse, or moves the mouse off of a pot of the game doesn't keep their cursor stationary, the pot should drop onto the heating plate and start to heat up.

Functional Requirement Definition

If the user holds the mouse down on a pot, it should raise and cool down, if valid, other wise there should be an error response.

If the user stops holding the mouse down on a pot, the pot should lower and heat up.

Testing The Requirement

Testing the requirements should be simple because the GUI interactions are straightforward. We can check if the pot lifts up when we click on it unless it is blown up in which case there should be an error response. Also if we no longer hold the mouse on the pot it should lower. Also which can check to see if the rate of temperature increase changes in the two positions.

Use Case: Pot Blows Up

Actors: Timer

Description: One of the pots get too hot and blows up.

Data: Temperature of the pot. Rate of increase of the temperature of the pots. Number of pots not blown up already.

Stimulus: The timer detects the pot is too hot.

Response: The pot is visually rendered as being destroyed. Game ends if all pots are blown up.

If the timer detects that one of the pots is too hot, it should cause that pot to blow up. If there are no more pots left, the mini-game is lost. The scene is notified that the game is over and it may return to the shell with the result and allow the shell to display the score to the user and offer him/her to retry the game.

Functional Requirement Definition

If one of the pots is too hot, it should blow up and if all the pots are blown up then the mini is lost.

