Introduction to Programming Glossary

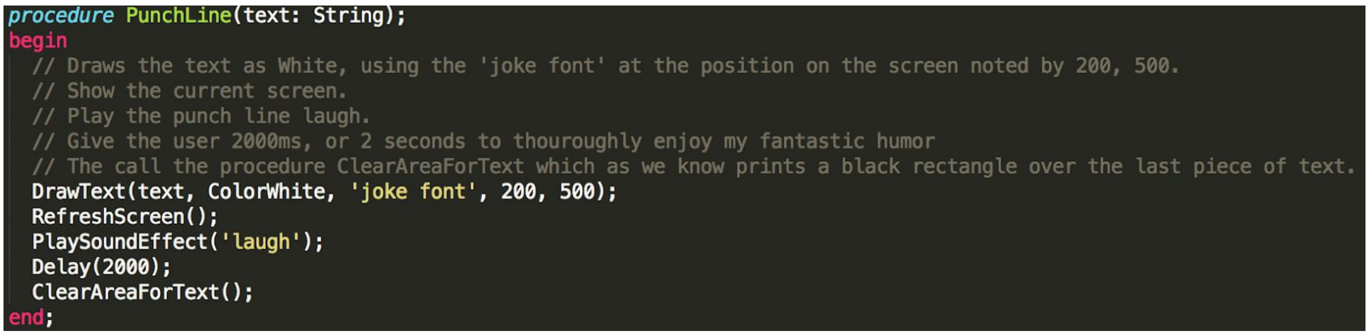
By Reuben Wilson (9988289)

Programming Overview

My *Knock Knock* program uses the *SwinGame* API. It's important that this link is understood because *SwinGame* is responsible for the way that most of the procedures used in this program behave.

Knock Knock consists of a list of procedures, which perform their own tasks. Most of these procedures use parameters or **data**, which are passed to the procedures in question. Here is an example of how one of my procedures uses defined parameters:

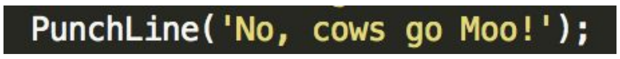
If we look at this procedure, *PunchLine(text: String)*, we know that it is going to be passed a parameter, in this case a string which is considered as data. By looking at the code comments in Fig 1 below, we can understand how this procedure is going to deal with the passed parameter.



```
procedure PunchLine(text: String);
begin
  // Draws the text as White, using the 'joke font' at the position on the screen noted by 200, 500.
  // Show the current screen.
  // Play the punch line laugh.
  // Give the user 2000ms, or 2 seconds to thouroughly enjoy my fantastic humor
  // The call the procedure ClearAreaForText which as we know prints a black rectangle over the last piece of text.
  DrawText(text, ColorWhite, 'joke font', 200, 500);
  RefreshScreen();
  PlaySoundEffect('laugh');
  Delay(2000);
  ClearAreaForText();
end;
```

Fig 1.

Now, lets look at how the procedure *PunchLine* deals with a defined parameter. In Fig 2 you can see that a string *'No, cows go Moo!'* is being passed to *PunchLine* and the resultant is that the output text is white, in a specific font and at a specific location, just like Fig 3 portrays.



```
PunchLine('No, cows go Moo!');
```

Fig 2.

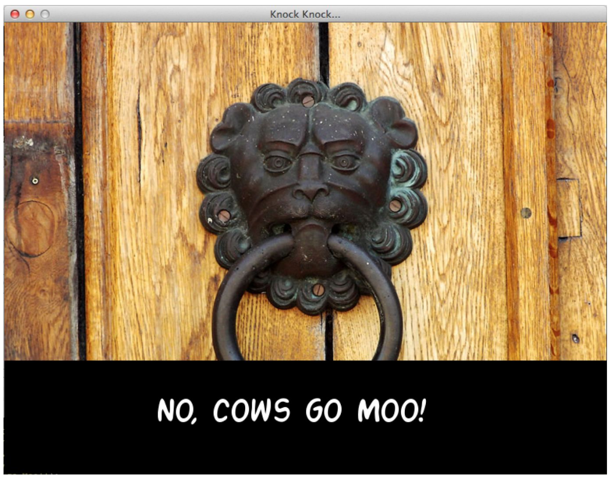
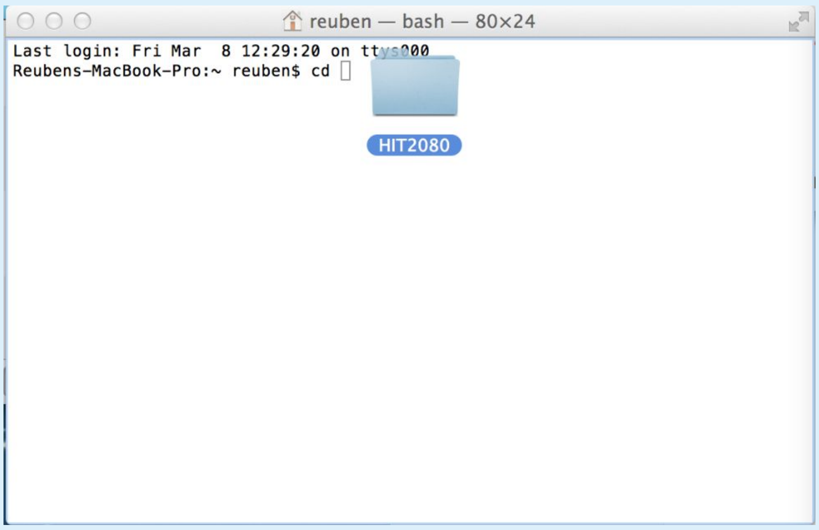


Fig 3.

