Soul Collector

Computer Science 483W Team – The U.N.

Chis Lawyer, David Tracey, Ed Lu,

Mujing Wang, Mustafa Al-Oreaydh, Sha Huang

Spring, 2013

Pennsylvania State University

Table of Contents

Contents

[1 Abstract 3](#_Toc352243247)

[2 Requirements 4](#_Toc352243248)

[2.1 Background 4](#_Toc352243249)

[2.2 Essential Solution 4](#_Toc352243250)

[2.3 Environment 5](#_Toc352243251)

[2.4 Implementation Outline 5](#_Toc352243252)

[2.5 Operational Assumptions 5](#_Toc352243253)

[3 Graphical Representation 6](#_Toc352243254)

[4 Diagrams 11](#_Toc352243255)

[4.1 Decomposition 11](#_Toc352243256)

[4.2 UML Diagrams 12](#_Toc352243257)

[4.3 State Diagram 14](#_Toc352243258)

[5 Modules 15](#_Toc352243259)

[5.1 Game Class 15](#_Toc352243260)

[5.2 Back-End Classes 15](#_Toc352243261)

[5.3 Front-End Classes 16](#_Toc352243262)

[6 Coding Standard 17](#_Toc352243263)

## 1 Abstract

Soul Collector is a top-down exploration game in which a dead protagonist attempts to escape a labyrinthine purgatory with multiple floors of traps and demons. In order to get through the maze, the player must utilize items that he finds to solve puzzles and escape the demons. The end goal is for the player to collect the pieces of his soul scattered across the various floors.

The game begins with the protagonist awaking in a random part of the purgatory with a memory of his death and a voice speaking to him. The voice explains to him that his soul has been broken through life and scattered throughout the labyrinth. If he can find all of the pieces of his soul before falling into darkness, he will be transported out of purgatory.

Light attracts demons (even though otherwise, they are blind) because they wish to destroy it; however, the player’s ability to see throughout the maze depends on light, whether it is the light from the Zippo he found in his pocket, or the light of the parts of his soul.

The player makes a choice between avoiding and killing the demons. Generally, it is more difficult to avoid the demons, and killing them seems like the best solution. However, the game has an implicit timer; as the player kills more demons (and slowly as time goes on), he becomes more beast-like. As the player transforms into a beast, he can utilize more direct and violent means of solving puzzles and defeating demons. If the player completely transforms into a beast, they will lose the will to escape, and become just like the rest of them - in other words, they lose. This mechanic requires the player to balance his humanity and will to escape. If the player chooses to use non-violent means to escape, they will advance through the maze by unlocking various doors with keys or through solving puzzles. But as more doors are opened, more demons are released into the playing area.

## 2 Requirements

### 2.1 Background

Monkey Island and other adventure games have existed since the beginning of the digital age. Recently, games utilizing excellent design have created horror games, such as Amnesia. However, no one (that we know of) has successfully blended the two genres together, creating a 2-D horror game with adventure elements. This is what we aim to create.

### 2.2 Essential Solution

We have described some of the mechanics the player will utilize in the abstract. Here, we will go more into detail.  
 The player will wake up within the labyrinth and a voice will speak to him, notifying him of the objective – to gather the parts of his soul. No other information is given at that time. The player will then take control of his avatar within the maze, with some controls described to him by way of dialog boxes, including the use of the lighter serving for his light source.

The measure of a player’s “demonic state” (how far he has gone in becoming a beast himself) will be measured on a bar on the player’s head’s up display (HUD), along with the player’s health. This HUD will be rendered as various labels and bars, on top of the portion of the maze the player can see. The player will be controlled using the arrow keys on the keyboard for movement.

The player’s sight will be limited if he does not have his lighter out. He will be able to see for only a very short distance in the dark, but monsters will also not be able to see him. The first encounter with a demon will be a scripted sequence in which there is a light in a hallway, and the monster rushes to attack it and extinguish it. In this way, the player will know that a demon will attack upon seeing a light.

Various puzzles will be scattered throughout the labyrinth. For example, one puzzle we may include is a door puzzle. If the player has transformed into a strong enough demon, he may simply break down the door using demonic powers, but this will further advance his demonic transformation. If the player does not wish to keep transforming, he may go and look for a key instead, or perhaps some gasoline to light the door on fire. Either option would entail further searches through the maze, making the player open more doors and let more demons loose.

The levels of the maze will be interconnected. The player will not be able to do them in any order they choose, but one level will be connected to many, allowing for some choice. This also means that demons from one level may be able to wander to other levels, making those more difficult. There will be breaks between the levels where the player does not have to fight or avoid monsters, so that the game will not feel completely tense all the time. These sections will most likely have non-violent puzzles in them.

### 2.3 Environment

The project will be cross-platform, so we discuss requirements for each OS.