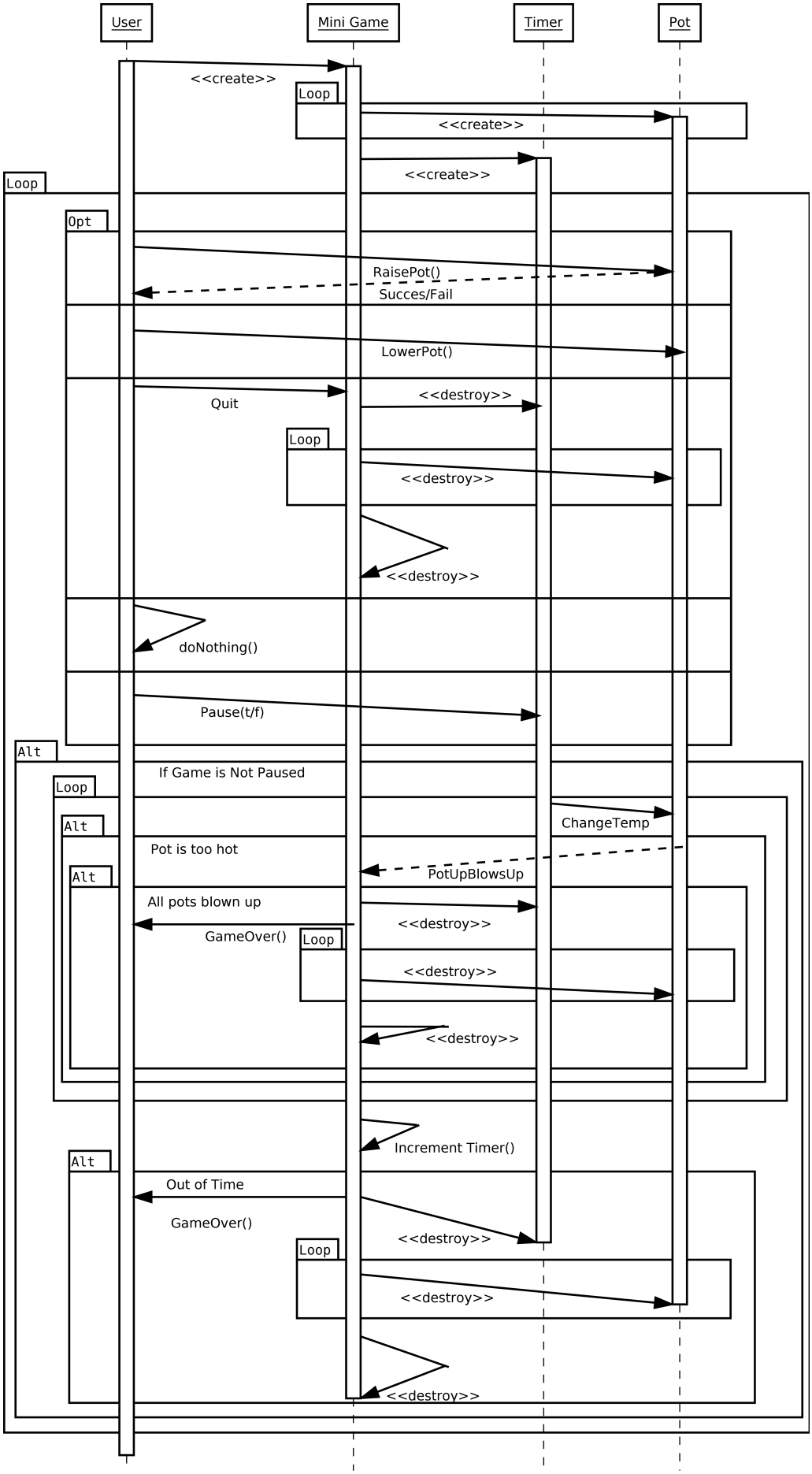
Testing The Requirement

Testing the requirement is easy since without any user interaction the pots should blow up after an interval of time. We can check to make sure that the timer accurately keeps tack of the temperatures of the pots and blow up if they get too hot. Also we can check to see if when the last pot blows up the mini game is lost and the necessary action are taken.

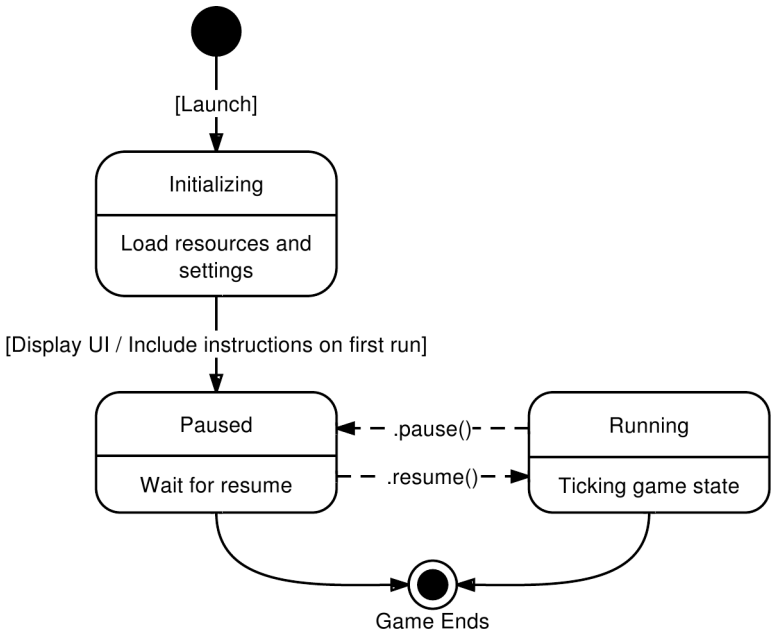
3.3.2 Data Dictionary

Identifier	Description	Type
location	Information corresponding to the location of the pot in the scene	int x, int y
temperature	The temperature of the pot	integer
temperature_delta	The amount that the temperature increments	integer
temperature_limit	The temperature at which the solution in the pot becomes ruined	integer
is_down	A flag that is true if the pot is on the heating plate	boolean

3.4 Sequence Diagram



3.5 State Diagram



4 Testing

To maintain software reliability, a testing infrastructure may be required and employed throughout development. To facilitate this engineering technique, python provides a unit testing framework called unittest'. This framework allows developers to prevent the creation of new bugs through regression testing. Although regression testing is theoretically expensive due to the nature of being redundant, the python API facilitates efficiency in conjunction with good programming practices.

To increase the benefits of using python's unittest framework, it is recommended that the development team take advantage of other testing techniques that naturally work with regression testing, such as test-driven development, and that they structure the application internally such that it is easy to find defects.

Test-driven development relies on developers writing a unit test before developing a bug-fix or implementing a feature. The unit test is designed to fail at first, but work after sufficient code has been put in place. The developer confirms that his fix works, and that he didn't introduce new problems, by running the unit test with all the previously existing unit tests through the regression testing infrastructure.

An additional method of maximizing the testability of the mini-game is in how well the testing team simulate and automate UI events. On the highest level, they could test the system

by reproducing events from the window manager, or some application external to the game. This level of abstraction and disconnection can produce exceptional overhead and can be costly to set up. The next level down is to use python's unittest framework. Pygame has a test' library, in pygame.test, that may be useful to use with python's unittest; but, it is poorly documented and it is uncertain it has even been developed to a degree where it is useful. Nevertheless, using unittest, and pygame's more commonly used components, it is possible to create events at a high level, pass them to the pygame application, and access low-level data structures to verify the completeness and health of the system.