Programming Command Line Tools:

The following tools are necessary to compile and run programs from the command line:

Command	Description and Example

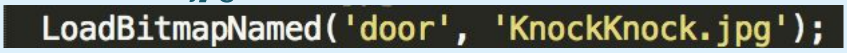
cd	Stands for 'change directory'. My current directory is */Users/reuben/* and I wish to change to the directory */Users/reuben/HIT2080*. This is achieved by executing the following command in terminal:
	
	cd HIT2080/
	
	On OSX, you can simply drag your desired directory into terminal as depicted below:
	
	
	`Last login: Fri Mar  8 12:29:20 on ttys000`
	`Reubens-MacBook-Pro:~ reuben$ cd`

fpc	Stands for 'Free Pascal Compiler'. This is the library, which is used to execute code written in Pascal. E.g. I've a file in the directory */Users/reuben/HIT2080* titled *helloworld.pas* that I wish to compile. This is achieved by executing the following command in terminal (assuming my current directory is the above stated): *fpc –S2 helloworld.pas*
./helloworld	This command is used to run a program titled 'helloworld' which has been compiled using the demonstrated understanding from above. Assuming I've compiled my file titled *helloworld.pas* with no errors, I can now run the compiled executable using this command in terminal: *./helloworld*
./build.sh	This command uses a file titled *build.sh* to execute a predetermined compile script. Let's assume my *build.sh* file is written as follows: *#!/bin/sh* *echo "Starting build..."* *fpc -S2 helloworld.pas* *echo "Done!"*

	So in this instance, if I were to run *./build.sh*, it would print the message *"Starting build…"*, compile my *helloworld.pas* file and finally print *"Done!"*
./run.sh	This command uses a file titled *run.sh* to find and run a file, which has already been compiled using the *./build.sh* command. *./build.sh* will compile to a project source directory. *./run.sh* will then run the compiled output in that directory.

Programming Terms:

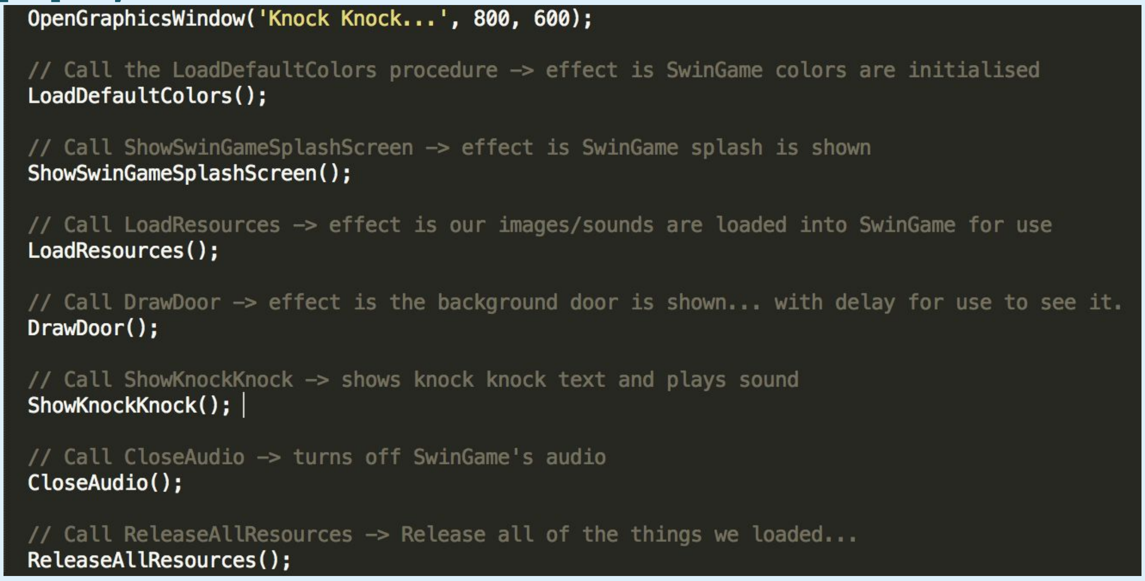
The following terms have specific meanings for software developers. It is important to understand these terms, as they will be used to communicate ideas.

Term	**Description**
Statement	An instruction inside the program. E.g. This statements purpose is to load an image called *'KnockKnock.jpg'*.  `LoadBitmapNamed('door', 'KnockKnock.jpg');`
Expression	A value used within an instruction, or statement.
Identifier	These are used to name sections of code.

Programming Concepts:

The following concepts are central to procedural programming.

Concept	Description

Sequence	A *sequence* is a specific list of instructions, which can be made up of any number of events. When the program is run, none of the listed instructions can be skipped. Each has to be run in the order in which they are specified.
	
	E.g. This particular sequence of codes states a specific order in which the procedures are run. The order is as follows:
	
	1. *OpenGraphicsWindow();*
	2. *LoadDefaultColors();*
	3. *LoadResources();*
	4. *DrawDoor();* etc.
	