Requirements for the Windows version of the product are:

* Microsoft Windows XP
* Microsoft C++ 2012 runtime

Requirements for the Linux version of the product are:

* A recent variant of Linux (executables will be created for popular distributions, such as Debian and Fedora)
* C++ libraries
* SFML runtime library

Requirements for MacOS have yet to be determined.

### 2.4 Implementation Outline

The development of this project will be in Visual Studio 2012. We will use SFML for graphics, sound and input, and Git for source control.

### 2.5 Operational Assumptions

We assume the computer is running at least Windows XP, Mac OS 10, or some recent version of Linux that supports shared objects. The system should have at least 256MB RAM, as well as some sort of graphics capability with VBO support. The system must have internet capability in order to download the game; the game will not be distributed on physical media.

We assume that the monitor has at least 800x600 pixels resolution. The player must also have a keyboard with arrow keys. The files, including images, maps, and DLLs, that ship with the game must be in an uncorrupted state.

## 3 Graphical Representation

Here we have some concept art of what the game might look like when it is finished.

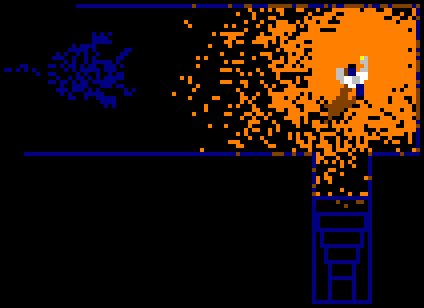


*Figure 1: The menu screen, which shows up when the user first launches the game. The “Play” button takes the player to Fig. 3 or 4. The “Settings” button takes the player to Fig.4. The “Help” button displays the controls, as well as some information about the authors.*



Figure : The settings menu, where the player can change the way the game is displayed and played. The “Resolution” drop-down menu changes the size of the window, as well as the color depth (bits per pixel). The “Difficulty” chooser changes how difficult the game is, including aspects such as player health, monster health, and size of the lighted area. Finally, the “control scheme” chooser changes whether the game is controlled by mouse or keyboard. The back button returns the player to the main menu, in Fig. 1.

Lit area



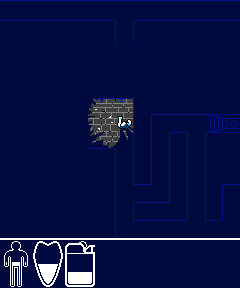
Monster, obscured

Stairs to another area

Player

Figure : In-game concept art. The player, along with a lit area, is next to a monster. Stairs, displayed toward the bottom, take the player to another level of the dungeon. The monster is only an outline because it is not inside the lit area.

Obscured walls (out of sight, but previously discovered)

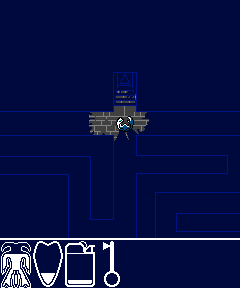


HUD, from right to left: humanity, health, and lighter fuel.

Player

Lit area

Figure : Another (updated) in-game screenshot, this one showing the HUD (heads up display) containing the player’s health, transformation state, and lighter fuel.



Key, picked up from the maze

Locked door

Player, now a monster

Figure : Another (updated) in-game screenshot, showing the HUD where the player has a key in his inventory and is close to being a full demon; note the changed humanity bar.

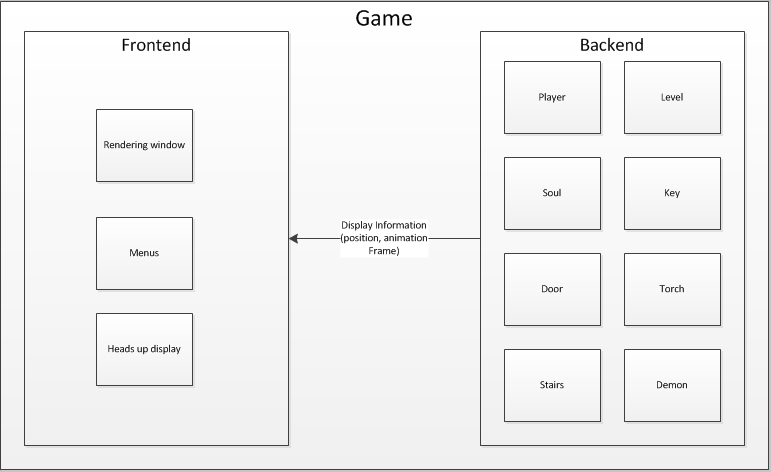


Figure : If the player loses all his/her health, the game ends and this screen is displayed.

## 4 Diagrams

### 4.1 Decomposition

This diagram shows how the game is decomposed into two modules; the frontend and the backend.

Figure : Decomposition diagram showing, at a high level, how the front and back ends communicate.