Furthermore, developers could design the internal application structure such that the UI elements and their corresponding functionality are sufficiently separated so they can be reproduced - for instance, by calling a method - without the need for simulating Pygame's event objects. In fact, decoupling an application's UI and its core logic is an important engineering practice that has been gaining widespread popularity because of its more powerful and flexible design. This is the motivation for model-view-controller relationships. An example of this is in Qt. In Qt version 4, in comparison with version 3, all data produced by the WYSIWYG editor is void of substance in and of itself so that it must manually (by the developer) be given a widget to associate with. To illustrate, one may compare this to having a painting and placing it on a canvas. In Qt version 3, the canvas and the painting were produced together. In version 4, they structured it so that the painting must be placed on a canvas by the developer. In addition to that UI change, they separated views and models in many widgets where it was beneficial to do so - like in tables, lists, and combo boxes - to enforce model-view-controller relationships.

5 Other Requirements

5.1 Non-Functional Requirements

Regarding performance and gameplay, there are two primary requirements that must be met for basic acceptability.

Firstly, the game should maintain an average frame rate of 30 frames per second throughout the run-time. If the system detects that the frame rate drops below 25 frames per second across a three second time delta it should record and document the mini-game status so that the causes of this performance deficiency can be detected and optimizations can be created for those conditions.

Secondly, the mini-game shall be winnable. This is an important feature for balancing the game and making it fun. It is also challenging given the rules specified by the game designers. Part of the problems exist in the random nature of the mini-game. Pots are created with random temperature limits and increments. Another part of the problem is that there are four pots which, on average, increase at n per unit of time. This means that, when the user is not holding a pot, the system heats up at $4n$. When the user is holding a pot, the pot's temperature will decrease by $2n$. Which means that the system heats up at $3n - 2n$, or n. In both cases, the overall heat of the system increases at a positive rate. Therefore, the system is always heating up and there is no guarantee that it is possible for no pots to be ruined after constantly heating for 90 seconds, the duration of the mini-game. To account for this, we must either vary the temperature delta for each pot by enough such that lifting one up can reduce the overall system heat or at least adjust it so that the overall system heat never gets too high in 90 seconds. At this time, the game rules are not defined well enough to develop a formula for this specification.

5.2 Internal Requirements

5.2.1 File Formats

Ogg is the required audio format for sounds played in the mini-game. Typically, libraries have best support for WAV files, because they are raw and uncompressed. The Ogg format is an open format so is well supported in the Pygame library. The advantage with the Ogg format or the WAV format is that files can be compressed.

PNG is probably the ideal image format, primarily because it provides lossless compression as well as an 8-bit transparency channel. JPEG images have better compression, but are lossy, produce artifacts - particularly when text is involved, and don't naively support transparency. Rendering JPEG images with transparency can be achieved using transparent colour keys in Pygame's surface objects; but, it only produces binary transparency which may be suboptimal for this project. For the time being, PNG images are preferred.

5.2.2 Animation

The Pygame library does not have data types specifically for animated sprites so we will have to provide our own framework to obtain that functionality. Support for animated objects should be straightforward to design. Apart of that design is the specification of images associated with animated sprites, which is as follows. Images for an animated sprites must be as wide as the

product of the width of one frame and the number of frames in the animation. The image then contains each frame side-by-side in a row to produce a single wide image. When the sprite renders, it slides along its image and draws the section of the entire image that corresponds to whichever frame is needed.

5.2.3 Event Handling

Event handling in the Pygame library is rudimentary. The API allows the mini-game to query Pygame to gain access to basic input events like user keypress and mouse related actions. However, a more elaborate mechanism may be required to achieve a more powerful and object-oriented design.

Robust GUI frameworks, such as Qt, operate by sending events to objects along a parent-child hierarchy. Developing around such a model may be critical to obtaining a flexible and scalable internal architecture for providing a coherent UI. An event dispatcher object, that implements an Observer design pattern, to organize and structure events could potentially meet this requirement.

5.2.4 Resource Management

Properly managing image and sound resources is important to developing an extensible application.

For images, careful planning is imperative for engineering a performance optimized structure. Problems to consider include, but are not limited to; the performance impact of copying images, caching data for shared use, and closing unused resources. One strategy to solve these problems is to use the COW semantic.