	The program relies on this sequence to be executed in order to run properly.
	
	`OpenGraphicsWindow('Knock Knock...', 800, 600);`
	
	`// Call the LoadDefaultColors procedure -> effect is SwinGame colors are initialised`
	`LoadDefaultColors();`
	
	`// Call ShowSwinGameSplashScreen -> effect is SwinGame splash is shown`
	`ShowSwinGameSplashScreen();`
	
	`// Call LoadResources -> effect is our images/sounds are loaded into SwinGame for use`
	`LoadResources();`
	
	`// Call DrawDoor -> effect is the background door is shown... with delay for use to see it.`
	`DrawDoor();`
	
	`// Call ShowKnockKnock -> shows knock knock text and plays sound`
	`ShowKnockKnock();`
	
	`// Call CloseAudio -> turns off SwinGame's audio`
	`CloseAudio();`
	
	`// Call ReleaseAllResources -> Release all of the things we loaded...`
	`ReleaseAllResources();`

Structured Programming

If you refer to my description of a sequence above, you can understand that sequences are important because they are running a set of instructions in an order, which is intended. The program will not branch off or skip any steps when the sequence is followed. That being said, if I were to write a program, I would intend for every line of code to be run, hence the sequence of these events is important.

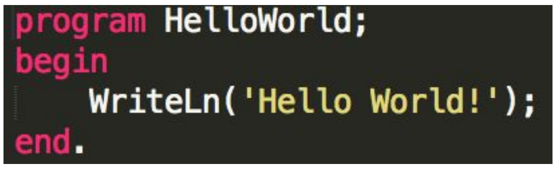
Program

	Details / Answer
A program is ...	A set, or sequence of instructions, which perform a specific task.
A program contains...	All sorts of goodies including functions, parameters, procedures

Example 1: HelloWorld.pas

This is a simple program that all programers have made along their path to programming glory!

So, what this little program does is uses the *WriteLn* procedure to print the text *'Hello World!'*. The string '*Hello World!'* is known as a parameter, which is being passed to the procedure *WriteLn.*



```
program HelloWorld;
begin
    WriteLn('Hello World!');
end.
```

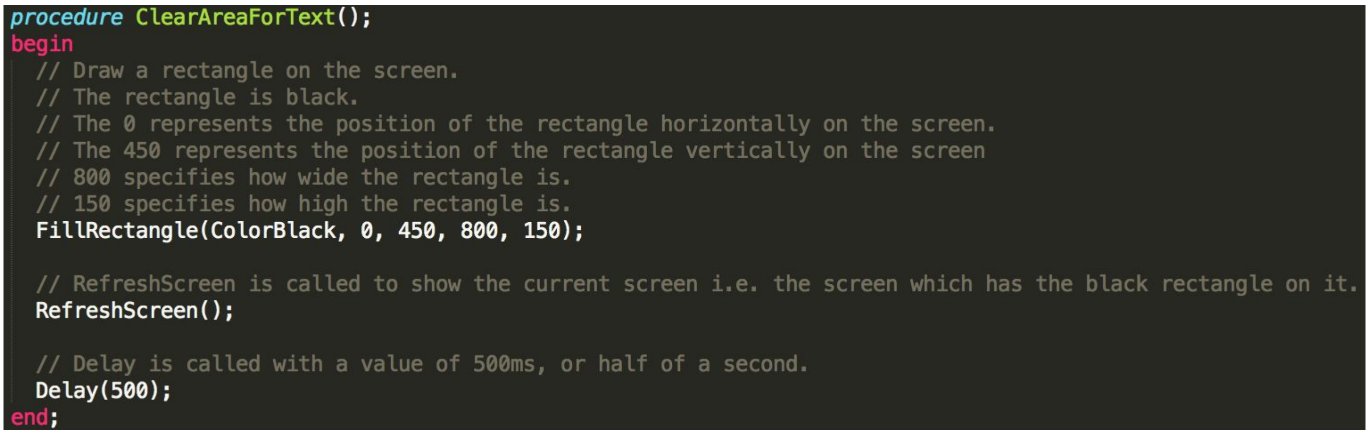
Procedure

	Details / Answer
A procedure is …	Named set of instructions that perform a specific task. Procedures can be defined and called upon in the program.
Procedures can contain	A plethora or instructions, ranging from printing a string on the screen as text or asking for data input, such as your name, loading images, sounds etc.

Example 1: ClearAreaForText

This procedure (Fig 4) simply draws a black rectangle at a specific area, which provides for a clear canvas for some text. Please refer to the comments in the code below to have an understanding of what the lines do.

Pascal Code:



```pascal
procedure ClearAreaForText();
begin
  // Draw a rectangle on the screen.
  // The rectangle is black.
  // The 0 represents the position of the rectangle horizontally on the screen.
  // The 450 represents the position of the rectangle vertically on the screen
  // 800 specifies how wide the rectangle is.
  // 150 specifies how high the rectangle is.
  FillRectangle(ColorBlack, 0, 450, 800, 150);

  // RefreshScreen is called to show the current screen i.e. the screen which has the black rectangle on it.
  RefreshScreen();

  // Delay is called with a value of 500ms, or half of a second.
  Delay(500);
end;
```

Fig 4.

Procedure Execution:

When a procedure is called from within a program, the computer then reads that procedures coded instructions and executes them.