### 4.2 UML Diagrams

These UML diagrams show the relationships between the objects within the game, as well as how they are defined. *N.B.* Since the game is currently in development, these UML diagrams are subject to change.

#### 4.2.1 Back-End

Note that the elements of the back-end are not strictly computational elements; they have sprites/textures and can draw themselves. This decision was made so as not to make certain information redundant, such as position (which is held in the sprite class).



#### 4.2.2 Front-End



### 4.3 State Diagram

This state diagram shows the regular flow of the game.



## 5 Modules

In this section, we will discuss the different modules (classes) that will make up our project, as well as brief descriptions.

### 5.1 Game Class

The **Game** class serves as the bridge between the back-end and the front-end. It takes care of input detection (and sends the input to the correct back-end classes), as well as rendering and taking care of menus. Only one should be instantiated, and that instance should be alive for the entire duration of the program.

### 5.2 Back-End Classes

* **Level**: Contains the tile information for a single map, including tilesets, layers, and the tilemap. It also includes a list of the entities within that map. Parses ‘.tmx’ files for information. Responsible for collision detection between various entities.
* **LevelManager:** Manages the loading of levels into the games, as well as levels that have already been loaded.
* **Entity**: Abstract base class for anything in the game that can be displayed within the level. Subclass of sf::Drawable. Contains a sf::Sprite and a sf::Texture, as well as animation information for that sprite and texture

Note: although the sf::Sprite does not fit into the back-end (it can be drawn), we decided to include it in Entity because doing so otherwise would require a duplicate ‘position’ field (already extant in the sprite class).

* **AnimManager:** Manages an Entity’s animations.
* **SoundManager:** Manages an Entity’s sounds.
* **Movable**: Abstract base class for any entity that is living and has its own intelligence. A better name for this class is, perhaps, “organism” or “living entity”, but we found these names a bit unwieldy. Includes the players and any monsters within the maze.
* **Player**: Subclass of Movable. Takes care of player movement, collision detection, items, and other.
* **Demon**: Subclass of Movable. Represents a monster within the maze. Takes care of its own movement. The Level object resolves collisions between these and the player. If the player collides with an attacking Demon, then he loses health; likewise for the Demon.
* **Rat**: Subclass of Movable. Represents a small organism within the maze. Runs away from the player when seen. Can be attacked to reduce sanity, but is harmless.
* **Activatable**: Abstract subclass of Entity. Any object in the maze that can be activated.
* **Door**: Subclass of Activatable. Has two states: open and closed. When closed, acts as a solid wall. When open, acts as an empty space. Can be broken if the player has sufficient power.
* **Button**: Subclass of Activatable. When activated, also activates another Activatable.
* **Torch**: Subclass of Activatable. When activated, acts as a stationary light similar to the one the player holds.
* **LightSource:** Does real-time lighting calculations.
* **Trap**: Subclass of Activatable. Activates when any other entity collides with it, dealing damage or trapping the entity.
* **Key**: Subclass of Activatable. Represents a key that can be picked up by the player which opens a specific door.
* **Soul:** Subclass of Activatable. Represents a piece of the player’s soul, part of the goal of the game.

### 5.3 Front-End Classes

* **HUDManager**: Manages the Heads-Up display, which can be seen in Fig. 4/5. Displays the player’s health, sanity, and items.
* **HUDElement**: Represents a single element (image & progress bar) in the HUDManager.
* **MenuManager**: Manages the menus, such as the main menu, the game over screen, and the settings menu.

## 6 Coding Standard

This section of the document describes the coding standard all programmers on the project must adhere to.

**Names and Casing**

All variables shall be camel cased, including those within a class. In contrast, all words within a function or class name shall be capitalized. All “#define” macro names shall be capitalized.

**Indentation**

All code within blocks (denoted by an opening and closing brace) shall be indented by one tab. If a block is nested within another, the block shall be indented by one tab further than its parent.

**Control Structures**

An opening brace should be placed directly after the control structure header, on the same line. In addition, all control structures must have braces. The closing brace should be on its own line.

**Functions**

A function implementation should have its opening brace on its own line, unlike a control structure header.

**Files**

All class and function definitions shall be placed in their own header files, separate from their implementations. These files are denoted using the “.h” file extension. All implementation will be placed in object files denoted by the “.cpp” file extension. All files should have the name of the class they are declaring/implementing as their names, save “main.cpp”, which defines the main function.

**Includes and Forward Declarations**

Wherever possible, forward declarations instead of “#include”s should be used. Includes within a header file are only necessary when the class declared is a subclass of another, or if the class contains an instance of another class.

**Parameter Passing**

All primitive parameters (integers, booleans) shall be passed by value. All non-primitive types must be passed by “const reference”, unless the class is to be modified, in which case is still must be passed by reference. This rule applies to all classes, including those from the C++ STL, the SFML/SFGUI libraries, and our own classes.