For sound resources, a manager object may be required to be the interface to Pygame's mixer API. This manager would need to manage sound pre-loading and caching, playing sounds simultaneously, and closing resources when they are no longer needed.

5.3 External Requirements

The mini-game is a component of a larger system, the entire game. As a result, the mini-game depends on the external shell for access to components that are shared to all mini-games and to obtain run-time dependent information. An interface between the mini-game and the shell should be specified at a low level and the necessary data transfer be sufficiently documented

to obtain a competent design. Some preliminary designations have been stated in the data structure of the scene component above.

5.4 Design Constraints

An important consideration in the development of the software architecture for the mini-game rests in Pygame's API for rendering. The nature of this API is such that rendering happens by blitting objects to a surface before being drawn on the screen. Some drawing operations can only be done on these surface objects. For instance, Pygame's transform library, which is used for rotating, scaling, etc. operations, only operates on surfaces.

The surface object can be viewed as an abstraction of an image. It is essentially a collection of pixels with properties and methods you would expect from an ordinary image. Surfaces have colour palettes, support for transparency, widths and height, can be copied or drawn on each other (pasted), and can provide direct access to a pixelarray. But, Pygame has expanded surface objects to have more elaborate functionality and optimization for the Pygame environment; for instance, surfaces can have parents. In Pygame, surfaces are not of a resource archetype like an image. In this respect, surfaces are more powerful. However; Pygame demands that specific, constrained, and thought-out drawing methods be employed in the software architecture to produce a coherent and maintainable UI.

6 Estimated Resource Requirement

- Two people contributing 50 hours total.
- Two work stations with project work suites installed
- Internet (to look at examples and forums)
- Animation PNGs (for the pots and the avatar)
- Image of stovetop with burners
- Images of thermometers (and/or any other method of temperature display the game design team desires)
- Backgrounds images

6.1 Learning

It is most likely that the people involved in this project are not entirely familiar with the pygame environment and will have to research the keywords and coding styles in order to accomplish their task.

2 people approx. 6 hours.

Total: 12 hours

6.2 GUI/Loader

The GUI will require a little bit of overhead to get started. It must load and arrange the graphics, create hotspots and determine their sizes, and arrange all the animations appropriately.

1 person, approx. 10 hours.

Total: 10 hours

6.3 Refresh Screen

The graphics will have to update in real time.

1 person approx. 1 hour.

Total: 1 hour

6.4 Animation

Create a superclass of type animation in order to load each animation uniformly. Then subclass for each different png animation. 1 person Approx 5 hours.

Total: 5 hours

6.5 Functions

These will be required for the various use-case scenarios described in section 3. When the user takes an action, a function must be called in response.

6.6 Administrative Functions (start, exit, pause, resume)

These will be relatively simple functions. Start, pause, resume will only require the manipulation of a Boolean value. Exit will have to tie up all the loose ends of the program and then close it (or return to the main program).

1 person Approx. 1 hour

Total: 1 hour

6.7 Callbacks and Asynchronous Procedures (Timer)

The timer runs continuously, as it counts down the remaining time in the minigame.

1 person Approx. 30 minutes

Total: 0.5 hours

6.8 Testing/Debugging

This is a highly graphical minigame, which means that most of the bugs will come in displaying images, and positioning them correctly in relation to each other. In addition, GUI interfaces tend to be easily broken by users, and thus need to be tested extensively.

2 people Approx. 8 hours.

Total: 16 hours

6.9 Miscellaneous

Difficulties usually come up at some point in the process of programming, and so there is a small gap of time left unassigned in order to compensate for this lost time.

4.5 hours.

7 Glossary

GUI or **UI**: (Graphical) User Interface. In the context of this mini-game, any user interface will be graphical.

Transparent colour keys: When rendering 2D sprites. A popular technique to achieve transparency without a normal image channel is to designate a particular colour as the transparent colour key. When a sprite is rendered, any pixels of that transparent colour key are ignored and the result effect is binary transparency.

Pixelarray: A Pygame object that provides direct pixel access to a surface through an interface similar to a regular python list.

COW: COW, copy-on-write, is an implicit sharing technique for optimizing resource access. The notion is that a resource that is accessed multiple times only needs a single instance of itself

instead of a separate copy for each accessing object. Only when that shared copy is modified will it produce a new copy for the object requesting a modification.

Minimum hardware requirements: The minimum hardware requirements for an application to run this mini-game properly is equivalent to the hardware specifications of the computers in the senior computer lab. This definition is subject to change.