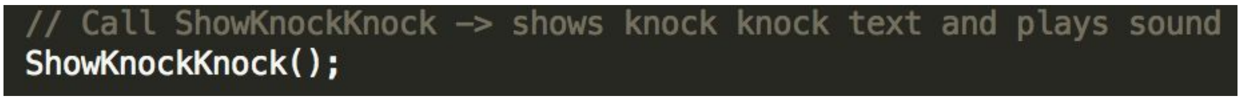
Say I have a procedure, which is instructed to print my name to the screen, when the computer reaches this procedure; it reads what instructions belong to it and executes them. In this case, the instructions were to print my name

Procedure Call

	Details / Answer
A procedure call …	Is when a program has reached an instruction to execute a procedure. In Fig 5, the *ShowKnockKnock* procedure is being called.
Parameters allow you to …	Pass data to a procedure.

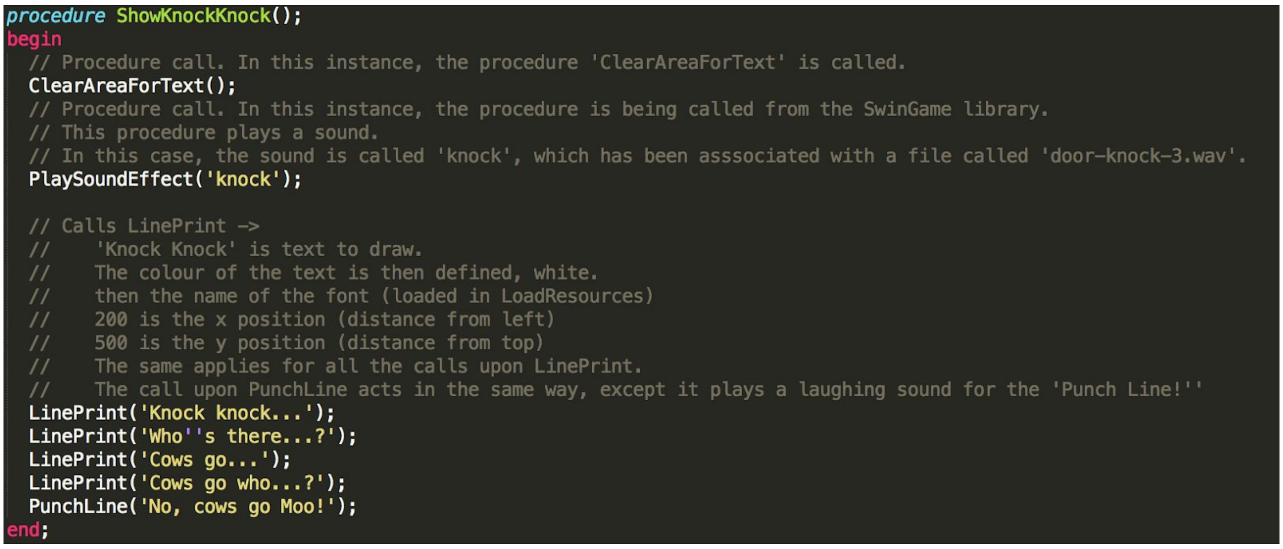
Example 1: ShowKnockKnock

This is a call upon the procedure *'ShowKnockKnock'* (See Fig 5). So now, the instructions, which have been defined for this procedure are called upon. See Fig 6 for the procedures code!



```
// Call ShowKnockKnock -> shows knock knock text and plays sound
ShowKnockKnock();
```

Fig 5.



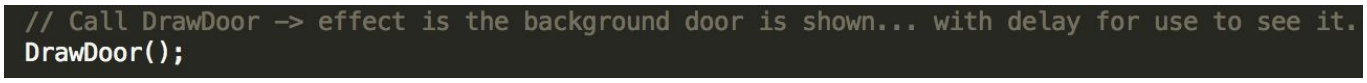
```
procedure ShowKnockKnock();
begin
  // Procedure call. In this instance, the procedure 'ClearAreaForText' is called.
  ClearAreaForText();
  // Procedure call. In this instance, the procedure is being called from the SwinGame library.
  // This procedure plays a sound.
  // In this case, the sound is called 'knock', which has been asssociated with a file called 'door-knock-3.wav'.
  PlaySoundEffect('knock');

  // Calls LinePrint ->
  //    'Knock Knock' is text to draw.
  //    The colour of the text is then defined, white.
  //    then the name of the font (loaded in LoadResources)
  //    200 is the x position (distance from left)
  //    500 is the y position (distance from top)
  //    The same applies for all the calls upon LinePrint.
  //    The call upon PunchLine acts in the same way, except it plays a laughing sound for the 'Punch Line!''
  LinePrint('Knock knock...');
  LinePrint('Who''s there...?');
  LinePrint('Cows go...');
  LinePrint('Cows go who...?');
  PunchLine('No, cows go Moo!');
end;
```

Fig 6.

So, This procedure is responsible for printing my *'Knock Knock'* joke to the screen. The comments in this code define clearly what this procedure does!