**Comments & file headers**

Each header file shall have a header, denoting a brief description, the author of the file, the date, and an (optional) extended description in Doxygen format. Each function declared within the header shall have a brief overall description, a description of parameters and return values, and an (optional) extended description, also using the Doxygen format. All data members shall have a brief description.

The code following is an example of all the rules discussed previously.

**Foo.h**

/\*\*

\* \brief Simple description of the class.

\* \author Edward Lu

\* \date February 2 2013

\* Extended description.

\* May span several lines.

\*/

//Must include because foo is a subclass of baz

#include “Baz.h”

//Do not need to include because it’s only used as a parameter

class Qux;

class Foo : public Baz

{

public:

/\*\*

\* \brief Gets bar.

\* \return The value of bar.

\* Some extended description can go here.

\*/

bool GetBar();

/\*\*

\* \brief Does something with qux.

\* \param someQux An instance of qux.

\*/

void DoQux(const Qux& someQux);

private:

/\*\* Some variable. \*/

int bar;  
}

**Foo.cpp**

#include “Foo.h”

//Now we need to include qux, because we are using its members

#include “Qux.h”

Foo::GetBar()

{

return bar;  
}

Foo::DoQux(const Qux& someQux);

{

bar = someQux.GetData();  
}

## 7 Tests

### 7.1 Unit Tests

**Player movement test**

1. Open Player.cpp. Scroll down to Player::Update and place a breakpoint at the start of the "if(moveUp)" block of code.
2. Run the game while debugging. Press the "start" button.
3. Press the 'w' key.
4. Step through the code until the line "movement.y += std::max(-moveSpeed, (float)nearest);" is encountered.
5. Step once more. Verify that the value of "movement.y" has changed to "(0.0, -2.0)".

**Level Loading Test**

1. Open Game.cpp. Scroll down to “Game::Init()” and place a breakpoint on the levelManager.LoadMap() line of code.
2. Run the game in debug mode. The breakpoint should trigger nearly immediately.
3. Step forward once. Verify that levelManager.LoadMap() returned true.
4. Verify that there are no warnings in the console.

**Torch Activation Test**

1. Open Torch.cpp. Go to Torch::Update, and place a breakpoint within the “if(IsActive()…)” block of code.
2. Run the game in debug mode.
3. Find a torch within the level and activate it by pressing space while the player is colliding with it.
4. Verify that the breakpoint was hit. Continue.
5. Verify that the torch is visibly lit.

**Key Activation Test**

1. Open Key.cpp. Go to Torch::Update, and place a breakpoint within the “if(IsActive()…)” block of code.
2. Run the game in debug mode.
3. Find a key within the level and activate it.
4. Verify that the breakpoint was hit. Continue.
5. Verify that the key no longer appears within the level.
6. Verify that when Space is hit again, the breakpoint is not hit.

**Key Obtain Test**

1. Open Player.cpp. Go to Player::AddKey(), and place a breakpoint within the function.
2. Run the game in debug mode.
3. Find a key within the level and activate it.
4. Verify that the breakpoint was hit. Step forward once.
5. Verify that the door’s name associated with the key is now within the list.

**Stairs Activation Test**

1. Open Stairs.cpp. Go to Torch::Update, and place a breakpoint within the “if(IsActive()…)” block of code.
2. Run the game in debug mode.
3. Find some stairs within the level and activate them.
4. Verify that the breakpoint was hit. Continue.
5. Verify that the level has changed.
6. Activate the stairs again. Verify that the breakpoint is hit once again and that the level changes back.

**Magnitude Test**

1. Open main.cpp. Insert a line of code at the beginning of “int main()”

magnitude(sf::Vector2f(3.0, 4.0));

1. Set a breakpoint at the new line of code. Run the program.
2. Step forward once. Verify that the function returns “5.0”.

**Demon Attack Test**

1. Open Demon.cpp. Place a breakpoint within “Demon::Attack()”, within the “if(state==CHARGING)” block of code.
2. Run the program. Press the Play button.
3. Find a Demon within the level and walk up to it.
4. Wait until the Demon charges at the player.
5. Verify that when the Demon collides with the player, the breakpoint is hit.

### 7.2 System Test

1. Run the program.
2. Hit the play button.
3. Verify that the game loads the level.

### 7.3 Deployment Test

**Windows 7**

1. Run the installer, “SCollectorSetup.msi”, on Windows 7 without Visual Studio installed.
2. Open the shortcut in the start menu, named “Soul Collector”.
3. Verify that the game runs. Hit the “Play” button.
4. Verify that the game loads the first level, and that there are no warnings in the log file.

**Windows XP**

1. The procedure is the exact same as on Windows 7.

### 7.4 Acceptance Test