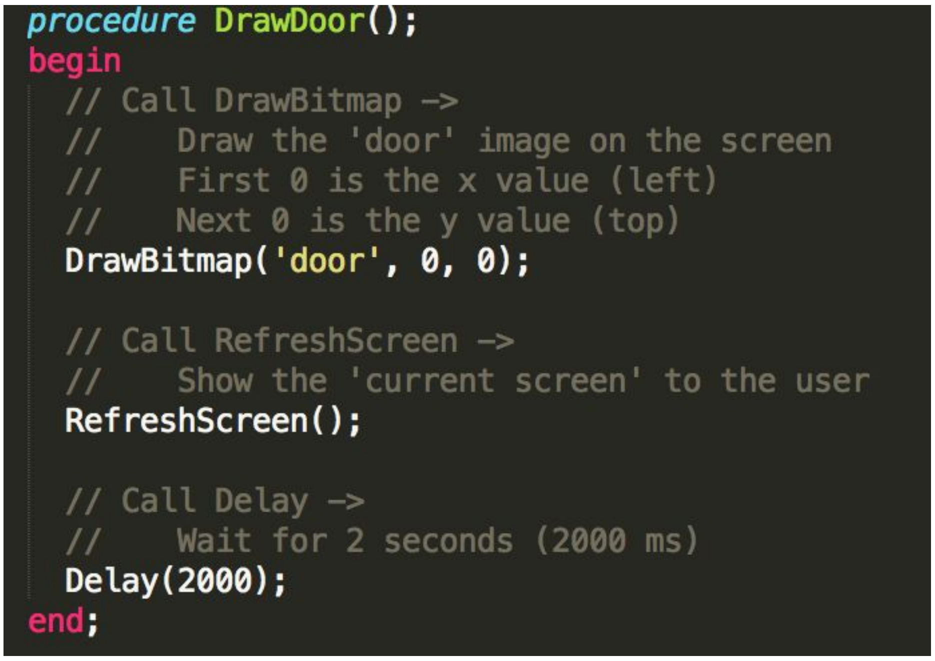
Example 2: DrawDoor

This is a call upon the procedure *'DrawDoor'* (See Fig 7). So now, the instructions, which have been defined for this procedure are called upon. See Fig 8 for the procedures code!



```
// Call DrawDoor -> effect is the background door is shown... with delay for use to see it.
DrawDoor();
```

Fig 7.



```
procedure DrawDoor();
begin
  // Call DrawBitmap ->
  //    Draw the 'door' image on the screen
  //    First 0 is the x value (left)
  //    Next 0 is the y value (top)
  DrawBitmap('door', 0, 0);

  // Call RefreshScreen ->
  //    Show the 'current screen' to the user
  RefreshScreen();

  // Call Delay ->
  //    Wait for 2 seconds (2000 ms)
  Delay(2000);
end;
```

Fig 8.

So, This procedure is responsible for printing my door to the screen, the door is the background image for the joke. The comments in this code define clearly what this procedure does, including positioning of the image on the screen.

Actions Performed

When a procedure call is executed the computer performs the following steps:

1. The computer reaches the procedure call.
2. It remembers that is at that step, it places the procedure in the stack.
3. The computer then goes back through the code to find that particular procedures instructions.
4. Once the computer knows the instructions for that procedure, it executes them.

Knock Knock!

Here you will find three screenshots of my *Knock Knock program running. You will also find the code for this program attached!*

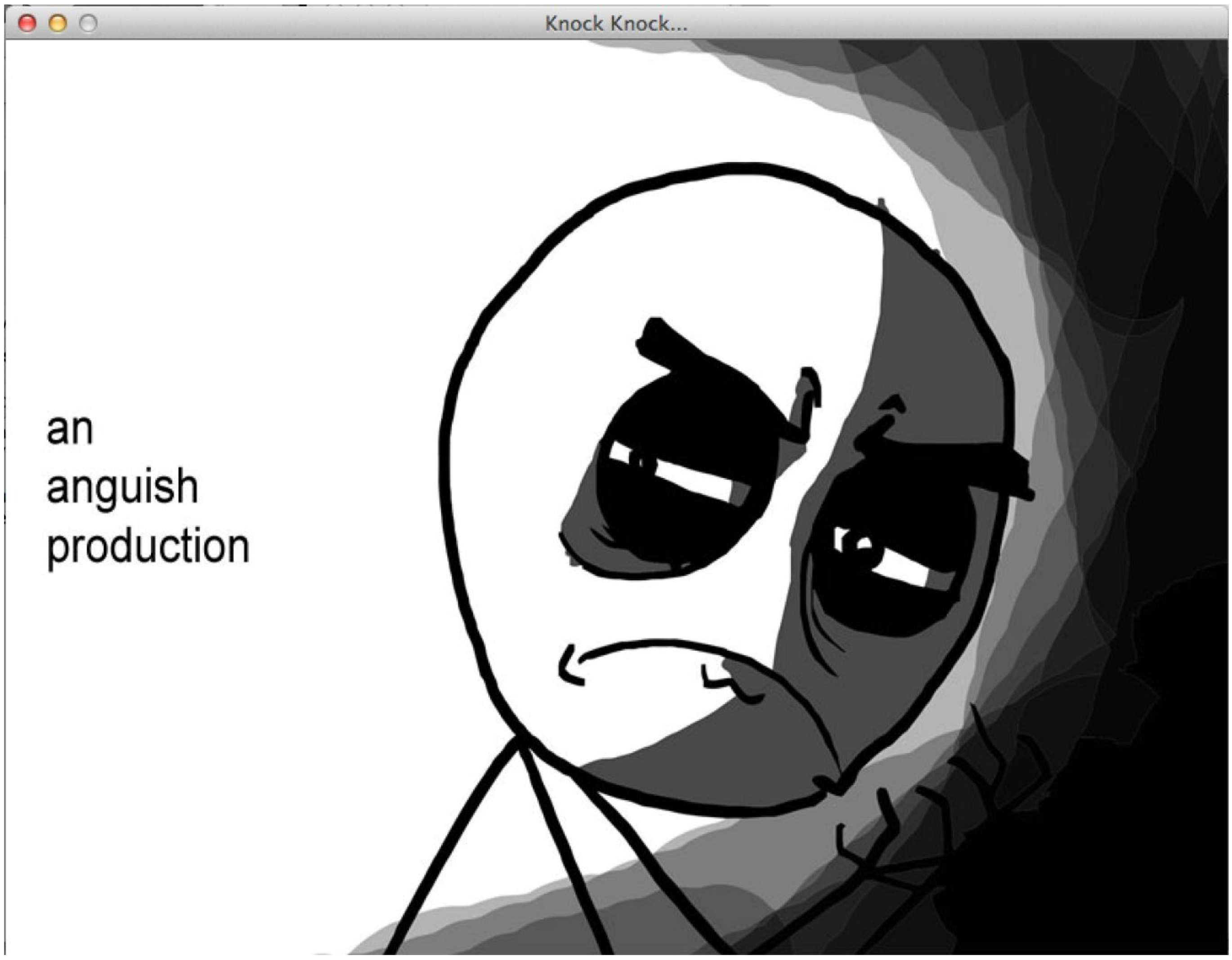


Fig 9.

My customised splash screen for *Knock Knock.*

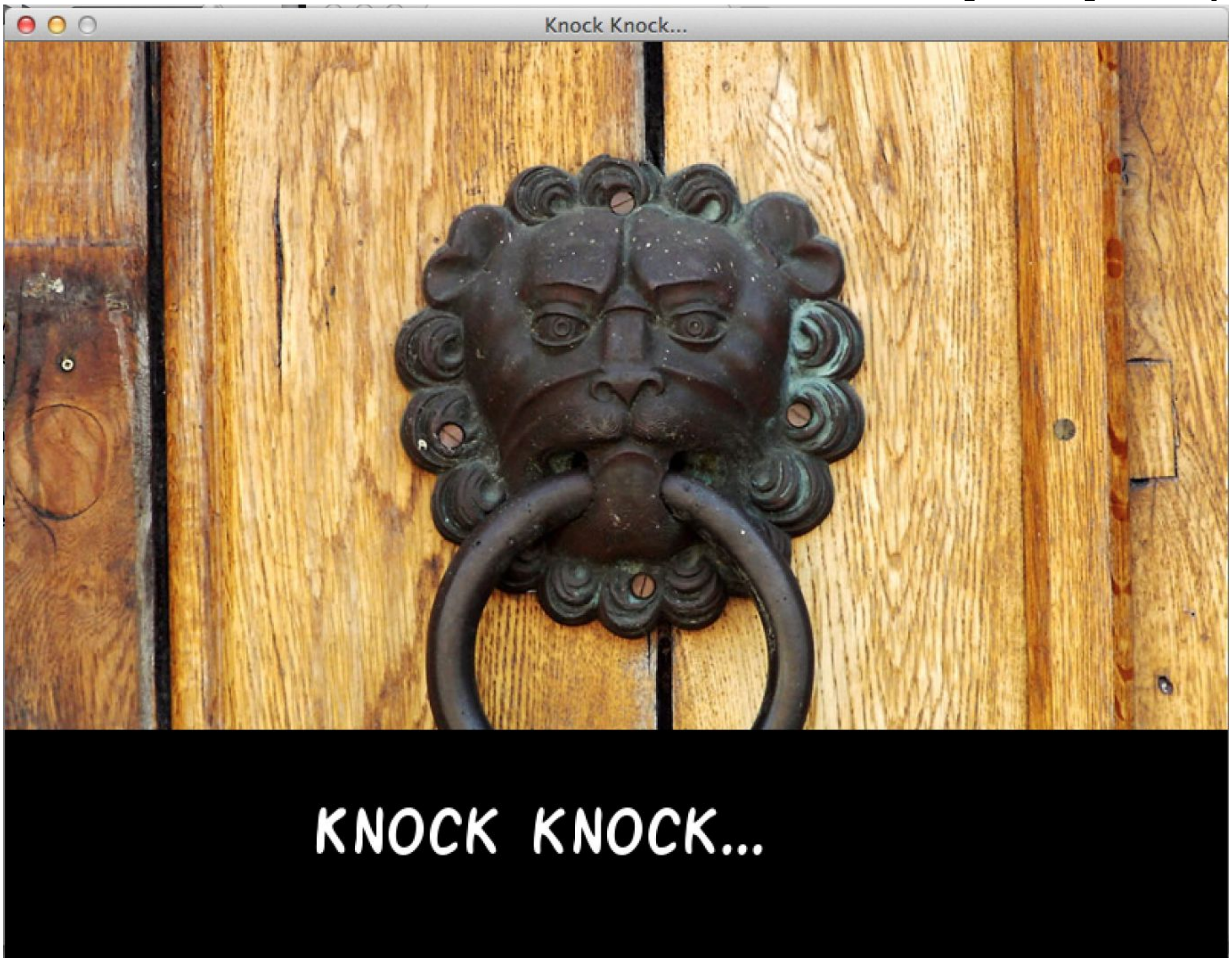


Fig 10.

The joke gets underway!

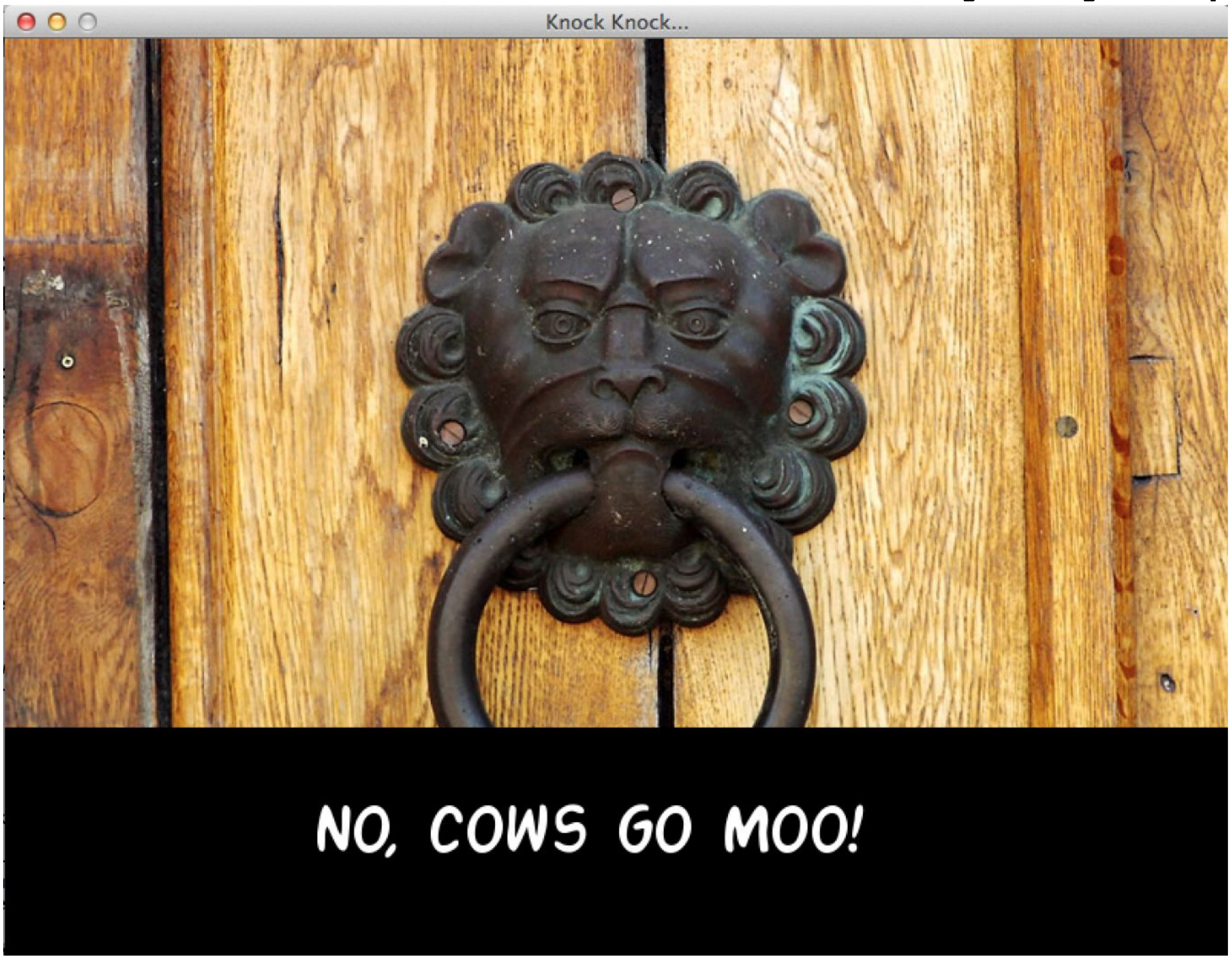


Fig 11.

The punch line!

Extension 1 – Custom Splash!

You can see my custom splash outline in Fig 9 on page 11. It uses sound and obviously a custom name and image. Refer to my *Knock Knock* code for an understanding of how I implemented it.

Extension 2 – Drama 101

Here is my structure chart for my *Drama 101* application.

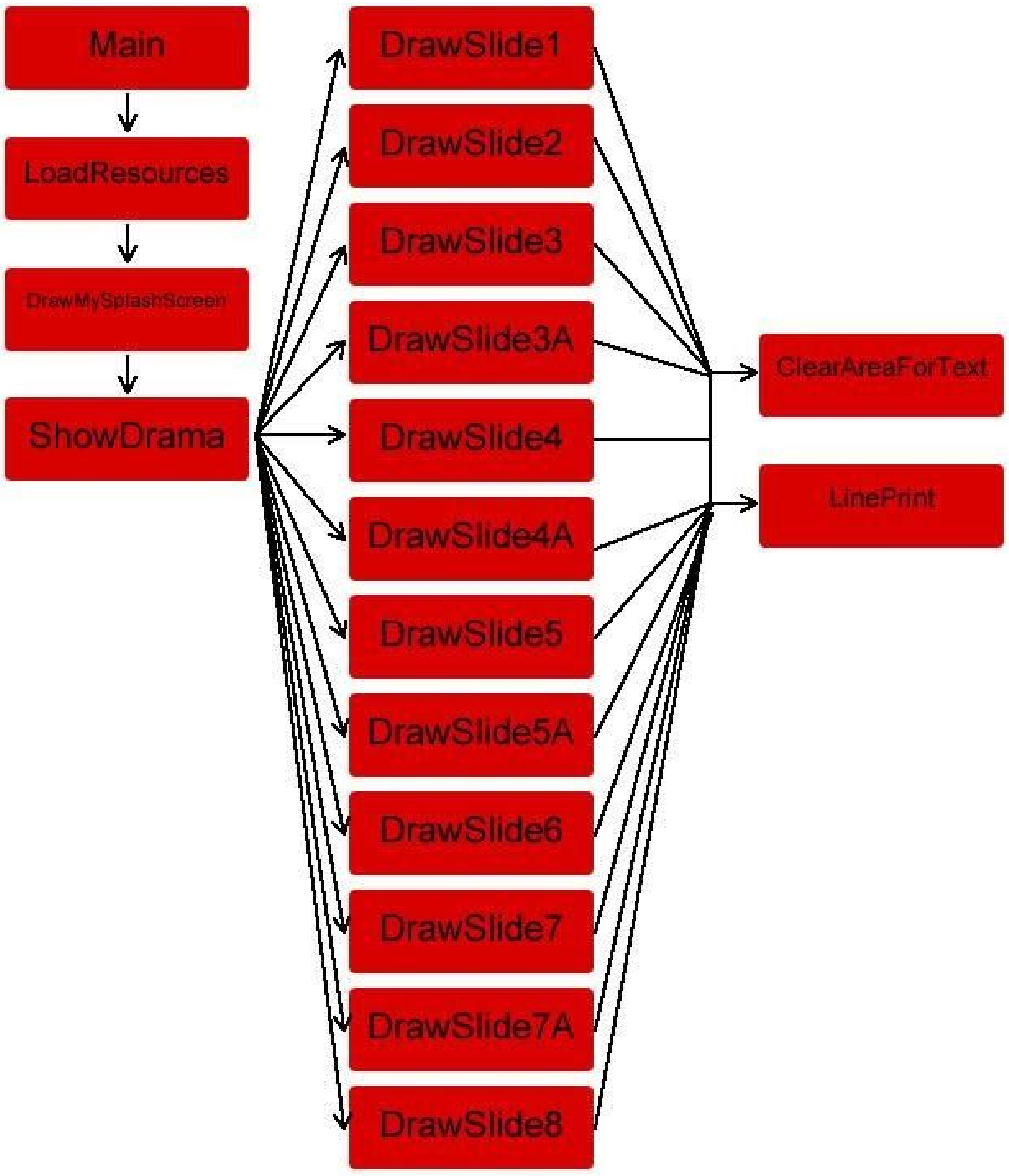

Main
LoadResources
DrawMySplashScreen
ShowDrama
DrawSlide1
DrawSlide2
DrawSlide3
DrawSlide3A
DrawSlide4
DrawSlide4A
DrawSlide5
DrawSlide5A
DrawSlide6
DrawSlide7
DrawSlide7A
DrawSlide8
ClearAreaForText
LinePrint

Fig 12.

Fig 13.

A screenshot of the final scene in my *Drama 101* program. It also uses my custom splash screen outlined in Fig 9.

The code?!

Please find the code for my *Drama 101* attached.

Extension 3 – Custom Program

I'm aiming to write a small RPG, based around the concepts that the early Zelda and Mario games are built upon. The main character will be central to the game play experience and story emersion.

I'm aiming to have items, which the player can acquire to change the way the character interacts with the game environment. These items may include temporary statistical boosts (speed, damage) and health boosts amongst other things.

The game environment will consist of items, physical environment (ground, objects) and non-player computers which will influence the way the user interacts with the game

I'm aiming to make the game experience evolve over three short levels with a key event at the end of each level so the user feels a sense of progress and satisfaction at each stage.

Some of the tasks that the computer will need to undertake in order for this game to work properly are considered below:

1. Sound: The game will rely heavily upon sound for user emersion. Each level will have it's own soundtrack. There will also be sounds played when the player acquires statistical boosting items, defeats non-player computers and succumbs to environmental related incidents like death, gaining life etc.
2. Numbers and statistics: The character that the user assumes will rely on a health point life system. The character will have three lives; each with the capacity of one hundred health points making a total of three hundred health points. So, non-player computers will deal set amounts of damage reflected by whole integers (subtract health points). Beneficial items will give life, or reverse damage (additional health points).
3. Items: A system will be implemented to track items (there will only be a few, I promise) and which ones the character is using, if any.
4. Environments (images and videos): For the start menu, it would be ideal to have a rolling video of gameplay in the background; this would require a procedure to handle videos and images.

The playable levels will be graphical images with set boundaries. For example, the physical ground that the player can walk on would have to be defined as a surface. Dead space like the sky would have to be defined as a non-physical entity that cannot be walked on.

5. Time: Time will be used to track how long a beneficial item can be used for as well as giving the player an idea of how long they have been playing their current level.